

Référence

Pratique de Silverlight

Conception d'applications
interactives riches

Éric Ambrosi

Réseaux
et télécom

Programmation

Développement
web

Sécurité

Système
d'exploitation

PEARSON

Pratique de Silverlight

Éric Ambrosi

Avec la collaboration d'Aude Mousset
et Thierry Bouquain

PEARSON

The Pearson logo consists of the word "PEARSON" in a blue, sans-serif, uppercase font. Below the text is a yellow swoosh that starts under the 'P', curves under the 'A' and 'R', and ends under the 'N'.

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Édition et
et mise en pages : Dominique Buraud

ISBN : 978-2-7440-4125-9
Copyright © 2010 Pearson Education France
Tous droits réservés

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

Préface

IX

Partie I

Approche de Silverlight

1. Introduction	1
1.1 Qu'est-ce que Silverlight ?	2
1.2 De .Net 1 à Silverlight	3
1.3 Les avantages de Silverlight	4
1.4 La suite Expression Studio	6
1.5 Positionnement métier	7
1.5.1 Designer interactif	7
1.5.2 La chaîne de production	8
1.6 Langages de développement et choix	9
1.6.1 Langages accessibles	9
1.6.2 CLR, DLR et langages non compilés	10
2. Le couple XAML / C#	13
2.1 XML	13
2.1.1 À quoi sert le XML ?	13
2.1.2 La grammaire	14
2.1.3 L'ambiguïté des relations familiales	15
2.2 XAML, un langage déclaratif puissant	16
2.2.1 L'utilité du XAML	16
2.2.2 Comparaison XAML / C#	17
2.2.3 Afficher un visuel XAML	18
2.2.4 Espaces de noms	19
2.3 Les fondamentaux du langage C#	20
2.3.1 C# et l'API Silverlight	20
2.3.2 Introduction à la programmation orientée objet	21
2.3.3 Une première application en mode console	22
2.3.4 Les types	23
2.3.5 Déclarer des méthodes	31
2.3.6 Héritage et implémentations	34
3. HelloWorld	37
3.1 Une première application Silverlight	37
3.2 Architecture d'une solution	40
3.3 Le conteneur racine	42

3.4	Ajouter du texte	42
3.4.1	Créer le champ texte	42
3.4.2	Alignement	43
3.5	Tester et compiler	44
3.5.1	Première compilation	44
3.5.2	Une application 100 % Silverlight	45
3.5.3	Fichiers générés	47
3.5.4	Processus de compilation	49
4.	Un site plein écran en 2 minutes	51
4.1	Les projets de type site Internet	51
4.2	Créer des conteneurs redimensionnables	53
4.2.1	Créer le pied de page	53
4.2.2	Créer le menu du haut	55
4.2.3	Créer la grille centrale	60
4.3	Le composant bouton	61
4.3.1	Définir un nom d'exemplaire	61
4.3.2	Afficher une bulle d'information au survol	62
4.3.3	Choisir un curseur de survol personnalisé	63
4.3.4	La propriété Content	64
4.4	Ajouter de l'interactivité	67
4.4.1	Utiliser les comportements	67
4.4.2	Mode plein écran	69
4.4.3	Spécificités du mode plein écran	70
4.4.4	Détecter le changement d'affichage	71
4.5	Fichiers déployés	72
5.	L'arbre visuel et logique	73
5.1	Composants visuels	73
5.1.1	Familles de composants	73
5.1.2	Arbre d'héritage des classes graphiques	75
5.2	Principe d'imbrication	81
5.2.1	Ordre d'imbrication	81
5.2.2	Accéder aux objets de l'arbre visuel	83
5.2.3	Parcourir la liste des enfants	87
5.3	Ajouter des enfants à la liste d'affichage	91
5.3.1	Mécanisme d'instanciation d'objets graphiques	91
5.3.2	Ajouter et insérer des objets dans l'arbre	92
5.3.3	Erreurs levées	94
5.3.4	Créer un menu dynamique	95
5.3.5	Affecter les propriétés Child et Content	97
5.3.6	Événement diffusé	98
5.4	Supprimer des objets de l'arbre visuel	99
5.4.1	Les différentes méthodes	99
5.4.2	L'effondrement de la liste d'enfants	100
5.4.3	Désactiver des objets	103
5.4.5	Événement diffusé	104
5.5	Échanges d'index	105
5.5.1	Échange d'index du point de vue conception	105
5.5.2	Une méthode d'extension pour <code>UIElementCollection</code>	106
5.5.3	Échange d'index du point de vue design	108

Partie II

Interactivité et création graphique

6. Animations	113
6.1 Introduction	114
6.1.1 Qu'est-ce qu'une animation ?	114
6.1.2 La cadence des animations au sein de Silverlight	115
6.1.3 Une première animation	116
6.1.4 Les différents types d'animation	118
6.1.5 Les différents types d'interpolation	120
6.1.6 La classe Storyboard	123
6.2 Animer avec Expression Blend	124
6.2.1 Les bonnes pratiques	124
6.2.2 Créer une animation d'introduction	125
6.2.3 Déclencher des animations	128
6.2.4 Les transformations relatives	129
6.3 Gérer l'accélération	132
6.3.1 Les principes	132
6.3.2 Une animation de rebond en une seule clé d'animation	133
6.3.3 Améliorer l'animation d'introduction	137
6.4 Animer avec C#	138
6.4.1 L'intérêt d'animer avec C#	138
6.4.2 Instanciation dynamique de ressources Storyboard	139
6.4.3 Affectation dynamique de Storyboards	144
6.4.4 Dupliquer un Storyboard créé dans Blend <i>via</i> C#	146
6.5 Les transformations relatives	150
6.5.1 Principes	150
6.5.2 Tester la présence d'une instance TransformGroup	151
6.5.3 Affecter la propriété RenderTransform	153
6.5.4 La bibliothèque ProxyRenderTransform	154
6.5.5 Effets antagonistes et complémentaires	158
6.6 Animer des particules	160
6.6.1 Exemples d'un fond sous-marin	161
6.6.2 Créer l'animation des bulles	165
7. Boutons personnalisés	169
7.1 Créer un lecteur vidéo	169
7.1.1 Mettre en place le projet	170
7.1.2 Insérer une image d'arrière-plan	170
7.1.3 Le rôle du composant Grid	171
7.2 Style visuel	172
7.2.1 Créer un style personnalisé	172
7.2.2 Naviguer entre style, modèle et application principale	174
7.3 Bouton générique	177
7.3.1 La propriété Content	177
7.3.2 Liaison de modèles	178
7.4 Le gestionnaire d'états visuels	183
7.4.1 Évolution de l'expérience utilisateur	184
7.4.2 Au sein d'un control	187
7.4.3 Au niveau de l'application	196
7.5 Le bouton interrupteur ou ToggleButton	201
7.5.1 Créer le bouton	202
7.5.2 Le système d'agencement fluide	203

8. Interactivité et modèle événementiel	205
8.1 Les fondements du modèle événementiel	205
8.1.1 Le pattern Observateur	206
8.1.2 Introduction au couplage faible	207
8.1.3 Souscrire à un événement en C#	208
8.1.4 Cas pratique	209
8.2 Supprimer l'écoute d'un événement	211
8.2.1 Principe	211
8.2.2 Un cas concret d'utilisation	212
8.3 Le couplage faible en pratique	215
8.3.1 Principe	215
8.3.2 Simplifier le code du lecteur vidéo	217
8.3.3 Affectation dynamique d'animations	219
8.4 Propagation événementielle	220
8.4.1 Principe	221
8.4.2 Un exemple simple de propagation	222
8.4.3 Éviter la diffusion d'événements	223
8.4.4 Arrêter la propagation événementielle	225
8.4.5 Glisser-déposer et capture des événements souris	228
8.5 Les comportements	232
8.5.1 Comportements <i>versus</i> programmation événementielle	232
8.5.2 Les différents types de comportements	232
8.5.3 Créer des comportements personnalisés	234
9. Les bases de la projection 3D	245
9.1 L'environnement 3D	245
9.1.1 Introduction	246
9.1.2 Le plan de projection	247
9.2 Les propriétés 3D	249
9.2.1 Rotation 3D	249
9.2.2 L'axe de rotation	250
9.2.3 Les axes de translation	251
9.2.4 Accès et modification <i>via</i> C#	253
9.3 La caméra	257
9.3.1 Position de la caméra dans l'espace de projection	258
9.3.2 L'angle d'ouverture	259
9.3.3 Les conteneurs 3D	260
9.3.4 Le point de fuite	262
9.4 Introduction aux matrices	263
9.4.1 Définition et principes	263
9.4.2 Les matrices 3D	267
9.4.3 Opérations sur les matrices	269
9.4.4 Initialiser une vue perspective	272
10. Prototypage dynamique avec SketchFlow	277
10.1 L'environnement	277
10.1.1 Le prototypage	277
10.1.2 Qu'est-ce que SketchFlow ?	279
10.1.3 Le flux de production	282
10.2 Prototype simple	283
10.2.1 Problématique cliente	283
10.2.2 Le projet SketchFlow	284
10.2.3 La carte de navigation	287

10.3	Le lecteur SketchFlow	290
10.3.1	Navigation	291
10.3.2	Collaboration transverse avec SketchFlow	292
10.4	Interactivité	298
10.4.1	Importer des fichiers PSD	298
10.4.2	Navigation utilisateur	300
10.4.3	Simuler un flux d'utilisation	302
10.4.4	États visuels	303
10.5	Interface riche	304
10.5.1	Écran <i>versus</i> composant	305
10.5.2	L'exemple du configurateur riche	306

Partie III

Conception d'applications riches

11.	Ressources graphiques	315
11.1	Qu'est-ce qu'une ressource	316
11.1.1	Définition et types de ressources	316
11.1.2	Les dictionnaires de ressources	319
11.1.3	Appliquer des ressources	324
11.2	Les pinceaux	327
11.2.1	Couleurs et pinceaux	328
11.2.2	Les pinceaux d'images et de vidéos	336
11.3	Les polices de caractères	340
11.3.1	Afficher et mettre en forme du texte	340
11.3.2	Polices personnalisées	344
11.4	Styles et modèles de composants	348
11.4.1	Les styles	348
11.4.2	Affectation dynamique	350
11.4.3	Organiser et nommer les ressources	351
11.4.4	Les modèles	353
11.5	Le modèle <code>Slider</code>	355
11.5.1	Principes et architecture	356
11.5.2	Un <code>Slider</code> personnalisé	358
11.6	Les liaisons	360
11.6.1	Créer une liaison	360
11.6.2	Les données fictives	367
11.6.3	Le contexte de données	371
11.6.4	Paramètres optionnels	374
11.7	Le modèle <code>ListBox</code>	375
11.7.1	Structure et principes	375
11.7.2	Modifier l'apparence d'une liste d'éléments	377
11.7.3	<code>DataTemplate</code> <i>versus</i> <code>ListBoxItem</code>	379
12.	Composants personnalisés	385
12.1	Contrôle utilisateur	385
12.1.1	Un héritage pertinent	386
12.1.2	Navigation page par page	386
12.2	Boîte de connexion	390
12.2.1	Conception visuelle	390
12.2.2	Validation par liaison de données	393

12.2.3	Rafraîchissement manuel de liaison de données	398
12.2.4	Notification de changement de propriété	399
12.2.5	Événements personnalisés	403
12.3	Contrôles personnalisables	409
12.3.1	ColorChooser et ColorPicker	410
12.3.2	Créer des contrôles	411
12.3.3	Le contrat de modèle	413
12.3.4	Les propriétés de dépendance	423
13.	Médias et données	429
13.1	Chargement de médias	429
13.1.1	Chargement dynamique d'images	430
13.1.2	Formats vidéo et modes de diffusion	438
13.1.3	Un lecteur vidéo simple	444
13.1.4	Chargement dynamique de vidéos	449
13.2	Conception MVVM	452
13.2.1	Principes	452
13.2.2	Modèle	454
13.2.3	Vue-Modèle	457
13.3	Chargement de données	459
13.3.1	L'objet WebClient	460
13.3.2	Introduction à LINQ	463
13.3.3	Expression lambda	467
13.3.4	Consommer du XML avec LINQ	469
13.3.5	Charger un flux JSON	472
13.3.6	Sécurité interdomaines	477
Index		481

Préface

Cela fait maintenant trois années que j'utilise Silverlight et avant lui WPF. Lorsque l'on me demande de qualifier Silverlight, le premier mot qui me vient à l'esprit est moderne. Moderne, l'environnement de travail *via* Expression Blend, logiciel de conception, que l'on pourrait qualifier d'hyper contextuel, et *via* Visual Studio porté en WPF pour sa version 2010. En tant que concepteur, c'est pour moi à ce jour le meilleur outil et produit créé par Microsoft. Moderne, le flux de production et les enjeux humains qu'il engendre dans celle-ci. Silverlight porte en lui une réflexion sur nous-mêmes et sur la manière dont nous catégorisons les acteurs de la production et leurs compétences. Que l'on soit développeur, graphiste, designer, architecte, directeur technique ou artistique, il est impossible de rester insensible à cette mutation qui s'opère autour de nous. Jusqu'à récemment, le design graphique ne concernait pas ou peu le monde du développement applicatif, mais le Web et son évolution ont servi de catalyseur et ont bousculé nos habitudes de production. Moderne enfin, parce que Silverlight représente le fer de lance des technologies dans ce domaine et qu'il échappe aux habitudes et aux clichés.

Les applications évoluent et, de ce fait, la communication qui a souvent fait défaut dans le passé entre techniques et créatifs est remise en jeu. Les frontières s'effacent à différents niveaux. Ainsi, une application perçue comme riche, l'est indépendamment de son support ou de son contexte de diffusion, qu'elle soit bureautique ou Internet. Ainsi, développeurs et designers éprouvent aujourd'hui la nécessité de communiquer et de travailler de concert pour obtenir des applications de meilleure qualité. L'un sans l'autre, ils ne peuvent plus répondre au renouvellement constant des problématiques clientes. Ainsi, les compétences de chacun ne sont plus enfermées et catégorisées aussi simplement qu'auparavant. Silverlight offre, de ce point de vue, un environnement de développement performant, robuste et complet. Il ne se contente pas de porter l'existant .Net dans une application compilée pour le Web ; il correspond pleinement à l'évolution des mœurs et des attentes des internautes et leur propose de nouvelles expériences enthousiasmantes.

Ce livre est bâti sur ces principes, c'est une référence pratique de Silverlight qui vous accompagnera en production. Il s'adresse aux designers interactifs, aux développeurs ainsi qu'à tous les concepteurs souhaitant apprendre le framework Silverlight. Il expose les principes et concepts de cette plateforme et les illustrent par de nombreux exemples téléchargeables. Il s'articule autour des deux axes de conception que sont le design et le développement, et les met en perspective l'un par rapport à l'autre. Si vous êtes designer interactif, ce livre vous donnera une meilleure compréhension des contraintes liées au développement d'applications. Il vous immergera progressivement dans l'univers du développement orienté objet tout en vous apportant les clés de la conception graphique propre à Silverlight. Si vous êtes développeur, il vous sensibilisera aux contraintes nouvelles en matière de design, de prototypage et d'expérience utilisateur. Il facilitera le dialogue et la communication avec les autres acteurs de la production. Quel que soit votre cas, cet ouvrage est conçu pour fluidifier vos productions et établir les bases d'une réflexion intermédiaires respectueuse de chaque profil.

Prérequis

Ce livre s'adresse à deux types de public différents : les designers et les développeurs. La difficulté est progressive. Toutefois, quelques sections peuvent s'adresser à l'un de ces profils en particulier. Ainsi, ne vous formalisez pas sur certains termes utilisés car même sans pratiquer le design ou le développement, il est nécessaire d'apprendre le vocabulaire propre à ces métiers pour communiquer avec chacun des acteurs d'une production.

Dans cette optique, les copies d'écran d'Expression Blend sont en anglais. Il n'est cependant pas nécessaire de connaître la langue de Shakespeare pour comprendre ce livre. De manière plus générale, il est conseillé d'avoir des notions basiques en création graphique ou en développement quel que soit le langage et le niveau technique.

Ce livre se positionne comme un ouvrage d'apprentissage en mode pas à pas. Les trois premiers chapitres ainsi que celui dédié à SketchFlow (Chapitre 10) sont les seuls qui ne nécessitent *a priori* aucune des disciplines mentionnées précédemment.

Organisation de ce livre

Cet ouvrage se compose de trois parties :

- **Partie I, Approche.** Cette partie du livre donne une vision globale de la technologie Silverlight et de son positionnement. Nous y découvrons l'architecture de la plateforme ainsi que son histoire. Nous créons notre premier projet Silverlight et nous y abordons les principes élémentaires de conception liés à Silverlight. Nous la clôturons par un tour d'horizon des composants fournis par la plateforme et nous concevons un premier site brut plein écran.
- **Partie II, Interactivité et création graphique.** Cette partie s'axe essentiellement sur le graphisme et l'interactivité au sein de Silverlight. L'animation, le modèle événementiel, la conception d'interfaces graphiques, la personnalisation graphique de composants ou encore la 3D, sont autant de sujets étudiés. En fin de partie, nous abordons la conception de prototypes dynamiques *via* la technologie SketchFlow.
- **Partie III, Conception d'applications riches.** Le flux de production, le positionnement métier et l'architecture représentent les thèmes de cette partie. Au cœur de ceux-ci, nous étudions notamment la conception de contrôles, l'organisation et le rôle des ressources responsables de la répartition des tâches de chacun, ainsi que le modèle de conception MVVM et le chargement de données externes.

Les exemples

Ce livre contient de nombreux exemples sous forme d'exercices, de projets et de fichiers divers dont la majorité est téléchargeable sur le blog dédié au livre ainsi que sur le site de l'éditeur Pearson. Le blog est conçu dans le but de mettre à jour les projets et de garder un contenu vivant et cohérent. Il est disponible à l'adresse <http://referencesilverlight.tweened.org>. Certains projets doivent être récupérés sur ce dernier avant de pouvoir vous lancer. Il est organisé sous forme de treize catégories dont chacune correspond à un chapitre. Elles donnent accès à différents articles contenant les sources, les projets et certains schémas téléchargeables. Plusieurs composants et autres classes sont en téléchargement libre et peuvent vous aider dans vos productions. Certains

des contrôles ont été récemment ajoutés à la bibliothèque *SLExtensions* ou fondus avec ceux déjà existants. Dans tous les cas, ils sont accessibles sur mon blog <http://www.tweened.org> via la catégorie Composants.

Conventions typographiques

Les conventions typographiques adoptées dans ce livre sont les suivantes :

- **Police fixe.** Cette police met en valeur les éléments du langage.
- **Police fixe grasse.** Dans les extraits de code, cette police met en exergue les modifications apportées aux programmes.

INFO

Ce type de note fournit des précisions en marge du texte.

ATTENTION

Cette icône vous met en garde sur un point précis.

À propos de l'auteur

Issu d'un cursus scientifique à Paris VII, puis d'une formation aux métiers de la production audiovisuelle et de la post-production, Éric Ambrosi est co-fondateur de l'École Européenne Supérieure d'Animation, rebaptisée école Méliès, dont il fut le responsable pédagogique de 1999 à 2003. Il devient par la suite concepteur d'applications Web 2.0 Flash / Flex et responsable technique sur les outils de mobilité pour la société Atos Origin jusqu'à fin 2006. Il est aujourd'hui responsable pédagogique et directeur technique chez Regart.net pour les plateformes Microsoft Silverlight et WPF. Il a été nommé deux fois MVP, une première fois en tant que Client Application Developer pour l'année 2008-2009 et une seconde fois comme MVP Silverlight pour l'année 2009-2010.

Remerciements

Je remercie toutes les personnes qui ont cru en moi et m'ont accompagné tout au long de cet exercice périlleux et fastidieux d'écriture. Je remercie également mes parents et mes beaux-parents qui nous ont soutenus ma femme et moi tout au long de ce parcours.

Je remercie tous mes collègues et ex-collègues de Regart.net. Merci donc à Fatosé et Guylaine, les deux présences féminines de Regart.net. Un grand merci à Vincent Maitray pour m'avoir accordé les droits de la vidéo Petit Pan (utilisée au Chapitre 13), réalisée avec talent en une journée par ses soins. Je remercie également Matthieu Appriou pour ce joli visuel de première de couverture que j'ai repris pour le lecteur multimédia du Chapitre 11. Merci à mon collègue Nicolas pour son humour et son soutien durant ces dix derniers mois. Merci à Thibault pour toute cette énergie qu'il diffuse autour de lui. Merci en vrac à mes amis, Philippe et Samia, Cyril et Aurélie, Candice et Hervé, et Num et Say à qui j'ai pensé ces derniers mois. Ils m'ont supporté durant tout ce temps

par leur présence ou leurs coups de fil même si je ne les ai vu que de manière entrecoupée. Merci à toutes les personnes de Microsoft avec qui je collabore et qui me font confiance.

Un grand merci à mon éditrice Patricia Moncorgé, qui m'a permis d'écrire ce livre, et à ma relectrice Dominique Buraud, avec qui il est vraiment agréable de travailler. Toutes deux forment une très bonne équipe et m'ont permis d'accomplir ce voyage jusqu'au bout.

Merci à mon relecteur côté développement, Thierry Bouquain, compatriote MVP Silverlight qui m'a apporté beaucoup par ses conseils et le temps qu'il m'a alloué. Thierry est co-fondateur de l'entreprise Ucaya qui s'est illustrée sur des projets de Smooth Streaming pour les événements du Tour de France et de Roland Garros entre autres. Merci à Aude Mousset, ma relectrice côté design interactif et amie depuis trois ans, merci donc pour avoir pris le temps de me relire et d'être une force de proposition. Aude est graphiste de formation, experte sur Blend et a travaillé pour I-Breed et Microsoft durant ces dernières années. À travers son travail constant et de qualité, elle a ouvert la voie aux designers. Son aventure continue chez MPower en tant que Directrice artistique et Designer interactif. Merci donc à vous deux, Aude et Thierry, sans qui le livre ne serait pas ce qu'il est aujourd'hui.

Pour finir, je remercie également ma femme pour son soutien, pour m'avoir poussé à écrire ce livre, et pour sa compréhension dans cette épreuve. Merci de nous avoir donné cette jolie Gabrielle qui nous rend tellement heureux. Merci également pour notre fils qui va naître dans les prochains jours et qui fait déjà des siennes dès qu'il entend le rire de Gabrielle ou la musique autour de nous – je vous dédie ce livre.

Un mot des collaborateurs techniques

"La lecture de cet ouvrage renforce ma vision de l'évolution des sites Internet. La dimension interaction et design est de plus en plus présente et devient incontournable pour des sites de qualité. Les bonnes pratiques de développements décrites ici permettent de bien démarrer un projet où la méthode de travail et de collaboration entre designer, intégrateur et développeur est un des éléments clés du succès."

Thierry Bouquain, directeur technique et fondateur de Ucaya.

"En tant que graphiste, Silverlight est un nouveau terrain de jeux à explorer. Les possibilités sont multiples, la technologie est puissante, mais sans connaissances techniques cela peut s'avérer frustrant. Même si vous n'êtes pas développeur, faire un pas vers le côté obscur est un vrai plus dans le cadre de projets Silverlight.

Que vous produisiez en solo ou en équipe, que vous soyez du côté Blend ou du côté Visual Studio, ce livre est un bon moyen d'appréhender le workflow Silverlight. Vous rentrerez rapidement dans le vif du sujet : Éric Ambrosi partage sa maîtrise de Silverlight et vous permet d'appréhender des concepts d'intégration et de développement *via* de nombreux cas pratiques, pertinents et complets.

Un ouvrage indispensable pour tous les *devsigners*¹ !"

Aude Mousset, Directrice Artistique et Designer Interactif pour MPower.

1. Devsigner pour designer + développeur.

Partie I

Approche de Silverlight

1

Introduction

Dans ce chapitre, vous découvrirez la nouvelle plateforme de diffusion plurimédias interactive Silverlight et les raisons qui ont poussé Microsoft à sa conception. Vous apprendrez en quoi Silverlight hérite de la bibliothèque .Net et propose un nouveau modèle de conception avantageux. Nous évoquerons également la nouvelle suite d'outils Expression Studio dédiée aux graphistes et aux intégrateurs et comment ceux-ci se positionnent dans la chaîne de production.

Silverlight laisse aussi bien la place aux langages .Net C# qu'aux langages libres et dynamiques comme JavaScript. Prendre partie pour l'un de ces langages est l'une des étapes incontournables de sa prise en main. Nous listerons donc les critères qui orienteront votre choix parmi ceux-ci. De manière générale, Silverlight est fait pour améliorer notre quotidien de designer ou de développeur, ainsi que l'expérience utilisateur sur Internet. Nous examinerons les moyens mis en œuvre pour atteindre ces différents objectifs.

1.1 Qu'est-ce que Silverlight ?

Silverlight est un lecteur Internet existant sous forme de plug-in, anciennement connu sous le nom de WPF/E, initiales de *Windows Presentation Foundation Everywhere*. Les applications lisibles par ce lecteur sont des fichiers au format xap. Basé sur un moteur vectoriel puissant, il permet d'afficher du contenu dynamique et interactif pour le Web. Il facilite la conception et la diffusion d'applications riches sur Internet. Il fait partie de la plateforme de développement .Net (prononcer "dotte net") proposée par Microsoft, mais ne nécessite qu'une fraction héritée de celui-ci pour être exécuté, autrement dit, l'installation du lecteur seul suffit. Cette particularité lui permet d'être diffusé au sein de nombreux navigateurs sur Mac OS, Windows ou Linux (*via* le projet Moonlight pour Linux, voir Tableau 1.1).

Tableau 1.1 : Navigateurs et systèmes supportés.

<i>Système d'exploitation</i>	<i>Internet Explorer 8</i>	<i>Internet Explorer 7</i>	<i>Internet Explorer 6</i>	<i>Firefox 2 / 3</i>	<i>Safari 3 / 4</i>
Windows 7	Oui	-	-	Oui	-
Windows Vista	Oui	Oui			
Windows XP SP2	Oui	Oui	Oui	Oui	-
Windows 2000 SP4		-	Oui	-	-
Windows Server 2003 / 2008	Oui	Oui	Oui	Oui	-
Mac OS 10.4.8+ (PowerPC)		-	-	Oui	Oui
Mac OS 10.4.8+ (Intel)		-	-	Oui	Oui
Linux, FreeBSD, SolarisOS	Supporté par le projet Moonlight développé par Novell				

Les applications .Net traditionnelles reposent sur la notion de client lourd, c'est-à-dire qu'elles nécessitent l'installation de la bibliothèque .Net. Celle-ci peut peser jusqu'à 195 Mo pour la version 3.5. Le lecteur Silverlight échappe à ce type de fonctionnement peu souple car c'est un environnement d'exécution pour les applications de type client léger. Cela signifie que son poids est suffisamment négligeable (de 4 Mo à 7 Mo selon les systèmes d'exploitation) lors du téléchargement, pour que son installation soit rapide et les applications exécutées très légères (souvent inférieure à 1 Mo). Sous Vista et Windows 7, les droits nécessaires à son installation sont moins contraignants pour l'utilisateur de la machine cliente. Bien que l'on puisse avoir l'impression qu'il s'agit d'un détail, les réseaux d'entreprises actuels sont souvent constitués de centaines, voire de milliers d'ordinateurs, et les déploiements qui en découlent nécessitent d'importants moyens humains et techniques. Dans ces conditions, le déploiement de clients légers est une stratégie efficace à court et moyen termes puisque les applications développées pour ces plateformes sont de plus

en plus performantes et puissantes. Malgré un poids négligeable, Silverlight apporte un environnement de développement complet.

1.2 De .Net 1 à Silverlight

Comme nous l'avons vu précédemment, Silverlight est à la fois une évolution parallèle et un héritage de la plateforme de développement .Net (prononcé "dotte nette"). Qu'est-ce que .Net ? Quel est son but et comment est-elle née ? Répondre à ces questions nous permettra de mieux comprendre les orientations prises par les équipes d'ingénieurs ayant participé à sa conception et nous permettra plus facilement d'envisager les améliorations possibles dans le futur.

Il faut remonter le temps jusqu'à février 2002 pour voir la sortie de .Net 1.0, livré avec le Service pack 1 de Windows XP. Visual Studio.Net est à cette époque, l'outil de prédilection pour développer ce type d'applications. .Net est né des besoins d'interopérabilité (entre ancienne et nouvelle technologie) et de portabilité (valable pour les langages et les systèmes). Les objectifs sont simples :

- Proposer une meilleure plateforme de développement.
- Apporter un environnement indépendant du réseau ou du langage choisis quels qu'ils soient Visual Basic, C# ou C++, etc.
- Délivrer un unique modèle de conception quelle que soit la version de Windows : XP, 2000, Windows CE, etc.
- Concurrencer Java.
- Faciliter la création de services web.
- Accompagner les développeurs et assouplir la méthodologie de développement avec en point de mire l'industrialisation des processus.

Microsoft propose de nombreux systèmes d'exploitation et .Net uniformise le développement d'applications. Les applications développées en .Net ciblent essentiellement l'OS Windows. Par ailleurs Microsoft réalise une première percée significative pour scinder le design des interfaces et le code logique, grâce aux formulaires Windows, surnommés *Winforms*. Avec ce type d'architecture, apparaît la notion de code "behind". Un fichier va contenir le code nécessaire pour fabriquer l'interface visuelle. Celle-ci sera simplement créée par de simples glisser-déposer de composants dans la fenêtre de prévisualisation sous Visual Studio. Un autre fichier contient désormais le code logique, ou code "behind", sans lequel l'interface ne répondrait pas aux interactions utilisateurs et ne posséderait aucune logique. Sorti fin 2005, .Net 2 est une véritable amélioration mais n'apporte que peu de nouveautés. Il faut attendre janvier 2007 pour découvrir .Net 3 qui révolutionne le développement d'applications sous Windows.

.Net 3 n'est pas une refonte, mais une extension de .Net 2. Cette version apporte, en plus de C# 3 ou de technologies comme Linq, de nombreux bénéfices répartis en quatre modules :

- WCF (*Windows Communication Foundation*) facilite la communication entre les applications, les domaines ainsi que partage de bases de données.
- WCS (*Windows CardSpace*) pour l'authentification des utilisateurs.

- WWF (*Windows Workflow Foundation*) pour l'amélioration des flux de production.
- WPF (*Windows Presentation Foundation*) comme couche de présentation.

WPF est la technologie qui nous concerne : Silverlight en est un développement à la fois hérité et parallèle. On peut considérer que WPF représente le futur de la technologie Winforms. Tout en conservant les capacités précédentes, il intègre les profils du designer et de l'ergonome qui, dans 90 % des cas, étaient soit mis à l'écart, soit relégués au second plan. Microsoft comprend en effet que la fonctionnalité n'est plus la seule garantie d'une réussite commerciale ou technique. De plus en plus d'applications voient le jour et beaucoup sont fonctionnelles. Toutefois, seules très peu d'entre elles y ajoutent l'ergonomie et l'esthétisme, deux facteurs que l'on retrouve pourtant dans l'industrie automobile, les téléphones portables et de nombreux autres secteurs. Grâce à l'environnement WPF, vous pourrez changer le visuel d'une application dans sa globalité. Modifier le design d'une barre de défilement ou d'une liste d'éléments se révélera très simple. Tout ceci est réalisable grâce à WPF et au langage XAML (*eXtensible Application Mark-up Language*). Le XAML permet de formaliser n'importe quel dessin vectoriel ou composant visuel sous forme de balises XML. Un client peut dorénavant configurer le visuel d'une application en modifiant des feuilles de styles XAML et y apposer ainsi sa charte graphique sans faire appel à l'éditeur. Certes, ces opérations sont réalisables depuis longtemps en important des bibliothèques entières de classes, mais autant dire que les graphistes n'étaient pas vraiment présents dans le processus de création. *Exit* donc les interfaces grises, rectangulaires, héritées de Windows 3.1.

WPF aborde donc avec succès l'un des grands chantiers de l'informatique moderne : la séparation complète du fond et de la forme, elle dépasse de loin les Winforms sur ce point. D'autres technologies ont également essayé de répondre à cette problématique : XUL, de la fondation Mozilla, ou encore le langage FXG basé sur SVG du côté d'Adobe. WPF a pris de l'avance, mais concerne les développements sur système Windows pour les langages de hauts niveaux. À partir de ce constat, une nouvelle problématique apparaît. De manière générale Microsoft est avant tout éditeur de solutions, quel que soit le système d'exploitation, c'est là son cœur de métier. Il se doit donc proposer des solutions indépendantes du système. Afin de supprimer le couplage existant entre l'environnement de développement et les systèmes d'exploitation Windows ciblés, *WPF/Everywhere* voit le jour. C'est l'ancien nom de code de Silverlight. L'objectif est clair : puisque l'on crée des solutions autant faire en sorte qu'elles ciblent le plus de systèmes d'exploitation possibles. Le moyen le plus efficace consiste alors à proposer un lecteur Internet sous forme de plug-in. Les navigateurs les plus courants sont ciblés en premier, et Silverlight voit finalement le jour en novembre 2007. Cette initiative de Microsoft répond avant tout aux attentes du marché actuel qui évolue rapidement vers les applications en ligne et répond au quasi monopole de la plateforme Flash.

1.3 Les avantages de Silverlight

Développer avec Silverlight apporte beaucoup d'avantages, dont certains ne lui sont pas forcément propres, mais sont plutôt relatifs à l'ensemble des technologies asynchrones présentes sur le Web comme Shockwave, Flash, Silverlight et Ajax. Silverlight est avant tout orienté applications interactives riches et, dans ce cadre, il bénéficie également de toute l'expérience de Microsoft en matière de développement. Ce serait une erreur de penser que les graphistes sont délaissés car des efforts importants ont été réalisés pour fournir des outils de design et d'animations performants. Lors du rachat de Macromedia par Adobe, de nombreux acteurs de talent furent recrutés par Microsoft pour atteindre ces objectifs. Voici les avantages les plus significatifs de Silverlight :

- **Comportement asynchrone.** Ajax est la première technologie asynchrone au sein du navigateur, mais c'est aussi la plus ancienne. Apparu avec Internet Explorer 5 en 2001, cette technologie, rapidement adoptée, provient d'un objet en particulier (`XmlHttpRequest`) permettant d'effectuer des requêtes client serveur. La spécificité des technologies asynchrones réside dans le fait qu'elles ne nécessitent pas le rechargement complet de la page du navigateur à la réception des données serveur. L'utilisateur peut même continuer à utiliser le reste de l'interface sans provoquer d'erreur. Silverlight possède de nombreuses méthodes asynchrones, plus modernes qu'Ajax permettant non seulement les échanges dans un format texte, mais également dans un format binaire. Ainsi, l'échange d'images, de flux vidéo, de données typées est possible ; le mode peer-to-peer est également supporté.
- **Il ne dépend pas d'une technologie serveur spécifique.** Il vous suffit d'un simple client FTP et le tour est joué. Vous n'avez pas besoin d'une configuration serveur spécifique ou propriétaire. Un serveur Apache traditionnel fera très bien l'affaire pour diffuser votre site ou votre application Silverlight.
- **Ouvert aux technologies du Web.** Les langages dynamiques, comme JavaScript, ont la capacité de communiquer avec une instance de plug-in Silverlight de manière transparente. La communication est en réalité possible dans les deux sens. De plus la lecture, l'écriture ou l'envoi de données sont réalisables dans des formats courants, comme XML, ou JSON, de même que l'accès à n'importe quel type de technologie côté serveur. On peut ainsi interfacer sans problème une application Silverlight avec des services, côté serveur, codés en PHP, ASP, etc.
- **Il bénéficie d'une compatibilité multi-environnement.** Comme nous l'avons dit au début de l'introduction, Silverlight est accessible en tant que lecteur sur Mac OS, Windows et Linux (Moonlight) et sur les navigateurs Chrome, Firefox, Internet Explorer 6, 7 et 8, Safari. Votre application, bien que développée avec Visual Studio, Expression Blend ou même Eclipse, sera donc visible et accessible au plus grand nombre.
- **Les applications ne nécessitent pas de déploiement.** Autrement dit, une fois accessible à une adresse donnée, l'application est lisible et exécutable au sein des navigateurs par les internautes. Cela est peut être anodin ou normal si vous travaillez actuellement sur le marché du Web, mais la philosophie actuelle (en mutation) consiste encore à déployer des clients lourds et des applications locales poste par poste. Dans des réseaux comprenant des milliers de postes clients, le déploiement de telles applications est extrêmement coûteux en ressources humaines et techniques. Les applications Silverlight apportent une réponse efficace à ces problématiques de diffusion.
- **Les langages logiques .Net supportés sont gérés.** Le code géré (dit aussi managé) est l'opposé du code interprété. C# ou VB sont des codes compilés, JavaScript, PHP, HTML sont interprétés. Cela signifie qu'ils sont lus par le logiciel d'une machine cliente ou serveur dans le cas de PHP. Pour un langage comme JavaScript, cela se traduit par des différences de performances flagrantes selon le navigateur, mais aussi par des différences d'interprétations. Dans un site web traditionnel, l'aspect visuel, ainsi que la version du langage, peuvent donc varier fortement selon les navigateurs (IE, Firefox ou Safari par exemple). Au sein de Silverlight, toutes ces problématiques sont résolues, l'affichage sera le même quelle que soit la plateforme, la version du langage reste constante. On obtient également beaucoup de performances car le langage est géré. À la compilation, il est transformé et optimisé en langage intermédiaire. Par la suite, selon la machine cliente, le système d'exploitation et le navigateur, le lecteur s'adaptera pour l'optimiser en un langage machine adapté à chaque navigateur (voir Chapitre 3).

- **Il contient un moteur d’affichage vectoriel puissant.** L’ensemble des composants ou objets visuels sont vectoriels. Ils sont donc affichés grâce au moteur vectoriel Silverlight. De nombreuses problématiques, comme le poids des fichiers déployés ou encore la génération d’objets de manière dynamique, sont ainsi résolues. Les animations vectorielles sont non seulement possibles, mais également simples à réaliser. L’affichage de texte avec des polices de caractère embarquées, des vidéos ou des images est aisé à mettre en œuvre. Les visuels ne sont pas figés sous forme de bitmap ce qui facilite grandement la mise à jour des applications et le travail collaboratif.
- **Il améliore la collaboration entre designers et développeurs.** Héritant de WPF, Silverlight bénéficie de la même architecture de production. Cela signifie que le langage XAML est présent pour formaliser l’ensemble des éléments visuels d’une application. Tout ce que le graphiste va produire génère du XAML, de la mise en forme jusqu’au moindre pictogramme visuel. La conséquence directe est que la création d’un composant (ou contrôle) personnalisé, comme une mire de connexion par exemple, s’effectue en deux étapes distinctes complètement autonomes. Les designers interactifs ou les intégrateurs créent les éléments visuels et les animations. Le développeur ajoute le code logique. C’est une vision un peu simplifiée de la réalité, mais cependant assez proche. L’avantage réside dans le fait que les graphistes participent pleinement à la conception. De plus, ce qu’ils produiront sera fidèlement réutilisé par les développeurs et non recréé ou retraduit par ceux-ci. À tout moment, les designers pourront mettre à jour le visuel sans mettre en danger l’aspect fonctionnel ou modifier le code logique.

1.4 La suite Expression Studio

La gamme Expression Studio fait partie de la stratégie WPF. Cette suite de logiciels apporte un confort jamais atteint pour le développement d’applications Microsoft. Les objectifs de la gamme Expression Studio sont :

- Permettre la création d’applications riches connectées (*Rich Desktop Application*) ou d’applications riches pour navigateur (*Rich Internet Application*) via Expression Web ou Blend intégrant Silverlight.
- Améliorer la qualité graphique et l’ergonomie des applications en proposant de nouveaux outils aux profils de type graphiste, intégrateur ou designer dans l’environnement .Net et Silverlight, tout cela en scindant le fond et la forme. Ces outils facilitent la mise en place de un flux de production reposant sur la collaboration intermétiers.
- Proposer de nouvelles solutions aux besoins de la "vidéo à la demande" très présents sur le Web grâce à Silverlight, Expression Encoder et au format *wmv*.
- Répondre aux problématiques traditionnelles propres à la création de site Internet via Expression Web.

INFO

Pour rappel, Visual Studio est l’outil de développement phare de Microsoft. Du point de vue d’un développeur et concernant des projets standard, Visual Studio peut être utilisé seul. En effet, il est possible de bâtir un projet Silverlight sans besoin de logiciels de la gamme Expression Studio (Blend ou Design). Toutefois, cela ne serait pas pertinent, on pourrait en effet se demander quelle serait la valeur ajoutée de telles applications ? La gamme Expression facilite justement l’accès au développement à de nouveaux acteurs. Ces nouveaux profils, que sont l’intégrateur, l’ergonome et

le graphiste, par leur apport, enrichissent de manière considérable l'approche et le ressenti d'une application pour l'utilisateur. La gamme Expression Studio apporte une ouverture exceptionnelle au monde du graphisme qui, jusque-là, était laissée de côté. Blend et Design ont donc pour mission de fournir du contenu graphique et interactif, ainsi qu'une ergonomie accrue en comparaison des applications bureautiques ou Internet standard.

Voici une liste des outils compris dans la suite :

- Expression Web pour la création de sites Internet ;
- Expression Blend pour la conception d'interfaces riches ;
- Expression Design pour la conception graphique vectorielle ;
- Expression Encoder pour l'encodage des fichiers au format wmv en haute définition ;
- Expression Media pour l'organisation et la lecture des médias ;
- Visual Studio pour la conception, le développement et l'architecture d'applications.

Pour notre part, nous couvrirons la conception d'applications interactives riches *via* l'utilisation d'Expression Blend et de Visual Studio.

1.5 Positionnement métier

Comme nous le voyons, WPF et les outils de la gamme Expression sont sur le point de bouleverser les modes opératoires existant. La tendance actuelle repose sur un besoin devenu essentiel qui consiste à formaliser le graphisme grâce à des langages déclaratifs de type XML. Il est intéressant de noter que ce mouvement, initié par Mozilla avec le XUL et par Adobe avec le SVG il y a quelques années, est aujourd'hui concrétisé par Microsoft. Cependant, nous parlons bien de tendance, ce qui signifie que d'autres acteurs comme Adobe reprennent la même direction. Le FXG est la version Adobe du XAML, les deux langages se ressemblent fortement. L'enseignement que l'on peut tirer de cette situation est que les modes et flux de production s'uniformisent et sont en pleine mutation. Les prochaines années seront riches à tous points de vue. Les capacités de Silverlight, Flash et Ajax (et nouvellement XHTML 5) à proposer des langages performants et à diffuser du contenu innovant seront décisives. Ce qui est abordé dans cette section est donc valable pour tous les acteurs du marché quelle que soit leur provenance.

1.5.1 Designer interactif

Au sein de l'environnement .Net, trois logiciels de la gamme sont le théâtre de cette évolution : Expression Blend, Expression Design et Visual Studio :

- Expression Design concerne essentiellement les graphistes. Il contient l'ensemble des outils nécessaires à la création vectorielle. Cependant il joue aussi le rôle de passerelle et permet d'importer nativement de nombreux types de fichiers dont le format Adobe Illustrator ou le format Photoshop psd.
- Expression Blend cible avant tout les designers interactifs, les intégrateurs et les designers d'expérience Utilisateur. Les ergonomes peuvent aussi dans une certaine mesure créer des maquettes et prototype en partie réutilisables (voir Chapitre 10).

- Visual Studio comme outil de développement principal est conçue pour les développeurs et possède un puissant outil de débogage que Blend ne fournit pas. À ce jour, il est considéré comme l'un des meilleurs produits délivrés par Microsoft.

Le tableau 1.2 indique la localisation de Blend / Design dans la chaîne de production, ainsi que l'investissement à prévoir en fonction du rôle de chacun. On remarque que chacun peut l'utiliser, Blend étant un outil polyvalent et multiforme.

Tableau 1.2 : Positionnement de Blend au sein de la chaîne de production.

<i>Rôle</i>	<i>Outils et niveau d'utilisation</i>		
	<i>Expression Design</i>	<i>Expression Blend</i>	<i>Visual Studio</i>
<i>Graphiste</i>	Fort	Moyen	Aucune
<i>Designer interactif</i>	Moyen	Fort	Faible à moyen
<i>Développeur</i>	Aucune	Faible à moyen	Fort

Comme on peut s'en rendre compte, un nouveau profil émerge : le designer interactif. On peut considérer ce profil comme un nouveau genre de designer web. Dans la majorité des cas, c'est un graphiste ayant évolué et acquis une réelle culture de développeur ainsi qu'une connaissance aigüe de l'ergonomie. L'intégrateur HTML représente en partie ce type de profil. Il est à la fois confronté à des langages comme JavaScript, PHP (réalisation de modèles) et CSS, tout en conjuguant cette technique à une sensibilité de graphiste. La différence entre intégrateur et designer interactif repose essentiellement sur la partie animations et transitions. On peut voir le flasher comme designer interactif. Son objectif principal est de coupler le visuel, l'ergonomie, l'animation et les transitions. Il considère la logique applicative et la fonctionnalité comme importantes, mais coulant de source et plutôt dévolues aux développeurs.

Au sein des environnements WPF et Silverlight, le designer interactif tient un rôle crucial puisqu'il fait le lien entre les profils situés à chaque extrémité de la chaîne de production. C'est un élément fédérateur. Silverlight et WPF proposent un modèle qui facilite la collaboration, mais c'est le profil de designer interactif qui formalise et permet cette communication. On pourrait y voir un mélange des genres douteux, mais sans lui, on se retrouverait dans une situation délicate où les graphistes n'auraient finalement que peu d'influence sur la production.

1.5.2 La chaîne de production

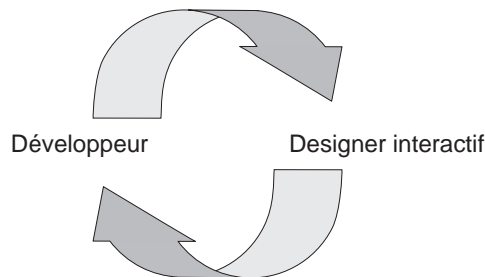
Tout projet conçu avec Blend s'inscrit dans le cadre WPF, les applications Silverlight sont un sous-ensemble de WPF. Cependant, au contraire des applications WPF bureautiques, les applications Silverlight sont multi-plateformes et multi-navigateurs. Les projets Silverlight suivent la logique classique de tout développement applicatif avec un renforcement singulier de la collaboration entre chaque pôle métier. Voici le détail de chaque étape et son impact métier :

- **Cahier des charges.** Profils concernés : client, D.A. / D.T., chef de projet, graphistes, développeurs, ergonomes et utilisateur final...

- **Maquette fonctionnelle et story-boards.** Profils concernés : 50 % développeurs et 50 % graphistes ou ergonomes. Réflexion papier commune sur la disposition des objets de l'interface et sur l'ergonomie de l'application. La navigation et la cinématique doivent être pensées à ce stade.
- **Squelette technique et maquetage.** Les collaborateurs du projet sont dans le même bureau et travaillent ensemble à :
 - l'élaboration du squelette technique, profils concernés : 95 % développeurs et D.T., 5 % graphistes ou ergonomes
 - la création de plusieurs maquettes sous Expression Design ou Illustrator "Look & Feel", profils concernés : 5 % développeurs et 95 % graphistes ou D.A.Cette phase se termine lorsque l'une des maquettes est validée par le client.
- **Production.** Cycle de "va et vient" dit itératif entre chaque pôle métier qui est une interaction continue entre les profils. Durant cette phase chaque pôle métier fournit à l'autre les éléments dont il a besoin afin d'avancer dans le développement du projet.
 - Développement de la partie logique de l'application, profils concernés : développeurs et directeur technique ;
 - Intégration graphique des éléments composant l'application, profils concernés : graphistes, designers interactif ou intégrateurs, directeur artistique.

Figure 1.1

Le cycle itératif de production.



Un designer interactif est toujours nécessaire. Toutefois, si le projet nécessite peu de ressources et qu'il est de faible envergure, ce rôle est dévolu à un graphiste ou à un développeur, ou encore aux deux à tour de rôle selon les besoins.

1.6 Langages de développement et choix

Choisir un langage de développement n'est pas seulement un choix pragmatique, cette décision est étroitement liée à la sensibilité et à l'histoire personnelles de chacun. Nous allons toutefois lister les langages, et essayer de nous décider pour l'un d'eux de manière impartiale.

1.6.1 Langages accessibles

Plusieurs langages sont accessibles aux développeurs dans la gamme Expression Studio, ils sont catégorisés en trois types distincts. La première catégorie concerne les langages managés propres à la CLR (*Common Language Runtime*). On y compte :

- **Le langage déclaratif XAML.** XAML signifie *eXtensible Application Mark-up Language*. Ce langage permet de formaliser le graphisme : les courbes vectorielles, les dégradés de couleurs ou les couleurs unies, les styles, les composants visuels, les animations, les modèles de composants qui correspondent à la forme et au graphisme de ceux-ci.
- **C#.** Le langage logique que nous utiliserons tout au long du livre (voir Chapitre 2).
- **Visual Basic.** Langage de haut niveau, comme C#, mais plus spécifique dans son écriture, nous ne l'utiliserons pas dans ce livre.

La deuxième catégorie concerne les langages managés dynamiques gérés par la DLR (*Dynamic Language Runtime*). Ils ne seront pas couverts ici. Parmi eux on trouve :

- **IronPython.** C'est le portage du python pour la plateforme .Net, vous trouverez plus d'informations à l'adresse suivante : <http://www.codeplex.com/IronPython>.
- **IronRuby.** C'est la version du langage Ruby propre à .Net, vous trouverez plus d'informations à l'adresse suivante : <http://www.ironruby.net/>.
- **JScript.** C'est un langage créé par Microsoft qui jouit de la norme ECMA.
- **JavaScript.** Version spécifique du langage du même nom mais compilée par la DLR

La troisième catégorie fait référence au langage non compilé (donc interprété) JavaScript, cela engendre moins de possibilités et d'accès au lecteur Silverlight, mais l'accès est transparent.

Le XAML est le seul langage dont vous aurez besoin quel que soit le code logique que vous choisirez. Celui-ci est orienté présentation, c'est un langage déclaratif de type XML (voir Chapitre 2).

1.6.2 CLR, DLR et langages non compilés

Nous aborderons en détails les notions de CLR et de DLR au Chapitre 3. Sachez simplement que la CLR est le compilateur gérant les langages natifs de la plateforme .Net, donc de l'environnement Silverlight. Au sein de Silverlight, deux langages sont gérés par défaut dans les projets : Visual Basic et C#, mais la CLR donne accès à d'autres langages moins connus comme F# par exemple. D'une toute autre manière, la DLR ouvre et enrichit Silverlight d'autres langages dynamiques qui ne sont pas forcément hérités de ou propres à .Net. Un langage dynamique est un langage capable de faire évoluer sa structure à l'exécution, autrement dit un langage dont il est facile d'étendre les capacités des classes ou objets natifs dynamiquement. Ce type de langage n'est pas réellement dans la culture originelle de Microsoft mais les anciens dogmes provenant des années 80 sont en cours de mutation. L'évolution de C# en est un flagrant exemple. La DLR est en réalité une surcouche à la CLR, elle permet à des langages dynamiques d'accéder au lecteur Silverlight en communiquant avec la CLR. JavaScript est un cas particulier, il peut être géré de deux manières, soit par la DLR, soit de manière transparente. Lorsqu'il est géré par la DLR, sa mise en production est, au départ, plus complexe mais dans ce cas, il accède entièrement à la CLR et bénéficie des avantages liés à cette dernière. Dans le cas d'une utilisation transparente, il donne directement accès à une petite partie des capacités du lecteur Silverlight, mais les performances à l'exécution sont beaucoup moins élevées puisqu'il n'y a pas de compilation.

Le choix du langage dépendra essentiellement de votre culture de développeur. Cependant, décider d'utiliser ou non directement JavaScript comme langage non compilé est un peu moins évident, vous devrez vous poser deux questions :

■ Quel type de projet souhaitez-vous construire ?

- Si vous souhaitez construire un projet Silverlight à faible interactivité, comme des bannières ou de petits sites événementiels, JavaScript paraît plus adéquat car sa mise en œuvre, ainsi que l'architecture projet, est moins complexe. Le plus logique sera alors de créer un site web Silverlight dans lequel JavaScript assurera la logique.
- Si vous souhaitez au contraire créer une application riche ayant une forte interactivité utilisateur, optez pour C#, Visual Basic ou n'importe quel langage géré par la DLR (selon votre culture). Tous ces langages ont accès à l'ensemble des fonctionnalités du lecteur Silverlight et permettent des performances bien supérieures à l'intégration simple de JavaScript. Bien que cela soit réellement possible, réaliser une bannière avec C# est sans aucun doute disproportionné.

■ Quel est votre corps de métier ?

- Si vous êtes intégrateur HTML ou que vous cumulez les expériences dans ce secteur, vous avez sans doute déjà développé avec JavaScript et vous connaissez peut-être AJAX. Vous n'aurez donc pas d'effort de reformation dans un premier temps. Vous vous apercevrez cependant que certaines interactions seront moins faciles, voire impossibles à mettre en place en JavaScript. Je vous encourage à envisager un apprentissage C# à moyen terme. Visual Basic est un très bon langage bénéficiant d'une grande communauté, mais sa syntaxe est très spécifique. C# est tout de même plus proche de ce que nous connaissons.
- Si vous êtes un développeur Java ou .Net, C# paraît tout désigné, la puissance de ce langage, son confort d'utilisation, son orientation vers des langages fonctionnels, surtout dans sa version 3 (avec les inférences de type par exemple) sont sans égales.

Pour notre part, nous utiliserons de façon significative le XAML et opterons pour le langage C#, assez proche de la norme ECMA (les origines de l'auteur...) sur de nombreux points. JavaScript ne nous servira qu'à l'intégration de Silverlight au sein d'une page HTML. Ce livre s'adresse en partie aux designers web, dans ce cadre et pour diverses raisons, nous n'aborderons pas Visual Basic, IronRuby, IronPython ou JScript.

Au Chapitre 2, afin de commencer notre apprentissage de Silverlight, en douceur, nous ferons un rapide retour sur les bases du langage XML et XAML, ainsi que sur C#. De cette manière, vous les assimilerez plus facilement par la suite.

Le couple XAML / C#

L'apparition du XAML constitue l'une des grandes nouveautés de .Net 3. Ce langage repose sur XML. Dans ce chapitre, nous reviendrons donc brièvement sur l'écriture et la grammaire XML. Puis nous verrons en quoi certains processus de création sont facilités grâce à XAML, et comment afficher un document XAML valide au sein d'un navigateur ou d'un éditeur de code simple.

Au même niveau que XAML, le langage C# permet d'ajouter de la logique à nos applications Silverlight. Nous dresserons donc un récapitulatif succinct de l'écriture afin de vous familiariser avec C# et la programmation orientée objet. Si vous développez avec un autre langage, comme JavaScript ou PHP, ou si vous êtes intégrateur, ce chapitre vous concerne. Ainsi, inutile de consacrer des heures à la lecture d'un ouvrage sur C#, dans un premier temps. Si vous êtes vous-même développeur .Net, vous pouvez passer votre chemin ou revoir les fondamentaux du langage.

2.1 XML

Le XAML repose sur XML. Connaître les règles d'écriture du XML revient donc à respecter celles du XAML. XML est l'acronyme de *eXtensible Mark-up Language*, c'est un langage déclaratif à balises.

2.1.1 À quoi sert le XML ?

Le XML n'a qu'un seul objectif : présenter d'une manière structurée et universelle des données. Il n'y a pas de vocabulaire lié à ce langage, seule la grammaire est importante. Celle-ci est très simple et ne connaît que quelques règles importantes. XML se veut donc le plus générique possible. Au contraire du XAML qui est conçu pour afficher un visuel et qui possède un vocabulaire spécifique, le XML ne possède pas de vocabulaire spécifique. Écrire ou comprendre la structure d'un fichier XML est à la portée de tous, développeurs ou non. Les noms des balises ne sont donc pas prédéfinis ou figés, mais arbitrairement choisis par le développeur ou le fournisseur de la source XML. Le transit de fichier XML est indépendant du réseau, de la plateforme ou de la connaissance préalable du fichiers XML. C'est aujourd'hui le format de communication le plus utilisé sur Internet. Vous connaissez sans doute les fameux flux RSS basés sur XML. Les flux RSS

ne sont rien d'autre que des fichiers de données XML respectant la norme RSS. C'est pour cette raison qu'un lecteur de flux RSS est capable de lire n'importe quelle source d'où qu'elle provienne tant que la norme RSS est respectée. Voici un exemple de contenu au format XML :

```
<DISCOGRAPHIE>
  <INTERPRETE prenom="Patrick" nom="Hernandez">
    <ALBUM annee="1979" titre="Born to be alive">
      <JACQUETTE url="www.amazon.com/images/P/Patrick.jpg" />
      <CHANSON>Born to be alive</CHANSON>
      <CHANSON>You Turn me On</CHANSON>
      <CHANSON>It Comes So Easy</CHANSON>
    </ALBUM>
    <ALBUM annee="1980" titre="Crazy day's Mystery nights">
      ...
    </ALBUM>
  </INTERPRETE>
  ...
</DISCOGRAPHIE>
```

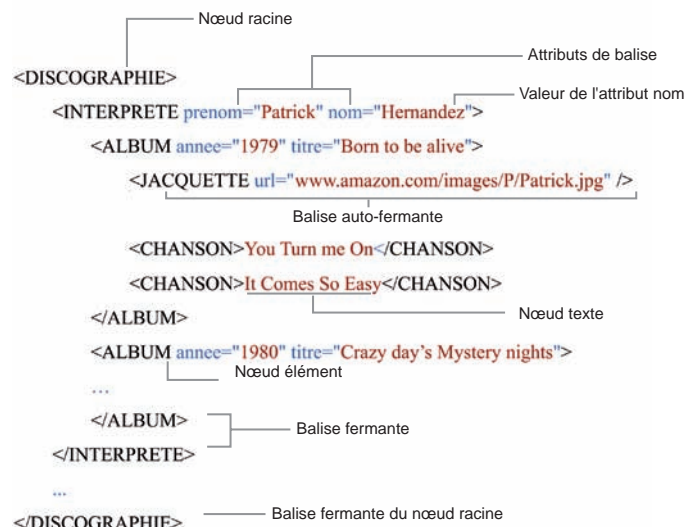
Vous remarquez que les données sont structurées suivant des relations familiales. Ainsi la balise INTERPRETE contient plusieurs balises ALBUM qui contiennent elles-mêmes plusieurs balises CHANSON. Les balises ALBUM sont sœurs les unes par rapport aux autres car elles sont situées sur le même niveau d'imbrication : elles sont toutes contenues par la balise mère INTERPRETE. Nous pouvons donc comparer une structure XML à un arbre généalogique. Toutefois les noms des balises sont arbitraires et ne relèvent pas du langage proprement dit. Nous allons maintenant étudier les règles d'écriture du XML qui s'appliquent également aux langages dérivés tels que XAML.

2.1.2 La grammaire

Le langage XML est constitué de balises et d'attributs. Les balises peuvent être ouvrantes, fermantes ou auto-fermantes. Les attributs sont définis au sein des balises ouvrantes ou auto-fermantes. Ils sont suivis de l'opérateur d'affectation = et leurs valeurs sont toujours entre guillemets (voir Figure 2.1).

Figure 2.1

Structure
d'un fichier XML.



Bien que les balises soient définies arbitrairement, il existe des règles d'écritures à respecter sous peine que le fichier ne soit pas lisible :

- La balise `xml` est proscrite. C'est une balise réservée, utilisée uniquement pour décrire l'encodage du fichier XML.
- Les balises ne doivent pas commencer par un chiffre ou par un caractère spécial.
- Le seul caractère spécial autorisé est `_`. Les autres ne doivent pas être contenus dans un nom de balise. Attention donc à éviter les caractères avec accent dont la langue française fourmille.
- Il doit y avoir un unique nœud racine. Dans le code XML précédent, la balise racine est `DISCOGRAPHIE`.
- Une balise ouverte doit impérativement être suivie d'une balise de fermeture qui commence par `"</"` ou être elle-même auto-fermante. C'est le cas de la balise `JACQUETTE`.
- Les attributs de balise, par exemple `annee`, ne doivent pas contenir de caractères spéciaux.
- Les valeurs des attributs de balise doivent toujours être entre guillemets ou entre simples apostrophes.

Vous pouvez également vous référer à des fichiers de type DTD (*Document Type Definition*). Ceux-ci contiennent une définition de type de document permettant de spécifier certaines règles.

2.1.3 L'ambiguïté des relations familiales

Décider arbitrairement d'une structure de données basée sur des relations familiales peut s'avérer très risqué. Dans un arbre généalogique, la structure familiale représente la vie réelle, il est donc facile de concevoir des données XML basées sur ce modèle. Toutefois, l'exemple de la discographie est intéressant car de nombreuses méthodes de tri existent dans ce cas. Nous avons classé notre discographie par interprète puisque les balises `INTERPRETE` contenaient un ou plusieurs albums. Cependant, nous aurions pu classer les albums au sein de balises `GENRE` contenant un attribut `type` avec des valeurs comme `funk`, `rap`, `disco`. Puis, à l'intérieur des balises `GENRE`, nous aurions pu avoir des balises `ALBUM`. Les données contenues auraient été les mêmes, mais leur structure aurait été différente. Nous aurions également pu classer les albums par date :

```
<DISCOGRAPHIE>
  <DATE annee="1979" month="05">
    <ALBUM titre="Born to be alive">
      <INTERPRETE prenom="Patrick" nom="Hernandez" />
      <JACQUETTE url="www.amazon.com/images/P/Patrick.jpg" />
      <CHANSON>Born to be alive</CHANSON>
      <CHANSON>You Turn me On</CHANSON>
      <CHANSON>It Comes So Easy</CHANSON>
    </ALBUM>
  </DATE>
  <DATE annee="1980">
    ...
  </DISCOGRAPHIE>
```

Bref, notre première structure aurait pu différer alors que nous avons les mêmes données. Il en est de même pour tous les langages dérivés du XML. Les conséquences pour les fichiers XML peuvent être importantes car cela peut rendre difficile les opérations de tri et d'accès aux données.

Pour les langages comme le XAML, les différentes approches d'imbrication de balises permettent une souplesse de conception unique. Nous allons donc nous intéresser d'avantage au XAML afin de comprendre en quoi ces relations peuvent engendrer des visuels différents et innovants.

2.2 XAML, un langage déclaratif puissant

2.2.1 L'utilité du XAML

Comme nous l'avons expliqué au Chapitre 1, le XAML, dérivé de XML, est le nouveau langage apporté par .Net 3. Ce langage permet simplement à un graphiste de formaliser n'importe quel visuel. Créer des éléments graphiques, grâce à ce langage, peut donc être réalisé de deux manières :

- La première consiste à écrire du XAML *via* n'importe quel traitement de texte ou éditeur de code. C'est l'approche que va adopter le développeur. Un outil comme NotePad suffit, mais Visual Studio est recommandé car le développeur bénéficie de l'IntelliSense. Ce concept facilite grandement la programmation en évitant aux concepteurs de connaître le vocabulaire d'un langage dans sa totalité. Cela suppose que le graphiste code le visuel, ce qui n'est pas l'idéal car concevoir du graphisme de cette manière se révèle être une démarche abstraite et indirecte, qui fait appel à peu de sensibilité. L'apprentissage technique du langage est tel que le coder directement dans un premier temps nuit à la création graphique.
- La deuxième méthode passe par l'utilisation d'un outil dédié aux graphistes ou aux designers interactifs. Dans ce cas, deux logiciels de la suite Expression répondent à cette problématique : Expression Blend et Expression Design. Avec ces deux logiciels, les graphistes peuvent concevoir le design d'éléments visuels ou d'une application. Ces logiciels ont la capacité de traduire le visuel en XAML. Une fois le design créé, le développeur peut non seulement en avoir un aperçu, mais aussi récupérer le code XAML généré de manière transparente. Si vous travaillez avec des outils tels que Adobe Illustrator ou Photoshop, il est tout à fait possible d'importer les fichiers produits par ces logiciels au sein d'Expression Design ou Expression Blend directement. Dans ce cas, les fichiers psd ou ai sont transformés en bitmap ou au format vectoriel XAML selon le type du fichier importé.

Le code généré est directement réutilisable au sein d'une production, le développeur ne le modifiera que très peu et laissera la place au designer interactif. Le travail de ce dernier est donc réellement respecté. Voici une liste de ce qui peut être formalisé en XAML par le graphiste :

- les tracés vectoriels ;
- les couleurs unies ;
- les dégradés de couleurs ;
- Les animations ;
- les composants visuels ;
- l'agencement d'une interface visuelle ;
- les styles de composants ;
- les modèles de composants ;

- les filtres ;
- la 3D.

Comme nous pouvons le voir, le XAML est un formidable outil de communication conçu pour chaque acteur d'une production. L'ergonome, le directeur artistique, le graphiste, le designer interactif, l'intégrateur, le développeur et le directeur technique peuvent participer pleinement à la création d'une application sans empiéter sur le travail des uns ou des autres.

2.2.2 Comparaison XAML / C#

Ces deux langages sont de même niveau. Cela signifie que tout ce qui est créé en XAML peut l'être *via* C#, bien qu'il ne soit pas adapté à la création graphique. L'inverse n'est pas vrai, XAML est orienté visuel avant tout, ainsi il n'est pas possible de créer d'algorithmes quelconques sans C# ou un autre langage logique, comme VB.

Voici l'écriture d'un bouton en C# :

```
Button monBouton = new Button();
monBouton.Width = 156;
monBouton.Height = 80;
monBouton.Content = "Hello";
Canvas.SetTop(monBouton, 50);
Canvas.SetLeft(monBouton, 75);
monBouton.Background = new SolidColorBrush(Color.FromArgb(0xFF,
                                                             0xC3, 0x02, 0x02));
monBouton.Click += new RoutedEventHandler(button_Click);
monCanvas.Children.Add(monBouton);
```

Voici la même écriture en version XAML :

```
<Canvas x:Name="monCanvas" >
  <Button x:Name="monBouton" Width="156" Height="80"
    Content="Bonjour" Canvas.Top="50" Canvas.Left="75"
    Background="#FFC30202" Click="button_Click" />
</Canvas>
```

Dans ces exemples, nousinstancions un bouton à l'intérieur d'un conteneur de type Canvas. Comme nous le constatons, le XAML est bien plus simple à écrire. Par exemple, affecter une couleur se révèle très facile. De même, définir le bouton comme enfant d'un autre objet est très intuitif et moins verbeux qu'en C#. Cela provient de la nature même du XAML qui émane du XML. Vous pouvez avoir un aperçu du résultat à la Figure 2.2.

Figure 2.2

Visuel d'un bouton généré par XAML ou C#.

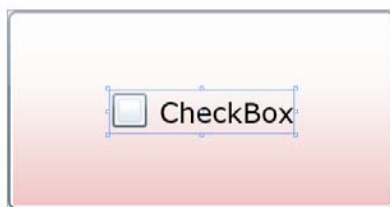


La propriété `Content` du bouton permet dans ce cas d'afficher le mot `Bonjour`. Cette propriété ne s'appelle pas `Label` car elle a la capacité d'afficher autre chose qu'une chaîne de caractères. Grâce à son écriture, très souple, s'appuyant sur le langage XML, le XAML permet d'imbriquer des objets complexes aisément. Pour cela, il suffit de définir l'objet imbriqué comme nœud élément enfant du bouton (voir la section 2.1.2 pour plus d'informations sur XML). Par exemple, on peut afficher une case à cocher dans le bouton au lieu d'un mot (voir Figure 2.3) :

```
<Canvas x:Name="monCanvas">
  <Button x:Name="monBouton" Width="156" Height="80"
    Canvas.Top="50" Canvas.Left="75"
    Background="#FFC30202" Click="button_Click" >
    <CheckBox Content="CheckBox" />
  </Button>
</Canvas>
```

Figure 2.3

Visuel d'un bouton généré par XAML avec case à cocher.



L'exemple de la Figure 2.3 ne serait pas réellement ergonomique pour l'utilisateur, imbriquer une case à cocher au sein d'un bouton n'a pas de réelle utilité, toutefois cela montre à quel point le XAML s'appuie sur les relations familiales héritées du XML. Vous pourrez donc imaginer toutes sortes d'interfaces visuelles et de nouvelles ergonomies pour vos applications. Ce n'est qu'un aperçu de ce dont le XAML est capable. Vous allez maintenant créer votre premier visuel XAML.

2.2.3 Afficher un visuel XAML

L'exemple précédent est illustratif, mais nous ne bénéficions d'aucun moyen pour le moment d'en avoir un aperçu. Bien sûr, vous pourriez utiliser Expression Blend ou Visual Studio, mais ces logiciels s'inscrivent dans une logique de projet. Vous souhaitez souvent avoir l'aperçu visuel d'un code XAML sans pour autant créer un projet ou une solution (voir Chapitre 3). Pour cela, il existe quelques éditeurs de XAML dont l'un s'appelle Kaxaml. Il a été conçu par Robby Ingebreetsen, vous pouvez le télécharger à cette adresse : <http://www.kaxaml.com>. Comme le dit son concepteur, l'objectif est de vous proposer une IntelliSense poussée tout en fournissant un aperçu de n'importe quel fichier XAML présent sur votre disque dur. Vous trouverez un fichier de démonstration, `preview.xaml`, dans le dossier *chap2* des exemples du livre.

Cependant Kaxaml n'est pas obligatoire. Il existe une autre solution si vous souhaitez envoyer votre travail à un collègue ou à votre directeur artistique, mais que ces derniers ne possèdent pas d'éditeur XAML. En réalité, ils n'en ont pas forcément besoin tant qu'aucun code logique n'est lié au code déclaratif. Tout fichier XAML que vous créez est lisible par votre navigateur, si le lecteur Silverlight est installé, et si ce fichier est correctement formaté. Dans ce dernier cas, le lecteur Silverlight compile directement le fichier XAML à l'exécution et l'affiche dans le navigateur (voir Figure 2.4).

Figure 2.4

Fichier preview.xaml
directement lisible
par Internet Explorer
ou Kaxaml.



Même si Kaxaml est pratique pour lire des fichiers, il ne permet pas réellement à un graphiste d'élaborer des éléments visuels complexes. Nous utiliserons donc exclusivement Expression Blend et Visual Studio pour concevoir nos projets.

2.2.4 Espaces de noms

Pour visualiser correctement un code déclaratif XAML, vous devez impérativement utiliser les espaces de noms conformes au développement XAML. Voici une portion du code où vous pouvez apercevoir les espaces de noms dans les lignes en gras ; il permet d'afficher la Figure 2.4 :

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="640" Height="480">
  <Grid x:Name="LayoutRoot" Background="White">
    <Border Height="134" Width="624" Canvas.Left="8" Canvas.Top="8"
      CornerRadius="24,24,24,24">
      <Border.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
          <GradientStop Color="#FF002352"/>
          <GradientStop Color="#FF68A9DE" Offset="1"/>
        </LinearGradientBrush>
      </Border.Background>
      <Grid Margin="24,20,8,20">
        <TextBlock HorizontalAlignment="Left" VerticalAlignment="Top"
          FontSize="24" Foreground="#FFFF"
          Text="Pratique de Silverlight" TextWrapping="Wrap"/>
        <TextBlock HorizontalAlignment="Left" FontSize="16"
          Foreground="#FFFF" Text="Conception d'application
          interactives riches"
          TextWrapping="Wrap" Margin="0,39,0,20" />
      </Grid>
    </Border>
    ...
  </Grid>
</UserControl>
```

Vous remarquez que le code précédent répond bien au standard XML. Outre de nombreuses balises permettant de générer du contenu graphique, il s'y trouve un nœud XML racine `UserControl`. Celui-ci représente le premier objet visuel de l'application Silverlight. Au sein de cette balise, deux attributs nous intéressent particulièrement : `xmlns` et `xmlns:x`. Les initiales `xmlns` signifient XML *NameSpace*. Ces attributs sont des espaces de noms, c'est-à-dire qu'ils font référence à des dictionnaires nécessaires au développement d'applications basées sur WPF et XAML. Ces espaces de noms doivent toujours se situer dans le conteneur XAML racine. Le premier fait référence à la bibliothèque WPF (*Window Presentation Foundation*) dont Silverlight émane. C'est l'espace de noms par défaut. Grâce à lui, nous avons accès aux objets complexes de cette bibliothèque, comme `Border`, `Grid` ou `TextBlock`. En interne, cette bibliothèque est basée sur le langage XAML. C'est pourquoi on a besoin d'un deuxième espace de noms, `xmlns:x`, qui fait pour sa part référence au

langage XAML proprement dit. À chaque fois que nous ferons référence à cet espace de noms, les balises ou attributs devront être préfixés de `x:`. C'est, par exemple, le cas de l'objet grille qui contient l'attribut `x:Name` :

```
<Grid x:Name="LayoutRoot" Background="White">
```

Pour adjoindre un comportement logique aux objets visuels, il vous faudra une classe dite partielle. Celle-ci devra être référencée au sein du nœud racine de cette manière (voir Chapitre 3 pour les classes partielles) :

```
<UserControl x:Class="maClasse" ...
```

Cependant, gardez bien à l'esprit qu'un navigateur renverra une erreur d'affichage lors de la lecture directe d'un fichier XAML si celui-ci référence une classe partielle. Dans ce cas, il vous faudra compiler les fichiers XAML et C# pour en faire un unique fichier xap lisible par le lecteur Silverlight. Nous allons maintenant nous concentrer sur le code logique C# qui contribue largement à la robustesse des applications Silverlight.

2.3 Les fondamentaux du langage C#

Cette section ne concerne que les développeurs non .NET ou les designers interactifs et intégrateurs ayant des bases de programmation dans n'importe quel langage logique (JavaScript, PHP, ActionScript). Si vous êtes développeur .Net expérimenté, vous pouvez passer votre chemin. L'objectif est de présenter et de s'initier aux bases de la programmation orientée objet et de C#, de savoir déclarer des variables et des méthodes et de connaître les types C#.

2.3.1 C# et l'API Silverlight

Le langage C# est sorti dans sa première version avec la plateforme .Net 1. D'après Microsoft, ce langage est le plus performant en termes de lisibilité et de confort. Il allie la puissance de C++ et la simplicité de Visual Basic. Chaque version de C# est une révolution en soi. Pour ses versions 3 et 4, il a bénéficié d'améliorations visant à assouplir les règles et les méthodologies de développement. L'idée est de permettre des prises de décisions tardives dans les choix d'architecture en intégrant les avantages propres aux langages fonctionnels F# ou JavaScript. Au même titre que Visual Basic, il est géré par la CLR (*Common Language Runtime*), c'est-à-dire qu'il est tout d'abord compilé en langage intermédiaire avant de l'être en langage machine par le *Just In Time Compiler* (JITC). C# est un langage orienté objet. Les variables ou méthodes sont fortement typées. Toutefois les types (comme `string` ou `double`) n'appartiennent pas au langage lui-même, mais au *Common Type System* (CTS). Cela signifie qu'un type `string` en C# ou en Visual Basic est avant tout un type `string` géré par le CTS propre à .Net. C'est cela qui permet aux développeurs de coder dans différents langages. Le CTS est donc une composante importante de la CLR. Il permet l'interopérabilité des langages.

Le cœur du langage C# est totalement indépendant de l'API Silverlight. Une API est constituée d'une multitude de bibliothèques (ou espaces de noms). L'API (ou *framework*) Silverlight permet aux développeurs de concevoir des applications Silverlight. L'espace de noms principal est `System`. Par exemple, l'une des bibliothèques accessibles dans le framework Silverlight permet de sauve-

garder des données sur le poste client local. Il s'agit de la bibliothèque `IsolatedStorage`. Celle-ci contient plusieurs classes qui permettent d'atteindre cet objectif.

2.3.2 Introduction à la programmation orientée objet

La programmation orientée objet s'oppose à la programmation procédurale. Jusqu'au début des années 1980, avec les langages bas niveau proche de la machine, le code était organisé de manière linéaire : il était lu de haut en bas à la manière d'une procédure que l'on suit. Par la suite, avec des langages comme C++, puis des langages de haut niveau, la programmation orientée objet s'est imposée. La POO part du principe que la plupart des fonctionnalités ou composants d'une application peuvent être décrits sous forme d'objets. Par exemple, si l'on crée une application de gestion de parc automobile, on peut considérer les voitures comme un composant essentiel de l'application, les voitures seront donc envisagées comme des objets. Les objets sont en réalité des exemplaires de modèles d'objets appelés classes. Chaque exemplaire de la classe `Voiture` sera différencié par les valeurs de ses propriétés et de ses champs (voir la section 2.3.4.6). Par convention, on privilégie la notation Pascal où chaque nom de variable ou de méthode commence par une majuscule : `MaFonction`, `MaPropriété`, `MonChamp`. Pour les paramètres de méthode, on utilise plutôt la notation Camel : `monParametre`. Le modèle de `Voiture`, appelé classe, permet de décrire tout ce que peut faire une voiture, ainsi que tout ce qui la définit. Le mot-clé `class` permet de créer une nouvelle classe. Voici en C# comment générer une classe `Voiture` :

```
class Voiture {
    //champs de la classe voiture
    public string Marque;
    public string Modele;
    public string Imat;
    public double Longueur;
    public string TypeCarburant;

    //méthode qui permet à la voiture de rouler
    public void Roule(){
        // permet de tracer 'la voiture roule' dans une console
        Console.WriteLine("la voiture roule");
        //permet de laisser la console affichée
        Console.ReadLine();
    }

    //méthode qui permet à la voiture de s'arrêter
    public void Stop(){
        Console.WriteLine("la voiture stoppe");
        Console.ReadLine();
    }
}
```

Comme nous le voyons, tous les exemplaires émanant de la classe auront des champs et des méthodes (ou fonctions) propres au modèle `Voiture`. Toutefois, les valeurs affectées aux propriétés et aux champs pourront être différentes pour chacun d'eux, c'est cela qui permet de les différencier. Toutes les voitures ont une propriété couleur, mais celle-ci pourra posséder plusieurs valeurs : rouge, verte ou bleu selon l'exemplaire. Le mot-clé `new` permet de créer un nouvel exemplaire de la classe `Voiture`. Voici comment vous pouvez créer des exemplaires (également appelés instances) du modèle `Voiture` :

```
//voici un exemplaire de la classe Voiture
Voiture MaPorsche = new Voiture() { Marque="Porsche",
    Longueur=1.95,Modele="Carrera 4", TypeCarburant="essence"};
// voici un autre exemplaire
Voiture MaSuper5 = new Voiture() { Marque = "Renault",
    Longueur = 1.2, Modele = "Super5", TypeCarburant = "diesel" };
```

On se rend bien compte que chaque voiture instanciée possède les mêmes champs mais que ceux-ci contiennent des valeurs différentes. La déclaration et l'affectation de variables seront abordées à la section 2.3.4.

2.3.3 Une première application en mode console

Nous allons maintenant mettre en pratique notre apprentissage récent en créant une petite application en mode console. Ouvrez Visual Studio 2008 et choisissez Fichier > Nouveau projet C# > Application console. Entrez le nom de solution : ParcAuto. Voici le code que Visual Studio génère par défaut :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ParcAuto
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Toute application console possède une classe par défaut qui s'appelle Program, contenue dans un espace de noms correspondant au nom du projet. Celle-ci contient une méthode qui s'exécute par défaut : Main. Nous allons tracer un message dans la console Windows. Pour cela, ajoutez les lignes suivantes au sein de la méthode Main :

```
static void Main(string[] args)
{
    Console.WriteLine("Bonjour tout le monde !!");
    //ceci trace le message
    Console.ReadLine();
    //ceci empêche la fermeture de la console
}
```

Pour compiler notre première application et l'exécuter, rien de plus simple : appuyez sur la touche F5. Lorsque vous utilisez ce raccourci, vous lancez le mode Debug. Ce dernier vous permettra de corriger et de comprendre les dysfonctionnements de votre application lorsque vous compilerez ou durant son exécution. Si vous n'utilisez pas ce mode, votre application échouera en silence si elle contient des erreurs. Pour compiler et exécuter sans déboguer, il suffit de laisser appuyer sur les touches Ctrl et F5 en même temps. Si vous souhaitez uniquement compiler sans lancer votre application, vous pouvez utiliser le raccourci Ctrl+Maj+F5.

2.3.4 Les types

2.3.4.1 Variables et champs d'objets

Une variable permet de stocker des données. Voici comment déclarer une variable au sein d'une classe, dans ce cas la variable est un champ d'objet (en C#, on différencie champ de propriété) :

```
class Program
{
    int monEntier = 13;
}
```

Vous remarquez que le type est toujours précisé en premier, dans notre cas `int` signifie entier. Il vous faut, cependant, faire la différence entre la déclaration et l'affectation. Ce sont deux étapes distinctes et l'affectation peut être réalisée plus tard. Nous aurions tout aussi bien pu écrire :

```
class Program
{
    int monEntier;
    monEntier = 13;
}
```

Il arrivera souvent de déclarer une variable sans pour autant l'affecter. Par exemple, dans un jeu la variable `score` sera affectée dès son initialisation mais durant l'exécution du programme. Lorsqu'une variable de type entier est déclarée sans être affectée, sa valeur par défaut est 0. Ce comportement est dû au fait que le type `int` est un type primitif numérique.

2.3.4.2 Les types primitifs

Il est nécessaire de typer les variables d'une part par obligation due au langage, et d'autre part parce que cela permet d'améliorer les performances de vos applications. Le choix du type est très important dans ce contexte car c'est lui qui déterminera l'occupation mémoire allouée par défaut pour stocker la valeur. Parmi tous les types existants, on trouve les types primitifs. Ceux-ci acceptent des valeurs simples. Ils sont à la base de tous développements, sans eux aucune conception ne verrait le jour. Voici quelques types primitifs importants :

- `sbyte`. Il contiendra une valeur entière comprise entre -128 et 127. Son nom vient du fait que la valeur pourra être négative (s pour *signed*) et que le nombre de possibilités correspond à un octet soit 2^8 (256 chiffres). Il ne faudra pas confondre *Byte*, en anglais, et bit, qui signifie 0 ou 1.
- `byte`. Ce type peut accepter un entier entre 0 et 255.
- `uint`. Il accepte un entier entre 0 et 2^{32} , soit 4 294 967 295 possibilités. Le u de `uint` signifie que la valeur sera toujours non signée (u pour *unsigned*), donc toujours positive.
- `int`. Il représente le même nombre de possibilités mais contient la moitié des chiffres en dessous de 0. Soit entre -2 147 483 648 et 2 147 483 647.
- `decimal`. Il contient des valeurs à virgule. La plage des valeurs admises se situe entre 10^{-28} et 7.9×10^{28} .
- `double`. Ce type contient également des valeurs à virgule. Comme pour `decimal` le stockage des données est assez complexe, mais le nombre de valeurs acceptées est beaucoup plus grand : entre 5×10^{-324} et 1.7×10^{308} .

- **bool**. Il est le digne représentant de la logique de Mr George Boole. Ce mathématicien britannique est à l'origine d'une révolution des mathématiques et de la philosophie au XIX^{ème} siècle. On en trouve aujourd'hui de nombreuses utilisations dans l'industrie ou dans le secteur informatique. Ce type ne peut contenir que deux valeurs : vrai ou faux (`true` ou `false`) ; le bit de donnée repose sur cette logique.
- **string**. Il représente une chaîne de caractères. La valeur est toujours spécifiée entre guillemets. La chaîne de caractères contenue peut-être infinie, la mémoire utilisée sera allouée au fur et à mesure des besoins. Toutefois le type `string` n'est pas comme les autres types simples (voir la section "Le tas et la pile" dans ce chapitre).

2.3.4.3 Les types complexes

Stocker des valeurs simples ne suffit pas. Bien souvent, vous aurez besoin d'une variable ayant la capacité de contenir plusieurs données. Voici les types complexes :

- Les énumérations notées `enum`. Elles vous permettent de stocker un choix de valeurs arbitrairement. Si l'on reprend l'exemple de la voiture, nous pourrions par exemple proposer une énumération pour le type de carburant au lieu d'une chaîne de caractères. Nous pourrions écrire ceci :

```
class Program{
    Voiture MaDeLorean = new Voiture() { TypeCarburant =
        Carburant.Detritus, Marque = "DeLorean", Modele = "volante" };
}
enum Carburant{
    Diesel,
    Essence,
    propane,
    Detritus,
    Ethanol
}
class Voiture{
    public string Marque;
    public string Modele;
    public Carburant TypeCarburant;
}
```

Vous remarquez que l'énumération est en dehors de la classe, elle se suffit à elle-même. Elle pourrait également en faire partie. Au sein de la classe `Voiture`, nous déclarons un nouveau champ public de type `Carburant`. Le mot-clé `public` est un modificateur d'accès, nous verrons les modificateurs d'accès plus loin dans ce chapitre. L'affectation se produit au niveau de la classe `Program`, à l'instanciation d'une nouvelle voiture grâce à cette ligne :

```
Voiture MaDeLoreane = new Voiture() { TypeCarburant =
    Carburant.Detritus, Marque = "DeLorean", Modele = "celle qui vole"
};
```

Il faut savoir qu'une valeur d'énumération possède toujours un type sous-jacent. Par défaut ce type est `int` mais vous pourriez le changer.

- Les structures notées `struct`. Il s'agit de l'équivalent d'une classe, mais elle est gérée différemment en mémoire. Il faut envisager une structure comme une définition d'objets. Dans la majorité des cas, ce seront des objets simples. Par exemple, un point géographique est défini

grâce à deux coordonnées : la latitude et la longitude, toutes deux exprimées en degrés. Nous pourrions ainsi définir la position de notre voiture grâce à une structure qui s'appellerait-PositionGPS :

```
class Program{
    //ceci est une écriture pratique
    //pour créer un nouvel exemplaire de voiture
    Voiture MaDeloreane = new Voiture(){
        TypeCarburant = Carburant.Detritus,
        Marque = "DeLorean",
        Modele = "volante",
        MyPlace = new PositionGPS() { DegreLat=180, DegreLong=176 }
    };
}
struct PositionGPS{
    public int DegreLat;
    public int DegreLong;
}
class Voiture{
    public string Marque;
    public string Modele;
    public Carburant TypeCarburant;
    public PositionGPS MyPlace;
}
```

- Les tableaux notés Array et []. Ce sont des objets différents de ceux que nous venons de citer en cela qu'ils sont faits pour stocker de multiples données sous forme d'éléments indexés numériquement à partir de 0. Il faudra toujours préciser le type des objets stockés. Pour initialiser un tableau, il existe deux méthodes. La première consiste à utiliser le mot-clé new et à indiquer le nombre d'éléments qu'il pourra contenir. Dans d'autres langages comme Java ou Action-Script 3, ce type correspond à la classe Vector (vecteur). Voici un exemple d'instanciation :

```
Voiture[] MesVoitures = new Voiture[3];
```

Par la suite, on affecte une voiture à chaque index numérique du tableau comme ceci :

```
MesVoitures[0] = MaDeLorean;
MesVoitures[1] = MaPorsche;
MesVoitures[2] = MaSuper5;
```

Voici la seconde méthode appelée "écriture littérale" :

```
Voiture[] MesVoitures = { MaDeloreane, MaPorsche, MaSuper5 };
```

Il nous suffit maintenant d'accéder aux voitures du tableau. Pour cela, nous pouvons utiliser une boucle qui nous permettra de tracer le nom de chaque voiture dans une console Windows. La boucle foreach est la mieux adaptée à ce type de situation. Voici comment l'utiliser :

```
void TraceMesVoitures(){
    //Pour chaque élément de type Voiture dans le tableau MesVoitures
    foreach(Voiture V in MesVoitures){
        //cette ligne permet d'afficher le résultat dans une Console
        Console.WriteLine("j'ai une {0}", V.Marque);
    }
    //celle-ci permet de figer l'affichage de la console pour la lecture
    Console.ReadLine();
}
```

Concrètement, la boucle parcourt le tableau `MesVoitures`, récupère un exemplaire de `Voiture` à chaque index et trace une phrase en fenêtre de sortie. Cette boucle s'arrête d'elle-même lorsque le tableau est entièrement parcouru. Les tableaux possèdent la propriété `length` qui permet de connaître le nombre d'éléments contenus en leur sein. L'objet `List`, en C#, est plus souvent utilisé que l'objet tableau (`Array`) en raison des nombreuses capacités qu'il offre. Un tableau possède une longueur (ou nombre d'éléments) fixe. La classe `List` possède pour sa part des méthodes lui permettant d'ajouter ou de supprimer des éléments sans avoir besoin d'initialiser la longueur. Vous la trouverez dans l'espace de noms `System.Collections.Generic`.

2.3.4.4 Les modificateurs d'accès

Vous avez sans doute remarqué le mot-clé `public` présent devant la déclaration de certains membres de classe. C'est un modificateur d'accès. Ces derniers permettent de rendre accessible ou non des champs, des propriétés et des méthodes de classe. Le mot-clé `public` permet de rendre un membre de classe accessible depuis l'extérieur de la classe. C'est cela qui nous a permis d'instancier des voitures tout en initialisant leurs champs publics depuis l'extérieur. Voici un cas concret :

```
class Program{
    //ceci est une écriture pratique
    //pour créer un nouvel exemplaire de voiture
    Voiture maDeLorean = new Voiture();
    maDeLorean.marque = "DeLorean";
}
class Voiture{
    public string marque;
    Guid numeroSerie = new Guid();
}
```

La propriété `numeroSerie` n'est définie avec aucun modificateur d'accès. En fait, lorsque rien n'est précisé c'est le modificateur `private` qui est utilisé à la compilation pour un membre de classe. Pour une classe ce sera le mot-clé `internal`. Un champ de classe, défini avec `private`, n'est pas accessible en dehors de la classe, mais seulement en son sein. La meilleure pratique consiste toujours à laisser le moins de champs ou de méthodes visibles depuis l'extérieur. Votre classe doit toujours être fermée à la modification et toujours ouverte à l'extension. Seules les propriétés et méthodes utiles doivent être accessibles à celui qui instancie ou utilise votre classe. Dans le cas précédent, il est logique que le numéro de série reste privé et ne soit pas accessible au *quidam* de base. Le type `Guid` est en fait une valeur unique car le nombre de possibilités est quasi illimité. De plus, on affecte la valeur lors de l'instanciation de la classe `Voiture`. Voici la liste des modificateurs d'accès utilisables :

- `public`. Permet l'accès depuis n'importe quel endroit du code.
- `private`. Définit un membre de classe accessible uniquement depuis la classe contenant ce membre.
- `internal`. Autorise l'accès à toutes les classes de l'espace de noms dans lequel est la classe et aux espaces de noms amis.
- `protected`. Permet l'accès au code situé au sein de la classe ou dans une de ses sous-classes.

2.3.4.5 Conversion de type implicite et explicite

Il vous arrivera souvent dans un langage orienté objet de transformer le type d'une valeur en un autre type. Catégoriser des objets est une démarche qui permet d'optimiser les performances des applications. Toutefois catégoriser est une démarche rigide, même dans la vie courante. Il est toujours dangereux et rarement vrai à long terme qu'un objet soit d'un seul type. La classification des espèces, bien que nécessaire d'un point de vue scientifique, est remise en question à chaque fois qu'une espèce sortant du cadre est découverte. L'ornithorynque est le cas le plus flagrant qui démontre que classifier des espèces est hasardeux. Celui-ci pond des œufs, allaite ses petits, possède un bec de canard, s'appuie sur des pattes aux cinq orteils palmés. Bref, il regroupe de nombreux aspects que l'on croyait uniques au mammifères, aux reptiles et aux ovipares. Il ne rentre dans aucune catégorie simple et présente certains aspects de plusieurs d'entre elles (depuis la théorie de Darwin, la classification des espèces a évolué, il s'agit bien ici d'un exemple mettant en évidence la dangerosité de cette manière de raisonner). Pour cette raison et de nombreuses autres, il vous arrivera donc souvent, si vous commencez la POO, d'utiliser un type à la place d'un autre ; d'avoir un jour besoin d'un entier à la place d'un décimal ou de transformer un booléen en chiffre. Pour transformer un type en un autre type, vous utiliserez une opération de conversion implicite ou explicite appelée *transtypage* ou *casting* en anglais.

La conversion implicite est réalisée par le compilateur sans que vous ayez besoin de lui indiquer un mot-clé. Pour que le compilateur puisse réussir cette opération, il suffit juste que le type de destination accepte la valeur que l'on souhaite définir avec ce type. Par exemple, il est facile de transformer un type `byte` en type `uint` car un `byte` est toujours positif et la plage de valeurs possibles est contenue dans la plage de valeurs possibles de `uint` :

```
int MonEntier;
byte MonByte = 113;
MonEntier=MonByte;
// Dans ce cas, le type de b sera converti en int à la compilation,
// ce qui est faisable, puis a recevra la valeur de b.
```

Il est donc très facile de déduire les conversions implicites pour les types primitifs numériques. Le deuxième type de transtypage est un peu plus délicat à manier car il vous faudra préciser vous-mêmes la conversion souhaitée. Nous allons reprendre le même exemple, mais en affectant la valeur de type `byte` à un entier noté `int` :

```
byte MonByte2;
int MonEntier2 = 113;
MonByte2 = (byte)MonEntier2;
Console.WriteLine("type de MonByte2 :: " + MonByte2.GetType());
```

Dans ce cas, vous devrez mettre le type de destination entre parenthèse devant la valeur à convertir. Tout type possède une méthode `GetType` renvoyant le type. Si jamais vous oubliez de convertir explicitement, Visual Studio vous surligne la valeur affectée – dans notre cas, cela serait `MonEntier2`. Lorsque vous passez la souris au-dessus du surlignage, Visual Studio vous indique que vous avez peut-être oublié une conversion et vous précise qu'une conversion explicite existe. Cela n'est pas toujours le cas.

Que se passe-t-il si jamais une valeur numérique sort de la plage de valeur admise par le type de destination ? Par exemple, que se passerait-il si `monEntier3` était égal à 312 ? Le résultat correspond à la valeur à convertir, modulo (%) la longueur de la plage des valeurs du type de destination.

Modulo est un opérateur mathématique qui permet de calculer le reste d'une division entière. Dans notre exemple "312 modulo 256" nous renverra 56. 312 divisé par 256 est égal à 256 divisé par 256 (qui est nombre entier) plus 56 divisé par 256. 56 est bien le reste de la division. Ce sera donc la valeur affectée à notre variable de type byte :

```
int MonEntier3 = 312;
byte MonByte3 = (byte)MonEntier3;
Console.WriteLine("type de MonByte3 :: " + MonByte3.GetType() + " -
                    valeur :: " + MonByte3);
```

Un autre moyen de convertir explicitement une valeur est d'utiliser la classe Convert. Celle-ci possède des méthodes de conversion. Nous pourrions par exemple écrire :

```
byte MonByte4;
int MonEntier4 = 312;
MonByte4 = Convert.ToByte(MonEntier4);
Console.WriteLine("type de MonByte4 :: " + MonByte4.GetType());
```

L'avantage de cette méthode est qu'elle génère par défaut une erreur à l'exécution si jamais la valeur à convertir ne correspond pas à la plage du type de destination. Dans l'exemple précédent, nous aurons donc une erreur levée.

2.3.4.6 Différencier les champs et les propriétés

Jusqu'à maintenant nous n'avons utilisé que des champs de classe. Une propriété est un membre de classe stockant une valeur au même titre qu'un champ de classe. Cependant l'avantage d'une propriété est qu'elle permet l'ajout de logique lorsqu'elle est mise à jour ou lue. Définir une propriété est simple avec Visual Studio, il suffit de taper prop, suivi de deux tabulations pour générer automatiquement le code :

```
public string modele { get; set; }
```

Ceci est la nouvelle écriture écourtée des propriétés depuis C# 3. Vous remarquez le mot-clé get qui signifie obtenir et le mot-clé set qui signifie établir. Ces termes représentent deux méthodes qui vont nous permettre de contrôler les accès à notre propriété. En règle générale, une propriété est toujours accompagnée d'un champ privé, donc non accessible, qui est mis à jour par ces deux méthodes. Nous allons maintenant ajouter un champ privé et un peu de logique dans notre propriété :

```
class Program
{
    static void Main(string[] args)
    {
        Voiture maVoiture = new Voiture();
        Console.WriteLine("modele de la voiture :: " + maVoiture.modele);
        Console.ReadLine();
    }
}

class Voiture{
    // ceci est le champ privé modele de type chaîne de caractères
    private string _modele;
    public string Modele
```

```
{
    get{ // on met la logique concernant la lecture de la propriété
        //si le modèle est défini alors je renvoie le nom du modèle
        if ( _modele.Length > 0) return _modele;
        else return "N.C."; //si non je renvoie Non Communiqué
    }
    set{ // on met la logique concernant l'écriture de la propriété
        // si la valeur que l'on essaie d'établir n'est pas nulle
        // alors on affecte la nouvelle valeur du modèle
        if (value.Length > 0) _modele = value;
    }
}
```

Par défaut, une propriété générée par Visual Studio est publique. Si vous souhaitez créer une propriété uniquement accessible en lecture, il suffit de supprimer la méthode set. Vous pouvez également préciser un modificateur d'accès devant le mot-clé set, ce qui est plus élégant et avantageux. Le modificateur d'accès `internal` permettra par exemple d'accepter l'écriture de cette propriété uniquement aux objets contenus dans l'espace de nom ; `protected` permettra quant à lui de laisser l'accès en écriture aux classes héritant de `Voiture`.

2.3.4.6 Le tas et la pile

On répartit les types en deux grandes catégories : les références attribuées sur le tas et les valeurs attribuées sur la pile. Concrètement, il s'agit de deux manières différentes de stocker des données. La pile représente l'espace mémoire alloué pour les valeurs, alors que le tas représente l'espace mémoire gérant les références. La gestion de ces espaces mémoire est très différente. L'allocation mémoire est figée dans le cas de la pile mais peut être variable sur le tas. Vous pouvez considérer les références comme des variables qui pointent vers une adresse mémoire, l'affectation sera contenue à cette adresse. Le Tableau 2.1 vous présente ces types.

Tableau 2.1 : Types gérés par la pile et le tas.

<i>Valeurs attribuées sur la pile</i>	<i>Références gérées sur le tas</i>
byte / sbyte	Array
short / ushort	classe personnalisée
int / uint	string
long / ulong	interface
boolean	List
double, float, decimal	ObservableCollection
char	
énumération	
structure	

Vous remarquez que le type `string` est géré sur le tas. Chaque modification d'une chaîne de caractères entraîne une allocation mémoire et donc des accès coûteux en terme de performances. Il faudra utiliser la classe `StringBuilder` qui permet d'alléger les accès mémoire.

Différencier une référence d'une valeur est assez simple. Il faut se rappeler qu'une référence est une variable qui pointe à une adresse mémoire. Commençons par définir une valeur de type structure allouée sur la pile :

```
struct PositionGPS{
    public int DegreLat;
    public int DegreLong;
}
PositionGPS Gpa = new PositionGPS { DegreLat = 141, DegreLong = 76 };
PositionGPS Gpb = Gpa;
Gpa.DegreLat = 35;
Console.WriteLine(Gpb.DegreLat);
```

La console affiche 141, la structure `gpb` ne change pas car nous avons procédé à une simple copie de valeurs. Les deux variables ciblent un espace mémoire différent. Elles sont donc autonomes. Examinons le même code en remplaçant les structures par des classes personnalisées. Celles-ci sont attribuées sur le tas :

```
class Point{
    public int X;
    public int Y;
}
Point P1 = new Point(){ X = 141, Y = 76 };
Point P2 = P1;
P1.X = 35;
Console.WriteLine(P2.X);
```

Cette fois, la console affiche 35, la valeur du champ `X` de `P2` a changé car l'instance `P2` pointe vers la même allocation mémoire que `P1`. On procède à une copie de pointeur dans le cas d'une affectation de variables de type référence. `P2` et `P1` ciblent le même espace mémoire.

2.3.4.7 Les inférences de type

Un nouveau mot-clé est apparu avec C# 3 : `var`. Ce mot-clé vous évitera de préciser le type d'une variable lors de sa déclaration. Cela signifie que le type d'une variable déclarée avec `var` n'est fixé qu'au moment de l'affectation de la variable. Cette capacité est très pratique si vous ne connaissez pas à l'avance la valeur d'affectation. Toutefois le mot-clé `var` n'est pas utilisable pour déclarer des membres de classes, il sert essentiellement à définir des variables locales aux méthodes.

2.3.4.8 Les types anonymes

Les types anonymes ont un peu le même rôle que les structures. Comme pour celles-ci, l'objectif majeur est de créer un objet ne contenant que des propriétés (bien que les structures puissent faire mieux). C'est une reprise directe de ce que vous trouverez dans les langages dynamiques fonctionnels. Ces types anonymes ne peuvent être définis et affectés qu'au sein d'une méthode, grâce au mot-clé `var`. Il s'agit donc de variables locales. Voici comment créer une classe anonyme :

```
var MaVoitureAnonyme = new { Modele="307", Marque="Peugeot",
                             Prix=8950, Annee=2004 };
```

En réalité, le typage fort est présent, mais une classe anonyme, ainsi que les champs qui la composent, sont créés dynamiquement.

2.3.5 Déclarer des méthodes

2.3.5.1 Définition et appel de méthodes

Une méthode est une fonction appartenant à une classe. Son premier objectif est de centraliser la mise en œuvre d'une série d'instructions. Celle-ci sera exécutable à l'infini par simple appel du nom de la méthode, suivi de deux parenthèses (ouverte et fermée). Quels que soient les langages, les fonctions ou méthodes permettent la réutilisation du code. En langage C#, les fonctions globales qui n'appartiennent à aucune classe n'existent pas. C'est un abus de langage de dire qu'une méthode statique de classe (donc appartenant et exécutable par la classe) est une fonction globale. *A contrario* de C#, les fonctions sont considérées comme des objets dans les langages dits fonctionnels. Dans ces langages, dont JavaScript fait partie, une fonction peut exister par elle-même (car elle est considérée comme un objet). Ce n'est pas le cas en C#. Deux types de méthodes existent, les méthodes exécutables par la classe et les méthodes invoquées par les exemplaires ou instances de classe. Il faut tout d'abord définir une méthode pour l'invoquer par la suite. Voici une définition et un appel de la méthode rouler, propre aux exemplaires de la classe Voiture :

```
class Voiture{
    public void Rouler(){
        Console.WriteLine("La voiture roule 100Km");
    }
}
class Program{
    static void Main(string[] args){
        //On invoque la méthode rouler sur l'instance de Voiture MaDeLorean
        MaDeLorean.Rouler();
        Console.ReadLine();
        //la console tracera le message définit au dessus
    }
}
```

Comme vous pouvez le constater, le nom de la méthode est d'abord précédé du mot-clé `public` (voir "Les modificateurs d'accès"), puis du mot-clé `void` qui fait référence au type muet. Cela signifie que la méthode ne retourne aucune valeur. Dans le cas d'une méthode renvoyant une valeur, il suffit de préciser le type à la place de `void`. On pourrait déduire que `Rouler` modifie la position de la voiture et renvoie du dioxyde de carbone en même temps. Voici le même exemple mis à jour :

```
public class Voiture{
    public int Rouler(){
        Console.WriteLine("La voiture roule 100Km ")
        return 125;
    }
}
class Program{
    static void Main(string[] args){
        Voiture MaDeLorean = new Voiture(){
            TypeCarburant = Carburant.Detrinitus,
            Marque = "De Loreane", Modele = "volante",
            MyPlace = new PositionGPS() { DegreLat = 180, DegreLong = 176 }
        };
        //On invoque la méthode rouler sur l'instance MaDeLorean
    }
}
```



```

        Console.WriteLine("Elle rejette :: " + MaDeLorean.Rouler() +
                           " g CO2 / Km");
        Console.ReadLine();
        //la console tracera le message définit dans la méthode,
        //puis le message défini dans la méthode Main
    }
}

```

Comme nous l'avons déjà dit, la méthode `Main` de la classe `Program` est une méthode d'initialisation, c'est le point d'entrée de notre application console, c'est-à-dire que celle-ci sera exécutée dès le lancement de l'application par la CLR. C'est pour cela que sa définition commence par le mot-clé `static`. Les méthodes `static` présentent de nombreux avantages. Nous pourrions par exemple créer une méthode de ce type pour la classe `Voiture`, qui renverrait le nombre de voitures en circulation.

2.3.5.2 Les paramètres de méthode

Nous allons maintenant améliorer la méthode `rouler` en lui définissant un paramètre. Comme nous l'avons vu au début de ce chapitre, une méthode doit faciliter la réutilisation¹. Toutefois, on constate que notre méthode `rouler` ne permet à une voiture de rouler que 100 km. Il serait utile de pouvoir choisir le nombre de kilomètres à chaque appel de cette méthode. Pour cela, il nous faut ajouter un paramètre dans la définition de cette méthode :

```

class Voiture
{
    public string Marque;
    public string Modele;
    public Carburant TypeCarburant;
    public PositionGPS MyPlace;
    public int Rouler( int nombreKilometre)
    {
        Console.WriteLine("La voiture roule" + nombreKilometre + "km");
        return 125;
    }
}

```

Puis, nous invoquons celle-ci en lui passant le nombre de kilomètres attendus :

```

static void Main(string[] args)
{
    //On invoque la méthode rouler sur l'instance maDeLorean
    Console.WriteLine("Elle émet :: " + MaDeLorean.Rouler(113) +
                      " g CO2 / Km");
    Console.ReadLine();
    //la console tracera le message défini dans la méthode en
    //prenant en compte le nombre de kilomètres parcourus
}

```

2.3.5.3 Le tableau de paramètres

Dans certains cas, vous aurez besoin de passer un nombre variable de paramètres. C# propose pour cela une signature de fonction spécifique. Il vous faudra utiliser le mot-clé `params` entre les

1. La réutilisation est un principe connu des développeurs, qui consiste à réutiliser le code ou les fonctionnalités déjà conçues.

parenthèses en précisant le type des paramètres reçus. Le terme `params` représente le tableau des paramètres qui ont été passés lors de l'appel de la méthode. Si vous ne souhaitez pas préciser de type, il suffit de préciser `object`. Toutes les classes en C# héritent de `object`, ainsi vous pourrez spécifier n'importe quel type de paramètre. Dans l'exemple suivant, nous calculons le prix moyen d'une voiture :

```
static void Main(string[] args){
    Voiture MaDeLorean = new Voiture(){
        Prix=150000,
        TypeCarburant = Carburant.Detritus,
        Marque = "De Loreane",
        Modele = "volante",
        MyPlace = new PositionGPS() { DegreLat = 180, DegreLong = 176 }
    };
    Voiture MaPorsche = new Voiture() { Marque = "Porsche",
        Modele = "Carrera 4", TypeCarburant = Carburant.Essence,
        Prix=18500 };
    Voiture MaSuper5 = new Voiture() { Marque = "Renault",
        Modele = "Super5", TypeCarburant = Carburant.Diesel,
        Prix=1300 };
    //on trace directement le retour de la méthode CalculPrixMoyen
    Console.WriteLine("prix moyen des voitures :: {0}", CalculPrixMoyen
        (MaDeLorean, MaPorsche, MaSuper5) );
}
//on utilise le tableau de paramètre et on précise un type de retour
//car cette fonction renvoie la moyenne calculée
private static decimal CalculPrixMoyen( params Voiture[] mesVoitures){
    decimal Total = 0;
    foreach (Voiture v in mesVoitures){
        Total += v.Prix;
    }
    return Total / mesVoitures.Length;
}
```

2.3.5.4 Portée de variables

Les variables que vous définissez au sein d'une méthode ne sont accessibles qu'à l'intérieur de celle-ci. Ce sont des variables locales à la méthode. À la fin de l'exécution et dans le cas de variables de type valeur, celles-ci libèrent la mémoire qui leur est allouée. Ces variables ne sont donc pas accessibles depuis l'extérieur. La variable locale `total` de l'exemple précédent est donc inaccessible dans la méthode `main`. Cependant, vous pouvez très bien décider d'affecter une propriété ou un champ de classe. Ils peuvent être atteints depuis n'importe quel endroit au sein de cette classe mis à part au sein de membres ou propriétés statiques.

2.3.5.5 Les méthodes d'extension

Comme nous l'avons dit à plusieurs reprises, C# s'oriente vers les langages dynamiques. Les méthodes d'extension sont un nouveau pas dans cette direction. Celles-ci permettent d'ajouter de manière propre et efficace des méthodes à n'importe quel type. Une méthode d'extension doit toujours être `static` et `public` et appartenir à une classe `static`. Tout se passe au niveau de la signature de la méthode dont voici un exemple simple :

```
static class MyExtensionsMethods {
    public static bool IsBiggerThan (this int myInt, int compare)
    {
```

```
        return myInt > compare;
    }
}
```

Vous remarquez que le premier paramètre commence par le mot-clé `this`. Cela signifie qu'il fait référence à la variable de type entier qui va faire appel à la méthode. Le second paramètre est le premier paramètre de la méthode `IsBiggerThan` lorsque celle-ci sera appelée. Voici comment se déroule l'appel :

```
int monEntier = 37;
bool myBoolean = monEntier.IsBiggerThan( 13 );
```

2.3.6 Héritage et implémentations

2.3.6.1 Principes

Un des grands axes de la programmation orientée objet des années 80 à 90 est la capacité des classes à hériter d'autres classes. Dans l'application d'exemple, peut-être souhaitez-vous louer des voitures de type `Utilitaire`. Dans ce cas, il n'est pas nécessaire d'écrire une classe `Utilitaire` en reprenant chaque champ de la classe `Voiture`. Nous pouvons simplement considérer que la nouvelle classe `Utilitaire` héritera de la classe `Voiture`. Dans ce dernier cas, nous n'aurons qu'à le préciser au moment de sa définition comme ceci :

```
class Utilitaire : Voiture{
    public int Volume;
    public void Charger(){
        Console.WriteLine("on charge l'utilitaire");
        Console.ReadLine();
    }
    public void Decharger(){
        Console.WriteLine("on décharge l'utilitaire");
        Console.ReadLine();
    }
}
```

Le grand avantage réside dans le fait que nous n'avons pas à recoder une nouvelle classe entièrement. De ce point de vue, l'héritage en POO est une technique de développement qui permet la réutilisation. La sous-classe de `Voiture` (`Utilitaire`) bénéficie de tous les comportements et propriétés d'une voiture normale en plus des siennes. Voici comment créer une instance de la classe `Utilitaire` :

```
//une classe héritée de Voiture
Utilitaire MonKangoo = new Utilitaire() { Marque = "Renault",
Longueur = 1.2, Modele = "Super5", TypeCarburant = "diesel", Volume=6
};
```

La problématique n'est cependant pas si simple car imaginez un véhicule de type 4x4. Nous pourrions l'envisager comme un utilitaire ou un véhicule familial. Doit-il hériter dans ce cas de la classe `Voiture` ou de la classe `Utilitaire` ? Il n'y a malheureusement aucune bonne réponse à cette question en utilisant l'héritage, car il est impossible d'hériter de plusieurs classes en C#. Nous avons donc là un problème de conception important : n'oubliez pas qu'une fois la décision prise, vous ne pourrez plus revenir en arrière facilement. Votre développement subira à long terme

les décisions que vous aurez pris trop tôt face à la nécessité de commencer le développement. Bien que de nombreuses techniques permettent de concevoir un code souple à la modification, la programmation objet tente de classer les fonctionnalités par type, ce qui ne correspond pas toujours à une réalité. Les langages dynamiques permettent la prise de décisions tardives mais aussi de rectifier simplement des choix de conception. Plus vous décidez tard, meilleure est votre appréciation de la situation et des besoins. Nous verrons que C# s'oriente dans cette direction depuis sa version 3.

2.3.6.2 Surcharge de méthodes

Comme nous l'avons vu, toutes les classes de C# héritent de la classe `object`. Celle-ci possède une méthode `ToString` qui permet d'avoir une représentation sous forme de chaîne de caractères de l'instance sur laquelle elle est invoquée. Les classes C# ont pour la plupart une implémentation différente de cette méthode. Les classes personnalisées dont `Voiture` fait partie, implémentent par défaut la méthode `ToString` propre à `object`. Utilisons cette méthode sur notre classe `Voiture` et voyons le résultat :

```
Console.WriteLine( MaDeLorean.ToString());  
//renvoie ParcAuto.Voiture
```

Comme vous le constatez, la méthode retourne par défaut l'espace de noms et le type `Voiture`. Il peut être utile d'avoir un retour personnalisé pour la classe `Voiture`. Nous allons donc redéfinir notre propre méthode `ToString`, il suffit d'utiliser le mot-clé `override` :

```
public override string ToString(){  
    return "modèle :: " + Modele + " - marque :: " + Marque;  
}
```

Ce mot-clé permet d'outrepasser la méthode héritée de la classe de base. Ainsi, vous pouvez redéfinir cette méthode à votre convenance. Il est encore possible d'invoquer la méthode originelle de cette manière :

```
return "classe :: " + base.ToString()+" - modèle :: "  
    + Modele + " - marque :: " + Marque;
```

Le terme `base` fait référence à la classe dont on hérite, dans notre cas c'est `object`.

2.3.6.3 Déclarer des interfaces

Une interface est constituée des signatures de méthodes, de délégués et d'événements (voir Chapitre 8). Elle représente un contrat abstrait d'implémentation. Une classe implémentant une interface doit nécessairement contenir les mêmes signatures de méthodes, délégués et événements mais avec une définition concrète de ceux-ci (à l'exception des classes abstraites). Les interfaces (et les classes abstraites) permettent d'assouplir le développement et rendent le code plus facile à maintenir et évolutif. L'utilisation d'interfaces se révèle au final moins dangereux et plus facile à gérer que l'héritage. Pour implémenter une ou plusieurs interfaces, il faudra ajouter un signe : après le nom de la classe, suivi du nom des interfaces. Si la classe est héritée d'une autre classe, il faudra suivre cet ordre :

```
class MaClasse : ClasseDeBase, interface1, interface2,..., interfaceN  
class Voiture : Vehicule, IUtilitaire, IFamilial
```

Par convention, on préfixe les noms d'interface de I, ce qui donne plus de lisibilité.

Nous n'entrerons pas plus dans les détails de C# au sein de ce chapitre car, comme vous l'avez constaté, C# est un langage performant dont l'apprentissage nécessiterait un livre à part entière. Maintenant que vous êtes familiarisé avec cet environnement, vous allez créer votre première application Silverlight grâce aux outils proposés au sein de la gamme Expression.

Dans le prochain chapitre, vous utiliserez Expression Blend et aborderez les bases de la mise en forme XAML, ainsi que l'architecture des projets Silverlight.

HelloWorld

Nous allons concevoir une première application Silverlight avec le logiciel Expression Blend. Ce projet nous servira de base pour détailler l'architecture par défaut des projets Silverlight ainsi que l'organisation de l'interface Blend. Nous aborderons ensuite la notion d'arbre visuel et logique ainsi que l'utilisation de conteneurs. Nous listerons rapidement les composants de gestion de texte et les options d'alignement. Nous finirons ce chapitre par une première compilation de l'application, ce qui nous permettra d'énumérer les fichiers qu'elle produit tout en abordant ses mécanismes internes.

3.1 Une première application Silverlight

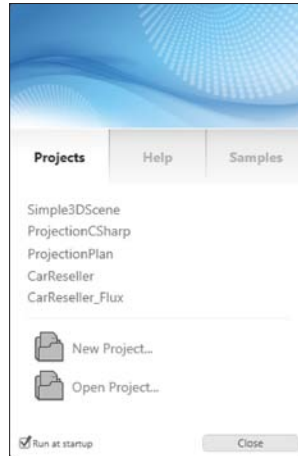
Commencez par créer un nouveau répertoire sur votre disque dur. Vous pouvez le nommer `Pratique_de_Silverlight`, par exemple. C'est dans ce répertoire que seront contenus tous les projets que vous allez créer dans ce livre. Vous devez vous munir de la dernière version d'Expression Blend afin de générer une solution Silverlight. Celle-ci est disponible en version d'essai sur le site de Microsoft : <http://www.microsoft.com/france/expression/>. De manière générale, vous trouverez les pré-requis logiciels sur le blog Tweened.org à cette adresse : <http://www.tweened.org/pre-requis-silverlight/>. Ils sont mis à jour à chaque nouvelle version de Silverlight.

Une fois que vous avez installé Blend et l'ensemble des pré-requis, démarrez l'application. Vous devriez avoir une interface correspondant à la Figure 3.1.

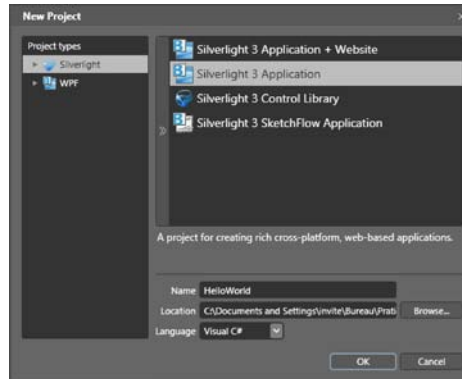
Blend vous permet d'afficher un écran de bienvenue. Cochez la case en bas à droite si vous souhaitez qu'elle apparaisse à chaque démarrage. Vous allez maintenant créer votre premier projet. Pour cela, sélectionnez l'onglet Projects de l'écran de bienvenue, puis cliquez sur New Project... Vous devriez voir un nouvel écran s'afficher (voir Figure 3.2).

Figure 3.1

Écran de bienvenue
d'Expression Blend.

**Figure 3.2**

Création d'un
nouveau projet.



Dans la boîte de dialogue affichée, plusieurs possibilités s'offrent à vous. Vous pouvez tout d'abord choisir entre deux familles de projets : WPF ou Silverlight.

Au sein de la famille Silverlight, quatre choix sont accessibles par défaut. Les solutions de type Silverlight 3 Application ou Silverlight 3 Application + Website sont très semblables. La première sera exécutée dans une page HTML générée dynamiquement lors de chaque compilation. *A contrario*, une solution de type site propose des fichiers HTML et JavaScript permettant d'intégrer l'application pour une mise en production et un déploiement rapide (voir Chapitre 4). Le troisième type, Silverlight 3 Control Library, génère un projet facilitant la centralisation de contrôles personnalisés. Créer des contrôles personnalisés avec Silverlight est tellement simple que cette manière de procéder est assez naturelle. Le dernier type de projet, Silverlight 3 SketchFlow Application, permet de concevoir, de manière efficace et rapide, l'architecture entière d'une application Silverlight. Nous aborderons la création de prototypes avec SketchFlow au Chapitre 10.

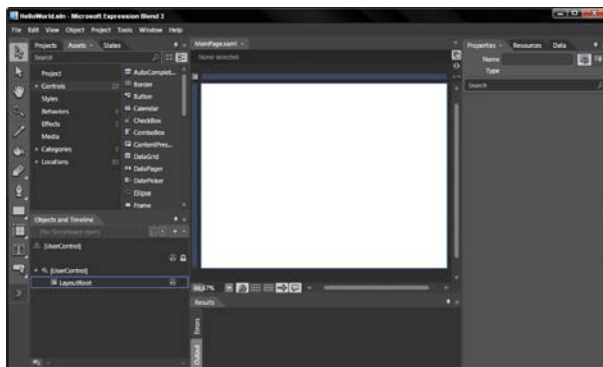
Choisissez Silverlight 3 Application, puis le langage C# (normalement sélectionné par défaut). Si vous êtes développeur Visual Basic, vous avez bien sûr la possibilité de choisir ce langage. Il faut également préciser le chemin d'accès au dossier que nous avons précédemment créé. Il va contenir l'ensemble de nos projets. Voici un exemple : C:\Documents and Settings\invite\Bureau\

Pratique_de_Silverlight\). Pour finir, entrez HelloWorld dans le champ Name, cliquez sur OK. Une autre méthode serait de fermer le panneau, d'ouvrir le menu File, puis de sélectionner New Project... Il n'y a aucune différence de résultat entre ces deux manières de faire.

Une fois cette étape réalisée, vous accédez à l'interface de Blend correspondant aux projets de type Silverlight. De légères différences d'interface existent entre les projets WPF et Silverlight. On a coutume de dire que Blend est réalisé avec Blend. Cela fait référence à la nature même de ce logiciel qui repose sur WPF. Vous devriez récupérer une interface correspondant à la Figure 3.3.

Figure 3.3

L'interface d'Expression Blend à l'ouverture du projet HelloWorld.



Blend a placé automatiquement, sur votre disque dur, un nouveau répertoire portant le nom du projet. Créer une application Silverlight, ou WPF, revient à générer un répertoire sur votre disque dur contenant un ensemble de fichiers par défaut. Comme vous le constatez à la Figure 3.3, l'interface de Blend est constituée de plusieurs parties distinctes :

- La barre de menu, située en haut, permet de créer des boutons et des composants personnalisés, mais donne également accès aux opérations sur les tracés, aux préférences du projet et à de nombreux autres menus inaccessibles d'une autre manière.
- Complètement à gauche de l'interface, vous trouverez une barre d'icônes. Chaque icône fait référence à un type d'outil. Ceux-ci sont catégorisés en cinq familles : les outils de sélection, de manipulation de la vue (le zoom par exemple), de modification, de création d'objets primitifs (tels que les rectangles, les tracés) et de création de contrôles (du plus simple conteneur aux objets visuels et logiques complexes comme la `ListBox`).
- En haut à gauche se trouve une série de panneaux gérant les états visuels, la liste des composants et des ressources, ainsi que l'arborescence du projet présent sur le disque dur. Par défaut, vous trouverez l'état visuel Base qui contient le visuel par défaut de l'application. Le panneau Projects contient une arborescence des fichiers nécessaires au fonctionnement de l'application (voir la section 3.2). La notion d'états visuels, telle qu'elle est réalisée dans Silverlight, est relativement innovante. Nous ajouterons des états visuels au sein de nos applications afin de scinder visuellement leurs fonctionnalités et de gérer les transitions (voir Chapitre 7).
- En bas à gauche se situe le panneau contenant l'arbre visuel et logique de notre application. Ce panneau affiche l'ensemble des composants d'une application. Ceux-ci sont hiérarchisés selon leur ordre d'imbrication et leurs liens de parenté. On peut considérer que l'arbre visuel et logique est la représentation graphique des liens familiaux entre les objets XAML. C'est également dans ce panneau que l'on pourra concevoir des animations

- Le centre de l'application correspond à l'espace alloué aux designers et aux intégrateurs pour concevoir les éléments visuels ainsi que l'architecture de l'interface utilisateur. Le rectangle blanc au milieu est la grille principale d'agencement, nommée `LayoutRoot`. La couleur d'arrière-plan par défaut est le blanc, mais vous pourriez décider de ne pas en définir dans l'optique de créer une application Silverlight transparente. Cette vue est donc réellement importante car c'est son ergonomie qui rend possible la participation des designers. Cet espace permet également d'afficher le code XAML généré par Blend lorsque le graphiste crée l'interface visuelle.
- En bas, au centre, le panneau sortie et erreur permet d'afficher des informations de sorties lors de la compilation de l'application Silverlight. Il permet également d'exposer les erreurs levées lors de la compilation ou de l'utilisation de Blend.

Complètement à droite se concentre une série d'onglets : Data, Properties et Resources. En voici le détail :

- Le panneau Data permet de gérer des sources de données externes ou propres à la solution. Il permet également de créer des jeux de données fictives et de simuler un flux RSS ou un tableau d'objets C#.
- Le panneau Properties est contextuel, c'est-à-dire qu'il est mis à jour en fonction de l'objet sélectionné dans l'arbre visuel et logique ou dans la fenêtre de design. Il dresse un inventaire complet de toutes les propriétés de l'objet en cours de sélection. Celles-ci sont parfois si nombreuses qu'un champ texte de saisie permet de les filtrer.
- Le panneau Resources est également contextuel au fichier XAML sélectionné ou à l'objet sélectionné. Il liste l'ensemble des ressources visuelles ou logiques accessibles. Nous consacrerons le Chapitre 11 aux ressources visuelles et logiques.

3.2 Architecture d'une solution

Le panneau Projects affiche l'ensemble des fichiers faisant partie de l'application. L'élément hiérarchique le plus élevé est la solution. Une solution est l'unité d'organisation principale. Celle-ci est constituée d'au moins un projet. Tous les fichiers nécessaires pour la conception ou produits lors de la compilation sont répartis au sein de divers projets. Chaque projet a pour but de gérer une application, un module, une fonctionnalité ou une bibliothèque de contrôles facilitant l'organisation du développement. On a donc, en premier lieu, une solution contenant un projet du même nom (voir Figure 3.4).

Voici la liste des fichiers générés, par défaut, au sein du projet lorsque vous créez une application Silverlight :

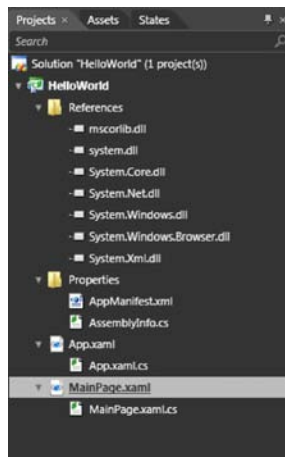
- Le fichier `MainPage.xaml`. Il contient le code XAML décrivant les objets graphiques et logiques de l'interface utilisateur. Par défaut, c'est également la première page chargée par l'application. Le travail du graphiste et de l'intégrateur se répercute dans un premier temps dans ce fichier. Lors de l'avancement du projet, d'autres fichiers XAML sont ajoutés, notamment des dictionnaires de ressources.
- Le fichier `MainPage.xaml.cs` contient le code logique C#. Celui-ci assure la partie fonctionnelle de la première page de l'application. Ce fichier concerne les développeurs C# et les intégrateurs. Aucun fichier logique JavaScript n'est présent dans ce type de solution. Toutefois, dans le cas de projets Visual Basic, le fichier aura une extension propre à ce langage.

- Le répertoire References contient toutes les bibliothèques C# ou assemblies nécessaires par défaut. Ce sont des fichiers dll qui permettent d'étendre à volonté les capacités fonctionnelles de l'application. Si vous souhaitez utiliser des formats de données comme XML ou JSON par exemple, vous devrez ajouter une référence sous forme de dll. Pour cela, il suffit de faire un clic-droit, puis de choisir Add Reference... ou Add Project Reference...
- Le répertoire Properties contient deux fichiers décrivant l'application dans ses grandes lignes : auteur, compagnie, objectif, version, etc.
- Les fichiers App.xaml et App.xaml.cs contiennent et centralisent du code inhérent au projet lui-même. Ils permettent de préciser quelle est la première page de l'application qui va être chargée. Ils peuvent également contenir des styles ou des modèles de composants accessibles pour toutes les pages XAML. Ces fichiers sont importants car ils représentent l'instance de l'application Silverlight dans la page HTML. Ils permettent donc de définir des comportements précis à l'initialisation ou à la fermeture de celle-ci.

Les fichiers énumérés ci-dessus ne sont pas les seuls pouvant faire partie d'une solution. De nombreux types de documents peuvent être utilisés au sein de solutions Silverlight. Ce sont généralement des ressources générées depuis des applications externes, comme Illustrator et Photoshop, ou créées au sein de Blend ou de Visual Studio. Vous pouvez à tout moment ajouter une nouvelle ressource, par exemple une police de caractère ou un fichier de code logique, à un projet existant.

Figure 3.4

Le panneau Projects.



Nous allons maintenant voir ce qui a été généré sur le disque dur. Ouvrez votre explorateur Windows afin d'accéder au répertoire contenant votre solution. Si celui-ci est sur votre bureau et que vous avez suivi la procédure indiquée, voici son chemin d'accès : C:\Documents and Settings\invite\Bureau\Pratique_de_Silverlight\. Votre répertoire doit contenir un dossier du nom de la solution. Celui-ci comprend un répertoire du même nom (correspondant au projet généré par défaut), ainsi que deux autres fichiers : HelloWorld.sln et HelloWorld.suo qui est un fichier caché contenant des paramètres propres à la solution (voir Figure 3.5).

Figure 3.5

Contenu du répertoire de la solution HelloWorld.



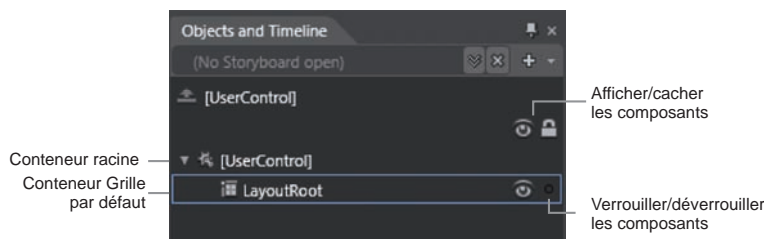
L'extension `.sln` indique un fichier solution. Il référence les projets contenus par la solution ainsi que diverses informations. Le répertoire `HelloWorld` possède un contenu correspondant à ce qui apparaît dans le panneau projet de Blend plus un répertoire `bin`. Celui-ci reçoit les fichiers binaires générés lors de la compilation (voir section 3.5.3). Lorsque vous ajoutez un fichier de type ressource ou code logique à un projet Silverlight, celui-ci est automatiquement placé dans le répertoire du projet. Cela permet au projet d'être autonome, déplaçable librement sur le disque dur ou encore d'être partagé.

3.3 Le conteneur racine

Maintenant que nous avons vu l'architecture d'une solution, nous allons étudier celle d'une page d'application Silverlight. Lors de l'initialisation d'une application Silverlight, celle-ci charge une page par défaut. La page en question est issue de la compilation des deux fichiers `MainControl.xaml` et `MainControl.xaml.cs`. Le premier fichier est de type XML, il contient donc un nœud racine. Ce nœud est le conteneur parent de tous les objets visuels et logiques de la page. Il est de type `UserControl`. Il détermine la dimension de la page, si elle possède un fond transparent ou encore tout ce qui a trait directement ou indirectement à son affichage. Les composants de type `UserControl` ne peuvent contenir qu'un unique objet enfant. Ainsi, déposer un composant directement dans un `UserControl` efface l'enfant qui s'y trouve éventuellement. Lors de la création d'un nouveau projet Silverlight, un contrôle Grid nommé `LayoutRoot` lui est ajouté, par défaut, comme élément enfant. Celui-ci peut, contrairement au composant `UserControl`, contenir plusieurs enfants visuels et logiques.

Figure 3.6

Arbre visuel et logique d'une page.



Comme vous pouvez vous en rendre compte, une page est constituée d'éléments imbriqués. Il n'y a donc pas réellement de notions de calques ou de couches. Un calque (ou une couche) est une unité d'organisation abstraite qui regroupe plusieurs objets sur une même profondeur visuelle. Les conteneurs Silverlight sont, quant à eux, des objets sélectionnables ayant un aspect physique et une logique d'agencement propre. Ce n'est pas un handicap, bien au contraire. Cela force à concevoir l'imbrication dès le départ et rend plus propre et efficace le développement d'applications ou de sites. L'ensemble des éléments imbriqués constitue l'arbre visuel et logique de l'application.

3.4 Ajouter du texte

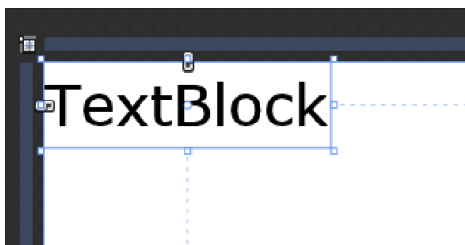
3.4.1 Créer le champ texte

Vous allez maintenant ajouter un champ texte centré. Pour cela, vous trouverez une icône de champ texte dans votre barre d'outils. Cliquez dessus et maintenez l'icône enfoncée. Trois pictogrammes

apparaissent, correspondant aux trois types de champs texte que vous pouvez créer : `TextBlock`, `TextBox` et `PasswordBox`. Le contrôle de type `TextBlock` vous permet d'afficher du texte. Les deux autres types autorisent l'utilisateur à saisir du texte lors de l'exécution de l'application. Le composant `PasswordBox` est utile pour la saisie de mots de passe utilisateur. Sélectionnez le premier type de champ (`TextBlock`). Nous allons ajouter une instance de type `TextBlock` dans notre conteneur `LayoutRoot`. Double-cliquez sur l'icône. Un nouveau `TextBlock` est automatiquement créé en haut à gauche, dans la grille `LayoutRoot`. À sa création un `TextBlock` est en mode édition de texte (voir Figure 3.7). Pour en sortir, appuyez sur la touche `Esc` ou cliquez hors du composant.

Figure 3.7

Création d'un `TextBlock`.



3.4.2 Alignement

Lors de sa création *via* un double-clic et au sein d'un contrôle de type `Grid`, un objet est toujours placé en haut à gauche. Vous allez centrer le champ texte au sein de la grille. Pour cela, sélectionnez-le dans l'arbre visuel, puis ouvrez l'onglet `Properties`. À l'intérieur de la catégorie `Layout`, vous trouvez deux séries d'icônes reflétant les options d'alignement ainsi que quatre champs de saisie juste en dessous. Ceux-ci permettent de spécifier des marges extérieures et doivent être utilisés conjointement avec les options d'alignement. C'est le paramétrage de l'ensemble de ces propriétés qui détermine le positionnement des objets au sein d'un conteneur. Assurez-vous que ces champs ont tous une valeur égale à zéro. Si un quelconque chiffre apparaît dans un de ces champs, il faut le remettre à zéro. Si vous avez créé le champ texte en le dessinant directement au sein de la grille, ces propriétés ne sont pas vides. Pour réinitialiser n'importe quelle propriété dans `Blend`, il vous suffit de cliquer sur l'icône carrée présente à droite de chacune des propriétés et de sélectionner `Rétablir`. Cliquez maintenant sur les deux pictogrammes qui permettent l'alignement horizontal et vertical (voir Figure 3.8).

Figure 3.8

Les options d'alignement.



Votre texte est maintenant centré. Les propriétés largeur et hauteur (`Width` et `Height`) possèdent une valeur `Auto`. Cela signifie que leur valeur s'adaptera automatiquement, soit au contenu du champ texte (la chaîne de caractères), soit au conteneur, en fonction des marges et de l'alignement spécifié. Vous allez paramétrer l'affichage de ce champ. Pour cela, ouvrez la catégorie `Text` du panneau `Properties`. Vous y trouverez plusieurs options, notamment pour choisir la police de

caractère ou encore la mise en forme du texte. L'une d'entre elles, `FontSize`, permet de modifier la taille de la police en pixels. Choisissez une taille d'au minimum 24 pixels. Si vous avez correctement configuré les options d'alignement du texte, celui-ci devrait s'étendre dans les deux sens, mais rester centré dans la grille principale. Dans l'onglet Common Properties, pour la propriété `Text`, entrez la valeur "HelloWorld" ou la chaîne de caractères de votre choix. Une fois de plus, le `TextBlock` s'étend dans les deux directions pour s'adapter à son contenu (voir Figure 3.9).

Figure 3.9

Le projet HelloWorld.



3.5 Tester et compiler

Contrairement aux langages tels que HTML ou JavaScript qui sont interprétés par le navigateur, le lecteur Silverlight exécute des fichiers de type xap compilables par Blend ou Visual Studio. La compilation permet de fusionner les fichiers contenant le code XAML et C# en langage intermédiaire. Cette étape est donc importante.

3.5.1 Première compilation

Vous allez maintenant compiler votre application. Pour cela, il y a trois solutions :

- Dans le panneau Project, après un clic droit sur la solution, sélectionnez Run Project.
- Au sein du menu Project, sélectionnez Run Project.
- Utilisez le raccourci clavier F5.

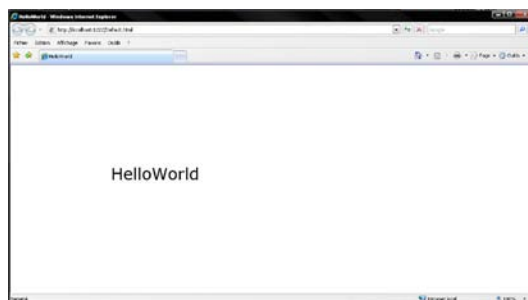
Ces trois méthodes compilent le projet et affichent le résultat dans votre navigateur configuré par défaut. Un lecteur autonome existe, mais tester ou déboguer une application Silverlight ne peut être réalisé qu'au sein d'un navigateur. Appuyez sur la touche F5. Une série de messages apparaît en fenêtre de sortie. Ils vous indiquent la progression de la compilation. La page HTML contenant votre application apparaît (voir Figure 3.10).

INFO

Lors de projets plus complexes, il peut arriver qu'une erreur soit levée à la compilation. Dans ce cas, l'application ne compile pas. Il vous faudra lire le message d'erreur présent en fenêtre de sortie. Toutefois Blend ne bénéficie pas d'un débogueur aussi puissant que celui de Visual Studio. C'est pourquoi les messages d'erreur sont parfois très génériques et peuvent ne pas vous donner d'indices suffisants pour résoudre le bogue.

Figure 3.10

L'application chargée dans le navigateur après compilation.

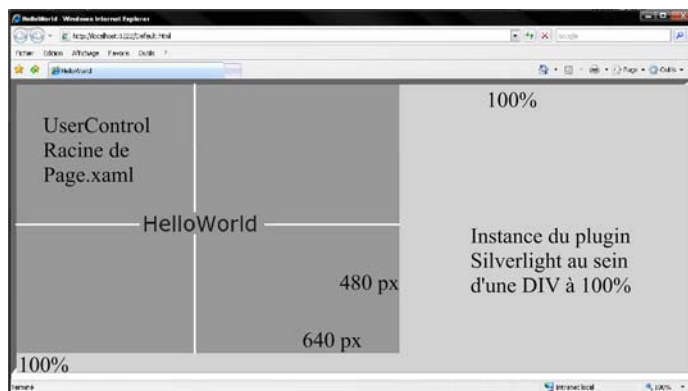


3.5.2 Une application 100 % Silverlight

Vous remarquez qu'au sein de notre page, le composant `TextBlock` n'est pas centré. En réalité, il est correctement centré, mais par rapport à la page de notre application chargée par défaut. Il est donc centré par rapport au conteneur principal du fichier `MainControl1.xaml` et son composant racine `UserControl1`. Si vous cliquez-droit n'importe où sur la page HTML, vous verrez un menu contextuel Silverlight. Celui-ci vous indique que l'instance du plug-in Silverlight occupe toute la place dans la page HTML. Cependant, par défaut, le `UserControl1` racine possède des dimensions fixes (640 pixels de large par 480 de hauteur). Pour le vérifier, sélectionnez dans Blend le `UserControl1` principal et vérifiez les valeurs de ses propriétés `Width` et `Height`. Pour résumer, la page HTML générée par défaut affiche l'instance du plug-in dans une balise de type `DIV` occupant 100 % de hauteur et de largeur, c'est-à-dire que l'instance Silverlight occupe toute la place au sein du navigateur. Toutefois, le `UserControl1` racine possède des dimensions fixes exprimées en pixels (voir Figure 3.11).

Figure 3.11

Mise en page par défaut d'une application Silverlight.



Si vous souhaitez centrer le `TextBlock` dans la page HTML, il vous faut donc faire en sorte que le `UserControl` racine de `MainControl1.xaml` s'adapte au navigateur continuellement. Pour cela il vous faut redéfinir la largeur et la hauteur du `UserControl`. Fermez votre navigateur et revenez sous Blend. Sélectionnez le `UserControl` dans l'arbre visuel, au-dessus du composant `Grid`, nommé `LayoutRoot` par défaut. Ensuite, définissez ses propriétés `Width` et `Height` sur `Auto` en cliquant sur le pictogramme situé à droite. La valeur `Auto` signifie que la largeur du composant va s'adapter à l'espace qui lui est permis dans la page HTML. Au sein de Blend, la largeur et la hauteur n'étant pas fixées en pixels, le composant s'adapte pour afficher au minimum le champ texte

contenu dans la grille LayoutRoot. Des icônes apparaissent autour du contour, elles indiquent que vous pouvez agrandir les dimensions du UserControl racine (voir Figure 3.12).

Figure 3.12

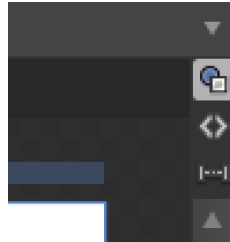
Mise à jour des propriétés du UserControl.



Les icônes les plus éloignées permettent d'étirer le projet pour simuler les dimensions fictives d'une page HTML. Vous pourrez par exemple obtenir un aperçu du redimensionnement de votre application dans une résolution plus faible. Pour gérer de manière précise les dimensions d'affichage fictives, vous devez passer en mode d'édition XAML. Pour cela, Blend propose un mode d'édition XAML. Cliquez sur l'icône représentant une balise en haut à droite de la fenêtre de design. C'est la deuxième icône en partant du haut (voir Figure 3.13). Elle permet d'accéder au mode XAML. La première icône, en haut, passera la vue en mode création. Vous pourrez également utiliser le mode mixte qui affiche à la fois le code XAML et le visuel en cliquant sur l'icône située en bas.

Figure 3.13

Bouton d'accès au mode d'édition XAML.



Une fois en mode XAML, modifiez les propriétés `d:DesignWidth` et `d:DesignHeight` du nœud UserControl racine en leur donnant respectivement les valeurs `800` et `600`. Vous pouvez voir un aperçu du code ci-dessous :

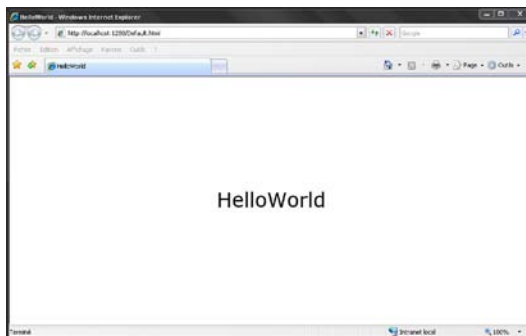
```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="HelloWorld.MainControl" Width="Auto" Height="Auto"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="600">
```

Votre application possède maintenant une résolution de 800 pixels de large et de 600 pixels de hauteur. Toutefois ces valeurs ne sont uniquement prises en compte et utiles que lorsque vous êtes sous Expression Blend. Elles sont pratiques pour avoir une prévisualisation de la mise en page dans cette résolution. Elles ne sont pas prises en compte lors de l'exécution de l'application dans une page HTML. Quoi qu'il en soit, l'instance de l'application Silverlight est dans une balise à 100 %, le UserControl occupera le maximum d'espace dans la page HTML car les propriétés

Width et Height sont en mode Auto et son alignement vertical et horizontal est en mode Stretch. Cette fois, le texte est correctement centré au sein du navigateur. De plus, lorsque vous redimensionnez la fenêtre du navigateur, le texte se repositionne dynamiquement (voir Figure 3.14).

Figure 3.14

Repositionnement centré du TextBlock.

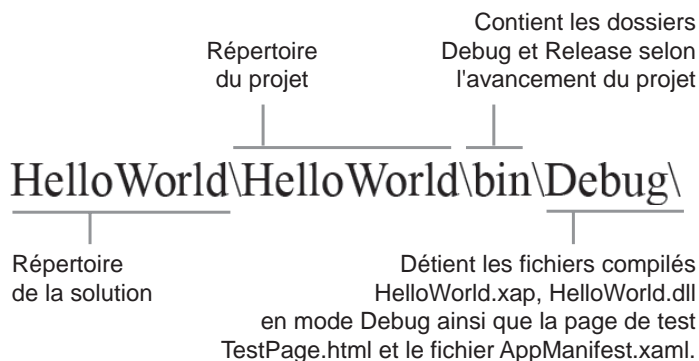


3.5.3 Fichiers générés

Une fois l'application compilée, votre navigateur se lance et affiche la page Silverlight. Le site s'affiche *via* le serveur web de développement lancé par Blend. C'est l'icône que vous pouvez apercevoir en bas à droite. Tous les fichiers relatifs à la création du site se situent dans le répertoire Debug (voir Figure 3.15).

Figure 3.15

Emplacement des fichiers générés.



Comme vous pouvez le voir, les fichiers produits à la compilation sont situés dans le répertoire Debug. Lorsque votre projet est en phase finale et prêt pour sa mise en ligne, vous pouvez, sous Visual Studio, définir l'application en mode Release. Cela permet de gagner en performances. Les bibliothèques nécessaires pour tracer des messages dans la fenêtre de sortie ou pour déboguer sont ignorées, le code est également beaucoup plus optimisé. De plus, c'est une bonne pratique, lorsque l'on arrive à une version stable, de compiler en mode Release, puis de la mettre en production. Dans tous les cas, vous trouverez au moins les fichiers suivants :

- `TestPage.html` est le fichier généré automatiquement lors de chaque compilation pour tester l'application Silverlight. Ce fichier propose une intégration minimale. Il contient simplement une balise object définissant l'instance du plug-in Silverlight dans la page web. Voici un exemple de la balise générée par défaut :


```

<div id="silverlightControlHost">
  <object data="data:application/x-silverlight-3,"
    type="application/x-silverlight-3" width="100%" height="100%">
    <param name="source" value="SL3WebSite.xap"/>
    <param name="onerror" value="onSilverlightError" />
    <param name="background" value="white" />
    <param name="minRuntimeVersion" value="3.0.40304.0" />
    <param name="autoUpgrade" value="true" />
    <a href="http://go.microsoft.com/fwlink/?LinkId=141205"
      style="text-decoration: none;">
      </a>
    </object>
  </div>

```

Vous pouvez remarquer que le premier paramètre indique le chemin d'accès vers le fichier xap à lire (voir HelloWorld.xap au point suivant). Le deuxième paramètre, `onerror`, redirige quant à lui toutes les erreurs à l'exécution ou lors de l'ouverture du fichier vers une fonction JavaScript qui gère leur affichage dans une balise DIV dédiée. Le paramètre `autoUpgrade` est par défaut à `true`. Cela signifie que celui-ci permet la mise à jour automatique de Silverlight lorsqu'une nouvelle version est disponible.

- `HelloWorld.xap`. C'est le format de fichier lisible par le lecteur Silverlight. Malgré les apparences, il n'est pas le produit direct de la compilation mais un fichier zip renommé avec l'extension xap. Il contient la dll de l'application ainsi qu'un manifeste décrivant tout ce qui est à l'intérieur. Dans notre cas, il possède `HelloWorld.dll`, le fichier produit par la compilation.
- `HelloWorld.dll`. C'est exactement le même fichier que celui situé dans le document xap. Il peut être référencé par n'importe quel autre projet Silverlight. Cela permet au projet qui le référence de pouvoir créer des instances de notre application. Vous pourriez ainsi créer un portfolio de tous les sites que vous avez réalisé en Silverlight, simplement en référençant leur dll et en les instanciant sous forme de miniature au sein d'Expression Blend.
- `AppManifest.xaml`. Le code montré ci-dessous expose le contenu de ce fichier :

```

<Deployment
  xmlns="http://schemas.microsoft.com/client/2007/deployment"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  EntryPointAssembly="HelloWorld"
  EntryPointType="HelloWorld.App"
  RuntimeVersion="3.0.40624.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="HelloWorld" Source="HelloWorld.dll" />
  </Deployment.Parts>
</Deployment>

```

Comme vous le constatez, il définit les informations indispensables permettant au lecteur Silverlight de lire le fichier xap. Le lecteur compare la version de Silverlight exposée par l'attribut `RuntimeVersion` à la valeur de la propriété `minRuntimeVersion` définie dans la balise `object` au sein du document `TestPage.html`. Si les versions définies dans chacun de ces fichiers ne correspondent pas, le lecteur peut lever une erreur.

3.5.4 Processus de compilation

3.5.4.1 Les classes partielles

Les classes partielles sont nées avec l'apparition de .Net 2. Elles avaient déjà pour but de séparer le fond et la forme au sein de fichiers distincts. Pour cela, le développeur pouvait glisser/déposer n'importe quel composant .Net au sein d'une fenêtre représentant l'interface visuelle. Le code logique était bien séparé du code décrivant l'interface visuelle mais ce dernier n'héritait pas du XML (contrairement à XAML). Lorsque le développeur compilait son application, celle-ci était donc générée à partir de deux fichiers. Ce mécanisme, permis grâce aux classes partielles, est exactement le même que celui utilisé aujourd'hui par Silverlight. Toutefois ce concept est bien plus efficace et pertinent avec cette technologie. Comme nous l'avons dit lors du précédent chapitre, une classe est un modèle d'objet. Une classe partielle est donc une partie d'un modèle d'objet contenue, la plupart du temps, dans un fichier séparé. Examinons le code XAML de notre projet HelloWorld :

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  x:Class="HelloWorld.MainPage"
  Width="Auto" Height="Auto" mc:Ignorable="d">...
</UserControl>
```

Vous remarquez que le UserControl principal contient la propriété x:Class qui a pour valeur HelloWorld.MainPage. Cela signifie que ce nœud est une définition de la classe MainPage héritant de la classe UserControl, et que cette définition est contenue dans l'espace de noms HelloWorld. Tout ce qui est déclaré dans le fichier XAML est donc un nouveau membre de cette classe. Le fichier MainPage.xaml.cs permet d'ajouter du code logique à l'application et aux objets déclarés dans MainPage.xaml. C'est lui qui permet les interactions utilisateur et la création d'algorithmes complexes. Voici un extrait de ce texte :

```
namespace HelloWorld
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            // Required to initialize variables
            InitializeComponent();
        }
    }
}
```

Cette classe se nomme MainPage. Elle hérite, elle aussi, de la classe UserPage et elle est contenue dans l'espace de nom HelloWorld. Vous remarquez l'utilisation du mot-clé partial. Ce mot signifie que la classe est partielle. Il s'agit donc de la même classe au sein de deux fichiers séparés. Chacun a pour but de définir une partie de celle-ci et non la totalité. C'est une approche très performante car XAML est bien plus pratique pour gérer l'arbre visuel et logique ainsi que les animations. C# sera, quant à lui, beaucoup plus performant pour développer l'aspect fonctionnel et les interactions complexes. Nous allons maintenant étudier les mécanismes du compilateur permettant de gérer deux langages aussi différents que C# et XAML.

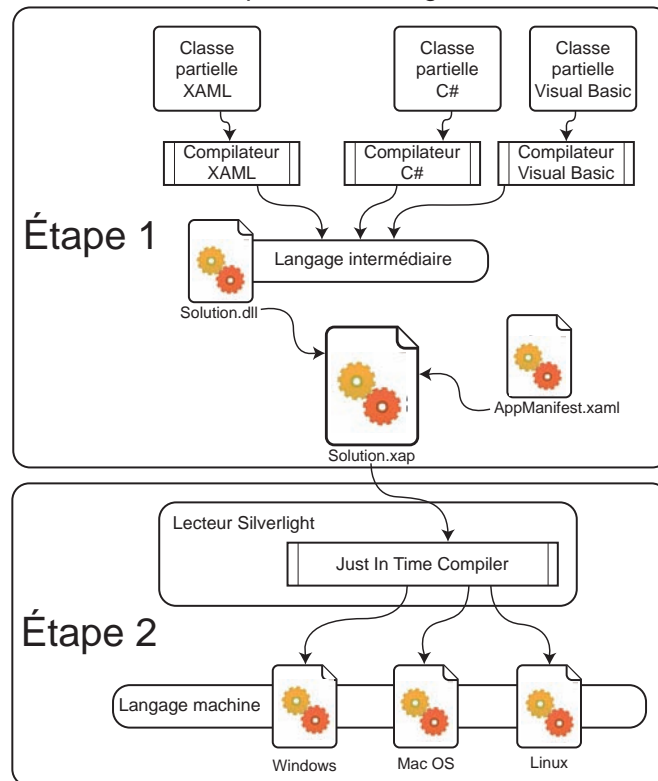
3.5.4.2 Just In Time Compiler

La compilation d'un projet Silverlight s'effectue toujours en deux étapes. La première consiste à transformer les langages de haut niveau gérés par la CLR comme C#, VB, XAML ou les langages gérés par la DLR, comme IronPython en langage intermédiaire. Quel que soit le langage d'origine, le langage intermédiaire résultant de la compilation est le même pour tous les langages gérés par .NET. La compilation est réalisée par le compilateur propre au langage dans lequel vous développez. Ce langage intermédiaire est contenu dans la dll qui est encapsulée au sein du fichier xap. Dans notre cas, elle se nomme "HelloWorld.dll" que nous avons évoquée plus haut. La deuxième étape est une compilation effectuée lors de la lecture du fichier xap par le plug-in Silverlight. L'implémentation du lecteur Silverlight diffère selon les systèmes d'exploitation et le navigateur utilisé. Cette compilation est donc à chaque fois différente mais l'objectif est le même : transformer le langage intermédiaire en langage machine. C'est le *Just In Time Compiler* qui gère cette étape. Son but est de compiler au dernier moment en langage machine afin d'obtenir les meilleures performances possibles selon le système. Comme il est différent et spécialisé selon les systèmes et les navigateurs, il produit du code machine plus adapté qu'un compilateur générique (voir Figure 3.16).

Figure 3.16

Compilation Silverlight en deux étapes.

Processus de Compilation Silverlight



Dans le prochain chapitre, nous plongerons dans l'interface d'Expression Blend pour créer un site Silverlight très simple et gérer le redimensionnement des éléments visuels.

Un site plein écran en 2 minutes

Au sein de ce chapitre, vous allez créer votre premier site Silverlight, puis vous identifierez les différences et avantages existants avec les projets de type application. Vous apprendrez à créer des boutons au sein de conteneurs spécifiques. Ceux-ci joueront le rôle de menus et rafraîchiront le contenu de la page principale. L'un d'eux permettra notamment l'affichage du site en mode plein écran. Le navigateur ne sera plus visible et l'application occupera tout l'espace sur l'écran lorsque ce mode sera activé. Vous verrez donc comment gérer le redimensionnement ou le repositionnement des objets dans la grille principale. L'interactivité utilisateur est abordée à travers trois méthodologies différentes : les comportements, l'utilisation du XAML ou la création de code logique C#. Pour finir, nous listerons les fichiers qui doivent être déployés sur le serveur.

4.1 Les projets de type site Internet

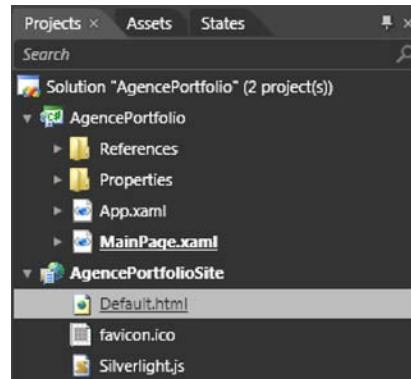
Les projets de type site permettent aux designers interactifs de se passer partiellement d'un développeur .Net ou d'un intégrateur HTML CSS. Ils fournissent par défaut les mécanismes d'intégration des applications Silverlight au sein d'une page web.

Créez un nouveau projet de type Silverlight Application + Web Site. Nommez-le AgencePortfolio. Vérifiez bien que le type de langage choisi est C# et que le répertoire contenant le projet s'appelle Pratique_de_Silverlight. Pour finir, cliquez sur OK. Votre panneau Projects affiche le visuel correspondant à la Figure 4.1 après un court instant.

Vous remarquez qu'il existe une nette différence avec les applications Silverlight 3 standard. La solution est constituée de deux projets au lieu d'un seul. Le premier projet est l'application Silverlight elle-même. Vous pouvez le constater car sa structure est la même que celle étudiée au Chapitre 3. Le deuxième projet représente le site web lui-même. Son nom est le même que celui de la solution, mais se termine par le suffixe Site. Trois fichiers sont visibles : `Default.html`, `favicon.ico` et `Silverlight.js`. Le premier représente la page HTML intégrant l'application Silverlight. Le deuxième correspond à l'icône par défaut du site dans la barre d'adresse. Le troisième fichier est un document JavaScript qui facilite et améliore l'intégration de Silverlight.

Figure 4.1

Arborescence d'un projet de type site Silverlight.



Dans le précédent type de projet, la page HTML se nommait `TestPage.html` au lieu de `Default.html`. Elle était générée par défaut lors de chaque compilation. Lorsque vous le modifiez directement dans le répertoire Debug, pour améliorer la mise en page ou l'intégration Silverlight, ces modifications étaient écrasées à chaque compilation. Elles n'étaient donc pas prises en considération. Cela n'était pas pratique et rendait obligatoire la sauvegarde de `TestPage.html` sous un nom différent. Vous n'avez pas ce type de problème avec notre projet, vous pourrez modifier à loisir `Default.html` car il n'est pas recréé à chaque compilation. Un serveur web est lancé avec sa racine pointant sur le dossier site. Le serveur web pointe directement sur le fichier quand le navigateur s'ouvre. Appuyez sur la touche F5 pour tester votre site Internet. Le navigateur affiche une page blanche car aucun élément visuel n'est contenu au sein de `LayoutRoot`. Au moment où vous testez votre site, le projet Application est compilé en premier, puis le projet `AgencePortfolioSite`, qui est défini en tant que projet de démarrage, est lancé. Le projet en gras au sein d'une solution est toujours le projet de démarrage. Ce projet est donc exécuté au sein du navigateur après la compilation réussie du projet Application.

Le deuxième projet vous sert donc de base pour produire une intégration HTML performante. En interne, son code est équivalent à celui utilisé pour lancer les applications classiques car il référence le fichier JavaScript `Silverlight.js` pour vérifier que le plug-in est déjà installé sur le poste client ou demander sa mise à jour. Le fichier `Silverlight.js` donne également accès aux propriétés de l'instance du plug-in représenté par la balise `Object`. Voilà les propriétés apportés par ce fichier :

- Il contient un mécanisme de détection du lecteur Silverlight, de sa version (si celui-ci est déjà présent) et propose l'installation du lecteur.
- Il facilite la conception d'une intégration avancée et permet l'utilisation de Javascript comme langage logique.
- Au même titre que la balise `Object`, il permet de gérer le préchargement de l'application Silverlight. Vous pourrez donc afficher un indicateur de préchargement pour les applications Silverlight complexes et celles dont le poids excède une certaine valeur. La décision concernant la valeur limite vous incombe et dépend du réseau sur lequel l'application est déployée ou le type de public ciblé.
- Il donne accès aux mêmes paramètres avancés que la balise `Object`. C'est le cas de la propriété `IsWindowLess`, équivalente à `windowless` pour la balise `Object` (permettant la création d'applications transparentes).

Lorsque vous avez compilé, un répertoire `ClientBin` a été automatiquement ajouté dans le projet Internet. Ce répertoire contient uniquement le fichier `xap`. Il est recopié par défaut à la fin de la compilation du projet de type application. Dans ce répertoire, vous ne trouverez aucun fichier `dll` ou propre au débogage car l'objectif est de rester le plus simple possible. Le dossier `ClientBin` est le dossier de publication. Les fichiers du dossier `bin\debug` sont empaquetés dans le fichier `xap` qui est déployé dans `ClientBin`. Vous obtenez ainsi le meilleur des deux mondes. Le projet Internet reste simple et épuré, mais le débogage et la réutilisation demeurent toutefois possibles.

4.2 Créer des conteneurs redimensionnables

Pour créer une interface redimensionnable, plusieurs possibilités s'offrent à vous. Vous pourriez, par exemple, positionner les menus directement à l'intérieur de la grille principale. Il est pourtant préférable de centraliser les objets ayant la même fonctionnalité au sein d'un conteneur dédié. Cela permet de les manipuler tous ensemble plus facilement, si besoin – il vaut mieux prévenir que guérir.

Nous allons créer deux types de conteneurs. Le premier, un `StackPanel`, contiendra les éléments du pied de page, essentiellement des champs texte cliquables. Le deuxième sera un conteneur de type `WrapPanel`. Il contiendra et agencera les menus présents dans le haut de notre page. Ceux-ci seront en réalité des composants de type `Button`. Notre site correspondra à la Figure 4.2. Cette figure est un croquis plus ou moins fidèle du résultat escompté. Ne vous formalisez donc pas sur les dimensions ou les rapports de proportions. Ce visuel nous sert avant tout de base de conception. Il a été réalisé avec *SketchFlow*, le nouvel outil de prototypage conçu par Microsoft, que étudierons au Chapitre 10.

Figure 4.2

Croquis du site plein écran.



4.2.1 Créer le pied de page

4.2.1.1 Créer et configurer le StackPanel

Vous allez commencer par les menus les plus simples : ceux du pied de page. Ils sont contenus dans un `StackPanel`. Ce type de conteneur permet d'empiler verticalement ou horizontalement un ensemble d'objets. Dans notre cas, il s'agira de champs texte de type `TextBlock`. Tout contrôle visuel est cliquable au sein des projets Silverlight ou WPF. Le `TextBlock` n'échappe pas à cette

4.2.1.2 Ajouter des champs texte et notion de contexte conteneur

Lorsque vous avez double-cliqué sur l'icône du `StackPanel`, vous l'avez directement imbriqué dans la grille `LayoutRoot`. Ce n'est pas un hasard. Vous avez pu réaliser cette opération car cette grille était sélectionnée comme contexte conteneur.

Le contexte conteneur est toujours entouré d'un liseré bleu ou jaune. Cela signifie que toutes les actions du graphiste, la création de composants, leur sélection ou encore leur modification, seront réalisées dans le conteneur sélectionné comme contexte actif. Pour créer une série de menus à l'intérieur du `StackPanel`, il vous faut donc cliquer sur celui-ci dans la vue de design pour le définir en tant que contexte actif. Ensuite, vous pouvez double-cliquer sur l'icône `TextBlock` dans la barre d'outils. Lors de chaque double-clic, vous imbriquez une nouvelle occurrence de `TextBlock` au sein du `StackPanel`. Répétez cette opération trois fois. Une fois les trois champs texte créés, définissez des marges à droite de 20 pixels pour les deux premiers. Vous pouvez sélectionner les deux éléments et spécifier leurs marges en même temps dans le panneau des propriétés. Ensuite, changez la valeur contenue dans la propriété `Text`, qui est accessible *via* le panneau des propriétés ou directement par double-clic sur le composant `TextBlock` dans la fenêtre de design (voir Figure 4.4). Vous pouvez également choisir une police afin d'affiner le visuel. Je vous conseille la police Trebuchet MS dans un premier temps car cela évite d'avoir à l'embarquer puisqu'elle est intégrée par défaut à Silverlight.

Figure 4.4

Menu bas du site
représenté par un
`StackPanel` contenant
plusieurs contrôles
`TextBlock`.



Vous remarquez que le `StackPanel` s'agrandit vers la gauche automatiquement à chaque nouvel élément visuel ajouté en son sein.

4.2.2 Créer le menu du haut

Nous allons maintenant créer le menu du haut. Le composant bouton (`Button`) propose des interactions visuelles avantageuses informant l'utilisateur que ce menu est réactif. Par exemple, le menu pourrait changer de couleur lorsqu'il est survolé.

4.2.2.1 Principe du `WrapPanel` et du mode Auto

Comme la navigation de notre site repose essentiellement sur le menu du haut, nous allons imbriquer des boutons au sein d'un conteneur. Afin de gérer les redimensionnements extrêmes de notre site, nous allons utiliser un composant `WrapPanel`. Celui-ci est proche du `StackPanel` dans son principe, c'est-à-dire qu'il empile les éléments les uns après les autres. Toutefois, il met à la ligne un élément visuel si jamais celui-ci tend à dépasser le bord droit ou le bord bas du conteneur selon son orientation (voir Figure 4.5).

Couplé au mode de redimensionnement Auto, propre aux propriétés largeur et hauteur, nous pourrions ajuster les dimensions dynamiquement à chaque remise à la ligne. Par exemple, si nous définissons la hauteur (`Height`) du `WrapPanel` sur Auto et son orientation à horizontale et un aligne-

ment vertical haut (éviter le mode Stretch dans ce cas, ainsi que les marges), alors le `WrapPanel` s'ajustera en hauteur pour chaque nouvelle ligne d'éléments visuels. Vous pouvez voir l'exemple précédent mis à jour en utilisant le mode Auto (voir Figure 4.6).

Figure 4.5

Principe d'imbrication au sein d'un `WrapPanel`.

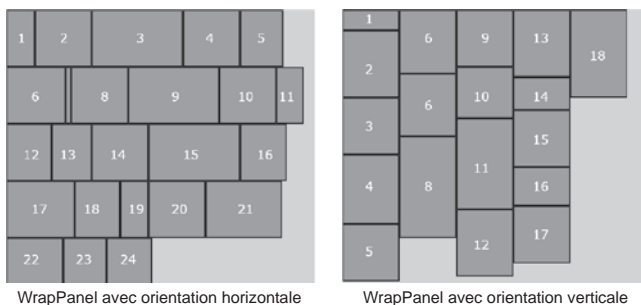
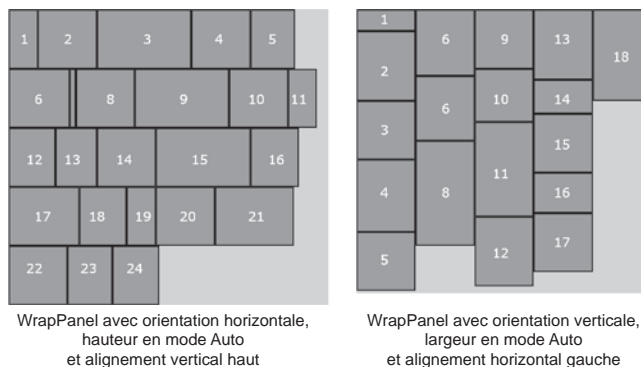


Figure 4.6

Principe d'imbrication au sein d'un `WrapPanel` avec la valeur Auto activée.



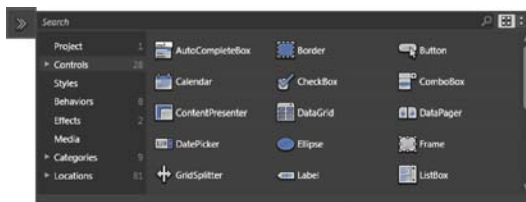
Comme vous pouvez le constater, sur le premier schéma de la Figure 4.6, les rectangles (22, 23, 24) ne sont plus coupés, la hauteur est réajustée en fonction du nombre de lignes. Sur le second schéma, il ne reste plus aucun espace à droite de l'élément 18 car la largeur du `WrapPanel` s'ajuste au contenu.

4.2.2.2 Créer et configurer le `WrapPanel`

Pour créer un conteneur de ce type, vous devez utiliser la bibliothèque de composants car il ne se trouve pas dans la liste des conteneurs proposés par défaut. Cliquez sur la dernière icône de la barre d'outils, celle ressemblant à une double flèche (»). Elle vous permet d'ouvrir la bibliothèque de composants. Sélectionnez le mode d'affichage avec grandes icônes pour identifier les contrôles disponibles (voir Figure.4.7).

Figure 4.7

Bibliothèque de composants Silverlight.



Le champ de recherche sert de filtre, entrez-y les lettres wr. Le contrôle de type `WrapPanel` n'est pas visible dans la liste. C'est en fait très logique, il n'est pas instanciable par défaut au sein des projets Silverlight. Pour y remédier, il vous suffit de télécharger et d'installer la bibliothèque nommée *Silverlight Toolkit*. Elle est maintenue par l'équipe Silverlight de Microsoft et regroupe de nombreux avantages dont certains contrôles qui ne sont par défaut accessibles qu'au sein des projets WPF. Afin d'assurer un maximum de fonctionnalités au sein de Silverlight, ces composants ont été redéveloppés et mis à disposition dans le Silverlight Toolkit. Vous le trouverez à l'adresse : <http://www.codeplex.com/Silverlight/>. Cliquez sur le menu Download (voir Figure 4.8).

Figure 4.8

Le portail CodePlex hébergeant le projet Silverlight Toolkit.



Une fois la bibliothèque installée, sauvegardez votre projet et relancez Blend. Cette fois-ci, le composant `WrapPanel` est affiché. Cela prend parfois quelques secondes car Blend doit explorer les nouvelles bibliothèques de contrôles installées. Sélectionnez le contrôle `WrapPanel`, son icône apparaît juste en dessous de celle permettant l'accès à la bibliothèque. Double-cliquez dessus, une instance de type `WrapPanel` est automatiquement créée au sein du contexte conteneur sélectionné. Dans notre cas, cela doit être `LayoutRoot`.

INFO

Vous pouvez également accéder à la liste des contrôles via le panneau Assets situé en haut à gauche. Ce panneau est parfois plus pratique d'utilisation, cela dépendra essentiellement de la manière dont vous utilisez Blend.

Nous allons l'ancrer sur les bordures droite, gauche et haute de la grille principale. Pour cela, vous allez définir une largeur et une hauteur en mode Auto, comme nous avons fait pour le menu du bas avec le composant `StackPanel`. Ensuite, cliquez sur l'icône d'alignement vers le haut et spécifiez une marge de 10 pixels par rapport au bord haut. Cliquez sur l'icône étirement horizontal pour activer ce mode. Définissez des marges à 30 pixels pour la bordure gauche et à 90 pixels pour la bordure droite. Cliquez sur l'icône à droite des propriétés Width et Height. Le `WrapPanel` n'a plus de hauteur. Il s'est adapté à son contenu qui est vide. Il possède donc une hauteur égale à 0 pixel. Au contraire, comme nous avons défini des marges à gauche et à droite ainsi qu'un alignement horizontal étiré, il possède une largeur égale à la largeur totale de l'application moins les marges (voir Figures 4.9 et 4.10).

Figure 4.9

Configuration des options de mise en forme du WrapPanel.

**Figure 4.10**

Visuel du composant WrapPanel Configuré.



Comme vous pouvez le voir, le WrapPanel est ancré en haut, à gauche et à droite. Les icônes représentant un verrouillage indiquent l'ancrage des objets ou non sur les bords du conteneur Grid principal. Elles sont accompagnées de nombres indiquant la valeur en pixel de chaque marge. Elles sont cliquables et permettent de modifier les options d'alignement horizontal et vertical.

4.2.2.3 Ajouter et configurer les boutons du menu

Vous allez maintenant ajouter des boutons correspondant à chacun des menus de la Figure 4.2. Sélectionnez le WrapPanel, puis double-cliquez cinq fois sur l'icône représentant un bouton au sein de la barre d'outils. Vous venez à l'instant d'imbriquer cinq boutons dans le conteneur WrapPanel. Vous auriez pu les dessiner directement au sein de ce conteneur. Cette manière de faire présente l'avantage d'éviter de préciser, pour chaque bouton, les dimensions en hauteur et largeur. Avec cette méthode, celles-ci ont été définies en mode Auto pour chacun d'eux. Les valeurs des propriétés Width et Height s'ajustent en fonction de leur contenu textuel, qui est par défaut la chaîne de caractère Button. Dès cet instant, le WrapPanel possède une hauteur qui s'ajuste à chaque bouton. Si vous définissez une police différente ou une taille différente, le bouton réajustera sa hauteur qui réajustera celle du WrapPanel. Si vous avez défini des marges verticales sur l'un des boutons, celles-ci affecteront directement la hauteur finale du WrapPanel.

Sélectionnez chaque bouton en maintenant la touche Maj enfoncée. Ensuite, au sein du panneau des propriétés, dans l'onglet des propriétés communes, spécifiez la police Trebuchet MS, ainsi qu'une hauteur de caractères de 14 pixels.

INFO

Le corps des polices est exprimé par défaut en pixels et non en points. Il est toutefois possibles de le modifier. Pour cela, ouvrez le menu Tools > Options..., choisissez ensuite l'onglet Units. Dans la liste sélectionnez Points au lieu de Pixels.

Désélectionnez tous les boutons en cliquant n'importe où sur la scène, puis cliquez sur le premier bouton et entrez la chaîne de caractères Nouveautés dans la propriété Content. Procédez de

même avec tous les autres afin d'avoir un menu correspondant à la Figure 4.2. Ensuite, créez un dernier bouton à l'extérieur du WrapPanel au sein du conteneur LayoutRoot. Il doit être ancré en haut à droite et posséder des marges de 10 pixels en haut et de 8 à droite. Dans sa propriété Content, insérez la chaîne de caractères Plein écran. Ce bouton ne fait pas partie directement du WrapPanel car il ne donne pas accès au même niveau d'utilisation. Il nous permettra de passer alternativement du mode plein écran au mode d'affichage normal. Vous devriez maintenant avoir un visuel similaire à la Figure 4.11.

Figure 4.11

Menu du haut avec un bouton Plein écran.



Pour finir, sélectionnez le UserControl racine et définissez une largeur et une hauteur en mode Auto, cela vous permettra de profiter du navigateur à 100 %. Étirez ensuite la vue de design avec les manipulateurs pour simuler une largeur de 640 pixels et une hauteur de 320 pixels. Voici le code XAML généré en arrière-plan par Blend, pour la totalité de la grille principale :

```
<Grid x:Name="LayoutRoot" Background="#FFFFFF" Margin="0,0,0,0" >
    <controlsToolkit:WrapPanel Height="Auto" HorizontalAlignment="Stretch"
        VerticalAlignment="Top" Width="Auto" Margin="30,10,90,0"
        d:LayoutOverrides="Width">
        <Button Height="Auto" Width="Auto" Content="Nouveautés"
            Margin="0,0,20,0" FontSize="14" FontFamily="Trebuchet MS"
            Visibility="Visible"/>
        <Button Height="Auto" Width="Auto" Content="Portfolio"
            Margin="0,0,20,0" FontSize="14" FontFamily="Trebuchet MS"
            Visibility="Visible"/>
        <Button Height="Auto" Width="Auto" Content="Médias"
            Margin="0,0,20,0" FontSize="14" FontFamily="Trebuchet MS"
            Visibility="Visible"/>
        <Button Height="Auto" Width="Auto" Content="Savoir faire"
            VerticalAlignment="Center" Margin="0,0,20,0" FontSize="14"
            FontFamily="Trebuchet MS" Visibility="Visible"/>
        <Button Height="Auto" Width="Auto" Content="Contact"
            Margin="0,0,0,0" FontSize="14" FontFamily="Trebuchet MS"
            Visibility="Visible"/>
    </controlsToolkit:WrapPanel>
    <StackPanel Height="Auto" HorizontalAlignment="Right" Margin="0,0,30,8"
        VerticalAlignment="Bottom" Orientation="Horizontal">
        <TextBlock Text="A propos " TextWrapping="Wrap" Margin="0,0,30,0"
            FontFamily="Trebuchet MS" Foreground="#FF6C6C6C"
            FontWeight="Bold" FontSize="12"
            TextDecorations="Underline"/>
        <TextBlock Text="Qui sommes nous ? " TextWrapping="Wrap"
            Margin="0,0,30,0" FontFamily="Trebuchet MS"
            Foreground="#FF6C6C6C" FontWeight="Bold" FontSize="12"/>
        <TextBlock Text="Newsletter" TextWrapping="Wrap"
            FontFamily="Trebuchet MS" Foreground="#FF6C6C6C"
            FontWeight="Bold" FontSize="12"/>
    </StackPanel>
    <Button Height="45" Width="60" Content="Plein écran" Margin="0,10,8,0"
        HorizontalAlignment="Right" VerticalAlignment="Top" FontSize="10"
        Visibility="Visible"/>
    <Grid Margin="30,70,30,50" Background="#FF8C8C8C"
        Visibility="Collapsed">
        <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
            Text="Contenu des Pages" TextWrapping="Wrap" FontSize="36"
            FontFamily="./segoepr.ttf#Segoe Print" Foreground="#FFE2E0E0"
```

```

        FontWeight="Bold" />
    </Grid>
</Grid>

```

En très peu de lignes, nous sommes parvenus à créer une interface redimensionnable. Vous remarquerez que la balise `WrapPanel` est préfixée de `controlsToolkit`. Cela signifie qu'il n'est pas référencé dans l'assembly *System.Windows.dll* par défaut. En réalité, lorsque vous avez ajouté ce composant dans l'application, Blend a automatiquement référencé deux bibliothèques issues du Silverlight ToolKit : *System.Windows.Controls.dll* et *System.Windows.Controls.Toolkit.dll*. Pour le vérifier, il suffit d'ouvrir votre panneau projet et de déplier le répertoire *Reference*. Blend a également fait en XAML l'équivalent d'un `using` C# pour référencer l'espace de noms `controlsToolkit`. Voici le code ajouté dans le `UserControl` racine :

```

xmlns:controlsToolkit="clr-namespace:System.Windows.Controls;
assembly=System.Windows.Controls.Toolkit"

```

Lorsque vous concevrez et partagerez vos composants personnalisés, les personnes désirant les utiliser agiront de la même manière. Elles devront donc ajouter la bibliothèque contenant vos composants en tant que nouvelle référence. Toutefois Blend fait ce travail pour vous dans la plupart des situations. Pour l'instant, ce visuel est un peu brut, mais nous apprendrons rapidement à créer des composants et des styles personnalisés.

4.2.3 Créer la grille centrale

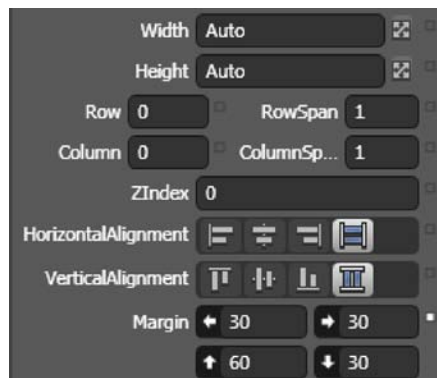
Vous allez créer un conteneur `Grid` situé au centre de la grille principale. Il contiendra le contenu des pages de notre site plein écran. Sélectionnez le conteneur `LayoutRoot` pour le définir comme contexte conteneur, puis dessinez une grille au centre.

4.2.3.1 Gérer le redimensionnement

Affectez à cette grille une hauteur et une largeur automatique avec un alignement étiré (*Stretch*). Pour éviter qu'elle ne recouvre les autres objets, elle doit posséder des marges d'environ 30 pixels à gauche, 60 pixels en haut, 30 pixels à droite et 30 pixels en bas (voir Figure 4.12).

Figure 4.12

Paramétrage de l'alignement de la grille centrale.



Pour finir, créez un composant `TextBlock` centré et sans marges au sein de la grille avec dimensions en mode *Auto*.

4.2.3.2 Modifier la couleur d'arrière-plan

Afin de rendre la grille plus visible lors de la compilation, définissez une couleur d'arrière-plan gris clair. Pour cela sélectionnez, en haut du panneau des propriétés, l'attribut Background présent dans l'inspecteur de couleurs. Ensuite, cliquez sur la deuxième icône en partant de la gauche. Elle permet de spécifier une couleur unie (voir Figure 4.13).

Comme vous pouvez le remarquer, le code couleur hexadécimal est codé sur 4 octets soit #FFFFFF.

Figure 4.13

Paramétrage de la couleur d'arrière-plan de la grille centrale.



INFO

Un octet représente 8 bits, c'est pour cela que l'on parle d'images 32 bits (4x8). Un bit possède une valeur 0 ou 1. Une valeur 8 bits représente donc une combinaison de 8 chiffres possibles de 0 ou 1, soit 2^8 possibilités. En hexadécimal, cela se traduit par FF, ce qui donne en tout 256 possibilités, de 0 à 255. Dans la plupart des logiciels de graphisme, la couche de transparence n'est pas directement affichée dans le code hexadécimal mais elle est plutôt affichée sur une échelle qui va de 0 à 100 % d'opacité. Blender propose les deux affichages, ce qui évite les ambiguïtés. La couleur #00FF0000 ne sera donc pas visible car le premier couple de 0 indique qu'il n'y a aucune opacité.

4.3 Le composant bouton

Étudier les propriétés du composant bouton permet de comprendre de nombreux principes propres au framework Silverlight car celles-ci sont communes à de nombreux autres composants.

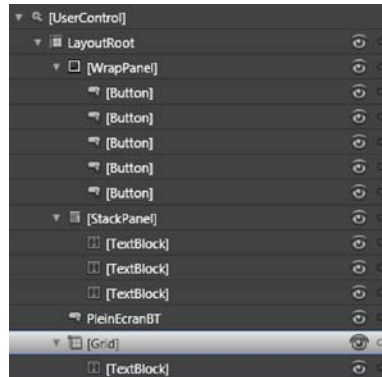
4.3.1 Définir un nom d'exemplaire

Donner un nom d'exemplaire aux objets permet d'y accéder facilement depuis le code logique. Une fois nommés, les objets deviennent des champs de la classe associée au UserControl. Ces champs sont définis par défaut avec le modificateur d'accès `internal`. Ainsi vous pourrez atteindre leurs méthodes, cibler leurs propriétés dynamiquement ou encore écouter les événements qu'ils diffusent (comme le clic de la souris). Sélectionnez le bouton plein écran, celui-ci ne possède pour l'instant pas de nom d'instance. Pour savoir si un composant possède un nom d'exemplaire, il suffit de regarder dans l'arbre visuel. Si le composant est décrit par son type entre crochet,

cela signifie qu'il n'est pas nommé. Vous pouvez également regarder dans le panneau des propriétés si son attribut Name, situé tout en haut, est défini. Définir un nom d'instance est assez simple, vous pouvez soit double-cliquer sur l'occurrence directement dans l'arbre visuel, soit insérer une chaîne de caractères dans sa propriété Name. Définissez `PleinEcranBT` comme nom d'instance. Par la suite, nous définirons un comportement propre à ce bouton pour passer en mode plein écran (voir Figure 4.14).

Figure 4.14

Différence d'affichage entre objets nommés et anonymes.



4.3.2 Afficher une bulle d'information au survol

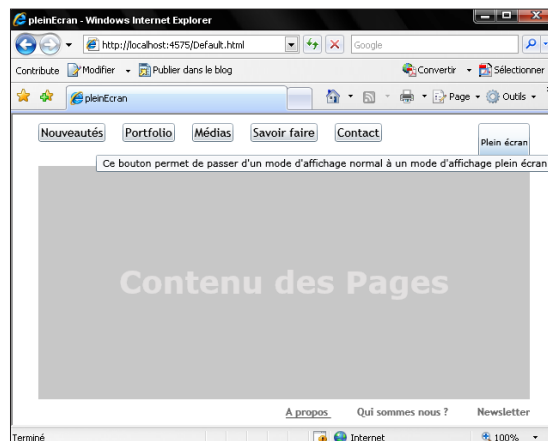
On peut assigner un `ToolTip` à tout élément visuel. Cette propriété est dite attachée, elle n'est pas définie sur l'objet, elle est fournie par la classe statique `ToolTipService`. Pour mieux comprendre ce principe, insérez la phrase suivante dans l'attribut `ToolTip` : "Ce bouton permet de passer d'un mode d'affichage normal à un mode d'affichage plein écran". Ouvrez le mode d'édition XAML. Voici ce que vous pouvez voir dans la balise `Button` :

```
ToolTipService.ToolTip="Ce bouton permet de passer d'un mode d'affichage
normal à un mode d'affichage plein écran"
```

Compilez votre application en appuyant sur le raccourci F5 (voir Figure 4.15).

Figure 4.15

Affichage de l'info bulle au survol du bouton.



Comme vous pouvez le constater, la propriété `ToolTip` est fournie par `ToolTipService` :

```
ToolTipService.ToolTip="..."
```

Pour finir, si vous souhaitez aligner le bouton `Newsletter` et le bouton plein écran verticalement à gauche (voir Figure 4.15), il suffit de définir une marge de 30 pixels à droite pour le bouton `PleinEcranBT`.

4.3.3 Choisir un curseur de survol personnalisé

Tout élément visuel possède par défaut la propriété `Cursor`. Celle-ci permet de définir un curseur système lors du survol d'un objet. Selon le système d'exploitation sur lequel vous serez, vous aurez donc des curseurs avec des visuels différents. Cela peut vous permettre, par exemple, d'indiquer à l'utilisateur un chargement de données en cours de progression. Il vous suffirait de spécifier un curseur d'attente (`Wait`) pour une liste de données. Lors de la réception des données, vous pourriez ensuite redéfinir un curseur standard. Cette propriété prend comme valeur une constante de la classe `Cursors` (au pluriel).

4.3.3.1 Définir un curseur en C#

Si vous n'êtes pas un familier de .Net, cette méthode est intéressante car elle permet de redéfinir à l'exécution le curseur d'un objet. La classe statique `Cursors` liste tous les curseurs disponibles lors du survol d'un objet graphique. Double-cliquez sur `MainPage.xaml.cs` et insérez le code en gras suivant :

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
        PleinEcranBT.Cursor = Cursors.Hand;
    }
}
```

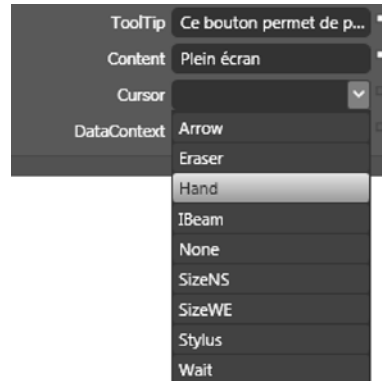
Assigner une valeur dans le XAML est équivalent à affecter une valeur dans le constructeur. Pour rappel, le constructeur d'une classe représente sa méthode d'initialisation, celle qui se lance à l'instanciation de l'objet. L'intérêt d'affecter les valeurs dans le code C# est de coupler l'assignation de ces valeurs à une logique métier.

4.3.3.2 Définir un curseur avec Expression Blend

Expression Blend permet de simplifier le processus pour accomplir cette opération. Il vous suffit d'ouvrir le panneau des propriétés propre au bouton plein écran, puis de définir un curseur de type `Hand` (voir Figure 4.16).

Figure 4.16

Affectation d'un
curseur de survol.



4.3.4 La propriété *Content*

La propriété *Content* est assez difficile à cerner au premier coup d'œil. Jusqu'à maintenant, nous l'avons utilisée pour afficher un texte au sein des boutons, ce qui constitue son comportement par défaut. Nous pourrions donc nous demander pourquoi cette propriété ne se nomme pas *Label* ou *Title*. Dans les faits, elle est capable de stocker beaucoup plus qu'une simple chaîne de caractères. Elle est typée *Object*, ce type représente la classe située au niveau le plus haut de l'arbre d'héritage. Des classes aussi différentes que *String*, *Panel* ou *ButtonBase* héritent par conséquent de la classe *Object*. La propriété *Content* permet donc d'afficher n'importe quel objet visuel ou non. Si l'objet à afficher n'est pas de type visuel (*UIElement*), il est converti en chaîne de caractères de type *string*, et cette dernière est affichée. Toutefois, la propriété *Content* ne peut contenir qu'un seul enfant. Tous les objets héritant de la classe abstraite *ContentControl* héritent par défaut de la propriété *Content*. Nous y reviendrons au Chapitre 5 dédié à l'arbre visuel et logique.

INFO

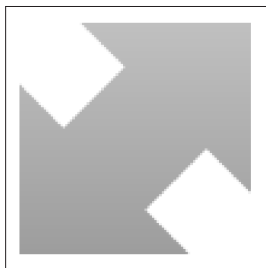
Une classe abstraite est une classe dont on ne peut pas créer d'instances directement. Ces classes servent de modèle et contiennent du code qui sera, soit surchargé par les classes qui en héritent, soit directement réutilisé. *Panel*, *ButtonBase*, *RangeBase* sont des classes abstraites qui sont héritées par de nombreuses classes – classes dont on peut créer des instances. Par exemple, *Slider* ou *ProgressBar* sont des classes que l'on peut instancier et qui héritent des méthodes et des propriétés de la classe abstraite *RangeBase*. En programmation orientée objet, une classe abstraite est l'un des moyens mis en œuvre en vue de la réutilisation. Toutefois, l'héritage multiple n'étant pas permis (contrairement à C++), ce n'est pas toujours le moyen le plus souple.

4.3.4.1 Créer l'icône de redimensionnement

Nous allons créer une icône de redimensionnement que nous placerons au sein du bouton plein écran. Cette icône est très simple à réaliser, et elle va nous permettre d'apprendre les opérations de base que Blend propose pour les tracés vectoriels (voir Figure 4.17).

Figure 4.17

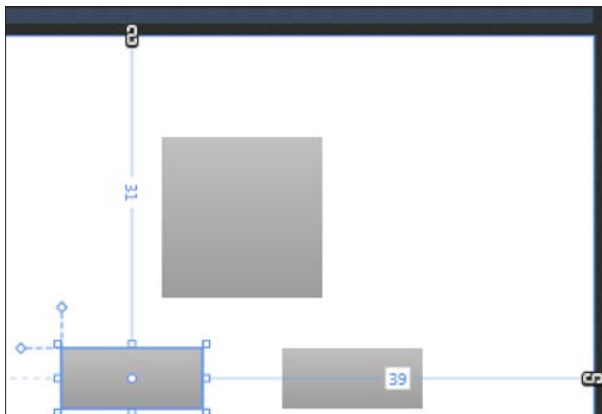
Icône de redimensionnement.



Dans un premier temps, faites un carré de 16 pixels de côté *via* la primitive Rectangle. Ensuite, légèrement en dessous, créez deux rectangles allongés de 6 pixels de hauteur par 14 de longueur. Alignez-les horizontalement côte à côte en gardant un espace entre eux de 8 pixels (voir Figure 4.18).

Figure 4.18

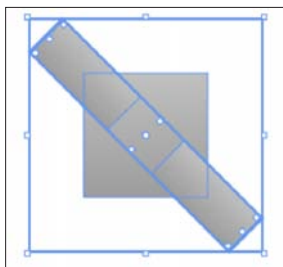
Création de l'icône, étape 1.



Une fois les rectangles réalisés, sélectionnez-les, puis ouvrez le menu Object et, au sein de l'onglet Combine, cliquez sur l'option Unite. Vous obtenez une primitive de type Path. Les deux rectangles ont donc été convertis en tracé vectoriel. Dans l'arbre visuel, en lieu et place des deux rectangles, vous remarquerez qu'est apparu un objet [Path]. Sélectionnez le carré, puis le nouveau tracé, et cliquez-droit dessus. Grâce au menu Align, centrez les deux objets horizontalement et verticalement. Ensuite, au sein du panneau des propriétés, dans l'onglet des transformations RenderTransform, cliquez sur la deuxième icône en partant de la gauche. Vous ouvrez ainsi le panneau qui gère la rotation des objets. L'onglet RenderTransform est commun à tous les objets graphiques. Entrez la valeur de 45 degrés dans le champ Rotation (voir Figure 4.19).

Figure 4.19

Création de l'icône, étape 2.



Pour finir, désélectionnez tout, et dans l'ordre, cliquez sur le tracé (les deux anciens rectangles), puis sur le carré, en maintenant la touche Maj enfoncée. Ensuite, ouvrez le menu Object, puis Combine et sélectionnez l'option Subtract. Le carré s'est vu retirer tout l'espace occupé par le tracé vectoriel (voir Figure 4.16). Il ne nous reste plus qu'à insérer cette image dans notre bouton plein écran en l'affectant à sa propriété Content. Cela est assez simple à faire : glissez l'icône au-dessus du bouton. Dès que le message vous indiquant la possibilité d'imbriquer l'élément apparaît, cliquez sur la touche Alt. Relâchez le bouton de la souris. C'est fait, l'icône est intégrée au bouton plein écran. Afin d'obtenir un visuel élégant, vous pouvez redimensionner le bouton à 18 pixels de hauteur et de largeur, puis sélectionner le tracé à l'intérieur et lui affecter des dimensions en mode Auto. Cette dernière modification lui permet de s'adapter à son conteneur (le bouton). Pour ma part, j'ai également défini des marges extérieures (Margin) en haut et à droite de 8 pixels sur le bouton (voir Figure 4.20).

Figure 4.20

Application finale sans interactivité.



Finalement, on considère les objets bénéficiant de la propriété Content comme des conteneurs à enfant unique. Au sein de la plateforme Silverlight, de nombreux composants utilisateur en bénéficient. Les primitives échappent assez logiquement à cette particularité. Le site, sans interaction logique, est disponible en ouvrant *pleinEcran_maquette.zip* dans le *chap4* des exemples du livre.

4.3.4.2 Affecter la propriété Content en C#

Vous l'aurez compris, au sein de Silverlight presque tous les objets sont imbriquables les uns dans les autres. Mais comment faire en C# pour affecter la propriété Content avec un élément existant ou instancié dynamiquement ?

Pour répondre à cette question, nous allons affecter un composant Image à la propriété Content du bouton Contact. Ce type de composant permet d'afficher des images bitmap. Pour cela, nommez le bouton ContactBtn, puis téléchargez dans votre répertoire projet l'image *IcôneContact.png*, présente dans le dossier *chap4* des fichiers d'exemples. Ensuite, au sein de Blend, cliquez-droit sur le projet et sélectionnez l'option Add Existing Item... Dans la fenêtre d'exploration, sélectionnez l'image téléchargée et cliquez sur OK. Cette opération vous a permis d'ajouter l'image en tant que ressource. Double-cliquez sur *MainPage.xaml.cs*. Voici le code C# commenté permettant d'affecter l'image à la propriété Content :

```
public MainPage()
{
    InitializeComponent();
    PleinEcranBT.Cursor = Cursors.Hand;
```

```

        //on écoute l'événement Loaded diffusé lors du premier affichage
        //du UserControl soit this
        Loaded +=new RoutedEventHandler(MainPage_Loaded);
    }

    private void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        // 1 - ici on crée un composant Image
        Image myImage = new Image();
        // 2 - puis on définit son mode de redimensionnement
        myImage.Stretch = Stretch.None;
        // 3 - on crée ensuite un objet Uri qui pointe vers
        // notre fichier image à utiliser
        Uri adresseImage = new Uri("IcôneContact.png",UriKind.Relative);
        // 4 - on crée un objet de type ImageBitmap en lui spécifiant
        // l'adresse relative que nous venons de définir
        BitmapImage bi = new BitmapImage( adresseImage );
        //5 - on précise au composant la source de l'image qu'il doit afficher
        myImage.Source = bi;
        //6 - pour finir, on ajoute le composant Image à la
        // propriété Content de notre bouton Contact
        ContactBtn.Content = myImage;
    }

```

Vous remarquez que l'image n'a pas été copiée dans le répertoire `ClientBin`. En réalité, elle a été compilée au sein du fichier `xap`, plus exactement dans le fichier `dll` contenu dans le `xap`. Il faudra toujours utiliser un objet de type `BitmapImage` que l'on affectera ensuite à la propriété `Source` du composant `Image`. Finalement, la partie la plus simple est d'attribuer le composant `Image` à la propriété `Content` de notre bouton. Pour affecter un composant plus complexe, tel que `Grid`, à cette propriété, nous aurions simplement pu écrire `ContactBtn.Content = new Grid();`. Cette grille aurait pu, à son tour, contenir des objets visuels, il n'y a pas limite. Le code n'est pas entièrement expliqué car nous ferons référence au mode redimensionnement ainsi qu'au chargement d'image ultérieurement.

4.4 Ajouter de l'interactivité

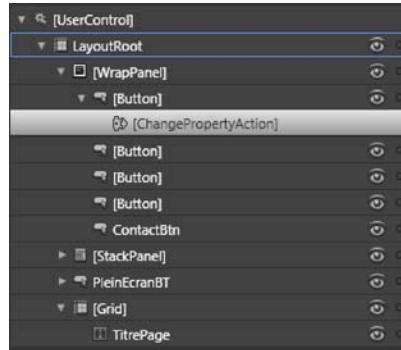
Nous allons, dans un premier temps, utiliser les comportements. Depuis sa version 3, Blend permet de coder dans les langages `C#` et `Visual Basic`. Comme `Visual Studio`, il propose une aide au code, `IntelliSense`, pour ces deux langages ainsi que pour le code déclaratif `XAML`. Nous utiliserons le `C#` pour passer du mode plein écran au mode d'affichage normal.

4.4.1 Utiliser les comportements

Nous allons nous contenter de simuler le changement de page. À chaque fois que nous cliquerons sur un bouton, le champ texte situé au milieu de la page sera mis à jour. Nommez le champ texte contenu dans la grille au centre de notre site, `TitrePage`. Ensuite, ouvrez la bibliothèque, puis l'onglet `Behaviors`. Celui-ci contient une liste de comportements. Glissez, puis déposez le comportement `[ChangePropertyAction]` sur le premier bouton du menu. Le comportement apparaît dès lors dans l'arbre visuel (voir Figure 4.21).

Figure 4.21

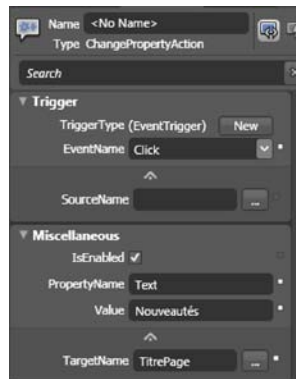
Ajout d'un comportement au premier menu.



Nous allons maintenant configurer ce comportement pour qu'il change le champ texte `TitrePage` lors d'un clic sur le bouton. Pour cela, après avoir sélectionné le comportement dans l'arbre visuel, ouvrez le panneau des propriétés. À chaque fois que l'utilisateur cliquera sur le premier menu, nous affecterons dynamiquement la propriété `Text` du champ `TitrePage` avec la valeur `Nouveautés` (voir Figure 4.22).

Figure 4.22

Configuration du comportement.



L'événement `Click` représente l'interaction utilisateur qui déclenchera le changement de valeur. Le paramètre `SourceName` est optionnel, il indique quel objet est à la source de l'événement diffusé. Comme le comportement est créé sur le bouton, c'est lui qui, par défaut, diffusera l'événement `Click`. `PropertyName` permet de spécifier la propriété affectée lors du `Click`. Pour modifier le contenu textuel d'un composant `TextBlock`, il faut affecter la propriété `Text`. Le champ `Value` représente la valeur de cette propriété. `TargetName` est l'objet ciblé, dans notre cas c'est le champ texte `TitrePage`. L'icône avec les trois points, située à droite du champ de saisie (☰), vous permet de naviguer dans l'arbre visuel et logique afin de rechercher l'objet cible. Vous devez maintenant répéter cette opération pour les autres menus. Pour accomplir cette tâche rapidement, vous pouvez copier-coller ce comportement sur les autres menus, puis changer le paramètre `Value` pour chaque menu.

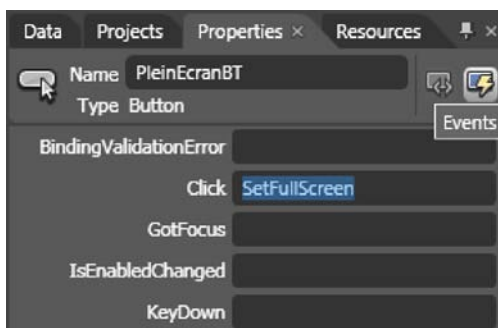
4.4.2 Mode plein écran

Le mode d’affichage plein écran est très pratique pour certains types d’applications, par exemple pour afficher de la vidéo ou un contenu nécessitant une grande résolution. Cela permet à l’utilisateur de s’immerger complètement dans l’application en faisant abstraction des menus propres au navigateur. Pour des raisons de sécurité, certaines possibilités du lecteur Silverlight sont désactivées dans ce mode. Pour passer d’un affichage normal au mode plein écran, une seule ligne de code C# et un paramétrage XAML suffisent.

La première chose à faire est de définir une méthode C# qui se déclenchera lorsque l’événement Click est diffusé par le bouton plein écran. Pour cela, sélectionnez ce bouton, puis ouvrez le panneau des propriétés. Cliquez sur l’icône en forme d’éclair qui se trouve en haut à droite de ce panneau. Vous venez d’ouvrir la liste des événements disponibles pour le bouton plein écran. Pour l’événement Click, définissez la méthode SetFullScreen (voir Figure 4.23).

Figure 4.23

Le panneau Events.



Voici le code du bouton généré en XAML :

```
<Button x:Name="PleinEcranBT" Height="18" Width="18" Margin="0,8,8,0"
HorizontalAlignment="Right" VerticalAlignment="Top" FontSize="10"
Visibility="Visible" ToolTipService.ToolTip="Ce bouton permet de
passer d'un mode d'affichage normal à un mode d'affichage plein écran"
BorderThickness="1,1,1,1" BorderBrush="#FFACACAC" Click="SetFullScreen" >
  <Path Stretch="Fill" Stroke="{x:Null}" Height="Auto" Width="Auto"
    UseLayoutRounding="False" Data="M4.242609,0 L16,0 L16,11.757387
    L12.94972,8.7071075 L8.7070818,12.949746 L11.757336,16 L0,16
    L0,4.2426682 L3.0502257,7.2928939 L7.2928643,3.0502553 z">
    <Path.Fill>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FFC1C1C1" Offset="0"/>
        <GradientStop Color="#FF9E9E9E" Offset="1"/>
      </LinearGradientBrush>
    </Path.Fill>
  </Path>
</Button>
```

XAML permet de déclarer des méthodes d’écoute sur les événements diffusés. Ce n’est pas toujours l’idéal car C# offre plus de souplesse de ce point de vue. Toutefois, dans une certaine mesure, cela donne aux designers interactifs une certaine autonomie. En phase finale de développement, c’est plutôt au développeur de gérer la diffusion et l’écoute des événements. Les performances de l’application sont en partie liées à la notion de programmation événementielle (voir Chapitre 8). Lorsque vous avez défini la fonction d’écoute, Blend a automatiquement créé une méthode corres-

pondante dans le fichier `MainPage.xaml.cs` et a ouvert ce fichier. Supprimez la ligne commentée et remplacez-la par la ligne en gras du code suivant :

```
private void SetFullScreen(object sender, RoutedEventArgs e)
{
App.Current.Host.Content.IsFullScreen =
!App.Current.Host.Content.IsFullScreen;
}
```

Le code précédent est déclenché lorsque l'utilisateur clique sur le bouton. La ligne C# en gras est assez simple, la propriété booléenne `IsFullScreen` est affectée de l'opposé de sa valeur actuelle. Par exemple, si celle-ci est `false` (donc si l'application n'est pas en mode plein écran), alors elle passe en mode plein écran. Si elle est `true`, c'est-à-dire en mode plein écran, alors on repasse en mode normal. Cette syntaxe nous a évité d'utiliser une condition `if`, ainsi qu'un code fastidieux. Testez votre site, redimensionnez la fenêtre du navigateur et cliquez sur les menus. Le positionnement et l'affichage des menus s'adaptent automatiquement, même au sein d'une fenêtre de 640 par 480 pixels. Le `WrapPanel` gère le retour à la ligne des menus si besoin. Pour finir, cliquez sur le bouton plein écran plusieurs fois pour vérifier que le code logique C# correspond au besoin.

4.4.3 Spécificités du mode plein écran

Pour des raisons de sécurité, dès lors que vous êtes passés en mode plein écran, un message vous l'indiquant est apparu durant quelques secondes (voir Figure 4.24).

Figure 4.24

Message d'indication plein écran.



Ce n'est pas la seule contrainte de sécurité imposée par le mode plein écran. Ce mode ne peut pas être défini par d'autres événements que ceux découlant des interactions utilisateur. Ainsi, vous ne pouvez pas forcer le mode plein écran dès le démarrage de l'application par exemple. Dans ce cas, affecter la propriété `IsFullScreen` ne provoque aucun changement d'affichage. Le mode plein écran limite également l'utilisation du clavier. Cela évite à l'utilisateur de saisir des données confidentielles. Ce mode d'affichage peut en effet simuler n'importe quel type d'interface. Voici la liste des touches auxquelles vous pouvez accéder :

- le pavé des flèches directionnelles (Up, Down, Left et Right) ;
- la barre d'espace (Spacebar) ;

- la touche Tabulation (Tab) ;
- les touches Page Précédente et Suivante (Page Up, Page Down)
- la touche Maison et Fin (Home, End) ;
- la touche Entrée (Enter).

Plusieurs instances du plug-in Silverlight ne peuvent pas être en même temps en mode plein écran. En réalité, lorsqu'une application Silverlight plein écran perd le focus, celle-ci revient à un mode d'affichage normal. Par exemple, le simple fait, sur Windows de basculer, *via* la combinaison de touches Alt+Tab, d'une application à une autre fait perdre le focus à l'application Silverlight. Deux applications Silverlight ne peuvent donc pas être en mode plein écran en même temps car l'une d'entre elles a forcément perdu le focus utilisateur.

ATTENTION

La dernière particularité du mode d'affichage plein écran est d'empêcher l'affichage d'applications transparentes (*via* la propriété `windowless`). L'arrière-plan de l'application sera alors complètement opaque. Lorsque l'utilisateur quittera ce mode, l'application pourra à nouveau bénéficier de la transparence.

4.4.4 Détecter le changement d'affichage

Il peut être très utile d'écouter l'événement `FullScreenChanged` de l'instance du plug-in. Cet événement est déclenché lors du changement de mode. Le code suivant permet d'afficher la résolution de l'écran utilisateur dans le champ `TitrePage` à chaque changement de mode :

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ...
    //cette ligne permet d'écouter l'événement FullScreenChanged
    App.Current.Host.Content.FullScreenChanged += new System.
        EventHandler(Content_FullScreenChanged);
}
private void Content_FullScreenChanged(object sender, EventArgs e)
{
    double l = App.Current.Host.Content.ActualWidth;
    double h = App.Current.Host.Content.ActualHeight;
    TitrePage.Text = "largeur :: " + l + " - hauteur :: " + h;
}
```

Comme vous le constatez, les dimensions de l'application ne sont pas récupérées grâce aux propriétés `Width` et `Height` car celles-ci ont été définies en mode `Auto`. Par contre, elles peuvent l'être *via* les propriétés `ActualWidth` et `ActualHeight`. Ces propriétés contiennent les valeurs absolues en pixels du conteneur racine `UserControl`. Le Chapitre 8 fournit plus d'information sur l'écoute d'événements.

INFO

Un autre événement existe, `Resized`, mais il n'est pas diffusé lors du changement de mode d'affichage. Il est diffusé lors que l'utilisateur redimensionne l'instance du plug-in Silverlight. Il permet, par exemple, de repositionner des objets contenus de manière animée en fonction des nouvelles dimensions du plug-in.

4.5 Fichiers déployés

Les fichiers à déployer sont tous contenus dans le projet `AgencePortfolioSite`. Pour déployer votre application, il vous suffit de copier le contenu de ce répertoire sur votre serveur web *via* une connexion FTP. Au sein de votre répertoire distant, vous devrez donc avoir les fichiers :

- `Default.html`, la page HTML qui contient l'application Silverlight.
- `Silverlight.js`, fichier JavaScript qui facilite et améliore l'intégration de Silverlight.
- `ClientBin/AgencePortfolio.xap`, le fichier Silverlight compilé.

Le site finalisé est disponible dans l'archive *pleinEcran_interactif.zip* du *chap4* des exemples.

Au prochain chapitre, nous étudierons les mécanismes liés à l'arbre visuel et logique, ainsi que les méthodes permettant de manipuler la liste d'affichage. Comme nous le verrons, Blend propose de nombreux outils permettant aux designers de faire des applications contextuelles capable d'adapter l'affichage aux besoins et contraintes des utilisateurs.

L'arbre visuel et logique

Jusqu'à maintenant, nous avons construit des interfaces sans réellement nous préoccuper de l'architecture ou de l'imbrication des composants. Dans ce chapitre, nous listerons les familles de contrôles graphiques, de la simple forme primitive jusqu'au composant de données élaboré, en affinant notre analyse au fur et à mesure de notre avancée. Nous aurons ainsi une idée plus précise de leur rôle et de leurs capacités. Puis, nous utiliserons le projet précédent, `SiteAgencePortfolio`, pour apprendre à y manipuler, ajouter et modifier les enfants d'un composant conteneur. Nous aborderons les méthodes et les moyens techniques offerts par Silverlight pour concevoir l'imbrication, l'architecture et le design visuel de manière complémentaire. À cette fin, nous étudierons chaque cas aussi bien du point de vue d'un designer interactif que de celui d'un développeur C#.

5.1 Composants visuels

Au sein de Silverlight, tous les objets visuels sont des composants et font donc référence à une classe. La moindre primitive, qui semble pourtant anodine au premier abord, est un composant aux comportements étendus. Connaître les familles de composants est une chose naturelle pour un développeur car cette notion est directement liée à l'architecture de l'application qu'il développe. Pour un designer, cette démarche est moins intuitive car il est moins soumis à la technique qu'au résultat visuel final. Toutefois, un designer interactif doit se faire force de proposition sur l'expérience utilisateur et sur un ensemble de notions directement liées à l'architecture de l'application. Il est également très souvent conduit à modifier des composants existants pour arriver à ses fins. Il lui est donc nécessaire de connaître au moins partiellement les familles et types de contrôles visuels dont il peut profiter. Tous les composants que nous allons maintenant étudier peuvent et feront sans doute partie de l'arbre visuel d'une de vos applications Silverlight.

5.1.1 Familles de composants

De manière générale, on classe les composants par genre. Voici une classification possible des familles du point de vue d'un graphiste ou d'un ergonome. Toutefois, nous verrons que ces catégories ne sont pas si évidentes.

- **Les primitives vectorielles.** Les formes de type `Rectangle`, `Path`, `Line` ou `Ellipse` sont considérées comme des composants car elles font bien plus qu'afficher une forme ou un tracé. Elles sont interactives et possèdent des propriétés complexes que nous allons étudier.
- **Les conteneurs.** Il s'agit de composants à enfants uniques ou multiples. Ils sont à la base de tout visuel. Par exemple, le composant racine `UserControl` est considéré comme le conteneur principal de l'application. Nous verrons qu'il n'est pas forcément facile de les répertorier car de nombreux composants inattendus sont des conteneurs à enfants uniques.
- **Les contrôles de gestion et d'affichage de texte.** Ils vous permettront d'afficher du texte de différentes manières.
- **Les composants de liste de données.** Cette catégorie regroupe les listes (`ListBox`), les listes déroulantes (`ComboBox`) et les grilles de données (`DataGrid`). Le composant `AutoCompleteBox` en fait également partie, il est à mi-chemin entre le champ de saisie et la liste déroulante.
- **Les gestionnaires de médias.** Silverlight est une plateforme plurimédias avant tout : elle peut diffuser du son, des images et de la vidéo dans divers formats, grâce à trois composants essentiels : `Image` et `MediaElement` pour le son et la vidéo et `MultiScaleImage` basé sur la technologie `DeepZoom`.
- **Les composants d'interfaces utilisateur.** C'est la catégorie fourre-tout dans laquelle vous trouverez les boutons (`Button`), les barres de défilement (`ScrollBar`), les cases à cocher (`CheckBox`) ou encore les barres de progression (`ProgressBar`)... Pour résumer, c'est tout ce qui n'est pas défini dans les catégories citées précédemment.

Répertorier n'est toutefois pas si simple. Pour le démontrer, nous allons essayer de catégoriser certains des composants que nous avons utilisés dans le projet `SiteAgencePortfolio`. Prenons, par exemple, le cas de la grille (`Grid`). Elle peut-être classée dans le genre conteneur. Cela est d'autant plus visible qu'elle peut contenir plus d'un objet enfant. Le `WrapPanel` ou le `StackPanel` sont eux aussi dans cette catégorie. La seule différence entre ces contrôles concerne les contraintes d'agencement subies par leurs objets enfants. Le bouton semble être facile à classer lui aussi car c'est avant tout un objet assurant la gestion de comportement utilisateur. Il se retrouve donc dans la dernière catégorie des composants d'interface utilisateur. Toutefois, les boutons, comme de nombreux autres composants, possèdent la faculté de contenir un unique objet enfant. Deux des boutons que nous avons instanciés sur la scène ont été utilisés de cette manière et possèdent un enfant. C'est grâce à la propriété `Content` que nous avons pu imbriquer un enfant en leur sein. Cela fait-il d'eux des conteneurs ? Dans une certaine mesure, oui. Grâce à diverses techniques de programmation orientée objet, dont la composition et l'héritage, ces boutons possèdent naturellement cette capacité. Toutefois, leur rôle premier n'est pas d'agencer des objets. Ils n'ont pas la vocation des conteneurs spécialisés. De la même manière, le pied de page de notre site est constitué de menus représentés par des `TextBlock`. Comme tous les objets visuels, ces `TextBlock` sont cliquables et rien ne nous empêche de les utiliser comme des boutons sans interactions visuelles. De plus, chaque `TextBlock` contient du texte, soit *via* sa propriété `Text`, soit en imbriquant des balises de type `Run`. L'utilité principale du `TextBlock` est d'afficher du texte, nous le classons donc dans la catégorie des gestionnaires de texte. La nature même du XAML génère ce type d'ambiguïté de par les relations familiales propres aux arborescences XML, mais facilite la construction d'interfaces riches et non rigides. Dans la section suivante de ce chapitre, nous allons essayer de comprendre pourquoi cette classification est simpliste en étudiant l'arbre d'héritage des objets d'affichage.

5.1.2 Arbre d'héritage des classes graphiques

5.1.2.1 De Object à FrameworkElement

La notion d'héritage remonte aux débuts de la programmation orientée objet. Lorsqu'une classe hérite d'une autre classe, elle récupère les méthodes, les champs et les propriétés de celle-ci. Ainsi, le code créé pour la classe mère, et échu aux classes filles, n'est écrit qu'une seule fois par le développeur. L'héritage fait partie des nombreuses techniques de réutilisation de code existantes en POO. Comme de nombreuses autres plateformes, Silverlight est basé sur ce concept et les ingénieurs l'ayant conçu ont privilégié cette méthodologie. En tant que designer interactif ou développeur, la connaissance de la bibliothèque vous permettra de concevoir vos propres composants de manière optimisée et perspicace en héritant de la classe adéquate, qui n'aura ni trop ni pas assez de fonctionnalités. La classe mère de toutes les autres est `Object`. Elle possède plusieurs méthodes dont toutes les autres classes héritent et qu'elles utilisent. En voici une liste non exhaustive :

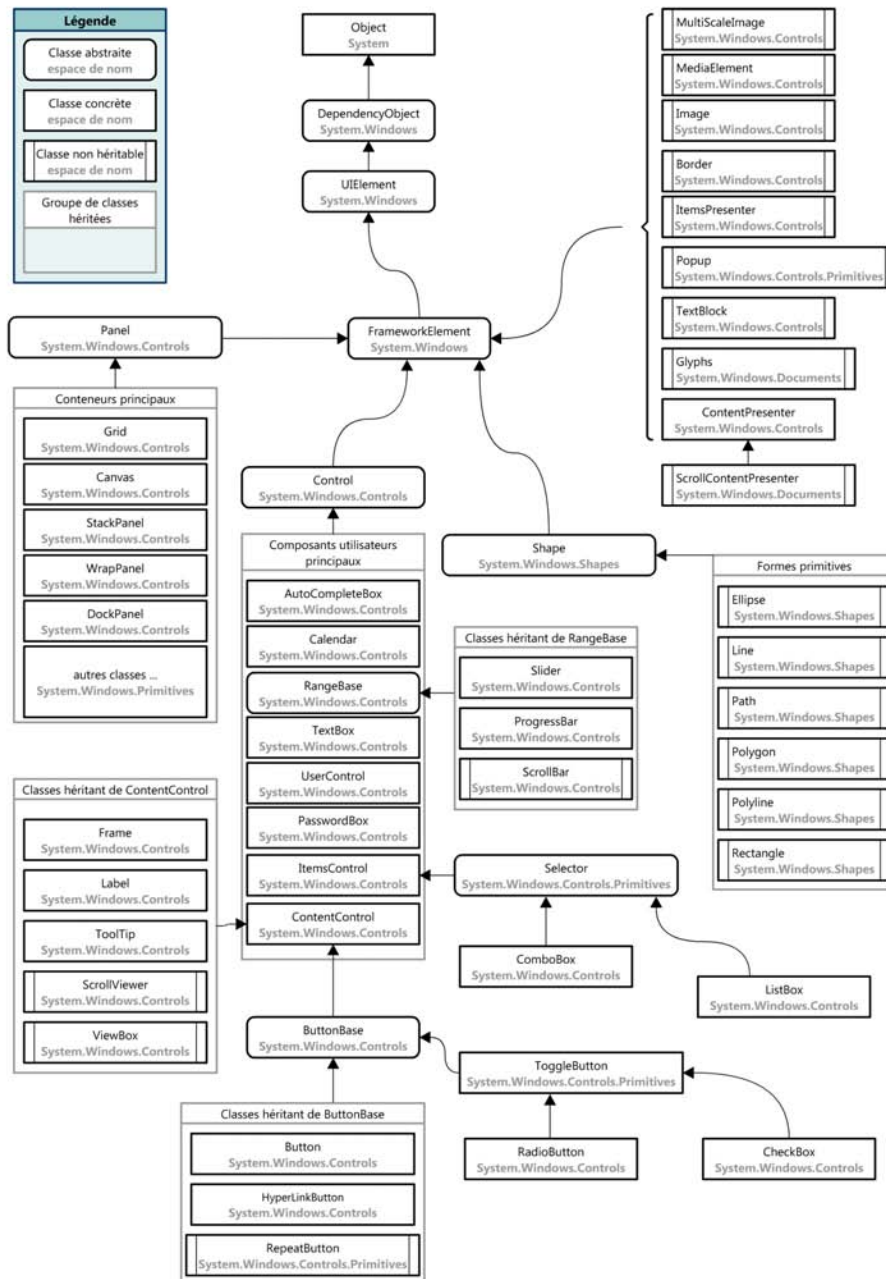
- `Equals`. Cette méthode permet de comparer les valeurs contenues par deux références d'objet.
- `GetType`. Renvoie le type de l'instance qui l'invoque, ce qui est très pratique pour étudier et énumérer à l'avance un type inconnu.
- `MemberwiseClone`. Renvoie une nouvelle référence clonée à partir de l'instance qui l'invoque.
- `ReferenceEquals`. Permet de comparer deux références pour savoir si elles possèdent la même allocation mémoire.
- `ToString`. Lors de son appel, elle renvoie la valeur de l'instance sous forme de chaîne de caractères. Il peut être pratique d'outrepasser la méthode fournie par défaut pour générer une représentation personnalisée si celle implémentée ou héritée par défaut ne correspond pas à votre besoin.

Nous allons maintenant détailler les classes graphiques qui hérite de `Object`. Dans le schéma de la Figure 5.1, en dessous de `Object`, vous trouvez `DependencyObject`. Cette classe est abstraite, cela signifie qu'elle ne peut pas être directement instanciée. Elle fournit les mécanismes de base propres aux objets graphiques de la plateforme Silverlight, dont le principe de `DependencyProperty`, aux classes en héritant. L'animation, l'imbrication et la notion de liaison (*Binding*) prennent appui sur ce principe. C'est là le cœur de la puissance et de la souplesse de Silverlight. Héritant de `DependencyObject`, la classe abstraite `UIElement` définit les mécanismes d'affichage propres aux objets graphiques. Elle possède les méthodes et propriétés permettant de calculer la taille et le positionnement de son instance dans l'espace, elle pose donc les bases du système d'agencement graphique. C'est dans cette classe que les propriétés d'opacité, de visibilité et de masque de découpe sont, par exemple, pourvues. Elle définit également les événements propres aux interactions utilisateurs : `MouseEnter`, `MouseLeave`, `MouseLeftButtonUp`, `KeyUp`, etc. La classe abstraite `FrameworkElement` hérite à son tour de `UIElement` et y ajoute le système de disposition visuel de Silverlight : le SLS pour *Silverlight Layout System*. Le SLS – ou système d'agencement de Silverlight – repose sur les méthodes `ArrangeOverride`, `MeasureOverride`, sur les propriétés `Width`, `Height`, `MinWidth`, `MinHeight`, `MaxWidth`, `MaxHeight`, `ActualWidth`, `ActualHeight` et, enfin, sur l'événement `SizeChanged`. La classe `FrameworkElement` définit également `Loaded`, diffusé lorsqu'une nouvelle instance est initialisée au sein de l'arbre visuel et logique de l'application. Pour finir, bien que les bases de la liaison de données soient définies en majeure partie dans la classe `DependencyObject`, c'est la classe `FrameworkElement` qui résout l'accès aux membres de classe en fournissant la propriété `DataContext` nécessaire à la liaison de données. Finalement,

FrameworkElement donne naissance à la majeure partie des classes concrètes donc des composants visuels instanciables dans Silverlight.

Figure 5.1

Arbre d'héritage des classes d'objets graphiques.



5.1.2.2 Formes primitives

Commençons par les primitives. Elles héritent toutes de la classe abstraite `Shape`, qui fournit les propriétés nécessaires pour affecter le remplissage du fond et le contour d'une forme primitive grâce aux propriétés `Stroke` et `Fill`. `Shape` permet également de personnaliser la forme du contour, ses extrémités et ses sommets. Toutes les classes héritant de `Shape` sont verrouillées, il est donc impossible de les étendre *via* l'héritage. La classe `Path` est particulière car le tracé vectoriel est défini par la propriété `Data`. Toutefois, les propriétés `Width` et `Height` sont tout de même utilisées. Il faudra modifier la propriété `Stretch` de l'objet `Path` pour déterminer de quelle manière le tracé remplit les dimensions imposées en largeur et en hauteur. Par défaut, la propriété `Stretch` a pour valeur `Fill`. Cela signifie que le tracé s'étirera pour remplir complètement les dimensions du composant `Path` en largeur et en hauteur.

5.1.2.3 Conteneurs primitifs

Cette catégorie concerne une dizaine de classes qui, pour la majorité, sont fermées à l'extension et héritent en droite ligne de `FrameworkElement`. Elles ont pour but d'afficher ou de mettre en valeur un contenu simple et spécifique. Ainsi, vous pouvez y trouver les composants suivants :

- `TextBlock`. Affiche du texte.
- `Glyph`. Composant de gestion de texte bas niveau. Il permet d'afficher et de mettre en forme n'importe quel caractère *via* ses propriétés `UnicodeString` et `Indices`. Il offre plus d'options et plus de capacités que `TextBlock`, mais il est plus complexe à mettre en œuvre.
- `Image`. Afficher et charger des images aux formats JPEG et PNG. Le format PNG 32 bits est supporté, ce qui vous permettra d'afficher des icônes transparentes par exemple.
- `MediaElement`. Contrôle qui permet de lire de la vidéo et du son aux formats WMV, WMA, MP3, ainsi que les fichiers encodés en H.264, ce qui étend ses capacités au format MP4 depuis la version 3 de Silverlight. Le logiciel Expression Encoder possède la capacité d'encoder la vidéo dans tous ces formats.
- `MultiScaleImage`. Communément appelé composant `DeepZoom`, il prend en paramètre un ensemble d'images qu'il chargera par la suite dynamiquement afin de proposer une expérience utilisateur spécifique. Pour vous familiariser avec ce comportement utilisateur, vous pouvez vous rendre sur le site <http://www.laguna-coupe.com/>.
- `Popup` et `Border`. Conteneurs à enfant unique. Ils possèdent tous les deux la propriété `Child` qui leur permet de contenir n'importe quel `UIElement`. `Border` peut être considéré comme un rectangle amélioré car il possède des propriétés de remplissage et d'affichage avantageuses. `Popup` permet, quant à lui, d'afficher une information au-dessus des autres composants constituant l'arbre visuel et logique. Il ne s'agit pourtant pas d'une fenêtre modale.
- `ItemsPresenter`. Ce composant est assez particulier. Il est intégré au sein des composants de type `ItemsControl` et des classes en héritant. Il permet d'établir la position et l'espace alloués aux objets (`ListBoxItem`) d'une `ListBox` par exemple. Il est donc étroitement lié à la propriété `ItemsSource` d'une `ListBox`.
- `ContentPresenter`. Gère le comportement de glisser-déposer et l'imbrication d'enfants uniques propres aux composants de type `ContentControl`. Il fait donc partie intégrante de leur structure.

5.1.2.4 Composants personnalisables

Parmi les contrôles que nous avons évoqués, seuls quelques-uns implémentent la capacité d'être personnalisable tant au niveau de leur style que dans leur forme visuelle. Si l'on prend le cas du composant `Image`, celui-ci ne peut être directement paramétré pour posséder une bordure ou un fond. Pour cela, vous devrez utiliser la technique de composition propre à la programmation orientée objet. Vous devrez donc créer un contrôle, qui s'appellera par exemple `Visionneuse` et qui en interne possèdera un `Border` ainsi qu'un composant `Image`. Si vous souhaitez que votre contrôle `Visionneuse` soit paramétrable par un designer interactif, vous devrez le faire hériter de la classe `Control`. Celle-ci hérite de `FrameworkElement` et apporte, en plus, tous les mécanismes inhérents à la personnalisation du visuel d'un composant. Tous les contrôles que nous allons à présent évoquer héritent de cette classe et sont donc personnalisables par un graphiste, un designer ou un intégrateur. Nous ne traiterons pas de tous les composants car ils sont nombreux. En voici une liste non exhaustive :

- `AutoCompleteBox` est un composant à mi-chemin entre un `TextBox` et une liste déroulante (`ComboBox`). De ce fait, il hérite directement de `Control` et non de la classe `ItemsControls` ou `Selector`. Il s'agit en fait d'un champ de saisie qui propose une liste de choix au fur et à mesure de la saisie utilisateur.
- `RangeBase` est une classe abstraite apportant les mécanismes, méthodes et propriétés de base propres aux objets de défilement ou de progression tels que `Slider` ou `ProgressBar`. Si vous souhaitez faire un `Slider` à double curseur, vous devrez étendre cette classe.
- `ItemsControl` constitue le fondement des composants exposant une liste d'objets. Ceux-ci sont, la plupart du temps, reliés à un contexte de données ou directement fournis par le développeur à partir d'un fichier XML, d'une base de données ou d'un service web.
- `TextBox` et `PasswordBox` sont deux composants permettant la saisie de données utilisateur. Alors que `TextBlock` n'est représenté visuellement que par le texte qu'il affiche, ceux-ci possèdent en leur sein une bordure extérieure ainsi qu'un fond et de nombreuses propriétés leur permettant d'être personnalisés.
- `ContentControl` est une classe abstraite apportant avec elle la capacité de contenir n'importe quel objet enfant. Cette capacité fait que, par défaut, tous les objets en héritant possèdent en interne un composant de type `ContentPresenter` et une propriété `Content`. Cette dernière est typée `Object`. Lorsque vous glissez un tracé vectoriel dans un bouton, vous affectez sa propriété `Content`, elle-même liée au `ContentPresenter` situé au sein du bouton. Cette capacité en fait une classe très puissante et très souple d'utilisation. De très nombreux composants héritent de cette classe du fait de cette capacité. Il arrivera sans doute que vous ayez besoin d'étendre cette classe pour vos propres besoins.
- `ButtonBase` hérite de `ContentControl` et apporte une gestion simplifiée et directe du clic gauche de la souris. Les classes en découlant, comme `Button`, `RadioButton` ou `CheckBox`, constituent la majeure partie des contrôles utilisateur au sein d'une interface. Il est à noter qu'il n'y a pas de différence fonctionnelle entre un `ToggleButton` et une `CheckBox`. La seule différence entre ces deux composants est purement visuelle.
- `ToolTip` est un peu spécifique. Cette classe permet d'afficher une bulle d'information lorsque la souris survole un objet graphique. Toutefois, il est impossible d'instancier cette classe directement, vous devrez passer par la classe statique `ToolTipService` et ses méthodes `SetTool-`

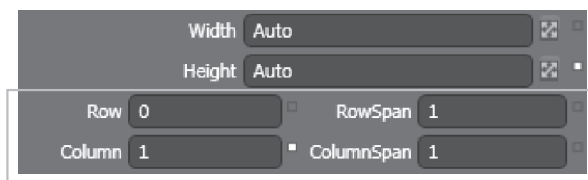
Tip et GetToolTip. ToolTip hérite de ContentControl, elle peut donc afficher n'importe quel enfant de type UIElement ou héritant de UIElement. Pour ajouter une bulle d'information à un objet graphique, il vous suffira d'affecter la propriété ToolTip (du même nom que la classe, attention donc aux confusions) de cet objet. Cette propriété est attribuée par défaut aux objets de l'arbre visuel par la classe ToolTipService. ToolTip est une propriété dite attachée, elle n'est pas héritée mais attribuée dynamiquement. Nous reviendrons sur cette notion tout au long de ce chapitre.

5.1.2.5 Conteneurs de mise en forme

Comme nous l'avons vu lors des chapitres précédents, les conteneurs à enfants multiples facilitent la mise en page et la création d'applications redimensionnables. Ce chapitre leur est dédié en grande partie, car ils constituent le fondement de toute application Silverlight. Ces conteneurs particuliers héritent de la classe abstraite Panel. De nombreuses notions et propriétés propres à ces composants sont basées sur les mêmes concepts existant au sein du langage HTML. Lorsque vous imbriquez un contrôle au sein d'un conteneur, l'objet imbriqué reçoit automatiquement des propriétés dites attachées. Par exemple, lorsque nous avons utilisé la grille d'alignement au Chapitre 2, les objets imbriqués en son sein ont automatiquement reçu des propriétés attribuées par la grille (voir Figure 5.2).

Figure 5.2

Propriétés héritées dynamiquement d'un conteneur Grid.



Exemples de propriétés attribuées dynamiquement à un objet par son conteneur de type Grid

Voici la liste des conteneurs de type Panel les plus courants :

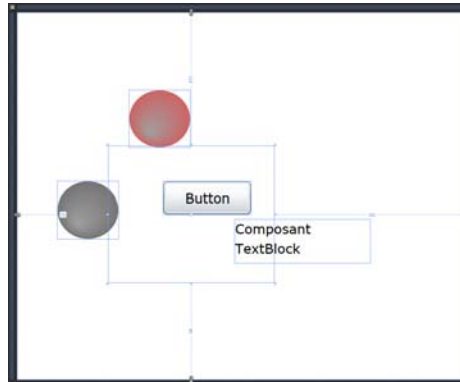
- **Canvas.** C'est le conteneur le moins contraignant et le plus simple. Il faut considérer le conteneur Canvas comme un point dans l'espace à partir duquel les objets qui y sont contenus sont positionnés. Les enfants de ce conteneur peuvent donc avoir des coordonnées positives ou négatives. Ils peuvent donc être en dehors du Canvas, mais sont affichés dans tous les cas. Le redimensionnement d'un Canvas *via* ses propriétés Width et Height n'affecte en rien la position ou l'affichage de ses enfants (voir Figure 5.3).

Les objets contenus au sein d'un Canvas récupèrent automatiquement les propriétés Top et Left attribuées par celui-ci. Lorsqu'une propriété est de type attachée, cela est visible dans le XAML. Les propriétés Top et Left sont préfixées de Canvas, comme le montre le code XAML ci-dessous :

```
<Canvas Margin="75,110,161,80">
    <Button Height="27" Width="72" Content="Button" Canvas.Left="45"
Canvas.Top="29" />
</Canvas>
```


Figure 5.3

Composants imbriqués au sein d'un Canvas.



- `DockPanel` permet de docker des objets en leur attribuant un espace dans une des quatre directions. Les composants qu'il contient pourront être dockés à gauche, en haut, à droite ou en bas du `DockPanel`. L'ordre dans lequel ceux-ci sont imbriqués, ainsi que les valeurs `Width` et `Height` de ces composants, déterminent l'espace qui leur est alloué et celui restant pour les autres composants du `DockPanel`. À cette fin `DockPanel` bénéficie de la propriété booléenne `LastChildFill`, qui permet de spécifier si le dernier élément imbriqué remplit ou non la totalité de l'espace restant. Par défaut cette propriété est à `true`, ce qui n'est pas le cas pour le `DockPanel` de la Figure 5.4 puisqu'il reste une place libre.

Figure 5.4

Composants imbriqués au sein d'un DockPanel.



- `Grid` est un composant d'agencement très puissant et souple d'utilisation. Il est assez complexe et permet de résoudre la majorité des problématiques d'agencement. C'est pour cette raison que ce composant est directement placé au sein du `UserControl` racine lors de la création d'un projet Silverlight. Ce composant propose des options de mise en page très semblables à ce que vous pouvez réaliser en HTML et CSS avec les balises de type `DIV`. Il s'agit en réalité d'une grille qui peut contenir un nombre indéfini de colonnes et de lignes. Comme vous avez pu le voir dans les chapitres précédents, il contraint automatiquement les objets qui sont imbriqués en son sein.
- `StackPanel` permet d'empiler les objets les uns après les autres dans la direction horizontale ou verticale. Comme vous l'avez constaté, les options de marges influencent directement le positionnement des objets empilés les uns par rapport aux autres.
- `WrapPanel` possède le même comportement que `StackPanel`, mais renvoie à la ligne automatiquement (voir Chapitre 4).

Ces composants répondent à la grande majorité des besoins en matière d'agencement. Toutefois, vous devrez étendre la classe `Panel` pour créer vos propres conteneurs personnalisés. Cette classe possède à cette fin deux méthodes virtuelles, `MeasureOverride` et `ArrangeOverride`. Vous devrez donc les surcharger lorsque vous créerez vos propres conteneurs. Tous les composants héritant de `Panel` partagent la propriété `Children`. Cette dernière est un ensemble de contrôles `UIElement` donnant accès aux enfants du conteneur, qui seront forcément accessibles car ce sont des objets graphiques. Ils sont donc au moins de type `UIElement`.

Ce qu'on appelle l'arbre visuel et logique d'une application est en fait l'ensemble des objets affichés ainsi que leurs relations enfant parent. Trois propriétés permettent ainsi d'imbriquer les éléments : `Content` pour les contrôles de type `ContentControl`, `Children` pour les conteneurs héritant de `Panel` et `Child` pour tous les autres conteneurs spécifiques, tels que `ViewBox` par exemple. Nous allons maintenant nous intéresser à la propriété `Children` et apprendre à gérer la liste d'affichage d'un conteneur de type `Panel`.

5.2 Principe d'imbrication

5.2.1 Ordre d'imbrication

La propriété `Children` de la classe `Panel` étant une collection, il est possible d'accéder à ses enfants par un index numérique correspondant à son ordre d'imbrication. L'ordre d'imbrication influe sur deux pôles : le design pur et la conception.

5.2.1.1 Du point de vue design

Le design est influencé car l'ordre d'imbrication impacte directement l'ordre de superposition des objets les uns par rapport aux autres, de la même manière que les calques sous Photoshop ou Expression Design (voir Figure 5.5).

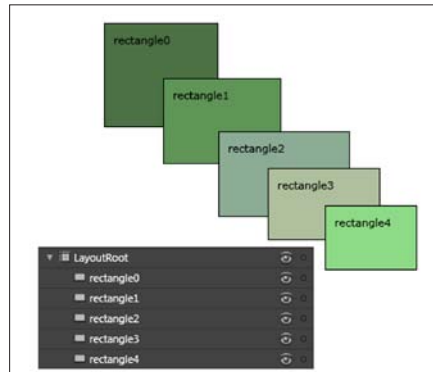
Voici un exemple d'agencement au sein d'une grille. Comme vous le remarquez, `rectangle0` est en dessous des autres d'un point de vue visuel. Il est situé à l'index 0 de la liste d'affichage, toutefois au sein de Blend il est représenté en haut de la liste d'affichage dans l'arbre visuel. Blend affiche par défaut l'arbre visuel en respectant l'ordre de création XAML. Le code XAML ci-dessous correspond à l'arbre visuel affiché à la Figure 5.5 :

```
<Grid x:Name="LayoutRoot" Background="#FFFFFF">
  <Rectangle x:Name="rectangle0" Fill="#FF4B7145" Stroke="#FF000000"
    Height="108" HorizontalAlignment="Left"
    Margin="137,89,0,0" VerticalAlignment="Top" Width="118"/>
  <Rectangle x:Name="rectangle1" Fill="#FF5F9657" Stroke="#FF000000"
    Height="89" HorizontalAlignment="Left"
    Margin="198,146,0,0" VerticalAlignment="Top" Width="122"/>
  <Rectangle x:Name="rectangle2" Fill="#FF8CAC95" Stroke="#FF000000"
    Margin="255,201,249,191"/>
  <Rectangle x:Name="rectangle3" Fill="#FFB0BF9D" Stroke="#FF000000"
    Margin="306,239,218,167"/>
  <Rectangle x:Name="rectangle4" Fill="#FF8DDB86" Stroke="#FF000000"
    Height="67" HorizontalAlignment="Right"
    Margin="0,0,180,136" VerticalAlignment="Bottom"
    Width="95"/>
</Grid>
```

L'ordre proposé par défaut est donc assez logique d'un point de vue architecture. L'arbre visuel peut au départ vous induire en erreur si vous êtes habitué à d'autres logiciels car l'objet le plus haut dans l'arbre est celui situé le plus à l'arrière-plan par rapport aux autres. Vous pouvez à tout moment changer ce mode de présentation en cliquant sur le bouton situé en bas à gauche de l'arbre visuel (📄). Toutefois, vous vous habituerez très vite à cette représentation car elle est cohérente avec l'ordre des balises XAML générées.

Figure 5.5

Ordre d'imbrication et index d'affichage.



5.2.1.2 Du point de vue conception

D'un point de vue conception, l'ordre d'imbrication est déterminant car il impose souvent le positionnement des objets au sein des conteneurs. L'agencement des enfants au sein d'un contrôle StackPanel ou WrapPanel, par exemple, en est une conséquence directe comme vous avez pu le constater au Chapitre 4 (voir Figure 5.4 et 5.6).

Figure 5.6

Influence de l'index de l'objet enfant au sein d'un WrapPanel.



Le bouton le plus à droite au sein du WrapPanel est ContactBtn. Il s'agit en fait du dernier bouton imbriqué et du seul qui est nommé. Il possède l'index 4. L'index influence donc également l'architecture même de l'application. Voici le code XAML représenté à la Figure 5.6 :

```
<controls:WrapPanel Height="Auto" HorizontalAlignment="Stretch"
    VerticalAlignment="Top" Width="Auto" Margin="30,10,90,0"
    d:LayoutOverrides="Width">
    <Button Height="Auto" Width="Auto" Content="Nouveautés"
        Margin="0,0,20,0" FontSize="14" FontFamily="Trebuchet MS"
        Visibility="Visible" />
    <Button Height="Auto" Width="Auto" Content="Portfolio"
        Margin="0,0,20,0" FontSize="14" FontFamily="Trebuchet MS"
        Visibility="Visible" />
    <Button Height="Auto" Width="Auto" Content="Médias"
        Margin="0,0,20,0"
```

```

        FontSize="14" FontFamily="Trebuchet MS"
        Visibility="Visible"/>
<Button Height="Auto" Width="Auto" Content="Savoir faire"
        VerticalAlignment="Stretch" Margin="0,0,20,0"
        FontSize="14" FontFamily="Trebuchet MS"
        Visibility="Visible"/>
<Button x:Name="ContactBtn" Height="Auto" Width="Auto"
        Content="Contact" Margin="0,0,0,0" FontSize="14"
        FontFamily="Trebuchet MS" Visibility="Visible"/>
</controls:WrapPanel>

```

INFO

Nommer les objets est d'une aide précieuse dans de nombreux cas. Même si ce n'est pas toujours obligatoire d'un point de vue technique, cela permet aux développeurs et aux designers interactifs de communiquer simplement et d'éviter les malentendus. Ces deux profils sont concernés par le nommage. Donner un nom à un objet consiste à définir sa fonctionnalité et par conséquent une partie de l'architecture. Le développeur est en général très concerné par cette phase, il doit donc veiller à ce que les intégrateurs ne perdent pas de temps sur la compréhension de l'arbre visuel.

Parfois, le développeur voudra manipuler un objet non nommé ou réorganiser une interface en arrangeant la liste d'affichage. Le graphiste, quant à lui, souhaitera tout de même gérer la superposition des éléments les uns entre les autres sans s'occuper directement de l'architecture ou de la conception technique. Nous allons aborder les méthodes et techniques qui répondent à ces problématiques en nous efforçant de garder un flux de production souple et en préservant les rôles de chacun.

5.2.2 Accéder aux objets de l'arbre visuel

5.2.2.1 Accéder à un composant nommé

Un objet nommé est facile à cibler car lorsqu'un objet est nommé, il devient un champ privé de la classe de l'application. Il faut en effet garder à l'esprit que tout ce que nous produisons au sein du fichier XAML est en réalité créé au sein de la classe partielle. Un développeur n'a qu'à écrire le nom d'un objet déclaré dans le XAML suivi d'un point pour accéder à tous ses membres. Nous pourrions de cette manière facilement accéder au bouton `ContactBtn` (voir Figure 5.7).

Figure 5.7

Accéder aux objets nommés.



Toutefois, vous n'aurez pas toujours la possibilité de connaître le nom d'un objet à l'avance ou encore de nommer tous les objets dans Blend. Nous allons donc apprendre à accéder à un objet anonyme.

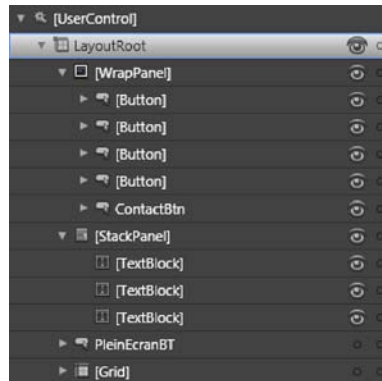
5.2.2.2 Accéder à un composant anonyme

Ouvrez le projet `SiteAgencePortfolio` créé lors du précédent chapitre. Pour atteindre un objet présent au sein de la liste d'affichage d'un conteneur, il suffit de cibler son index. Nous allons changer le texte du bouton anonyme représentant les nouveautés par le texte `Accueil` en procédant par étape. Étudions l'arbre visuel et logique de notre projet (voir Figure 5.8).

Le bouton des nouveautés est situé à l'index 0 du `WrapPanel`. Pour l'atteindre, nous devons utiliser la propriété `Children` du `WrapPanel`. Cependant, le `WrapPanel` n'est lui-même pas nommé. Nous devons donc également utiliser l'index enfant de celui-ci au sein du conteneur nommé `LayoutRoot`.

Figure 5.8

Arbre visuel du projet `SiteAgencePortfolio`.



Voici comment accéder à l'index 0 *via* la propriété `Children` d'un composant de type `Panel` :

```
MonConteneur.Children[0]
```

Pour accéder à l'objet `WrapPanel`, qui est le premier enfant de `LayoutRoot`, on écrira donc :

```
LayoutRoot.Children[0];
```

Pour des raisons de conception, `Children` est une collection d'`UIElement`. Les enfants d'un conteneur peuvent être de type différent les uns en les autres, mais ils héritent tous de la classe `UIElement` mère de toutes les classes d'affichage. Elle semble donc idéale pour représenter le type général des objets stockés dans la propriété `Children`. Nous n'aurons donc pas accès dans un premier temps aux membres de la classe `WrapPanel` mais plutôt aux membres de la classe `UIElement`. Nous devons donc transcrire la référence récupérée en `WrapPanel`, puis stocker sa valeur au sein d'une référence de même type :

```
WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
```

Vous pouvez à tous moments récupérer le type d'un objet *via* la méthode héritée de `Object`, `GetType()`. Voici comment faire :

```
LayoutRoot.Children[0].GetType();
```

La méthode `GetType()` renvoie un objet `Type`. Cet objet fournit toutes les informations que vous souhaitez obtenir sur la classe réelle de l'objet. Vous pouvez récupérer le nom de la classe *via* sa propriété `Name`. Il suffit ensuite de l'afficher dans le champ texte au centre du site, de cette manière :

```
WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
if (menuHaut != null)
{
    string nomType = menuHaut.GetType().Name;
    TitrePage.Text = nomType;
}
```

Lorsque vous compilez, vous pouvez constater qu'il s'agit bien d'un `WrapPanel`. Si vous souhaitez vérifier qu'il contient les menus, il faut regarder le nombre de ses enfants grâce à la propriété `Count` de `Children` :

```
WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
if (menuHaut != null)
{
    string nomType = menuHaut.GetType().Name;
    TitrePage.Text = nomType + " - Nb enfants : " + menuHaut.Children.Count;
}
```


Pour modifier le champ texte du premier bouton, nous devons le cibler *via* `Children`, accéder à sa propriété `Content` et affecter celle-ci :

```
Button un = menuHaut.Children[0] as Button;
un.Content = "Accueil";
```

5.2.2.3 Gérer les erreurs d'accès

Que se passe-t-il lorsque nous essayons d'accéder à un index indéfini de la liste d'affichage ? Pour le savoir, il vous suffit d'essayer d'accéder à l'index 5 de la liste d'enfants du `WrapPanel`. Celui-ci ne possédant que 5 enfants, le dernier index utilisé est 4. Pour générer cette erreur, il faut donc écrire :

```
Button Un = menuHaut.Children[5] as Button;
```

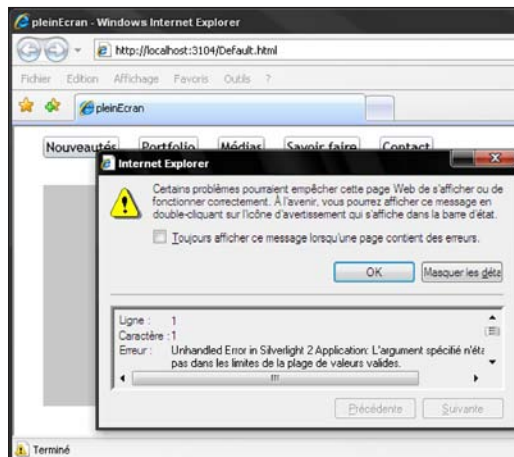
Compilez *via* le raccourci F5. Aucune exception n'est levée en apparence, mais si vous regardez en bas à gauche du navigateur Internet Explorer, vous apercevrez une icône jaune . Celle-ci vous indique une erreur au sein de l'application Silverlight. Double-cliquez dessus pour faire apparaître le détail de l'erreur rencontrée (voir Figure 5.9).

En réalité, Blend ne possède pas de vrai débogueur comme Visual Studio. Les erreurs sont levées, mais n'interrompent pas l'exécution de l'application. Il n'est donc pas possible de poser des points d'arrêt, d'afficher les valeurs des variables durant l'exécution ou d'afficher directement la ligne de code fautive ou l'exception levée. Le designer qui ne remarquerait pas l'icône d'avertissement aurait la fausse impression qu'il s'agit d'un échec en silence. De plus, ainsi que vous pouvez le constater à la Figure 5.9, le numéro de ligne spécifié dans le panneau d'erreur dans le navigateur n'est pas vraiment le bon. Cependant, un designer interactif trouvera cela déjà très utile car l'erreur renvoyée est la bonne :

L'argument spécifié n'était pas dans les limites de la plage de valeurs valide.
nom du paramètre : index

Figure 5.9

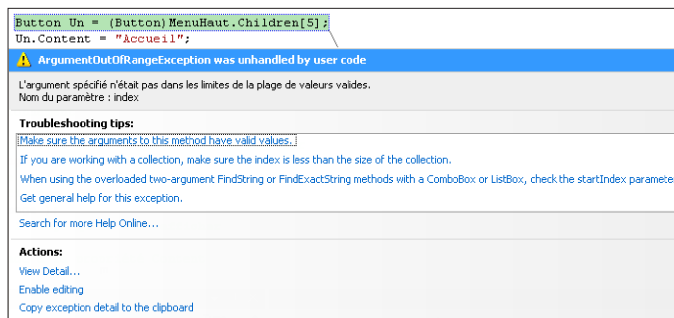
Erreur levée au sein du navigateur.



Visual Studio offre un environnement plus confortable pour déboguer une application. Nous allons compiler la solution dans ce logiciel. Cliquez-droit sur la solution dans le panneau Projet et sélectionnez Ouvrir dans Visual Studio. Une fois le projet ouvert sous Visual Studio, recompilez-le via le raccourci F5. Dès le chargement de l'application, le débogueur lève une exception et affiche la ligne de code posant problème (voir Figure 5.10).

Figure 5.10

Erreur levée par le débogueur.



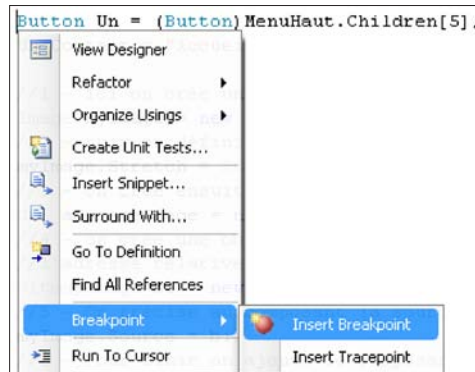
Il est ainsi plus rapide et pratique de trouver d'où vient notre erreur car si le message est le même, la ligne de code fautive est correctement localisée.

INFO

L'un des avantages majeurs du débogueur est de pouvoir récupérer les valeurs des références durant l'exécution de l'application. Il suffit pour cela de positionner votre souris au-dessus d'une variable lorsqu'une erreur est levée ou lorsque vous avez posé un point d'arrêt. Pour poser un point d'arrêt, cliquez à gauche de la ligne sur laquelle vous le souhaitez dans l'éditeur de code. Vous pouvez également cliquer droit sur la ligne où vous désirez que l'exécution soit stoppée, puis sélectionner Breakpoint (Point d'arrêt) et Insert Breakpoint (Ajouter un point d'arrêt) – voir Figure 5.11.

Figure 5.11

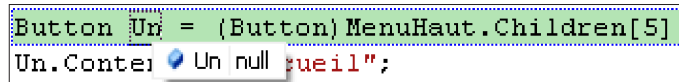
Insérer un point d'arrêt.



Examinons maintenant la valeur de l'instance du bouton Un à l'exécution. Pour cela, positionnez votre souris au-dessus de sa référence (voir Figure 5.12).

Figure 5.12

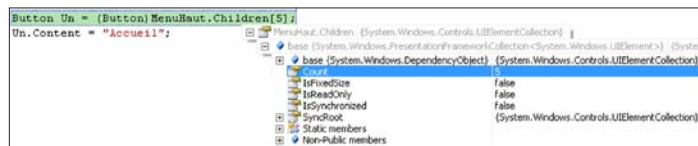
Valeur de la référence
Un de type Button.



La valeur du bouton Un est null. L'expression à droite du signe égal ne renvoie donc aucune valeur. Positionnez votre souris au-dessus de la propriété Children pour explorer son contenu (voir Figure 5.13).

Figure 5.13

Exploration de la
propriété Children
d'un conteneur grâce
au débogueur.



Nous nous apercevons que cette propriété ne possède que 5 éléments, indexés de 0 à 4. Cibler l'index 5 revient à cibler un sixième élément indéfini dans cette collection. Nous mettons donc facilement en évidence l'erreur levée.

Parfois, pour diverses raisons, par exemple pour montrer l'application à une tierce personne, vous ne souhaitez pas lancer le débogueur de Visual Studio. Il suffit dans ce cas d'utiliser le raccourci Ctrl+F5. Celui-ci permet une compilation équivalente à celle fournie par Blend en ne lançant pas le débogueur.

5.2.3 Parcourir la liste des enfants

Nous allons maintenant apprendre à parcourir la liste des enfants grâce aux boucles. Il peut être très pratique de parcourir la collection d'enfants Children pour affecter ou récupérer les propriétés de ceux-ci. Nous allons procéder par étapes. Tout d'abord, nous allons créer une méthode de parcours dans la classe MainPageMainPage. Celle-ci aura pour but de créer une représentation de l'arbre visuel sous forme de chaîne de caractères. Pour visualiser la chaîne de caractères, il vous faudra créer un composant TextBlock à qui nous affecterons la propriété Text. La méthode doit

recevoir comme paramètre le conteneur de type `Panel` que nous souhaitons parcourir ainsi que la valeur de tabulation. En voici une ébauche :

```
private void ParcourArbreVisuel ( Panel container, string tab )
{
}
}
```

Voyons comment l'écrire avec une boucle `for` :

```
private void ParcourArbreVisuel ( Panel container, string tab )
{
    // Les deux écritures suivantes sont utilisables
    //int Lng = container.Children.Count;
    var Lng = container.Children.Count;

    for (var i=0; i < Lng; i++)
    {
    }
}
```

Créez maintenant le `TextBlock` n'importe où dans la grille principale `LayoutRoot` et nommez-le `ArbreTxt`. Ce champ texte va nous permettre d'afficher notre arbre visuel. Chaque fois que nous trouverons un enfant, nous concatènerons le nom de sa classe suivi de son nom d'instance si celle-ci est nommée :

```
private void ParcourArbreVisuel ( Panel container, string tab )
{
    // Les deux écritures suivantes sont utilisables
    //int Lng = container.Children.Count;
    var Lng = container.Children.Count;

    for (var i=0; i < Lng; i++)
    {
        //on récupère le nom de la classe de chaque enfant
        ArbreTxt.Text += tab + container.Children[i].GetType().Name;

        // On récupère le nom de l'instance récupérée
        string nom = (container.Children[i] as FrameworkElement ).Name;

        // si l'instance est nommée
        if (!String.IsNullOrEmpty(nom))
        {
            ArbreTxt.Text += " - " + nom;
        }

        //On va à la ligne grâce au caractère d'échappement n
        ArbreTxt.Text += "\n";
    }
}
```

L'opérateur `+` au début de la boucle permet de concaténer des chaînes de caractères. Concaténer une chaîne revient à générer une chaîne en juxtaposant plusieurs autres chaîne de caractères les unes aux autres. La propriété `Name` est utilisée deux fois dans la boucle : la première fois pour récupérer le nom de l'objet `Type` retourné, la seconde fois pour retrouver le nom de l'instance.

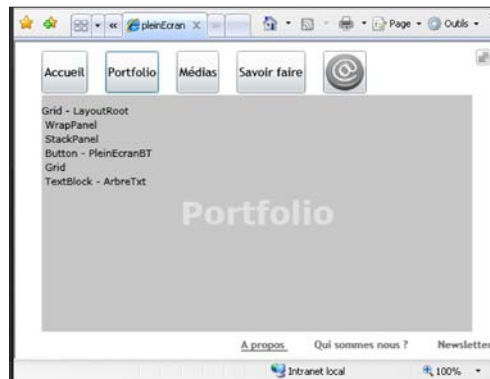
Toutefois, la propriété Name n'est pas héritée de UIElement, mais de FrameworkElement. Nous sommes donc obligé de transtyper (convertir le type) UIElement en FrameworkElement. Le mot-clé as permet de considérer un type en lieu et place d'un autre type. Pour finir, grâce au caractère d'échappement \, le caractère n est considéré comme un retour à la ligne. Cette méthode nous permet de lister tous les objets situés dans LayoutRoot. Il nous faut maintenant l'appeler au sein de la méthode MainPage_Loaded :

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ...
    ArbreTxt.Text = LayoutRoot.GetType().Name+" - "+LayoutRoot.Name+"\n";
    //appel de la méthode
    ParcourArbreVisuel( LayoutRoot, "\t" );
}
```

Lors de cet appel, nous listons le premier niveau d'imbrication, soit tous les objets présents dans LayoutRoot. Pour cette raison, nous passons la chaîne de caractères "\t" correspondant à une seule tabulation (voir Figure 5.14).

Figure 5.14

Affichage des enfants de LayoutRoot.



Nous n'avons affiché que le premier niveau d'imbrication, mais il serait intéressant de parcourir l'arbre visuel dans sa totalité. Pour cela, nous pouvons transformer notre méthode en méthode récursive. Une méthode récursive est une fonction capable de s'appeler elle-même tant qu'une condition est réalisée. Notre condition est simple : à chaque fois que nous listons un enfant, nous pouvons vérifier s'il est de type Panel. Dans ce cas il peut donc contenir des objets enfants. Vérifier le type Panel est pratique : que l'enfant soit un WrapPanel, un StackPanel ou un DockPanel importe peu car notre méthode sera ré-invoquée dans tous ces cas. Voici la méthode de parcours améliorée :

```
private void ParcourArbreVisuel ( Panel container, string tab )
{
    // Les deux écritures suivantes sont utilisables
    //int Lng = container.Children.Count;
    var Lng = container.Children.Count;

    for (var i=0; i < Lng; i++)
    {
        //on récupère le nom de la classe de chaque enfant
        ArbreTxt.Text += tab + container.Children[i].GetType().Name;
```

```

// On récupère le nom de l'instance récupérée
string nom = (container.Children[i] as FrameworkElement ).Name;

// si l'instance est nommée
if (!String.IsNullOrEmpty(nom))
{
    ArbreTxt.Text += " - " + nom;
}

// On saute une ligne grâce au caractère d'échappement n
ArbreTxt.Text += "\n";

// Si l'enfant que l'on vient de lister est aussi un Panel
// alors on appelle à nouveau la méthode de parcours

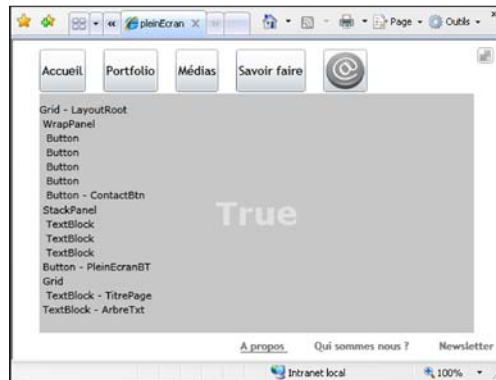
Panel panel = container.Children[i] as Panel;
if (panel != null)
{
    ParcourArbreVisuel( panel, "\t\t");
}
}
}

```

Vous devriez obtenir un résultat proche de la Figure 5.15.

Figure 5.15

Affichage récursif.



La boucle `for` n'est pas la plus adéquate car `Children` de type `UIElementCollection` implémente l'interface `IEnumerable`. Celle-ci décrit les méthodes permettant le parcours de la collection avec une boucle `foreach`. Le code est bien plus simple à écrire dans ce cas, le voici mis à jour :

```

private void ParcourArbreVisuel ( Panel container, string tab )
{
    foreach (UIElement enfant in container.Children)
    {
        ArbreTxt.Text += tab + enfant.GetType().Name;

        string nom = (enfant as FrameworkElement ).Name;

        if (!String.IsNullOrEmpty(nom))
        {
            ArbreTxt.Text += " - " + nom;
        }
    }
}

```

```

ArbreTxt.Text += "\n";

Panel panel = enfant as Panel;
if (panel != null)
{
    ParcoursArbreVisuel( panel, "\t\t");
}
}
}

```

Nous allons en rester là pour le parcours de l'arbre visuel. Toutefois, trois propriétés permettent l'imbrication, comme nous l'avons dit au début de ce chapitre : `Children`, `Child` et `Content`. Pour parcourir la totalité de l'arbre visuel de notre application, nous devrions également tester si l'enfant est de type `ContentControl` ou si celui-ci possède la propriété `Child`. Derrière ces propriétés peut se cacher une imbrication élaborée et complexe. Vous pouvez télécharger l'exercice *pleinEcran_parcours.zip* dans le dossier *chap5* des exemples.

5.3 Ajouter des enfants à la liste d'affichage

Ajouter des enfants en cours d'exécution est une opération courante. Cela nous permet de faire vivre et évoluer l'application. Les applications dynamiques apportent une expérience utilisateur très différente car elles évoluent lors de leur utilisation, dans des directions quelquefois surprenantes. Cela ajoute une profondeur aux applications et les rend plus attractives, donnant envie d'en explorer tous les recoins. Nous allons maintenant apprendre à réaliser ces opérations en ajoutant des objets à la volée, puis en reconstruisant la totalité de notre menu de manière dynamique.

5.3.1 Mécanisme d'instanciation d'objets graphiques

Cela peut paraître étrange, mais lorsque le graphiste crée des objets graphiques au sein de Expression Blend, il accomplit en réalité deux choses différentes. Il commence par instancier l'objet en le décrivant au sein d'une balise :

```

<Button Height="Auto" Width="Auto" Content="Portfolio" Margin="0,0,20,0"
        FontSize="14" FontFamily="Trebuchet MS" Visibility="Visible" />

```

Toutefois, cela ne suffit pas, le bouton ne s'afficherait pas s'il ne faisait pas partie de l'arbre visuel. Il doit donc appartenir à l'un des objets de cet arbre. Un conteneur de type `Grid` fait l'affaire.

La deuxième étape réalisée sans réellement y prêter attention est donc la création du lien de parenté avec son conteneur :

```

<Grid x:Name="LayoutRoot" Background="#FFFFFF" Margin="0,0,0,0" >
    <Button Height="Auto" Width="Auto" Content="Portfolio" .../>
</Grid>

```

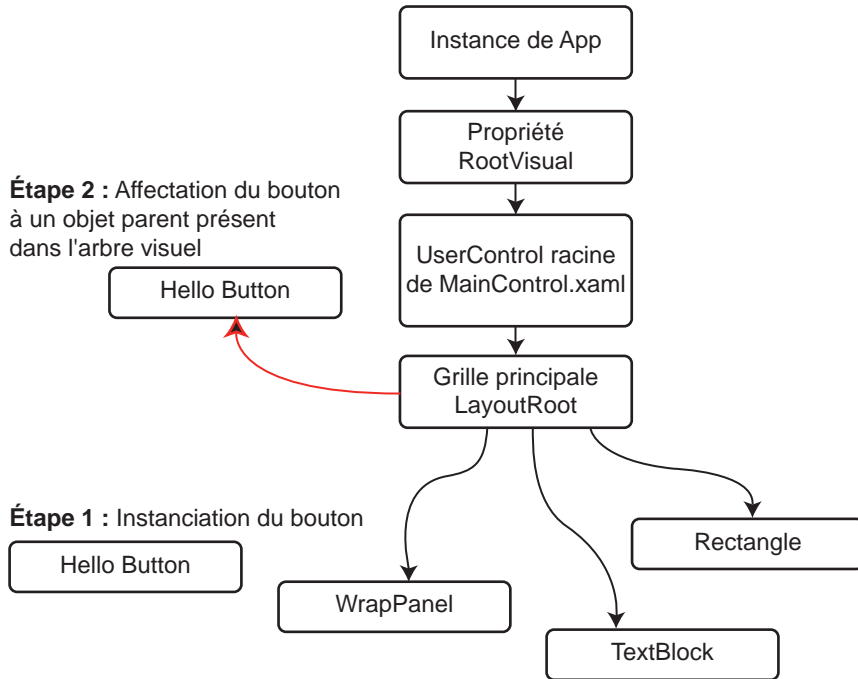
Au sein de Blend, cette étape est naturelle puisque l'on ne peut créer des objets qu'au sein de l'arbre visuel (voir Figure 5.16).

Figure 5.16

Mécanisme
d'ajout
d'objets
graphiques
à l'arbre
visuel.

Étape 2 : Affectation du bouton
à un objet parent présent
dans l'arbre visuel

Étape 1 : Instanciation du bouton



ATTENTION

L'instance du plug-in Silverlight possède la propriété `RootVisual` héritée de la classe `Application`. Celle-ci renvoie une instance du `UserControl` racine du fichier `MainPage.xaml`. Tout objet appartenant directement ou indirectement à la propriété `RootVisual` fait partie de l'arbre visuel de l'application. Lors de l'initialisation de l'application, cette propriété est en fait affectée au sein du fichier `App.xaml.cs`, dans la classe `App` :

```

private void OnStartup(object sender, StartupEventArgs e)
{
    // Chargement de l'arborescence contenue dans MainPage.xaml
    this.RootVisual = new MainPage();
}

```

Cette propriété est également accessible en C# à partir de n'importe quel fichier XAML. Vous pouvez l'atteindre en ciblant la propriété statique `Current` de la classe `App` :

```
App.Current.RootVisual
```

La propriété statique `Current` de la classe `App` renvoie l'instance actuelle de l'application.

5.3.2 Ajouter et insérer des objets dans l'arbre

Ajoutons un nouveau menu dans la barre. Nous allons réaliser la première étape consistant à instancier un bouton *via* C# :

```

WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
// on instancie le bouton
Button clientMenu = new Button();
// on lui spécifie de nouvelles marges
clientMenu.Margin = new Thickness (20,0,0,0);
// on lui affecte une nouvelle police
clientMenu.FontFamily = new FontFamily("Trebuchet MS");
// on définit une taille de police
clientMenu.FontSize = 14;
// on affecte sa propriété Content pour afficher une chaîne de caractères
clientMenu.Content = "Nos Clients";

```

Dans tous les cas nous utiliserons le mot-clé new. Cela semble étrange, mais la propriété Margin est de type Thickness. C'est une manière d'économiser le poids du lecteur Silverlight car la valeur est structurée de la même manière qu'un objet d'épaisseur, de type Thickness. Le premier chiffre indique la marge à gauche, puis le deuxième la marge haute, et ainsi de suite dans le sens des aiguilles d'une montre. Comme nous pouvons le voir, affecter une police est assez simple. La police Trebuchet MS possède l'avantage d'être embarquée par défaut dans le lecteur Silverlight. Il est donc facile de l'affecter au bouton. Vous remarquez que nous n'avons spécifié aucune largeur ou hauteur. Lorsque vous ne précisez pas ces valeurs, celles-ci sont par défaut en mode Auto. Pour finir, il suffit d'afficher un texte au sein de notre bouton. La seconde phase est plus facile, nous allons utiliser la méthode Add :

```

// on ajoute ce bouton à la liste des enfants du WrapPanel
menuHaut.Children.Add(ClientMenu);

```

La méthode Add n'ajoute pas les objets au hasard, en fait elle ajoute l'objet à la fin de la liste d'enfants, donc à un nouvel index. Notre menu contenait 5 éléments indexés de 0 à 4. À la fin du chargement de l'application, le menu contient désormais 6 éléments. Le sixième élément ajouté dynamiquement est situé au dernier et nouvel index, soit 5. Cela est particulièrement facile à constater dans un conteneur de type WrapPanel car l'index affecte directement l'ordre des objets imbriqués (voir Figure 5.17).

Figure 5.17

*Ajout d'une rubrique
Button avec la
méthode Add.*



La méthode Add n'appartient pas à l'objet lui-même mais à sa propriété de type UI-ElementCollection. Lorsque vous voudrez manipuler les enfants ou la liste elle-même, il faudra utiliser les membres (propriétés et méthodes) de celle-ci. Cette opération est finalement assez facile, toutefois, notre nouveau menu n'est pas vraiment à sa place. Il serait plus judicieux de le placer entre les menus Savoir faire et Contact. Cela est facile à réaliser grâce à la méthode Insert :

```

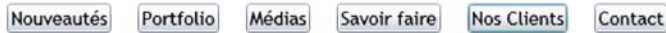
menuHaut.Children.Insert(4,ClientMenu);

```

Il faut commencer par préciser l'index auquel l'objet sera positionné, puis le composant graphique à insérer. Modifiez si besoin la propriété Margin pour une mise en forme adaptée (voir Figure 5.18).

Figure 5.18

*Insertion d'une
rubrique Button avec
la méthode Insert.*



Lorsque vous accomplissez une telle opération, vous décalez de 1 chaque objet imbriqué suivant l'index d'insertion au sein de la liste d'enfants.

5.3.3 Erreurs levées

Plusieurs erreurs peuvent être levées lors de l'appel de ces méthodes. La première arrive lorsque vous passez un index aberrant à la méthode `Insert`. Lorsque vous spécifiez un index dépassant le nombre d'enfants, soit `Children.Count`, le compilateur lève une erreur correspondant à celle levée lors d'une tentative d'accès à un index indéfini (voir section "Gérer les erreurs d'accès", plus haut dans ce chapitre). Il est en effet illogique d'insérer un objet à un index supérieur au nombre d'enfants. Si vous souhaitez connaître l'index du dernier enfant de la collection, il suffit de retrancher 1 à `Children.Count`. Insérer un enfant à l'index `Children.Count` revient à utiliser la méthode `Add`.

La deuxième erreur est levée si vous essayez de déplacer un objet graphique déjà présent dans la liste d'enfants d'un autre conteneur. Ce cas est plus pernicieux. L'une des règles d'affichage est qu'une référence d'objet graphique ne peut être située en même temps dans deux conteneurs différents. Lorsque vous attribuez un objet présent dans l'arbre visuel à un autre conteneur, une exception est levée. Elle vous indique que l'objet est déjà présent ailleurs (voir Figure 5.19). Pour vous en convaincre, il suffit d'ajouter au `StackPanel`, notre pied de page, un élément présent dans le `WrapPanel`, notre menu du haut de page :

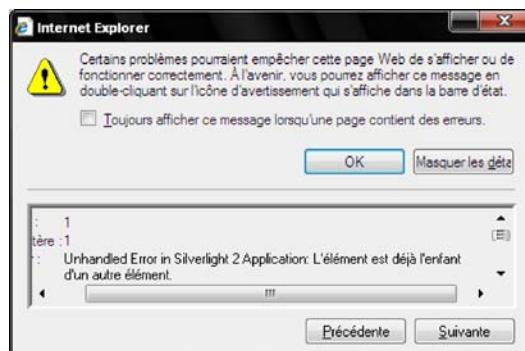
```
WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;

StackPanel footer = LayoutRoot.Children[1] as StackPanel;

Footer.Children.Add( menuHaut.Children[0] );
```

Figure 5.19

*Exception levée lors
de l'imbrication d'un
élément appartenant déjà
à un autre conteneur.*



Pour résoudre cette problématique, vous devrez supprimer l'enfant de la liste de son ancien conteneur, puis invoquer la méthode `Add` ou `Insert`. Pour savoir si un enfant est contenu au sein d'une liste de type `UIElementCollection` (`Children` est de ce type), il suffit d'utiliser les méthodes `Contains` ou `IndexOf` :

```

menuHaut.Children.Contains(Toto);
// renvoie false car Toto n'existe pas

menuHaut.Children.Contains(ContactBtn);
// renvoie true car Contact est bien un enfant de menuHaut

menuHaut.Children.IndexOf(Titi);
// renvoie -1 car Titi n'est pas présent dans la liste des enfants

menuHaut.Children.IndexOf (ContactBtn);
// renvoie 4 car ContactBtn est le 5ième éléments contenu

```

Pour de plus amples informations sur la suppression d'objets graphiques, vous pouvez consulter la section 5.4.

5.3.4 Créer un menu dynamique

Nous allons maintenant recréer le menu principal de toutes pièces. Pour cela, depuis la fenêtre de design, supprimons tous les boutons de menu du WrapPanel. Dans la méthode MainPage_Loaded, il nous faut également supprimer toutes les références vers ces menus sauf la ligne permettant de récupérer une référence du WrapPanel :

```
WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
```

Pour créer un menu à l'exécution, il est plus pratique de créer un tableau de chaîne de caractères. Ensuite, nous pourrons le parcourir. À chaque fois que nous trouverons un élément contenu dans le tableau, nous instancierons dynamiquement un nouveau bouton dans le WrapPanel. Le tableau doit être un champ privé de la classe MainPage, ainsi il sera accessible et modifiable à partir de n'importe quelle méthode de la classe. Voici comment le déclarer :

```

public partial class MainPage : UserControl
{
    string [] tabMenus = new string[]{"Accueil", "Portfolio", "Medias",
                                "Savoir faire", "Contact"};

    public MainPage()
    {
        InitializeComponent();
        Loaded +=new RoutedEventHandler(MainPage_Loaded);
    }

    private void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        PleinEcranBT.Cursor = Cursors.Hand;
        WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
    }
}

```

Ensuite, nous pouvons définir la nouvelle méthode CreateMenus sur la classe MainPage et l'appeler lors du chargement de l'application. La variable MenuHaut est une variable locale à la méthode MainPage_Loaded. Cela signifie que la variable n'existe et n'est accessible que dans cette méthode. Pourtant, nous en avons besoin au sein de la méthode CreateMenus pour ajouter des éléments graphiques à sa liste d'enfants. À ce stade, vous avez trois choix : soit vous créez un champ MenuHaut dans votre classe, puis vous l'affectez au sein de la méthode MainPage_Loaded, soit vous passez MenuHaut comme paramètre de la méthode CreateMenus ou alors vous faites

les deux. Il peut être pratique de créer les boutons de menu dans un autre conteneur si besoin. La méthode `CreateMenus` mérite donc un paramètre. Toutefois, nous pourrions également vouloir supprimer tous les objets contenus au sein du `WrapPanel` dans une autre fonction. Cela peut donc être utile d'en stocker une référence en tant que membre de la classe `MainPage`. Voici comment procéder :

```
public partial class MainPage : UserControl
{
    string [] tabMenus = new string[5]{ "Accueil", "Portfolio", "Medias",
                                         "Savoir faire", "Contact" };

    // on déclare un champ de type WrapPanel
    WrapPanel menuHaut;

    public MainPage()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }

    private void MainPage_Loaded(object sender, RoutedEventArgs e)
    {

        PleinEcranBT.Cursor = Cursors.Hand;

        // on affecte le champ de type WrapPanel
        menuHaut = LayoutRoot.Children[0] as WrapPanel;
        if (menuHaut!=null)
        {
            // on passe le conteneur menuHaut comme paramètre de CreateMenus
            CreateMenus(menuHaut);
        }

        // cette méthode pourra créer des menus quel que soit le conteneur ;)
        private void CreateMenus(Panel conteneur)
        {
            ...
        }
    }
}
```

Typier le paramètre en tant que `Panel` est pratique car nous pourrions spécifier n'importe quel conteneur héritant de ce type, pour centraliser les menus. La méthode `Insert` n'est pas idéale dans notre cas car il faudrait que des composants soient déjà présents dans notre `WrapPanel`. La méthode `Add` est la plus adéquate car elle ajoute les menus au fur et à mesure. L'ordre de notre tableau sera donc respecté. Voici la définition de la méthode `CreateMenus` :

```
private void CreateMenus(Panel conteneur)
{
    foreach(string titre in TabMenus)
    {

        // on instancie le bouton
        Button monMenu = new Button();
        // on lui spécifie de nouvelles marges
        monMenu.Margin = new Thickness (0,0,20,0);
        // on lui affecte une nouvelle police
        monMenu.FontFamily = new FontFamily("Trebuchet MS");
        // on définit une taille de police
```

```

        monMenu.FontSize = 14;
        // on affecte sa propriété Content pour afficher
        // une chaîne de caractères
        monMenu.Content = titre;
        // on ajoute ce bouton à la liste des enfants du WrapPanel
        conteneur.Children.Add(monMenu);
    }
}

```

Comme vous pouvez le voir, mis à part spécifier les marges de chaque bouton, nous n'avons rien à coder concernant le placement des menus. C'est le conteneur `WrapPanel` qui s'occupe de cette partie.

INFO

Dans cet exercice, nous aurions pu utiliser une classe `Rubrique` et créer un tableau d'instances de `Rubrique`. L'avantage est non seulement de pouvoir décrire et afficher plus qu'une simple chaîne de caractères, mais aussi de définir des événements et méthodes spécifiques. Ce genre d'instance est appelée `Value Object`. Ce sont des instances dont le rôle majeur est de contenir les enregistrements récupérés depuis une base de données tout en les adaptant au modèle objet. C'est généralement la méthodologie à suivre lorsque le site est véritablement mis à jour depuis l'extérieur.

Vous trouverez le projet finalisé dans l'archive *pleinEcran_dynamique.zip* du dossier *chap5*.

5.3.5 Affecter les propriétés *Child* et *Content*

Nous allons évoquer rapidement les propriétés `Child` et `Content`. Elles occupent une place importante en matière d'imbrication. Il suffit de voir le nombre d'objets héritant de `ContentControl` ou assurant une logique d'imbrication spécifique pour s'en convaincre. Ces propriétés pourraient en effet stocker un conteneur à enfants multiples, qui lui-même contiendrait de nombreux objets. Il n'y a pas de limites en la matière. La Figure 5.20 illustre un exemple d'imbrication réalisé dans Blend par un graphiste, avec les trois propriétés `Children`, `Child` et `Content`.

Figure 5.20

Exemple d'arbre visuel utilisant les trois propriétés d'imbrication.



Dans l'exemple de la Figure 5.20, le composant `Border` permet de gérer l'affichage et le graphisme de la barre de lecture. Sa propriété `Child` est affectée d'un `StackPanel` empilant horizontalement des boutons personnalisés *via* sa propriété `Children`. Le bouton `stop` utilise sa propriété `Content` pour être affecté d'un simple rectangle. Vous remarquez que le bouton `playPause` est un bouton de type `ToggleButton`. Pour rappel, la classe `CheckBox` hérite de `ToggleButton` et ne se différencie que par son visuel fait pour simplifier la vie au graphiste. Il possède deux états, l'un représentant une icône de lecture, l'autre une icône pause. Pour ajouter un enfant *via* les propriétés `Child` et `Content` en C#, il suffit de les affecter :

```

MonBorder.Child = new StackPanel();
MonBoutonStop.Content = new Rectangle();

```

N'oubliez pas cependant que si vous réaffectez ces propriétés à l'exécution, le nouvel objet affecté remplacera l'ancien :

```

MonBorder.Child = new StackPanel();
MonBorder.Child = new grid(); //

```

Le `StackPanel` ne sera jamais utilisé dans le code précédent car il est directement remplacé par une grille. Vous n'aurez pas d'erreur levée : le nouvel enfant écrase l'ancien.

5.3.6 Événement diffusé

Lorsque vous ajoutez une instance de type `FrameworkElement` à l'arbre visuel, elle diffuse l'événement `Loaded`. Nous allons écouter l'événement `Click` sur le bouton `ContactBtn`. Lorsqu'il sera diffusé, nous ajouterons le bouton `ContactBtn` dans le `WrapPanel` :

```

public partial class MainPage : UserControl
{
    private Button clientMenu;

    public MainPage()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }

    private void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        // on instancie le bouton
        clientMenu = new Button();
        ...
        // nous écoutons l'événement Loaded sur le bouton ClientMenu
        clientMenu.Loaded += new RoutedEventHandler(clientMenu_Loaded);
        ContactBtn.Click += new RoutedEventHandler(ContactBtn_Click);
    }

    private void ContactBtn_Click(object sender, RoutedEventArgs e)
    {
        WrapPanel menuHaut = LayoutRoot.Children[0] as WrapPanel;
        if (menuHaut != null)
        {
            // on ajoute ce bouton à la liste des enfants du WrapPanel
            menuHaut.Children.Insert(4, clientMenu);
        }
    }

    private void clientMenu_Loaded(object sender, RoutedEventArgs e)
    {
        // Lorsque le bouton est ajouté à l'arbre visuel,
        // l'événement Loaded est diffusé pour le montrer
        // on affecte le champ texte central
        TitrePage.Text = "Le bouton a été ajouté et initialisé.";
    }
}

```

Toutefois, il faut faire attention au fait que nous parlons bien d'arbre visuel et non de la propriété Children. Si jamais la propriété Children, à laquelle vous ajoutez un enfant, n'appartient pas à un objet de l'arbre visuel, alors l'événement Loaded ne sera pas diffusé :

```
private void ContactBtn_Click(object sender, RoutedEventArgs e)
{
    StackPanel unConteneur = new StackPanel();
    // on ajoute ce bouton à la liste des enfants du StackPanel
    menuHaut.Children.Insert(4,clientMenu);
}

private void clientMenu_Loaded(object sender, RoutedEventArgs e)
{
    // Lorsque le bouton est ajouté au StackPanel
    // l'événement Loaded n'est pas diffusé car
    // le StackPanel ne fait pas partie de l'arbre visuel
    TitrePage.Text="Ce message ne doit pas être tracé.";
}
```

Ainsi, le même bouton ajouté à un conteneur absent de l'arbre visuel et logique ne diffusera pas d'événement Loaded.

5.4 Supprimer des objets de l'arbre visuel

Nous allons maintenant apprendre à supprimer les enfants d'un conteneur. Comme nous allons le voir, les méthodologies sont très proches de celles que nous avons utilisées pour ajouter des enfants.

5.4.1 Les différentes méthodes

5.4.1.1 Supprimer des enfants de conteneur Panel

Trois méthodes sont accessibles pour supprimer un objet de l'arbre visuel et logique : Remove, RemoveAt et Clear. Ces méthodes sont invoquées par la propriété Children d'un conteneur de type Panel. Voici leur syntaxe :

```
// 1 - La méthode Remove reçoit la référence à l'élément contenu
menuHaut.Children.Remove( ContactBtn );

// 2 - La méthode RemoveAt doit recevoir l'index
menuHaut.Children.RemoveAt( 2 );

// 3 - Clear n'a besoin d'aucun argument, la collection
// de UIElement est simplement vidée de son contenu
menuHaut.Children.Clear( );

(ContactBtn.Parent as Panel).Children.Remove(ContactBtn);
```

La dernière ligne permet à un objet d'appeler la méthode Remove de son conteneur. Tous les objets de type FrameworkElement possèdent la propriété Parent. Celle-ci renvoie la référence du conteneur parent de l'objet. Dans certains cas, il peut arriver que vous ne connaissiez pas la référence du conteneur à l'avance. Cette méthode répond à cette problématique car chaque objet de l'arbre visuel possède une propriété parent non nulle.

5.4.1.2 Supprimer des enfants uniques

Supprimer l'enfant d'un conteneur tel que `Border` ou `ContentControl` est très simple. Pour ajouter un enfant au sein d'un conteneur à enfant unique, il nous a suffi d'affecter une instance d'`UI-Element` aux propriétés `Content` ou `Child`. Cette fois, il nous suffira d'affecter la valeur `null` :

```
MonBorder.Child = null;  
MonBouton.Content = null;
```

ATTENTION

Lorsque vous exécutez les méthodes de suppression de la liste d'enfants, vous ne supprimez que la valeur des propriétés `Children`, `Child` ou `Content`. Les références des enfants qui étaient imbriqués existent toujours en mémoire si celles-ci étaient des membres de classe. Nous verrons comment désactiver des objets en mémoire à la section "Désactiver des objets".

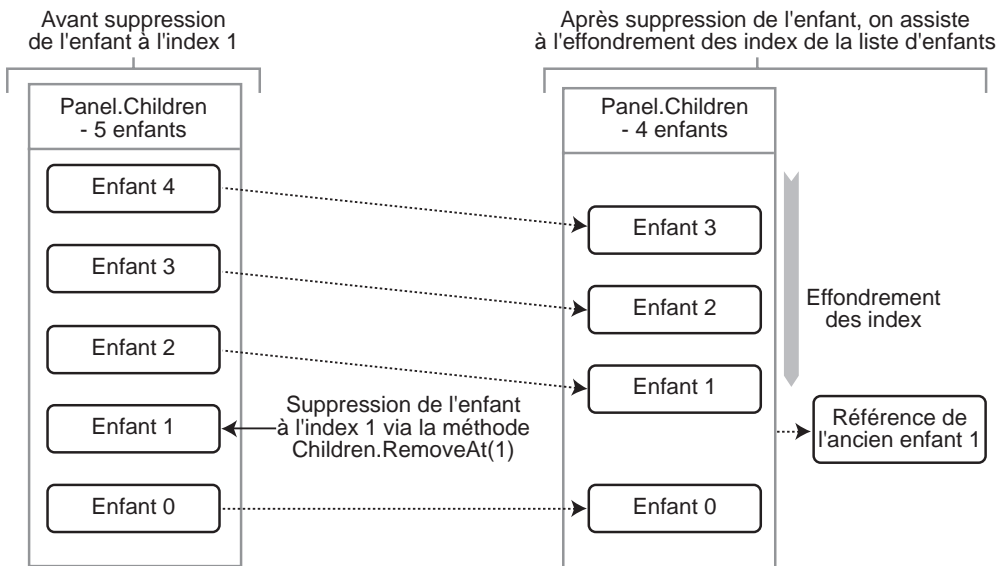
5.4.2 L'effondrement de la liste d'enfants

5.4.2.1 Principe

Comme vous avez pu le remarquer lorsque vous avez compilé votre application, la suppression d'un objet au sein du `StackPanel` a décalé chaque index suivant de -1 . Si, par exemple, vous avez supprimé le bouton à l'index 2, les objets présents à l'index 3 et 4 ont vu leur index passer respectivement à 2 et 3 (voir Figure 5.21).

Figure 5.21

Principe d'effondrement de la liste d'enfants.



Lorsque la liste d'enfants est modifiée, l'agencement du composant se met automatiquement à jour, ce qui peut être pratique pour mettre en forme des contenus dynamiques.

5.4.2.2 Trier une liste d'enfants

Vous serez donc souvent tenté d'utiliser l'effondrement des index pour mettre en forme ou dynamiser une application. Nous allons voir les limites et les contraintes de l'effondrement des index. Vous allez utiliser ce comportement pour supprimer des champs texte contenus dans un `StackPanel`. Ceux dont la propriété `Text` ne possède pas une chaîne de caractères précise seront effacés de la liste. Le `StackPanel` nous servira à cette occasion de liste de tri. Au sein d'un nouveau projet nommé `TriChildren`, instancions le tableau suivant comme champ de la classe `MainPage`, dans le code logique :

```
string[] moisAnnee = new string[12] { "janvier", "février", "mars",
    "avril", "mai", "juin", "juillet", "août", "septembre", "octobre",
    "novembre", "décembre" };
```

Positionnons un nouveau `StackPanel` dans `LayoutRoot`. Nous pouvons soit l'instancier dynamiquement, *via* C#, soit le créer *via* l'interface visuelle de Blend. Appelez-le `MonStackPanel`. Il serait pratique de l'imbriquer au sein d'un `Border` afin de définir un contour visuel. Nous allons parcourir le tableau et créer une occurrence de `TextBlock` pour chaque élément du tableau lu par la boucle :

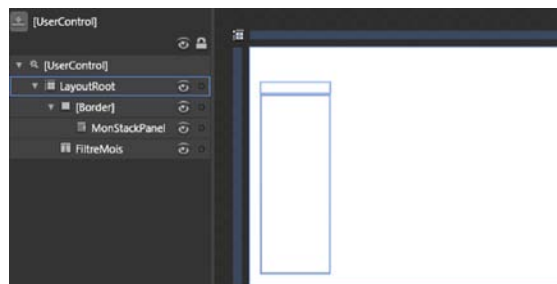
```
string[] moisAnnee = new string[12] { "janvier", "février", "mars",
    "avril", "mai", "juin", "juillet", "août", "septembre", "octobre",
    "novembre", "décembre" };

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    foreach (string mois in moisAnnee)
    {
        TextBlock moisTexte = new TextBlock();
        moisTexte.Foreground = new SolidColorBrush(Colors.Gray);
        moisTexte.FontSize = 16;
        moisTexte.FontFamily = new FontFamily("Trebuchet MS");
        moisTexte.Text = mois;
        monStackPanel.Children.Add(moisTexte);
    }
}
```

Plaçons maintenant un composant de champ texte de saisie, `TextBox`, au-dessus du `StackPanel` dans l'interface visuelle de Blend. Nous pouvons là aussi créer ces éléments par code. Appelons le champ de saisie `FiltreMois` (voir Figure 5.22).

Figure 5.22

Arbre visuel et interface de `TriChildren`.



Ajoutez maintenant la logique nécessaire pour trier les mois affichés dans le StackPanel. Pour cela, vous pouvez ajouter une méthode dans le panneau des événements du champ de saisie. Les champs de saisie TextBox possèdent l'événement TextChanged. Il vous suffit d'écrire le nom de la méthode, à droite de TextChanged. Elle pourrait se nommer OnChangedFiltre. Blend ajoute directement le code correspondant dans la classe MainPage :

```
private void OnChangedFiltre(object sender, TextChangedEventArgs e)
{
    // TODO: Add event handler implementation here.
}
```

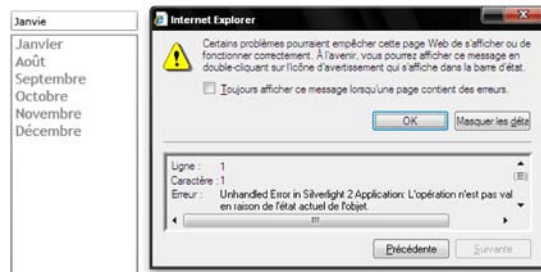
Lorsque le champ de saisie sera modifié par l'utilisateur, supprimez tous les champs texte du StackPanel ne contenant pas la chaîne de caractères entrée par l'utilisateur. Toutefois, vous n'effectuerez le tri qu'à partir d'un minimum de 3 caractères saisis :

```
private void OnChangedFiltre(object sender, TextChangedEventArgs e)
{
    // on vérifie qu'il y a au minimum 3 caractères entrés
    if ( FiltreMois.Text.Length>2 )
    {
        //on parcourt les enfants
        foreach (TextBlock moisTextBlock in MonStackPanel.Children)
        {
            // on récupère chaque chaîne de caractères
            string mt = moisTextBlock.Text as string;
            // si l'un des TextBlock du StackPanel ne contient pas la chaîne
            if ( !mt.Contains( FiltreMois.Text ) )
            {
                // on le supprime
                MonStackPanel.Children.Remove( moisTextBlock );
            }
        }
    }
}
```

Vous remarquez le typeage TextBlock dans la définition de la boucle foreach. Après tout, TextBlock est une sous-classe du type UIElement. Il faudra juste faire attention à ne positionner aucun autre type d'objet dans le StackPanel sous peine de lever une erreur. Compilez et testez votre application. Vous le constatez assez rapidement, le comportement du filtre est étrange. Une erreur est même levée (voir Figure 5.23).

Figure 5.23

Erreur levée lorsque le type attendu ne correspond pas à l'enfant.



Le message est clair, vous parcourez la liste et vous la modifiez en même temps pour la mettre en forme. Toutefois, vous ne réalisez pas une modification anodine dans le cas présent, vous supprimez définitivement un enfant de la liste. La boucle ne peut donc pas être lue correctement et vous

génère des erreurs d'accès. Dans ce cas précis, la méthode que nous employons n'est pas la bonne car n'oubliez pas que supprimer des objets modifie la mise en forme certes, mais affecte surtout l'arbre visuel de l'application et donc son architecture. Nous allons étudier une méthode plus appropriée à ce cas de figure dans la section suivante.

5.4.2.3 Suppression vs. Collapsed

Toute sous-classe de `UIElement` possède la propriété `Visibility`, qui doit être affectée d'une valeur de l'énumération `Visibility`. Attention à ne pas confondre la propriété d'objet et l'énumération (une liste de valeurs, voir Chapitre 3). L'énumération possède les valeurs `Visible` et `Collapsed`. Lorsque vous écrivez :

```
MonUIElement.Visibility = Visibility.Collapsed;
```

vous pratiquez un pseudo effondrement. Dans ce cas précis, le système d'agencement de *Silverlight* (*Silverlight Layout System*) ne prend pas en compte l'objet. Ses marges, sa largeur, sa hauteur et ses enfants sont simplement ignorés, l'objet n'est pas affiché et n'influence pas les autres. Pourtant celui-ci reste un élément de la collection `Children`, vous n'affectez donc en rien l'architecture de votre application. Il faut privilégier cette propriété aux méthodes de suppression lorsque vous souhaitez accomplir des opérations liées à la mise en forme. Cela est simplement plus intuitif, plus pertinent et plus simple. Voici notre code pour la liste de tri, mis à jour en utilisant cette propriété :

```
private void OnChangedFiltre(object sender, TextChangedEventArgs e)
{
    // on parcourt le tableau
    foreach (TextBlock MoisTextBlock in MonStackPanel.Children)
    {
        // on récupère chaque chaîne de caractères
        string Mt = MoisTextBlock.Text as string;
        // si l'un des TextBlock du StackPanel ne contient pas la chaîne
        if ( !Mt.Contains( FiltreMois.Text ) )
            MoisTextBlock.Visibility = Visibility.Collapsed;
        else
            MoisTextBlock.Visibility = Visibility.Visible;
    }
}
```

Le code est beaucoup plus simple dans ce cas. Avec les méthodes `Remove` et `RemoveAt`, nous aurions dû stocker les enfants supprimés dans un tableau temporaire pour obtenir un résultat équivalent et ne pas perdre leur référence.

5.4.3 Désactiver des objets

Nous pourrions nous demander l'intérêt que présente la désactivation des objets. Il faut admettre que plus une application est performante, plus elle est conviviale. D'un point de vue utilisateur, rien n'est plus énervant qu'effectuer des actions et voir leur résultat apparaître avec un temps de latence. Cela peut parfaitement se comprendre pour des accès distants aux bases de données mais c'est une problématique que l'on sait traiter et que l'utilisateur comprend. Pour des actions plus simples, comme naviguer dans un site ou déplier un menu par exemple, il est anormal d'avoir des temps de latence et cela révèle un développement un peu trop hasardeux. Il faut optimiser au maximum les performances de votre application. Pour cette raison, alléger les allocations mémoire

est une tâche réellement importante. Sans cela, l'application pourrait utiliser de plus en plus de mémoire vive et ralentir l'exécution.

Désactiver des objets est le processus inverse de celui consistant à ajouter des objets à l'arbre visuel. Pour ajouter un objet à la liste d'affichage, nous avons d'abord instancié cet objet, puis nous l'avons affecté comme enfant d'un conteneur `Panel` grâce aux méthodes `Add` et `Insert`. Les méthodes de suppression `Remove` et `RemoveAt` ne sont donc pas suffisantes pour désactiver complètement un objet en mémoire. Il faut faire exactement l'opposé de l'instanciation, toutefois une méthode aussi directe n'existe pas.

Les allocations mémoire sont gérées par le ramasse-miettes (ou *Garbage Collector*). C'est lui qui surveille et décide si un objet est désactivé ou non de la mémoire. Nous devons juste permettre au ramasse-miettes de jouer son rôle. Pour cela, la première étape consiste à supprimer de l'arbre visuel l'instance que vous souhaitez désactiver. Tant que celle-ci est présente dans l'arbre visuel, elle ne peut être désactivée puisqu'elle est référencée par une collection de composants `UIElement`. Nous invoquons donc en premier la méthode `Remove` ou `RemoveAt`. La seconde étape est de passer la valeur de la référence à `null` :

```
MonConteneur.Children.Remove(UnUIElement);  
UnUIElement = null;
```

Cette méthode est saine, toutefois vous ne saurez pas quand l'objet sera supprimé de la mémoire. Le langage C# est managé, cela signifie que les allocations mémoire ne sont pas gérées par le développeur C#, mais par le ramasse-miettes. Celui-ci gère la désactivation des instances de classes en fonction de leur occupation mémoire. Moins une référence prend de mémoire, moins elle est prioritaire. Une mauvaise pratique consiste à utiliser la méthode `Collect` du ramasse-miettes, pour forcer son passage. Cela est très coûteux en performances car le ramasse-miettes doit parcourir toutes les références et décider de les supprimer ou non :

```
MonConteneur.Children.Remove(UnUIElement);  
UnUIElement = null;  
GC.Collect();
```

Lorsque vous concevrez vos propres composants, vous devrez faire attention à ce qu'ils suppriment bien tous les écouteurs internes d'événements propres à l'application ou à des références externes. Sans cela, vos instances de composants personnalisés pourraient ne pas être collectées par le ramasse-miettes (voir Chapitre 12).

5.4.5 Événement diffusé

Nous avons vu qu'à chaque fois qu'un objet est ajouté à l'arbre visuel, soit à l'objet `RootVisual`, celui-ci diffuse l'événement `Loaded`. Il vous suffira donc d'écouter cet événement pour savoir si l'objet est ajouté à l'arbre visuel. Un événement semblable n'est pas diffusé lorsqu'un objet est supprimé de l'arbre visuel. Vous pouvez à la place utiliser l'événement `LayoutUpdate`. Toutefois, cet événement est diffusé lors de chaque modification d'agencement. Il est donc diffusé de nombreuses fois, même lorsqu'un objet est ajouté à l'arbre visuel. Voici une méthode pour vérifier qu'un enfant vient d'être supprimé de l'arbre visuel :

```
private void MainPage_Loaded(object sender, e)  
{  
    MonStackPanel.LayoutUpdated += MonStackPanel_LayoutUpdated;
```

```

    }

    private void MonStackPanel_LayoutUpdated(object sender, EventArgs e)
    {
        if (MonStackPanel.Children.Contains(MonBouton)
            { /* objet non supprimé */ }
        else
            { /* objet supprimé */ }
    }

```

Dans tous les cas, il vous faudra tester si l'objet a été supprimé. Ce n'est pas l'idéal d'un point de vue performance car l'événement `LayoutUpdate` est diffusé très souvent.

5.5 Échanges d'index

Nous avons abordé de nombreux mécanismes de l'arbre, mais nous n'avons pas pratiqué d'échanges d'index de la liste d'enfants. En fait, `Children` ne possède pas une méthode spécifique assurant directement cette opération. Nous allons étudier les différentes manières de procéder dans cette section.

5.5.1 Échange d'index du point de vue conception

Comme aucune méthode n'existe, nous allons coder notre échange d'index en prenant soin d'éviter les erreurs d'accès, d'ajout ou même de suppression que nous connaissons bien maintenant. Voici le code permettant d'effectuer cette opération délicate :

```

public partial class MainPage : UserControl
{
    public Button ClientMenu;
    public WrapPanel MenuHaut;

    public MainPage()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }

    private void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        menuHaut = LayoutRoot.Children[0] as WrapPanel;

        ContactBtn.Click += new RoutedEventHandler(ContactBtn_Click);
    }

    private void ContactBtn_Click(object sender, RoutedEventArgs e)
    {
        // on récupère l'un des enfants dont on veut échanger l'index
        Button Enfant1 = menuHaut.Children[1] as Button;

        // on stocke les index dont nous aurons besoin par la suite
        int IndexContact = menuHaut.Children.IndexOf( ContactBtn );

        // Ensuite on supprime les enfants de la liste
        menuHaut.Children.Remove( Enfant1 );
        menuHaut.Children.Remove( ContactBtn );
    }
}

```

```

        // Puis on les réinsère en respectant un ordre précis
        // toujours commencer par réinsérer l'élément
        // que l'on souhaite positionner à l'index le plus bas
        menuHaut.Children.Insert(1,ContactBtn);

        // pour finir, on ajoute à la liste d'enfant l'élément
        // qui a l'index le plus haut
        menuHaut.Children.Insert(IndexContact,Enfant1);
    }
}

```

Comme vous pouvez le voir, nous suivons un ordre vraiment très précis pour éviter au maximum les erreurs d'accès à la liste. Échanger l'ordre des enfants modifie l'arbre visuel, mais ne lui retire pas d'enfants. Nous aurons donc le même nombre d'enfants au départ de notre code qu'à l'arrivée. Il nous faut dans un premier temps stocker les index que nous souhaiterons échanger par la suite. Si nous le faisons après avoir supprimé un enfant de la liste, ceux-ci ne correspondraient pas au bon emplacement dans la liste.

La deuxième étape consiste à supprimer les enfants à échanger de la liste. L'ordre importe peu. Pour finir, on les réinsère, mais pas n'importe comment. Dans notre cas, nous avons échangé le dernier enfant de la liste avec un enfant situé en plein milieu. Si nous commençons par réinsérer l'élément le plus haut dans la liste, nous courons le risque de spécifier un index hors de la portée maximale qui correspond à notre nombre d'enfants, `Children.Count`. Le code suivant renvoie une erreur car nous essayons d'insérer `Enfant1` à un index qui n'est pas encore accessible :

```

private void ContactBtn_Click(object sender, RoutedEventArgs e)
{
    // on récupère l'un des enfants dont on veut échanger l'index
    Button Enfant1 = menuHaut.Children[1] as Button;

    // on stocke les index dont nous aurons besoin par la suite
    int IndexContact = menuHaut.Children.IndexOf( ContactBtn );

    // Ensuite on supprime les enfants de la liste
    menuHaut.Children.Remove( Enfant1 );
    menuHaut.Children.Remove( ContactBtn );

    menuHaut.Children.Insert(IndexContact,Enfant1);
    menuHaut.Children.Insert(1,ContactBtn);
}

```

Cette méthode est assez fastidieuse. Réaliser un échange d'index sans connaître à l'avance les objets à échanger nous forcerait également à prévoir toutes les erreurs possibles. Avoir une méthode de `UIElementCollection` qui accomplirait directement cette opération serait un plus. C'est ce que nous allons aborder dès maintenant.

5.5.2 Une méthode d'extension pour *UIElementCollection*

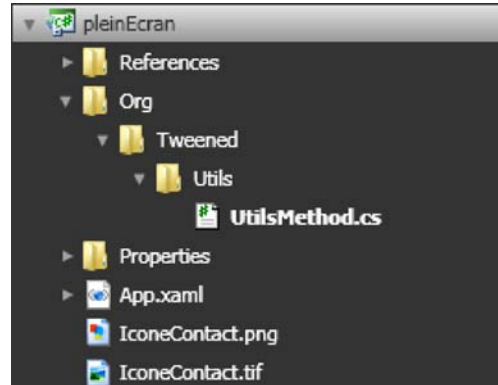
Les méthodes d'extension sont apparues avec C# 3. C'est un nouveau pas en avant pour rendre C# souple et faciliter l'ajout de méthodes personnalisées aux classes normalement fermées à l'extension. Voici la signature de la classe `UIElementCollection` :

```
public sealed class UIElementCollection :
    PresentationFrameworkCollection<UIElement>
```

Comme vous pouvez le constater, cette classe empêche tout héritage. Pourtant, les méthodes d'extension nous permettent de lui ajouter des fonctionnalités sans pour autant générer du code spaghetti. Cliquez-droit sur votre projet et sélectionnez Ajouter un nouvel élément. Dans la fenêtre qui vient de s'ouvrir, choisissez Class. Nommez-la *UtilsMethod.cs*, vous pouvez également la glisser dans un dossier spécifique pour éviter de mélanger les genres (voir Figure 5.24). La création de répertoires est accessible par un simple clic-droit sur le projet ou un autre répertoire.

Figure 5.24

*Arborescence de dossier pour le fichier *Utils.cs*.*



Comme vous pourriez avoir besoin de notre méthode plus tard, il vaut mieux inclure la future classe qui contiendra notre méthode dans un espace de nom spécifique. Voici le code contenant la définition de notre méthode d'extension :

```
namespace Org.Tweened.Utils
{
    public static class UtilsMethod
    {
        public static void SwapChildren(this UIElementCollection E,
                                         UIElement child1, UIElement child2 )
        {
            if ( E.Contains(child1) && E.Contains(child2) )
            {
                int Index1 = E.IndexOf(child1);
                int Index2 = E.IndexOf(child2);
                E.Remove(child1);
                E.Remove(child2);

                if (Index1>Index2)
                {
                    E.Insert(Index2,child1);
                    E.Insert(Index1,child2);
                }
                else
                {
                    E.Insert(Index1,child2);
                    E.Insert(Index2,child1);
                }
            }
            else
            {
            }
        }
    }
}
```

```

        throw new NotImplementedException("Au moins l'un des deux
            enfants n'est pas contenu dans la liste d'enfants.");
    }
}

```

Créer une méthode d'extension est relativement simple. L'objectif de ces méthodes est la décoration. Pas au sens artistique, bien sûr, mais d'un point de vue technique. Décorer signifie ajouter de nouvelles fonctionnalités et propriétés à un objet. La décoration est un concept permettant de résoudre de nombreuses problématiques de conception objet. Le mot-clé `static` est obligatoire aussi bien pour la classe que pour la méthode. Grâce à ce mot-clé, le simple fait de référencer notre espace de noms permet à la méthode d'extension d'être utilisée. La signature d'une méthode d'extension contient toujours au moins un premier argument commençant par `this`, celui-ci est suivi du type puis du nom du paramètre. Le mot-clé `this` indique au compilateur que le type qui le suit pourra utiliser la méthode. Les paramètres qui suivent sont quant à eux utilisés lors de l'appel. À ce moment, n'oubliez pas de référencer l'espace de noms *via* le mot-clé `using` :

```
MenuHaut.Children.SwapChildren(Enfant1, ContactBtn);
```

Vous n'aurez sans doute pas d'IntelliSense pour les méthodes d'extension au sein d'Expression Blend, mais vous en aurez au sein de Visual Studio. Compilez et testez votre application. Vous constatez que les boutons échangent bien leur place, toutefois contrairement à notre première version, la logique est complètement réutilisable au sein d'autres projets.

5.5.3 Échange d'index du point de vue design

Comme nous l'avons vu à la section 5.2, l'ordre d'imbrication est directement lié à l'ordre de superposition. Toutefois cela n'est pas forcément pratique car l'ordre d'imbrication est également lié aux contraintes de positionnement au sein d'un conteneur. Nous allons le démontrer à travers un mini exemple et trouver des solutions adaptées aux designers et aux développeurs.

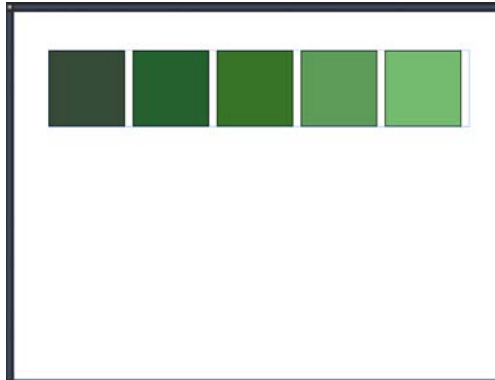
5.5.3.1 Créer le projet

Créez un nouveau projet de type application et nommez-le `SuperpositionOrder`. Dans le conteneur principal, créez un `StackPanel` avec une orientation horizontale. Celui-ci doit s'adapter à son contenu, vous devrez donc lui affecter une largeur et une hauteur en mode `Auto`. Faites en sorte qu'il soit centré horizontalement au sein de la grille principale. Ensuite, imbriquez à l'intérieur du `StackPanel` cinq objets `Rectangle` de différentes couleurs. Chacun doit faire 100 pixels de largeur par 100 pixels de hauteur. Pour finir, espacez-les en leur spécifiant une marge à droite (voir Figure 5.25).

Il faudrait agrandir un rectangle afin de voir comment ceux-ci se superposent, par exemple celui du milieu. Ceci n'est pas aussi facile à faire qu'il n'y paraît.

Figure 5.25

Exercice pratique
de superposition.

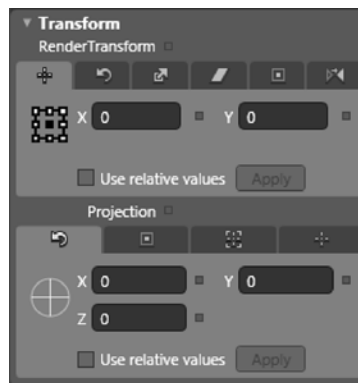


5.5.3.2 Introduction aux RenderTransform

Notre problème est simple : agrandir la largeur décalera les autres rectangles les uns par rapport aux autres car la largeur et la hauteur sont liées au comportement d'empilement du `StackPanel`. C'est une impasse si nous nous contentons de ces propriétés. Heureusement un autre type de transformation existe, les transformations vectorielles. Nous aborderons en profondeur ces transformations au Chapitre 6. Pour l'instant, nous nous contentons de les utiliser. Sélectionnez le rectangle du milieu, puis dans le panneau des propriétés, ouvrez l'onglet `Transform` (voir Figure 5.26).

Figure 5.26

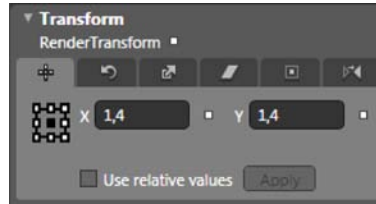
L'onglet des
transformations.



Cet onglet vous propose deux types de transformation : les transformations vectorielles de type `RenderTransform` et les transformations de type `Projection` qui permettent de gérer l'affichage d'objets en pseudo 3D (voir Chapitre 9). Nous allons utiliser les transformations vectorielles pour nous libérer partiellement des contraintes d'agencement propres au `StackPanel`. Cliquez sur l'icône de transformation d'échelle (📐), puis saisissez la valeur 1,4 dans les deux champs de saisie (voir Figure 5.27).

Figure 5.27

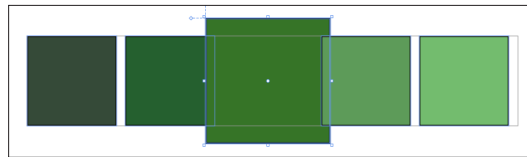
L'onglet des transformations d'échelle.



Les transformations de type `RenderTransform` sont totalement indépendantes de l'agencement imposé par le contexte conteneur. Elles permettent donc aux graphistes de s'affranchir des contraintes liées à la conception et à l'architecture. Notre `Rectangle` mesure désormais 40 % de largeur et de hauteur de plus et il ne décale plus les autres (voir Figure 5. 28).

Figure 5.28

Rectangle avec changement d'échelle.



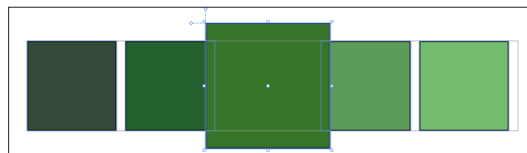
5.5.3.3 La propriété `ZIndex`

Comme vous pouvez le constater, notre `Rectangle` étant en troisième position dans l'ordre d'imbrication XAML, il apparaît au-dessus du rectangle à sa gauche et sous le rectangle à sa droite. Nous allons l'afficher au-dessus des autres rectangles. Sur Internet, vous trouverez de nombreux exemples dans lesquels un menu se superpose aux autres lors du survol de la souris. Pour réaliser ce type de menu, il faudrait nous affranchir, une nouvelle fois, des contraintes liées à l'imbrication.

Tous les conteneurs pouvant contenir plus d'un enfant possèdent la propriété attachée `ZIndex`, fournie par la classe `Canvas`. Par défaut, la propriété `ZIndex` contient la valeur 0. Comme tous les enfants d'un conteneur possèdent `ZIndex` à 0, c'est l'ordre d'imbrication XAML qui prime dans un premier temps. Il existe ainsi une compétition constante entre l'ordre d'imbrication, soit l'index de l'enfant dans la collection `Children`, et la propriété `ZIndex`. Lorsque deux enfants possèdent le même `ZIndex`, l'ordre de superposition est déterminé par l'index enfant de chacun. Au contraire, s'ils possèdent un `ZIndex` différent, l'ordre de superposition est défini par le `ZIndex`. La valeur `ZIndex` la plus haute représente l'objet affiché au-dessus des autres. Toutefois, cette propriété n'exerce son influence qu'au sein d'un même conteneur. Il nous faut la modifier pour le rectangle central. Pour cela, il vous suffit d'ouvrir le panneau des propriétés, dans les options d'agencement, puis de passer sa valeur à 1 (voir Figure 5.29).

Figure 5.29

Rectangle avec changement d'échelle et modification de la propriété `ZIndex`.



Pour mieux comprendre la compétition entre l'index enfant et le `ZIndex`, vous pouvez sélectionner tous les rectangles et passer leur `ZIndex` à 1. Notre rectangle du milieu passe à nouveau en dessous du rectangle de droite. Les `ZIndex` étant tous égaux, c'est l'ordre d'imbrication XAML qui prime à nouveau.

Vous êtes maintenant familiarisé avec les différents composants visuels proposé par le framework Silverlight. Dans le prochain chapitre, nous étudierons les mécanismes liés à l'animation propres à la plateforme Silverlight. Nous verrons en quoi Silverlight se révèle être un puissant moteur d'animation vectorielle. Nous aborderons donc la création d'animations dans Expression Blend ou avec C#.

Partie II

Interactivité et création graphique

6

Animations

Dans ce chapitre, nous étudierons en profondeur les mécanismes d'interpolations vectorielles propres à Silverlight. Pour cela, nous allons apprendre les principes fondamentaux de l'animation. Nous utiliserons notre projet de site plein écran comme point de départ pratique pour acquérir les concepts de base, ainsi qu'une aisance technique. Nous aborderons tout d'abord l'animation du point de vue d'un designer interactif *via* Blend. Dans un second temps, nous verrons que C# facilite et dynamise la création d'animations, mais qu'il simplifie et optimise également le flux de production sans délaissier le travail des designers ou des animateurs. Nous apprendrons en quoi les transformations relatives sont efficaces et incontournables en matière d'animation et comment les générer à l'exécution. Pour finir, nous aborderons et utiliserons le gestionnaire d'états visuels dont la bonne compréhension repose sur l'ensemble des notions apprises auparavant. Nous verrons quel est son impact en matière de conception applicative ou de flux de production dans notre quotidien.

6.1 Introduction

Silverlight est, entre autres, un moteur d'animations vectorielles. Depuis sa version 3, il permet de gérer la projection d'objets vectoriels en pseudo 3D. Cela signifie qu'il est capable de représenter un objet au sein des quatre dimensions que nous connaissons tous, soit x , y , z (la profondeur) et t pour le temps. Dans ce chapitre, nous n'étudierons pas les animations dans un espace 3D, mais uniquement 2D pour des raisons de clarté (si besoin, vous pouvez consulter le Chapitre 9 dédié à la 3D).

6.1.1 Qu'est-ce qu'une animation ?

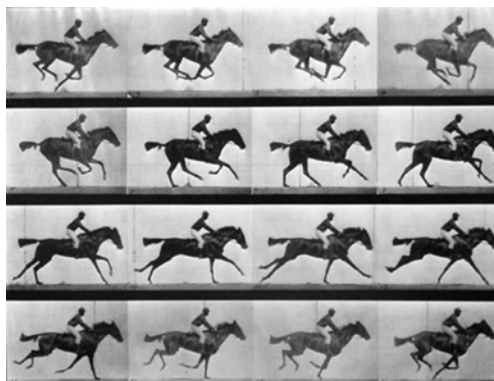
Quel que soit l'environnement de développement, créer une animation revient toujours à modifier la valeur d'une propriété d'un objet au cours du temps. Nous verrons à la section 6.3.2 que la valeur de départ peut-être récupérée implicitement. Cette particularité propre au lecteur Silverlight permet des comportements très puissants et beaucoup de souplesse de conception.

Dans le langage courant, une animation est forcément fluide, mais c'est un raccourci un peu rapide et incorrect. Gardez à l'esprit que c'est avant tout une propriété d'objet qui varie au cours du temps quel que soit le laps de temps écoulé entre deux valeurs de cette propriété. Ainsi, le rebond d'une balle se résume aux variations de ses coordonnées dans l'espace au cours du temps.

La nature offre de nombreux exemples d'animations invisibles à nos yeux, comme l'érosion d'une montagne ou la croissance d'un arbre. Les propriétés de ces objets évoluent tellement lentement dans le temps que voir ces phénomènes est simplement impossible à l'œil nu. Certains mouvements sont tellement rapides qu'il est également difficile de les analyser. Le galop du cheval en est un exemple flagrant, il ne fut réellement compris scientifiquement par Étienne Jules Marey et Eadweard Muybridge qu'à la fin du XIX^e siècle. Le naturaliste Étienne Jules Marey pensait qu'au grand galop aucune des pattes du cheval ne touchait le sol en même temps durant un cours instant. Cela est vrai, mais le démontrer n'était pas si simple. Pour le prouver, Eadweard Muybridge a découpé son mouvement à l'aide d'appareils photographiques alignés les uns à côté des autres. Les appareils photographiques de ces scientifiques étaient déclenchés de manière décalée dans le temps selon un intervalle régulier (voir Figure 6.1).

Figure 6.1

Découpage du galop d'un cheval par Muybridge.



La décomposition et l'étude des mouvements amena de grandes découvertes et perspectives. Ce laps de temps constant entre chaque photographie a introduit la notion de cadence de prise de

vues. Plus ce laps est court, plus la cadence est élevée. Par la suite, Muybridge inventa le premier appareil permettant d'afficher les images rapidement les unes après les autres. Cette invention préfigurait sans doute les prémisses du cinéma et de l'animation. Dans la même veine, les premiers dessins animés Disney étaient en huit images par seconde car cela évitait un travail de dessin trop fastidieux. Toutefois, l'œil humain ne percevant plus l'effet de saccade à partir de 24 images par seconde, ces 8 images par seconde étaient perçues par les spectateurs malgré le talent des animateurs. Les premiers films des frères Lumière connaissaient des limites équivalentes. Les cadences d'affichage, de prise de vues et d'animations furent, durant au moins 50 ans, le centre de nombreux efforts de la part des cinéastes et techniciens car celles-ci engendrent l'illusion du mouvement et son réalisme. Aujourd'hui encore, leur maîtrise permet de concevoir des effets visuels impressionnants et de mieux connaître notre environnement naturel.

6.1.2 La cadence des animations au sein de Silverlight

Au sein de Silverlight, la cadence des animations est de 60 images par seconde par défaut. Le laps de temps s'écoulant théoriquement entre deux images affichées est donc de 1/60 de seconde. Cette valeur est accessible à travers la propriété `MaxFrameRate` de l'instance `Silverlight`. Vous pouvez également afficher la cadence en bas de la fenêtre du navigateur *via* la propriété `EnableCounterFrameRate` à l'instanciation du lecteur `Silverlight` :

```
<div id="silverlightControlHost">
  <object data="data:application/x-silverlight," type="application/
    x-silverlight-2" width="100%" height="100%">
    <param name="source" value="ClientBin/sitet.xap"/>
    <param name="onerror" value="onSilverlightError" />
    <param name="background" value="white" />
    <param name="minRuntimeVersion" value="3.0.40128.0" />
    <param name="autoUpgrade" value="true" />
    <param name="maxFrameRate" value="30" />
    <param name="enableFrameRateCounter" value="true" />
    <a href="http://go.microsoft.com/fwlink/?LinkId=141205"
      style="text-decoration: none;"> 
    </a>
  </object>
```

Vous pouvez également définir ces deux paramètres à l'exécution en C# :

```
public MainPage()
{
    InitializeComponent();
    Loaded += new System.Windows.RoutedEventHandler(MainPage_Loaded);
    MouseLeftButtonDown += MainPage_MouseLeftButtonDown;
}

private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    App.Current.Host.Settings.EnableFrameRateCounter = true;
}

private void MainPage_MouseLeftButtonDown(object sender,
                                          MouseButtonEventArgs e)
{
    App.Current.Host.Settings.MaxFrameRate = 30;
}
```

Toutefois, comme dans n'importe quel autre moteur d'animation, et ainsi que vous pouvez le constater grâce à la propriété `EnableFrameRateCounter`, la cadence n'est pas constante. Ceci est dû aux algorithmes de calcul et aux fluctuations de performance des systèmes d'exploitation, mais également à de nombreux autres facteurs, comme l'occupation mémoire et processeur à l'instant où l'animation est visionnée. La cadence maximum n'est en réalité rien d'autre qu'une valeur souhaitée idéale. Silverlight n'est volontairement pas conçu pour gérer les animations traditionnelles de prime abord, c'est-à-dire les animations image par image. Il est orienté vers la diffusion d'applications riches avant tout.

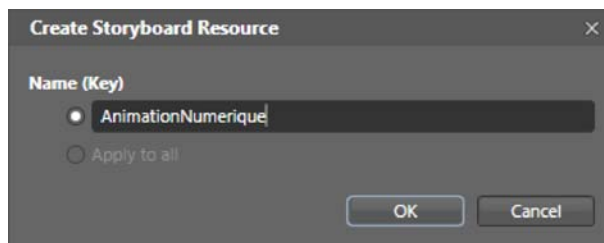
Il en découle deux spécificités essentielles propres à Silverlight. La première est que l'unité de temps est exprimée en secondes et non en images affichables comme sous Flash. La seconde particularité est que Silverlight dispose d'un moteur vectoriel "autodégradable". Cela signifie que si une animation est trop lourde pour être affichée correctement selon sa cadence, dans tous les cas, le moteur respectera au maximum le temps qui lui est imparti, même s'il lui faut ignorer l'affichage d'images intermédiaires de l'animation. Autrement dit, même saccadée, une animation ne dépassera pas une durée définie, car Silverlight privilégie le temps plutôt que l'affichage de toutes les étapes. De manière générale, l'œil perçoit l'intervalle entre deux images si la cadence est inférieure à 25 images par seconde. Il vaut donc mieux éviter les cadences trop faibles pour ne pas créer un effet saccadé disgracieux.

6.1.3 Une première animation

Vous allez maintenant créer votre première animation. Pour cela, ouvrez Blend et créez un projet de type application. Nommez-le `PremiereAnimation` par exemple. Nous allons commencer par un peu de pratique, puis nous examinerons ce qui a été produit côté XAML par Blend. Vous pouvez passer en espace de travail Animation *via* le raccourci F6. Dans le conteneur `LayoutRoot`, créez un champ texte de type `TextBlock`, puis entrez la chaîne de caractères `animation d'une valeur numérique` dans sa propriété `Text`. Nous allons créer une nouvelle animation de champ texte. Pour cela cliquez sur l'icône plus (+) dans le panneau au-dessus de l'arbre visuel et logique. Une nouvelle fenêtre s'ouvre, vous proposant de nommer la nouvelle animation que vous souhaitez créer. Nommez-la `AnimationNumerique` (voir Figure 6.2).

Figure 6.2

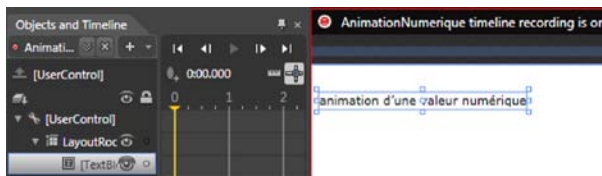
Fenêtre de création d'une nouvelle animation.



Vous remarquez que créer une nouvelle animation revient à créer une nouvelle ressource de type Storyboard. Cliquez sur OK. L'interface de Blend affiche désormais une ligne de temps exprimée en secondes. Celle-ci est située à droite de l'arbre visuel et logique. Chaque élément de l'arbre visuel peut posséder sa propre animation qui sera accessible dans la fenêtre de la ligne de temps. Une animation peut cibler une ou plusieurs propriétés. Un cadre rouge entoure la fenêtre de design (il est représenté en noir à la Figure 6.3).

Figure 6.3

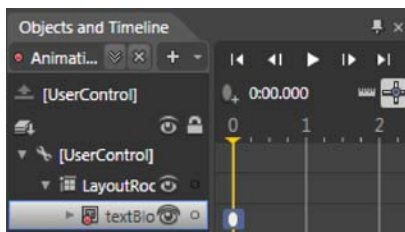
Arbre visuel attendant au panneau de la ligne du temps.



Dans la partie gauche de l'interface, l'encadré rouge indique que toute modification des objets entraînera la création d'une nouvelle clé d'animation ou la modification d'une clé existante. Attention à ne pas changer le type de valeur contenu par la propriété. Un dégradé ne peut pas être interpolé en couleur unie. Il faut un dégradé au départ et en fin d'animation. Déplacez l'instance du `TextBlock` de la gauche vers la droite. Dès que vous avez réalisé cette étape, une nouvelle clé d'animation est créée à la seconde 0. Elle est représentée par un ovale gris clair, à droite du champ texte (voir Figure 6.4).

Figure 6.4

Une nouvelle clé d'animation est créée à la seconde 0.



Lorsqu'un objet possède une clé d'animation, un point rouge apparaît en bas à gauche de son icône dans l'arbre visuel et logique.

ATTENTION

Une clé d'animation indique une modification de la valeur d'une propriété à un instant donné. Créer une clé d'animation alors qu'aucun changement ne survient peut être utile si vous poursuivez un but précis. Toutefois, évitez de polluer la ligne de temps en ajoutant des clés n'indiquant aucun changement de propriété.

Déplacez maintenant la tête de lecture à la seconde 2, représentée par une ligne jaune surmontée d'un triangle. La position de la tête de lecture permet d'afficher les objets vectoriels à un instant donné de l'animation. La déplacer revient à faire un arrêt sur image à n'importe quel instant de l'animation. Sélectionnez le champ texte, puis repositionnez-le n'importe où ailleurs au sein du conteneur `LayoutRoot`. Vous générez à nouveau une image clé, mais cette fois elle est positionnée à la seconde 2. L'image clé est automatiquement créée car Blender est en mode enregistrement d'animation. Vous pouvez voir le résultat de vos actions en jouant l'animation. Pour cela, il suffit de cliquer sur le bouton de lecture situé au-dessus de la ligne de temps (voir Figure 6.5).

L'aperçu de la lecture est moins performant que celui que vous obtiendrez en compilant le projet. Toutefois, compiler le projet ne déclenche pas l'animation au chargement. Il vous faudra déclencher l'animation à l'exécution, soit par C#, soit grâce aux comportements (voir section 6.2). Comme nous l'avons vu jusqu'à présent, tout ce qui est créé par le designer ou l'intégrateur au sein

de Blend est traduit en langage XAML. Nous allons maintenant étudier ce qui a été produit, ainsi que les classes de la plateforme Silverlight utilisées pour animer les objets.

Figure 6.5

Contrôler une animation sous Blend.



6.1.4 Les différents types d'animation

Fermez l'animation en cours pour éviter de la modifier. Il suffit pour cela de cliquer sur l'icône de fermeture de l'animation (✕) située à droite du nom de l'animation dans l'arbre visuel et logique. Lorsque vous avez créé l'animation, elle a été ajoutée en tant que *ressource*. Une ressource est un type d'objet particulier dont le but est de pouvoir être accessible et réutilisable au sein de votre projet. Il existe plusieurs types de ressources, mais elles possèdent toutes une portée d'utilisation. Dans le cas des animations, les ressources sont en majorité définies comme ressource de l'élément visuel le plus haut dans l'arbre visuel. Dans notre cas, l'animation sera stockée dans le composant UserControl racine. Elle sera donc utilisable et accessible à l'intérieur du nœud XAML UserControl racine. Voici le code XAML correspondant à notre animation :

```
<UserControl.Resources>
  <Storyboard x:Name="AnimationNumerique">
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
      TargetName="textBlock" Storyboard.
      TargetProperty="(UIElement.RenderTransform).
        (TransformGroup.Children)[3].(TranslateTransform.X)">
      <EasingDoubleKeyFrame KeyTime="00:00:00" Value="-25"/>
      <EasingDoubleKeyFrame KeyTime="00:00:02" Value="75"/>
    </DoubleAnimationUsingKeyFrames>

    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
      TargetName="textBlock" Storyboard.
      TargetProperty="(UIElement.RenderTransform).
        (TransformGroup.Children)[3].(TranslateTransform.Y)">
      <EasingDoubleKeyFrame KeyTime="00:00:00" Value="-20"/>
      <EasingDoubleKeyFrame KeyTime="00:00:02" Value="60"/>
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</UserControl.Resources>
```

Comme vous le constatez, l'animation est représentée par une ressource de type Storyboard. Elle possède un nom à l'instar des objets vectoriels que nous avons utilisés. À l'intérieur du nœud élément Storyboard, deux séquences d'animation de type DoubleAnimationUsingKeyFrames cohabitent. L'une cible la propriété X du nœud RenderTransform, l'autre la propriété Y de ce même nœud (voir section 6.3). Un nœud DoubleAnimationUsingKeyFrames décrit une animation d'une propriété de type Double par l'utilisation de clés d'animations. Comme nous le verrons dans ce chapitre, un changement peut être défini autrement que par des clés d'animation. De plus, il est possible d'animer des types non numériques. Pour le mettre en évidence, vous pouvez modifier la couleur du champ texte (propriété Foreground) à la seconde 2. Il vous faut au préalable accéder

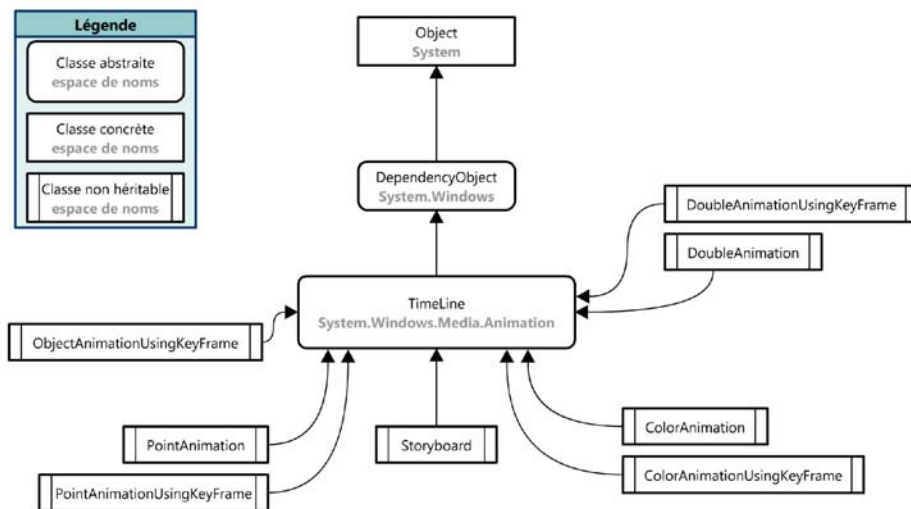
à l'animation pour la modifier. Pour cela, cliquez sur l'icône listant les animations créées (📄). Une fenêtre apparaît, sélectionnez l'animation. Une fois la couleur du champ texte modifiée, vous obtiendrez l'équivalent du code XAML généré ci-dessous :

```
<Storyboard x:Name="AnimationNumerique">
    ...
    <ColorAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
        TargetName="textBlock" Storyboard.TargetProperty=
            "(TextBlock.Foreground).(SolidColorBrush.Color)" >
        <EasingColorKeyFrame KeyTime="00:00:00" Value="Black" />
        <EasingColorKeyFrame KeyTime="00:00:02" Value="Red" />
    </ColorAnimationUsingKeyFrames>
</Storyboard>
```

Cette fois, Blend a généré une sous-séquence d'animation de type `ColorAnimationUsingKeyFrames` au sein de l'objet `Storyboard`. Vous n'animez plus de valeurs numériques, mais des valeurs de type `Color`. La Figure 6.6 liste une partie des classes et types d'animation permettant d'animer des objets au sein de Silverlight.

Figure 6.6

Les classes utilisées pour animer des objets.



La classe `Storyboard` assure le contrôle de tous les types d'animation, leur lecture ou leur pause, par exemple. De ce fait, vous pouvez considérer l'objet `Storyboard` comme l'unité d'organisation principale. L'interpolation des valeurs entre deux instants est gérée par les autres classes. Quatre grands types de valeurs peuvent être animés :

- `Double` représente les valeurs de types numériques. L'opacité, par exemple, sera animée grâce à la classe `DoubleAnimation`, car celle-ci est typée `Double`.

- **Point** correspond à une structure constituée d'une paire de valeurs nommées *X* et *Y* et de type *Double*. Pour résumer, il représente les coordonnées d'un point dans l'espace. Lorsque vous animez les sommets d'un tracé vectoriel (de type *Path*), en interne le XAML généré correspondra à *PointAnimation*.
- **Color** fait référence à toutes les classes utilisant la classe *Color*. Ainsi, animer les couleurs d'un dégradé linéaire (*LinearGradientBrush*) repose sur *ColorAnimation*.
- **Object** renvoie à n'importe quel type. *Silverlight* vous permet d'animer n'importe quel type de valeurs différentes des précédentes. Toutefois interpoler la visibilité (*Visibility*) ou une chaîne de caractères (*string*) ne produira pas une animation fluide.

Nous allons maintenant essayer de comprendre quels types d'interpolation sont fournis par *Silverlight* et comment les concevoir.

6.1.5 Les différents types d'interpolation

Comme vous pouvez le constater à la Figure 6.6, mis à part pour les animations ciblant les valeurs *Object*, deux grandes familles d'interpolations cohabitent pour chaque type. La première famille utilise des clés d'animation. La seconde possède une propriété pour la valeur de départ et une autre pour la valeur d'arrivée, cette dernière famille n'utilise donc pas de clé d'animation.

6.1.5.1 Interpolations par clé d'animation

Une clé d'animation est une classe constituée au minimum des propriétés *Value* et *KeyTime*. La propriété *Value* indique la valeur de la propriété animée. *KeyTime* définit l'instant auquel la propriété doit atteindre la valeur spécifiée :

```
<EasingColorKeyFrame KeyTime="00:00:00" Value="Black" />
<EasingColorKeyFrame KeyTime="00:00:02" Value="Red" />
```

Ainsi dans le code ci-dessus, à la seconde 0, la propriété animée est affectée de *Black* (noir) alors qu'à la seconde 2 celle-ci doit atteindre la valeur *Red* (rouge). Parce qu'il existe quatre grands types d'animation (voir section 6.1.4), il existe quatre grands types de clés d'animation (voir Figure 6.7).

Une classe abstraite représente chaque type de clé. Pour chaque type de clé (mis à part les clés faisant référence à l'animation *ObjectAnimation*), il existe quatre manières de jouer l'animation. Au sein de *Blend*, vous pouvez choisir le type d'interpolation en cliquant sur une image clé *via* le panneau des propriétés (voir Figure 6.8).

Par défaut, les clés d'animations sont de type *Easing*, c'est-à-dire qu'elles utilisent des équations d'accélération pour interpoler les valeurs entre elles. Par exemple, lors de la création de notre première animation, celle-ci utilisait une accélération linéaire :

```
<EasingColorKeyFrame KeyTime="00:00:00" Value="Black" />
<EasingColorKeyFrame KeyTime="00:00:02" Value="Red" />
```

Figure 6.7

Les différents types d'images clés et d'interpolation.

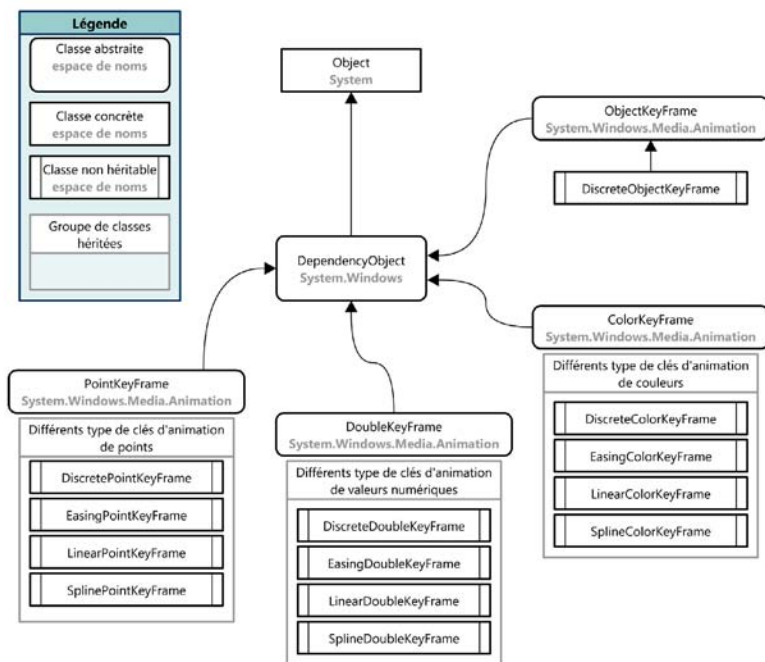
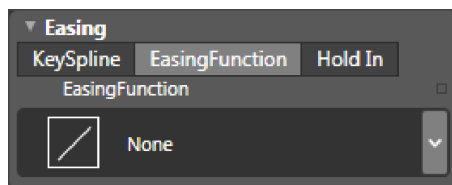


Figure 6.8

Choisir le type d'interpolations dans Blend.



Lorsqu'aucune accélération n'est définie, celle-ci est linéaire (Linear). Vous trouverez les différents types d'interpolation ci-dessous :

- **Discrete**. N'interpole pas les valeurs entre deux clés d'animation. L'animation est jouée de manière brutale. La propriété ciblée par l'animation ne change de valeur (Value) qu'à l'instant où la tête de lecture arrive au temps (KeyTime) défini dans l'image clé. L'animation n'est pas fluide mais saccadée. Il faudra cliquer sur l'onglet **Hold In** pour utiliser ce type de clé.
- **Easing**. Les clés préfixées par ce mot possèdent en plus une propriété (EasingFunction) permettant de définir une équation d'accélération. C'est exactement ce type que vous utiliserez pour faire rebondir une balle par exemple. Nous reviendrons sur ce type de clé à la section 6.3.
- **Linear**. Dans la vie réelle, aucun objet ne possède de mouvement linéaire. Plus vous utiliserez ce type de clé et moins votre animation sera intéressante car prévisible. Vous aurez parfois besoin de ce type d'animation pour les rotations par exemple. Toutefois, les animateurs traditionnels les évitent le plus possible. Il n'y a pas de moyen simple dans Blender de spécifier ce type de clé. On utilise plutôt une accélération linéaire – ce qui revient au même résultat.

- **Spline.** Vous pouvez définir vous-même la courbe d'accélération manuellement, ce qui peut être plus précis dans certains cas. Pour gérer vous-même l'accélération, cliquez sur l'onglet KeySpline (voir section 6.3).

6.1.5.2 Interpolations sans clé d'animation

Silverlight apporte beaucoup de souplesse et de facilité dans la création de transitions grâce aux propriétés **From** correspondant à la valeur de départ, **To** qui est la valeur de destination, et **By** une valeur de destination relative à la valeur de départ. Ces propriétés sont utilisables par les animations qui n'emploient pas de clés (indiquées par le suffixe **UsingKeyFrames**), soit :

- **ColorAnimation**
- **DoubleAnimation**
- **PointAnimation**

Voici une manière d'écrire une telle animation. Remplacez le XAML décrivant l'animation de positionnement X et Y par le code suivant :

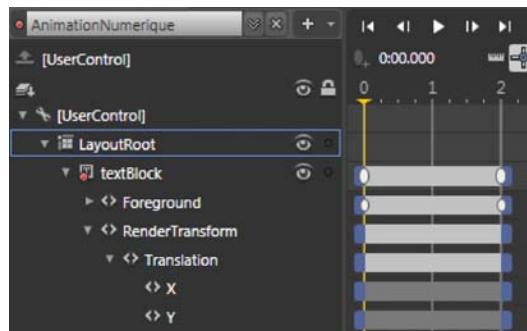
```
<DoubleAnimation BeginTime="00:00:00" From="-25" To="75"
    Duration="00:00:02" Storyboard.TargetName="textBlock" Storyboard.
    TargetProperty="(UIElement.RenderTransform).(TransformGroup.Children)
    [3].(TranslateTransform.X)" />

<DoubleAnimation BeginTime="00:00:00" From="-20" To="60"
    Duration="00:00:02" Storyboard.TargetName="textBlock" Storyboard.
    TargetProperty="(UIElement.RenderTransform).(TransformGroup.Children)
    [3].(TranslateTransform.Y)" />
```

Comme aucune propriété **KeyTime** n'est précisée, il vous faut définir la durée de l'animation *via* la propriété **Duration**. Ce type d'animation ne permet de définir qu'une valeur de départ et d'arrivée. Dans ce cas, Blend n'affiche pas de clés. Dépliez l'arborescence des objets animés pour visualiser les propriétés animées (voir Figure 6.9).

Figure 6.9

Affichage d'une animation sans clé d'animation.



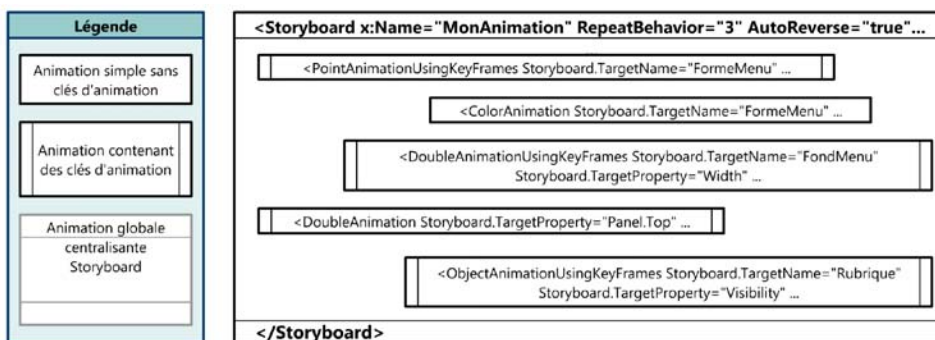
Blend ne crée ce type d'animation que dans le cas où vous utilisez le gestionnaire d'états visuels (voir section 6.4).

6.1.6 La classe Storyboard

Une instance de la classe Storyboard représente l'enveloppe de l'animation. Chaque occurrence de Storyboard peut avoir plusieurs sous-séquences d'animation distinctes possédant des comportements et des propriétés qui leurs sont propres. Par exemple, l'une de ces séquences ciblera un rond et l'autre un bouton. L'une pourra débuter à la seconde 2 du Storyboard, et l'autre ciblera une propriété de type couleur. Toutes les combinaisons sont possibles. Les animations sont conçues comme des objets indépendants complètement séparés de l'aspect visuel des objets. Autrement dit, elles n'appartiennent à aucun élément graphique, mais peuvent cibler n'importe lequel d'entre eux (voir Figure 6.10).

Figure 6.10

Principe du Storyboard.



La classe Storyboard possède des propriétés communes aux autres classes héritant de `Timeline`. Toutefois, c'est la seule classe possédant des méthodes de contrôle de l'animation, dont voici une liste :

- **Begin.** Cette méthode permet de lire l'animation depuis la seconde 0. À chaque appel de la méthode, le Storyboard est joué depuis le début.
- **GetCurrentState.** Cette méthode renvoie l'état actuel du Storyboard, s'il est en cours de lecture ou s'il est stoppé.
- **GetCurrentTime.** Retourne une valeur de type `TimeSpan` correspondant au temps écoulé depuis le départ de l'animation, et cela à l'instant où la méthode est invoquée.
- **Pause.** Met la lecture en pause.
- **Resume.** Redémarre une animation mise en pause.
- **Seek.** Cette méthode attend une instance de `TimeSpan` afin de positionner la tête de lecture à un temps donné de l'animation.
- **Stop.** Arrête la lecture du Storyboard et repositionne la tête de lecture à la seconde 0.

Les trois méthodes `SetTarget`, `SetTargetName` et `SetTargetProperty` sont statiques et définies dans la classe Storyboard. Elles sont donc invoquées par celle-ci. Elles sont très importantes car elles montrent à quel point le modèle d'animation de Silverlight est souple. Une animation au sein de la plateforme Silverlight est un objet indépendant de l'objet animé ou de la proprié-

té ciblée. Ces trois méthodes prennent en premier paramètre une instance de type `Timeline`, donc tout type d'animation. Les animations de type `ColorAnimation`, `Storyboard` ou `DoubleAnimationUsingKeyFrame` entre autres en font partie. En second paramètre, il vous faudra passer une valeur attendue par la méthode. La méthode `SetTarget` attend, par exemple, une instance de `DependencyObject` comme second argument.

- `SetTarget`. Précise la référence de l'objet graphique à animer.
- `SetTargetName`. Définit le nom de la cible à animer, elle est à utiliser côté XAML.
- `SetTargetProperty`. Cible la propriété à interpoler sur l'objet ciblé, par exemple l'opacité (`Opacity`) ou la largeur (`Width`).

Ces méthodes statiques sont très utiles. Prenons le cas d'un designer interactif. Il réalise une animation sous Blend pour un menu spécifique. L'un de ses collègues développeur souhaite réutiliser son animation pour la réaffecter à d'autres menus. Afin d'éviter un travail fastidieux de recopie et de réaffectation au sein de Blend, le développeur récupère la référence du `Storyboard` du designer, puis il réaffecte dynamiquement, comme cible, une autre instance de `DependencyObject` au `Storyboard` :

```
Storyboard.SetTarget( MonAnimationCool, MonNouveauMenu ) ;  
MonAnimationCool.Begin() ;
```

Nous allons maintenant passer à la pratique afin d'assimiler tous les concepts et principes que nous avons appris. Cependant, n'oubliez pas que Blend est un outil de mise en forme XAML. La nature profonde du XAML, basée sur les relations familiales, rend impossible la gestion de tous les cas d'imbrication ou d'écriture directement par l'interface graphique de Blend. Parfois, écrire ou copier-coller du XAML dans l'éditeur de code sera plus pratique et rapide, voire incontournable.

6.2 Animer avec Expression Blend

6.2.1 Les bonnes pratiques

Lorsque vous souhaitez animer un visuel, plusieurs bonnes pratiques importantes doivent être prises en compte. Si vous avez déjà de bonnes notions en animation traditionnelle, vous pouvez ignorer ce passage. Dans tous les cas, les règles sont faites pour être outrepassées, mais elles servent de cadre et évoluent avec les mœurs. Voici certaines d'entre elles :

- Évitez d'animer trop d'objets en même temps.
- Arrangez-vous pour que l'œil ne soit pas accroché à trop d'endroits en même temps. Essayez toujours de définir un sens de lecture et une unité d'animation.
- Les animations de menus ne doivent pas excéder une seconde. Dans le cas contraire, les animations deviennent prévisibles et pesantes lors d'une utilisation prolongée.
- Évitez autant que possible les animations linéaires. Rien n'est linéaire dans la nature, seules les mathématiques décrivent de tels mouvements. Cela donne un aspect plat, monotone et robotique aux animations. Parfois un simple ralenti fait des merveilles.

- Simplifiez vos animations et concentrez-vous avant tout sur les sensations que vous souhaitez créer. L'animation doit être complètement intégrée dans le site ou l'application et en prolonger l'unité graphique.
- Les transitions sont aussi importantes que les interfaces, elles représentent un lien logique et sensible entre chaque interface. Soignez-les particulièrement, mais restez sobre. Silverlight cible avant tout les applications riches.
- Essayez d'introduire des éléments inattendus au cours de vos animations. Cela ajoute de la profondeur à vos applications et l'utilisateur voudra les explorer un peu plus pour découvrir de nouveaux détails.
- Rythmez vos animations.

6.2.2 Créer une animation d'introduction

Dans Blend, ouvrez le projet `SiteAgencePortfolio` créé dans les précédents chapitres. Nous allons l'utiliser pour affiner notre compréhension du modèle d'animation à travers la création d'une animation d'introduction. Vous trouverez également ce projet dans l'archive *pleinEcran_maquette.zip* du dossier *chap6* des exemples du livre.

Nous allons découper les étapes de notre animation afin de l'enrichir. Cela permet de réfléchir et d'imaginer le tempo facilement sans pour autant nous plonger directement dans l'aspect technique de la réalisation. Voici chaque étape de notre animation d'introduction :

1. Rien n'est présent sur l'écran au lancement.
2. Les menus du haut apparaissent les uns après les autres avec un léger effet de déplacement vertical (du haut vers le bas). La durée totale de cette partie de l'animation ne doit pas excéder une seconde.
3. Le centre du site apparaît au milieu de l'écran.
4. Les liens en bas apparaissent comme s'ils émergeaient du cadre central en 4/10 de seconde environ.
5. Le bouton plein écran apparaît pratiquement en même temps.

En totalité, l'animation ne doit pas excéder trois secondes. Créez une nouvelle animation et nommez la ressource `Storyboard AnimIntro`. Vous n'êtes pas obligé de créer les séquences d'animation dans l'ordre chronologique. Vous allez commencer par créer l'apparition du centre de la page. Sélectionnez le composant `Border` et nommez-le `ContenuPage`.

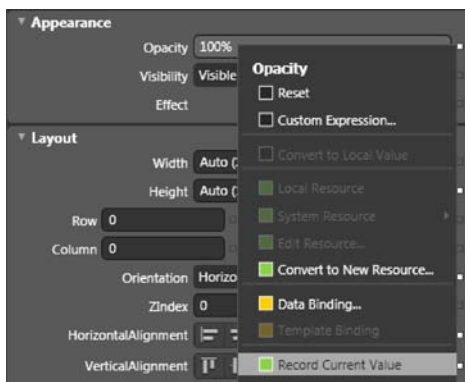
INFO

Nous nommons l'objet car en XAML une animation ne peut cibler que des objets nommés. Si vous l'animez sans lui donner de nom, Blend le nommera à votre place. Cela est gênant car le nom donné sera trop générique et peu explicite.

Une fois nommé, définissez son opacité à 0 %. Il s'agit bien de pourcentage car la plage de valeurs acceptées en C# ou en XAML se situe entre 0 et 1. Positionnez maintenant la tête de lecture à la seconde 1,6, puis cliquez sur l'icône carrée à droite de la propriété `Opacity`. Un nouveau menu apparaît. Cliquez sur l'option "Record Current Value" (voir Figure 6.11).

Figure 6.11

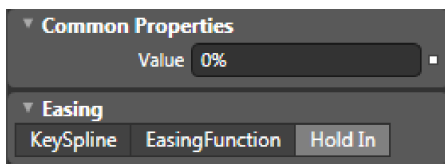
Création d'une nouvelle clé d'animation avec la valeur courante d'une propriété.



Vous venez de créer une nouvelle clé d'animation à la seconde 1,6. Cela signifie que durant la première clé et la seconde, l'opacité ne change pas et reste à zéro. Comme l'animation n'a pas besoin d'être fluide, vous pouvez également sélectionner la seconde clé, puis choisir l'option Hold In pour en faire une clé d'animation de type Discrete (voir Figure 6.12).

Figure 6.12

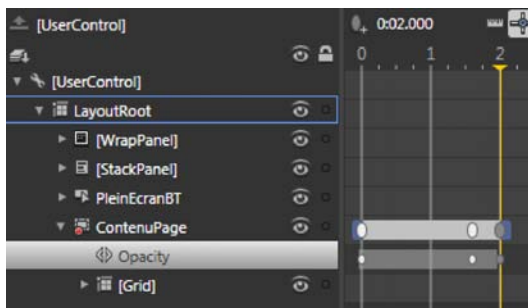
Clé d'animation sans interpolation.



Déplacez la tête de lecture à la seconde 2 et modifiez l'opacité avec une valeur de 100. Testez votre animation. Le centre du site reste transparent durant une seconde et demie, puis apparaît. Votre ligne de temps doit correspondre à la Figure 6.13.

Figure 6.13

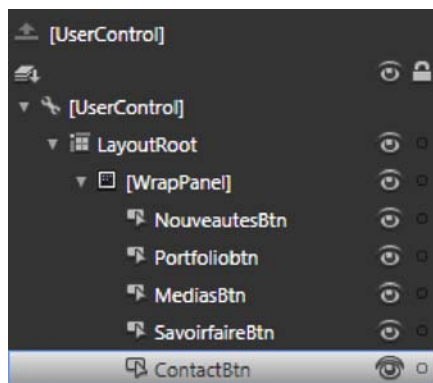
Animation du composant Border.



Chacun des objets animés du site va suivre cette logique d'animation avec quelques différences de tempo. Pour vous simplifier la vie, vous pouvez dupliquer le code XAML créé et cibler un autre composant comme, par exemple, le premier de nos menus qui est contenu au sein du WrapPanel1. Vérifiez tout d'abord que tous les boutons sont nommés pour qu'ils puissent être ciblés en XAML (voir Figure 6.14).

Figure 6.14

Nommage des menus.



Vous pouvez maintenant passer en mode d'édition XAML. Voici la partie du code générée par Blend :

```
<Storyboard x:Name="AnimIntro">
  <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
    Storyboard.TargetName="ContenuPage"
    Storyboard.TargetProperty="(UIElement.Opacity)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
    <DiscreteDoubleKeyFrame KeyTime="00:00:01.6" Value="0"/>
    <EasingDoubleKeyFrame KeyTime="00:00:02" Value="1"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

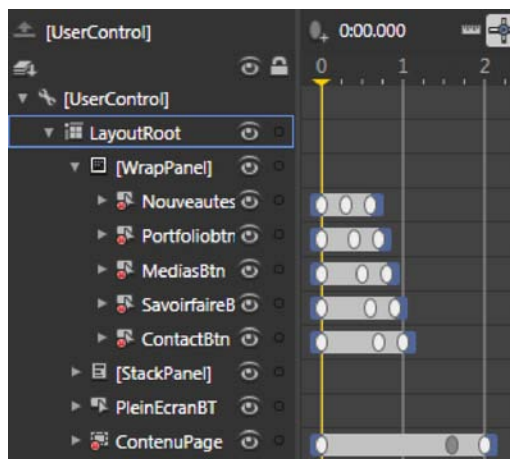
Animer d'autres objets graphiques est très simple à partir de cette base. Dupliquez le nœud XAML DoubleAnimationUsingKeyFrame et remplacez la cible, soit la valeur de Storyboard.TargetName, par NouveautesBtn. Ensuite, pour les deuxième et troisième clés d'animation, définissez la valeur de la propriété KeyTime respectivement à 0 seconde et 3/10 et 0 seconde et 6/10 :

```
<Storyboard x:Name="AnimIntro">
  <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
    TargetName="ContenuPage" Storyboard.TargetProperty=
    "(UIElement.Opacity)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
    <DiscreteDoubleKeyFrame KeyTime="00:00:01.60000" Value="0"/>
    <SplineDoubleKeyFrame KeyTime="00:00:02" Value="1"/>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
    TargetName="NouveautesBtn" Storyboard.TargetProperty=
    "(UIElement.Opacity)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
    <DiscreteDoubleKeyFrame KeyTime="00:00:00.3" Value="0"/>
    <EasingDoubleKeyFrame KeyTime="00:00:00.6" Value="1"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Répétez cette opération pour les autres menus en décalant la seconde et la troisième clés d'1/10^{ème} de seconde. Pour cela, copiez et collez le code XAML que vous venez d'écrire, puis changez le nom de l'objet ciblé à chaque fois (voir Figure 6.15).

Figure 6.15

Ligne de temps avec menus animés.



Testez votre animation. Il semble qu'il s'écoule un peu trop de temps entre l'apparition du dernier menu et l'apparition du fond central. Cela ne pose pas de problème. Vous pouvez glisser-déplacer les clés d'animation du fond central pour les rapprocher de celles des menus.

INFO

Comme vous l'avez constaté, concevoir une animation en XAML par de simple copier-coller ne prend que quelques secondes. Vous pourrez toujours réaliser en mode création vos propres animations. Passer par le code peut vous sembler éloigné de la création pure, mais cela est très rapide quand vous connaissez précisément vos objectifs. Vous pouvez également affiner les animations générées en mode création. Il suffira de les centraliser au sein d'un projet dédié pour les récupérer à n'importe quel moment.

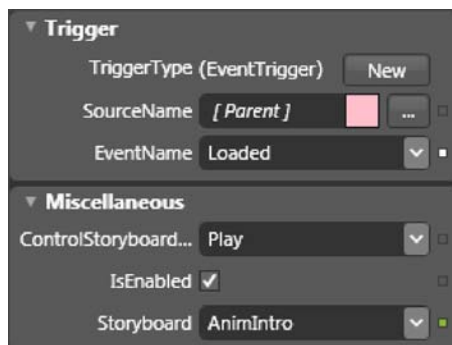
Nous allons maintenant finir l'animation en faisant apparaître le composant `StackPanel` contenant le pied de page, puis le bouton plein écran. Procédez exactement de la même manière et faites en sorte que le pied et le bouton apparaissent en dernier. L'animation d'introduction est presque terminée. Celle-ci est encore un peu fade car nous ciblons uniquement l'opacité des objets. De plus, nous n'avons défini aucune accélération sur chaque animation.

6.2.3 Déclencher des animations

Pour tester votre animation vous pouvez ouvrir la bibliothèque de composants, puis sélectionner l'onglet des comportements (*Behaviors*). Ensuite, glissez le comportement `ControlStoryboardAction` sur le composant `UserControl` racine. Celui-ci est ajouté au sein de l'arbre visuel, mais ce n'est pas un composant graphique. Il va simplement nous aider à jouer l'animation au chargement de l'application. Pour la propriété `EventName`, définissez la valeur `Loaded`. Pour le champ `Storyboard`, sélectionnez `AnimIntro` (voir Figure 6.16).

Figure 6.16

Configuration du comportement.



Ce comportement permet de jouer l'animation lorsque le contrôle UserControl racine est chargé. Toutefois l'action à entreprendre peut être différente. Par défaut, l'action est de lire le Storyboard. Appuyez sur la touche F5 de votre clavier pour tester votre animation dans des conditions réelles. Pour revoir plusieurs fois l'animation se jouer, vous n'avez qu'à rafraîchir la page au sein du navigateur. Nous apprendrons à la section 6.3 à contrôler des Storyboards avec C#.

6.2.4 Les transformations relatives

Pour donner un peu de relief, nous allons animer d'autres propriétés que l'opacité. Pour les menus, nous pourrions par exemple les faire apparaître tout en les déplaçant de haut en bas. Sélectionnez le premier menu, positionnez la tête de lecture à la seconde 0. Tout en maintenant la touche Maj enfoncée, appuyez deux fois sur la flèche du haut. De cette manière, le menu est déplacé de 20 pixels vers le haut. Déplacez la tête de lecture de 3/10^{ème}, puis repositionnez le menu de 20 pixels vers le bas. Celui-ci a repris sa position d'origine. À ce stade, notre objet est animé alors qu'il n'est toujours pas visible. Jusqu'à maintenant, pour décaler nos animations, nous avions besoin de trois clés d'animation. Toutefois, il existe une autre manière de procéder.

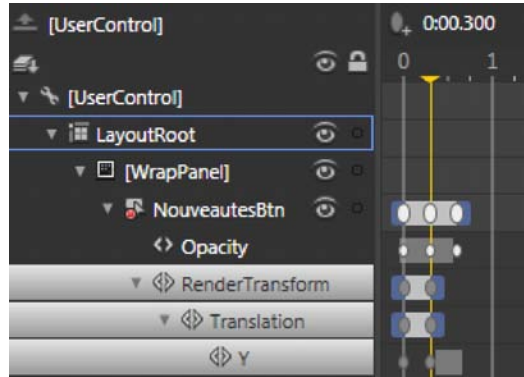
Toutes les classes héritant de la classe abstraite Timeline possèdent la propriété BeginTime. Dépliez l'arborescence du menu NouveautesBtn et sélectionnez la ligne RenderTransform. Passez ensuite en mode d'édition mixte pour faire apparaître le code XAML correspondant à ce nœud, modifiez la valeur de la propriété BeginTime à 0 seconde et 4/10 :

```
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00.3"
    Storyboard.TargetName="NouveautesBtn" Storyboard.
    TargetProperty="(UIElement.RenderTransform).(TransformGroup.
    Children)
    [3].(TranslateTransform.Y)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="-20"/>
    <EasingDoubleKeyFrame KeyTime="00:00:00.3000000" Value="0"/>
</DoubleAnimationUsingKeyFrames>
```

Vous venez de décaler l'animation de transparence de 3/10 de seconde. Cela vous permet d'animer le déplacement du menu au même instant où vous le faites apparaître. Au sein de Blend, la zone grise du déplacement est décalée vers la droite de 0/10^{ème} (voir Figure 6.17).

Figure 6.17

Décaler une séquence d'animation au sein de Blend.



Regardez maintenant la propriété ciblée par notre séquence d'animation :

```
Storyboard.TargetProperty="(UIElement.RenderTransform).
(TransformGroup.Children)[3].(TranslateTransform.Y)">
```

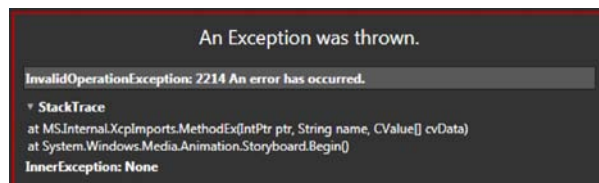
L'accès à la propriété Y n'est pas simple. En fait, dès que nous avons modifié la position de notre menu, Blend y a automatiquement généré un nœud de transformation relative. Pour rappel, les transformations relatives permettent de s'affranchir des contraintes liées à la mise en page du conteneur (voir section 5.5.3.2). Lorsque vous animez des objets, Blend privilégiera toujours l'utilisation des transformations relatives. Pour des raisons pratiques, le nœud `RenderTransform` est toujours généré par Blend de la même manière. Il existe toutefois de nombreuses façons d'écrire ce type de nœud. Ne vous formalisez pas si vous rencontrez une écriture différente. Voici le nouveau code XAML décrivant notre menu :

```
<Button x:Name="NouveautesBtn" Height="Auto" Width="Auto"
  Content="Nouveautés" Margin="0,0,20,0" FontSize="14"
  FontFamily="Trebuchet MS" Visibility="Visible"
  RenderTransformOrigin="0.5,0.5">
  <Button.RenderTransform>
    <TransformGroup>
      <ScaleTransform/>
      <SkewTransform/>
      <RotateTransform/>
      <TranslateTransform/>
    </TransformGroup>
  </Button.RenderTransform>
</Button>
```

La conséquence directe du ciblage d'un nœud `RenderTransform` par une animation est que si celui-ci n'existe pas réellement dans la balise du composant, Blend lève une erreur de ciblage. Pour le démontrer, il suffit de copier-coller notre nouvelle animation tout en ciblant les autres boutons à chaque fois et en décalant de 1/10^{ème} de seconde la valeur de `BeginTime` (voir Figure 6.18).

Figure 6.18

Erreur levée lorsque l'on cible un nœud RenderTransform inexistant.



Pour résoudre ce problème, vous devez coller le nœud `RenderTransform` suivant au sein de chaque menu :

```
<Button.RenderTransform>
  <TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform/>
    <TranslateTransform/>
  </TransformGroup>
</Button.RenderTransform>
</Button>
```

Chaque menu est maintenant animé du haut vers le bas. L'animation prend peu à peu plus d'ampleur. Vous pouvez procéder de la même manière pour le composant `StackPanel` représentant le pied de page. Vous risquez cependant de rencontrer quelques difficultés lors de la compilation. Le nœud `RenderTransform` que nous copions débute par `Button.RenderTransform`. Or, comme il est situé dans le nœud élément `StackPanel`, le compilateur lèvera une erreur. Pour éviter tout problème, il suffit de remplacer `Button` par `FrameworkElement` de cette manière :

```
<FrameworkElement.RenderTransform>
  <TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform/>
    <TranslateTransform/>
  </TransformGroup>
</FrameworkElement.RenderTransform>
</StackPanel>
```

INFO

La classe `FrameworkElement` est héritée de tous les composants visuels et elle possède la propriété `RenderTransform`. Pour cette raison, quel que soit le composant visuel auquel vous souhaitez ajouter un nœud `RenderTransform`, cette classe fera l'affaire pour le déclarer. En programmation orientée objet, ce concept très puissant est appelé polymorphisme. Il est réalisable à travers l'héritage de classes ou l'implémentation d'interfaces. Concrètement, vous l'utiliserez lorsque vous ne connaîtrez pas à l'avance le type de l'objet – ce qui est le cas ici. Comme les objets graphiques héritent tous de `FrameworkElement`, vous n'avez pas à connaître le type précis de chacun d'eux pour affecter le nœud `RenderTransform`.

Voici le code XAML permettant d'animer le pied de page :

```
<DoubleAnimationUsingKeyFrames BeginTime="00:00:01.3" Storyboard.
  TargetName="PiedDePage" Storyboard.TargetProperty=
    "(UIElement.RenderTransform).(TransformGroup.Children)[3].
    (TranslateTransform.Y)">
  <EasingDoubleKeyFrame KeyTime="00:00:00" Value="-20"/>
  <EasingDoubleKeyFrame KeyTime="00:00:00.3000000" Value="0"/>
</DoubleAnimationUsingKeyFrames>
```

Vous remarquez que le `StackPanel` est renommé en `PiedDePage`. L'animation subit un décalage de une seconde et 3/10 afin d'être lue en même temps que celle concernant l'opacité. Pour finir, nous allons faire apparaître le bouton plein écran. Positionnez la tête de lecture à la seconde 0,

puis modifiez l'échelle du bouton *via* l'onglet des transformations relatives. Définissez une échelle `ScaleX` et `ScaleY` à 0,5. Cela signifie qu'à cette seconde, il ne possédera que la moitié de ses dimensions d'origine. Déplacez ensuite la tête de lecture à la seconde 0 et 3/10 et définissez l'échelle à 1 pour `ScaleX` et `ScaleY`. Décalez cette animation pour qu'elle corresponde exactement à celle de l'opacité. Nous nous approchons d'un premier résultat, il nous reste encore quelques réglages pour voir aboutir l'animation. Vous pouvez récupérer cette étape de l'animation (*pleinEcran_mquetteAnimee_1.zip*) dans le dossier *chap6* des exemples.

6.3 Gérer l'accélération

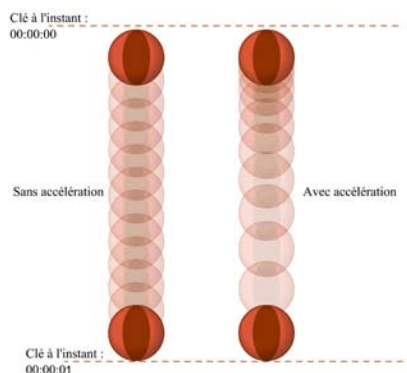
Comme nous l'avons évoqué dans les bonnes pratiques, il faut éviter les animations linéaires dans la majorité des cas. Silverlight vous permet d'éviter cet écueil grâce à trois types de mode d'animation : `Spline`, `Easing` et `Discrete`. Le mode `Discrete` n'est pas pertinent car aucune interpolation n'est réalisée par Blend. Dans ce cas, la transition est abrupte. Il nous reste les modes `Spline` et `Easing` qui font référence aux mêmes principes.

6.3.1 Les principes

Il nous faut d'abord comprendre ce qu'est une accélération avant d'aborder sa gestion au sein de Blend. Pour résumer, l'accélération, qu'elle soit positive ou négative, est une modification de la vitesse au cours du temps. La vitesse est le rythme du changement. Lorsque vous lâchez une balle au-dessus du sol, elle subit l'accélération de la pesanteur correspondant à un `G` (`G` pour Gravité), soit $9,82 \text{ m/s}^2$. Cela signifie que lorsque la balle n'est plus tenue, elle tombe de plus en plus rapidement vers le sol. S'il n'y avait pas d'accélération, elle tomberait de manière constante vers le sol (voir Figure 6.19).

Figure 6.19

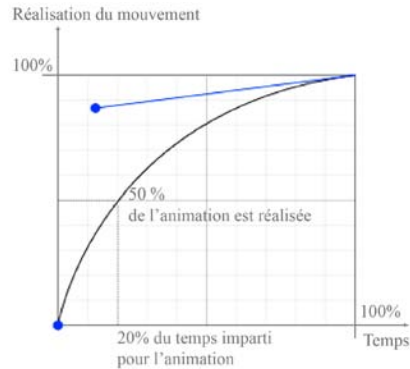
Mouvement sans accélération et avec accélération.



Dans le premier cas, le déplacement de la balle est constant, cela est visible car l'espace entre deux instants égaux est toujours le même. *A contrario*, dans le second cas, l'espace entre chaque affichage de la balle grandit car la vitesse augmente au fur et à mesure du temps. Il est également possible de représenter l'accélération grâce à un petit graphique. La Figure 6.20 illustre la représentation d'un ralenti. On s'aperçoit qu'à 20 % du temps de l'animation, 50 % de celle-ci est déjà effectuée. Cela signifie qu'il reste 50 % de l'animation à réaliser en 80 % de temps restant. La vitesse est donc en forte diminution.

Figure 6.20

Représentation d'une décélération sous forme de graphique.



Nous allons maintenant aborder ce sujet au sein de l'environnement Silverlight et étudier les moyens mis en œuvre par Blend pour gérer l'accélération. Dans Silverlight, l'accélération est toujours définie sur la clé d'arrivée. Lorsque vous souhaitez en créer une spécifique entre deux clés d'animation, il vous faudra donc sélectionner la clé d'arrivée, puis modifier ses propriétés. L'avantage de gérer l'accélération sur l'image clé d'arrivée est qu'une clé initiale n'est pas nécessaire pour créer une animation. Pour mieux l'expliquer et le démontrer, nous allons animer le rebond d'une balle avec une seule clé.

6.3.2 Une animation de rebond en une seule clé d'animation

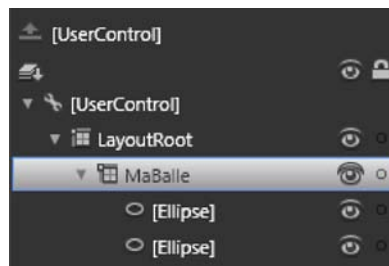
Une animation, dans Silverlight, peut-être définie grâce à une unique clé d'arrivée. Dans ce cas, l'animation est en fait une interpolation entre la valeur en cours de la propriété de l'objet et la valeur de destination définie par la clé. Cela signifie que l'animation sera calculée en fonction de la valeur de départ de manière dynamique, ce qui facilite grandement le travail. Nous allons étudier et créer ce type d'animation dans cette section. Nous aurons à nouveau l'occasion de le faire au Chapitre 7.

6.3.2.1 Créer la balle

Au sein de Blend, créez une nouvelle application Silverlight nommée AnimRebond. Nous allons commencer par créer une balle. Créez deux cercles parfaits l'un sur l'autre en instanciant le composant Ellipse. Sélectionnez-les tous les deux, puis groupez-les au sein d'une grille en utilisant le raccourci Ctrl+g. Nommez la grille MaBalle (voir Figure 6.21).

Figure 6.21

Imbrication de la balle.



Cachez l'instance d'Ellipse qui est à l'avant-plan. Cela vous permet de travailler sur l'autre de manière plus simple. Définissez-lui un dégradé radial partant de l'orange clair vers l'orange foncé. Pour créer un dégradé, au sein du panneau des propriétés, sélectionnez Fill puis le mode dégradé radial. Modifiez ensuite les picots pour gérer la couleur du début de dégradé et celle de fin de dégradé. Ceux-ci sont situés sur la réglette des dégradés (voir Figures 6.22 et 6.23).

Figure 6.22

Créer un dégradé radial.




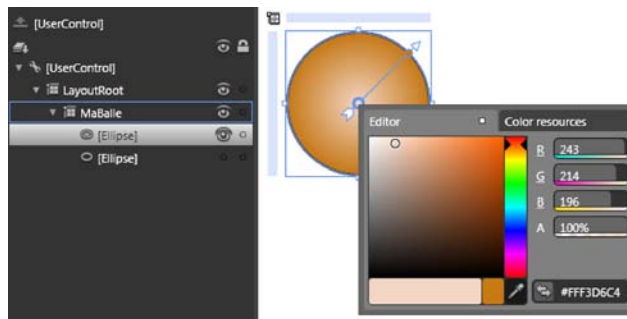
Depuis la version 3 de Blender, il est possible de modifier les picots directement au sein de la vue de création. Pour cela cliquez sur l'icône de gestion des dégradés  (voir Figure 6.23).

Figure 6.23

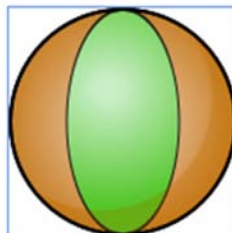
Modifier un dégradé radial au sein de la fenêtre de création.



Grâce à cet outil, vous pouvez directement, et de manière sensible, modifier et gérer le remplissage d'un dégradé. Faites apparaître le second cercle, puis au sein du panneau des transformations relatives, définissez une échelle en ScaleX à 0,6 et laissez l'échelle ScaleY à sa valeur. Définissez également un dégradé de couleur, mais dans les tons vert ou bleu. Le résultat est reproduit à la Figure 6.24.

Figure 6.24

Une balle en plastique.



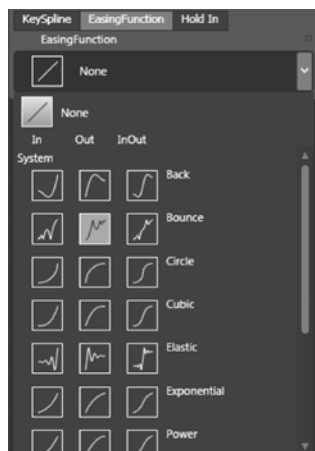
6.3.2.2 Les équations d'accélération

Maintenant que vous avez créé la balle, il ne reste plus qu'à réaliser l'animation. Pour cela, nous allons utiliser les équations d'accélération. Cliquez sur l'icône correspondante (+) et nommez l'animation AnimRebond. Une fois en mode d'enregistrement, déplacez le composant Grid Ma-Balle de 200 pixels vers le bas et de 100 pixels vers la droite. Vous venez de créer deux clés d'animation à la seconde 0, la première pour le déplacement en X et la seconde pour le déplacement en Y. Déplacez la clé principale à la seconde 2, puis dépliez complètement l'arbre visuel afin de sélectionner la clé générée par le déplacement vertical. Dans le panneau des propriétés, cliquez sur la liste déroulante contenant les équations d'accélération et choisissez celle décrivant un rebond à l'arrivée (voir Figure 6.25).

Vous pouvez régler cette accélération grâce à deux paramètres : le facteur de rebond (Bounciness) et le nombre de rebonds (Bounces). Plus le facteur de rebond est faible, plus les rebonds gagneront en amplitude. Plus le facteur sera élevé, moins les rebonds seront visibles. Pour chaque composant visuel, il est possible de définir, en une seule fois, une accélération différente pour chaque clé ou encore pour toutes les clés situées à la même seconde. Il suffit pour cela de configurer la clé d'animation située au même niveau que l'objet. De cette manière, toutes les propriétés sont interpolées de manière identique.

Figure 6.25

Choisir une accélération de type rebond à l'arrivée.

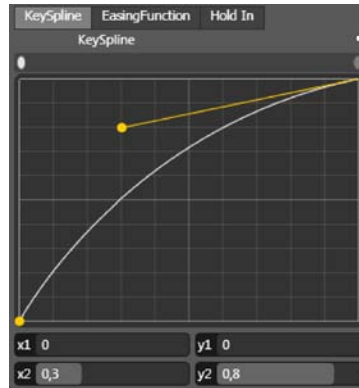


6.3.2.3 Les courbes d'accélération

Nous allons choisir un autre type d'accélération pour l'axe des X. Sélectionnez la clé correspondante et dans le panneau des propriétés choisissez l'onglet KeySpline. Configurez la courbe d'accélération manuellement pour créer un léger effet de ralenti. Déplacez le point jaune situé en haut à droite du graphique vers l'intérieur du graphe (voir Figure 6.26).

Figure 6.26

Personnaliser une courbe d'accélération.



L'extrémité de la tangente est déterminée en sortie de courbe par les valeurs X2 et Y2. Voici le XAML généré pour les deux types d'accélération que nous venons de définir :

```
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
    TargetName="MaBalle" Storyboard.TargetProperty="(UIElement.
    RenderTransform).(TransformGroup.Children)[3].
    (TranslateTransform.Y)">
    <EasingDoubleKeyFrame KeyTime="00:00:01" Value="200">
        <EasingDoubleKeyFrame.EasingFunction>
            <BounceEase Bounciness="2"/>
        </EasingDoubleKeyFrame.EasingFunction>
    </EasingDoubleKeyFrame>
</DoubleAnimationUsingKeyFrames>
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
    TargetName="MaBalle" Storyboard.TargetProperty="(UIElement.
    RenderTransform).(TransformGroup.Children)[3].
    (TranslateTransform.X)">
    <SplineDoubleKeyFrame KeyTime="00:00:01" Value="50"
        KeySpline="0,0,0.3,0.8"/>
</DoubleAnimationUsingKeyFrames>
```

Vous pouvez remarquer en gras les deux types de clés correspondants aux modifications que nous avons apportées. La propriété **KeySpline** contient les coordonnées des points de tangence soit, dans l'ordre, X1, Y1, X2 et Y2. Testez votre animation au sein de Blend puis dans votre navigateur préféré. Vous pouvez utiliser un comportement de type **ControlStoryboardAction** afin de déclencher la lecture de l'animation à l'exécution (voir la section 6.2.3). Comme vous le constatez, une seule clé d'animation est nécessaire.

Le principal avantage de ce comportement est de permettre une mise à jour plus facile de vos animations en évitant de figer par une clé leur point de départ. Nous allons le démontrer simplement. Fermez le Storyboard **AnimRebond** et déplacez votre balle n'importe où au sein du conteneur **LayoutRoot**. Puis recompilez votre application et testez-la. Comme l'animation utilise les transformations relatives, elle est indépendante des contraintes du conteneur. De plus, comme elle ne possède qu'une clé d'arrivée, elle a exactement le même effet quelle que soit la position par défaut de l'objet. Autrement dit l'animation est totalement indépendante de la scène principale ou de l'objet ciblé.

Pour vous en convaincre, créez un rectangle n'importe où dans `LayoutRoot` et nommez-le `MonCarre`. Créez-y un nœud de transformations relatives, `RenderTransform` (voir section 6.2.4), puis dupliquez l'animation en copiant-collant le code XAML.

INFO

Vous pouvez également dupliquer l'animation au sein de l'interface visuelle de Blend. Pour cela, ouvrez l'animation `AnimRebond` en cliquant sur l'icône de liste d'actions (🔍). Une fois à l'intérieur, cliquez sur l'icône déroulante (⌵) située juste à droite de l'icône d'ajout et sélectionnez l'option `Dupliquer`. Vous êtes désormais dans l'animation `AnimRebond_copy`. Cliquez à nouveau sur l'icône et choisissez `Renommer`.

Changez le nom par `AnimRebondCarre`. Passez en mode d'édition mixte, puis modifiez la cible des deux sous-séquences d'animation de type `DoubleAnimation` par `MonCarre`. Vous venez de dupliquer l'animation et vous avez changé sa cible en quelques secondes. Vous pouvez tester l'animation directement dans Blend.

INFO

Grâce à l'icône de liste d'actions, vous pouvez ajouter, supprimer, renommer et dupliquer une ressource `Storyboard`. Vous pouvez même inverser l'ordre des clés d'animation. Toutefois, cette dernière opération n'est pas sans surprises car seul l'ordre des enfants de la propriété `KeyFrames` est inversé. Autrement dit, le timing de l'animation ne l'est pas réellement. Il vous faudra donc un peu plus de travail si vous avez plus de deux clés pour la même propriété.

L'exercice finalisé est dans l'archive *AnimRebond.zip* du dossier *chap6* des exemples.

6.3.3 Améliorer l'animation d'introduction

Nous allons maintenant améliorer notre animation principale en gérant l'accélération des clés d'animation. Cela ne devrait pas poser beaucoup de problèmes. Nous allons cependant limiter à deux le nombre de types d'accélération. Ouvrez la solution faisant référence au projet `Site-AgencePortFolio` là où nous l'avions laissé à la section 6.2. Vous trouverez aussi ce fichier dans l'archive *pleinEcran_maquetteAnimee_1.zip* du dossier *chap6* des exemples du livre.

Sélectionnez la dernière image clé de chaque animation de modification relative, `RenderTransform`, et définissez une accélération de type `Back Out` (voir Figure 6.27).

Cette équation d'accélération est assez pratique car la valeur de la propriété dépasse un peu celle de destination avant de l'atteindre avec un effet de ralenti. Ce genre d'accélération est utile car impossible à réaliser avec les courbes d'accélération de type `Spline`. Il est en effet impossible de définir un point de tangence dont les coordonnées dépassent 1 en Y, soit 100 % de réalisation du mouvement.

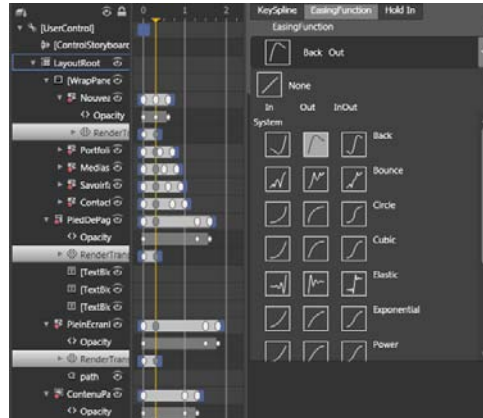
Si vous trouvez le tempo trop rapide, espacez légèrement chaque clé. De même, si l'œil de l'utilisateur n'est pas assez attiré en bas à droite, pour l'animation du pied de page en fin d'introduction, il vous suffit de ralentir l'animation. Toutefois, d'un point de vue design, de nombreux changements seront mis en œuvre. Le gris clair des contrôles par défaut est neutre et n'attire pas le regard.

D'autres moyens sont encore à notre disposition pour soutenir cette animation et l'intégrer au sein d'une charte graphique.

Un autre facteur est également très important : les dimensions du site. Comme le site occupe 100 % de la fenêtre du navigateur, si ce dernier accapare tout votre écran, les difficultés de décryptage pour l'utilisateur en seront accrues. Le redimensionnement est très pratique, mais peut influencer l'animation et son impact visuel au sein d'une application ou d'un site web. Pour finir, sélectionnez la dernière image clé de chaque animation d'opacité, puis définissez soit une courbe d'accélération soit une équation d'accélération avec ralenti.

Figure 6.27

Personnalisez
l'accélération des nœuds
RenderTransform.



6.4 Animer avec C#

6.4.1 L'intérêt d'animer avec C#

Il nous faut tout d'abord lever toute ambiguïté en précisant que l'animation d'un site incombe avant tout à l'animateur, au designer interactif ou au directeur artistique. Celle-ci est directement liée à la charte graphique, à l'expérience utilisateur et donc à l'ergonomie. Toutefois, notre but dans cette section est d'apprendre à créer des animations dynamiquement. Les langages C# ou VB ne remplaceront bien sûr jamais un logiciel comme Blend d'un point de vue créativité, mais l'utilisation d'un langage logique ouvre de nombreuses possibilités et offre une grande souplesse de production.

On le constate aisément avec des technologies du type *Processing*. Cette dernière présente l'avantage de générer des animations très esthétiques en temps réel. Pour de plus amples informations sur cette technologie, rendez-vous sur le site : <http://processing.org>. Toutefois ces technologies s'inscrivent souvent dans une démarche artistique. Les outils et les bibliothèques proposés sont par nature orientés vers le visuel et l'interactivité et non vers la fonctionnalité, au contraire de C#. La création d'animations *via* C# ou VB présente tout de même de nombreux intérêts si celle-ci est encadrée par des créatifs. Grâce à C# :

- On évite le travail rébarbatif de recopie des animations sous Blend sans enlever la conception de celle-ci aux animateurs.
- Il est possible d'ajouter une couche d'interactivité utilisateur supplémentaire en mettant à jour les animations dynamiquement.

- On peut déclencher des animations aléatoirement ou de manière rythmée.
- L'animation de particules est à notre portée. Cela nous permet de simuler des fluides, de la fumée, de la pluie ou un vol d'oiseau.

Tous ces avantages sont attrayants et justifient un investissement dans ce domaine. Nous allons aborder chacun d'eux d'un point de vue pratique.

6.4.2 Instanciation dynamique de ressources *Storyboard*

Ouvrez le projet *AnimRebond* que vous avez réalisé dans un précédent exercice (*AnimRebond.zip*). Dans un premier temps, nous allons créer la même animation que celle que nous avons réalisée, mais uniquement avec C# pour apprendre les concepts.

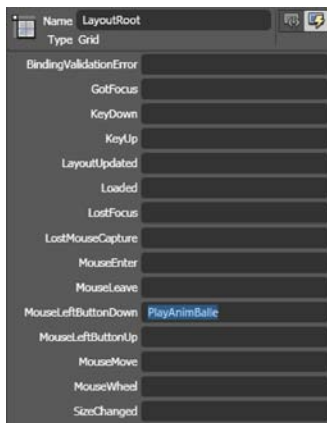
6.4.2.1 Créer l'animation de rebond en C#

La première étape consiste à dupliquer la balle. Nommez la nouvelle balle *MaBalle2*. Sélectionnez la grille *LayoutRoot*, puis dans le panneau des événements de l'objet entrez *PlayAnimBalle* pour l'événement *MouseLeftButtonDown* (voir Figure 6.28).

Blend ouvre automatiquement le fichier de code logique C# correspondant à notre fichier XAML et crée une méthode *PlayAnimBalle*. Elle se déclenchera à chaque fois que l'utilisateur cliquera sur la grille principale. Cette méthode nous permettra de déclencher la lecture du *Storyboard*, que nous allons créer par code.

Figure 6.28

Définir une méthode d'écoute pour l'événement *MouseLeftButtonDown* de l'objet *LayoutRoot*.



Le code ci-dessous expose la création d'une nouvelle ressource *Storyboard* :

```
public partial class MainPage : UserControl
{
    Storyboard AnimRebondCode;

    public MainPage()
    {
        InitializeComponent();
    }
}
```

```

        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }
    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        //on commence par instancier une nouvelle ressource Storyboard
        AnimRebondCode = new Storyboard();

        //on crée ensuite la sous-séquence d'animation
        //en lui définissant une nouvelle destination
        DoubleAnimation DaY = new DoubleAnimation();

        //on définit la valeur d'arrivée de la propriété
        //qui sera ciblée, on a le choix entre To et By
        DaY.To = 200;

        //DaY.By = 200;
        //By permet d'ajouter une valeur de destination
        //relative à la valeur actuelle

        //Puis une nouvelle équation d'accélération pour la
        //sous-séquence d'animation
        DaY.EasingFunction = new BounceEase();

        //La classe Storyboard nous permet de modifier l'objet
        //ciblé par l'animation via la méthode statique SetTarget
        Storyboard.SetTarget(AnimRebondCode, MaBalle2);

        //La classe Storyboard nous fournit également une méthode
        //pour cibler la propriété à animer
        Storyboard.SetTargetProperty(DaY, new PropertyPath("UIElement.
            RenderTransform).(TransformGroup.Children)[3].
            (TranslateTransform.Y)"));

        //On ajoute la sous-séquence d'animation comme enfant du Storyboard
        AnimRebondCode.Children.Add(DaY);

        //l'étape finale facultative consiste à ajouter le Storyboard
        //aux ressources du UserControl racine, soit
        //this.Resources.Add("AnimRebondCodeClé", AnimRebondCode);

    }

    private void PlayAnimBalle(object sender, MouseButtonEventArgs e)
    {
        AnimRebondCode.Begin();
    }
}

```

Deux ou trois concepts méritent d'être expliqués. Tout d'abord, la création, puis l'utilisation d'un Storyboard contiennent pratiquement les mêmes étapes que l'ajout d'un objet FrameworkElement à l'arbre visuel d'une application. On le déclare comme membre de classe pour y accéder depuis n'importe quelle méthode. On l'instancie ensuite au chargement de l'application. Pour finir, on l'ajoute comme enfant de la propriété Resources. Cette dernière étape est nécessaire uniquement si vous souhaitez utiliser le Storyboard comme ressource, ce qui n'est pas obligatoire depuis Silverlight 3. Il est conseillé de ne pas l'ajouter par défaut afin de faciliter la libération des ressources en cas de suppression du Storyboard. Cette propriété est propre à tous les objets de type FrameworkElement. N'importe quel objet au sein de l'arbre visuel peut donc posséder des ressources. Nous pourrions donc très bien avoir le code XAML suivant :

```
<Grid ... >
  <Button Width="100" Height="30">
    <Button.Resources>
      <Storyboard x:Name="monAnimAccessibleDansButton" ... >
        ...
      </Storyboard>
    </Button.Resources >
  </Button>
</Grid >
```

Comme nous l'avons montré à la section 6.1.4, les objets de type Storyboard sont considérés comme des ressources car ils ne possèdent pas de représentation visuelle concrète. La propriété Resources fait référence aux dictionnaires de ressources. Elle implémente l'interface IDictionary. Chaque ressource, lorsqu'elle est ajoutée au dictionnaire, doit posséder une clé d'accès. C'est à cela que sert la chaîne de caractères passée en premier paramètre :

```
Resources.Add("AnimRebondCodeClé", AnimRebondCode);
```

Le second argument représente la référence du Storyboard que nous souhaitons ajouter. Une autre difficulté que vous pouvez rencontrer est le ciblage de la propriété à animer. Dans la ligne de code ci-dessous, nous spécifions le chemin d'accès pour la propriété Y de type RenderTransform :

```
Storyboard.SetTargetProperty(DaY, new PropertyPath("(UIElement.
RenderTransform).(TransformGroup.Children)[3].
(TranslateTransform.Y)"));
```

Cela peut paraître un peu barbare, mais vous pouvez copier-coller le chemin d'accès grâce au code XAML généré automatiquement par Blend. De plus, ce type de chemin n'est nécessaire que pour les transformations relatives. Pour d'autres propriétés propres aux objets eux-mêmes, comme Opacity ou Width, vous pouvez utiliser des membres statiques de classe suffixés de Property :

```
PropertyPath pp = new PropertyPath(Canvas.WidthProperty);
Storyboard.SetTargetProperty(DaY, pp);
```

Pour finir, il ne faut pas oublier de cibler l'objet à animer. À cette fin, utilisez la méthode statique SetTarget de la classe Storyboard. Vous devez préciser l'instance de type Timeline et la cible animée par cette dernière :

```
//La classe Storyboard nous permet de modifier l'objet
//ciblé par l'animation via la méthode statique SetTarget
Storyboard.SetTarget(AnimRebondCode, MaBalle2);
```

Nous allons maintenant pousser ce concept un peu plus loin.

6.4.2.2 Mise à jour dynamique de l'animation

Pour l'instant, tout ce que nous faisons peut être réalisé dans Blend. Vous allez ajouter un peu de logique afin d'illustrer l'intérêt de C#. Lorsque l'utilisateur cliquera sur la grille principale, la balle se déplacera aux coordonnées où l'événement se sera produit. Tout d'abord, vous allez afficher les informations concernant le clic du bouton de la souris sur LayoutRoot.

Créez un nouveau membre de classe de type TextBlock, nommé InfoTxt, puis ajoutez-le comme enfant du conteneur LayoutRoot lors du chargement de l'application :

```

TextBlock InfoTxt = new TextBlock();
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    LayoutRoot.Children.Add(InfoTxt);
    // le code vu précédemment dans cette méthode...
}

```

Il faut également modifier la méthode `PlayAnimBalle` pour récupérer les coordonnées de la souris lors de l'événement `OnMouseDown` :

```

private void PlayAnimBalle(object sender, MouseButtonEventArgs e)
{
    Point Position = e.GetPosition(null);
    InfoTxt.Text = Position.ToString();
    AnimRebondCode.Begin();
}

```

La structure de type `Position` contient deux valeurs de type `double` correspondant à X et Y. Celle-ci est récupérée grâce à l'argument de type `MouseButtonEventArgs`. Nous reviendrons sur ce type d'argument au Chapitre 8. Le champ texte indique désormais les coordonnées de la souris à l'instant où vous cliquez sur le conteneur `LayoutRoot`. Pour se déplacer aux coordonnées X et Y, nous devons définir une deuxième animation ciblant cette fois l'axe des X :

```

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    LayoutRoot.Children.Add(InfoTxt);

    //on commence par instancier une nouvelle ressource Storyboard
    AnimRebondCode = new Storyboard();

    //on crée ensuite la sous-séquence d'animation
    //en lui définissant une nouvelle destination
    DoubleAnimation DaY = new DoubleAnimation();
    DoubleAnimation DaX = new DoubleAnimation();

    #region plus besoin de ce code
    //on définit la valeur d'arrivée de la propriété qui sera ciblée
    //on a le choix entre To et By
    //DaY.To = 200;
    //DaX.By = 200;
    //By permet d'ajouter une valeur relative à la valeur actuelle
    #endregion

    //Puis une nouvelle équation d'accélération pour la
    //sous-séquence d'animation
    DaY.EasingFunction = new BounceEase();
    DaX.EasingFunction = new BounceEase();
    //La classe Storyboard nous permet de modifier l'objet
    //ciblé par l'animation via la méthode statique SetTarget
    Storyboard.SetTarget(AnimRebondCode, MaBalle2);

    //La classe Storyboard nous fournit également une méthode
    //pour cibler la propriété à animer
    Storyboard.SetTargetProperty(DaY, new PropertyPath("(UIElement.
    RenderTransform).(TransformGroup.Children)[3].
    (TranslateTransform.Y)"));
    Storyboard.SetTargetProperty(DaX, new PropertyPath("(UIElement.
    RenderTransform).(TransformGroup.Children)[3].
    (TranslateTransform.X)"));
}

```

```
//On ajoute la sous-séquence d'animation comme enfant du Storyboard
AnimRebondCode.Children.Add(DaY);
AnimRebondCode.Children.Add(DaX);

}
```

Il suffit maintenant de modifier notre animation pour que les valeurs de destination soient mises à jour. Toutefois, il faut faire la différence entre les coordonnées de la souris, que nous récupérons sur le conteneur, et les transformations relatives X et Y que nous affectons. Le code suivant est un début mais n'est pas suffisant :

```
private void PlayAnimBalle(object sender, MouseButtonEventArgs e)
{
    Point Position = e.GetPosition(null);
    InfoTxt.Text = Position.ToString();
    (AnimRebondCode.Children[0] as DoubleAnimation).To = Position.Y;
    (AnimRebondCode.Children[1] as DoubleAnimation).To = Position.X;
    AnimRebondCode.Begin();
}
```

Vous remarquez qu'il y a un décalage entre le point de destination des animations X et Y et l'endroit où vous avez cliqué. Ceci est dû aux transformations relatives. Dans le code précédent, à chaque fois que vous cliquez, vous ajoutez les coordonnées de votre souris à la position initiale de la balle au sein du conteneur LayoutRoot. Pour remédier à ce problème, vous avez deux choix. Le plus simple consiste à positionner MaBalle2 en haut à gauche de LayoutRoot en supprimant les marges et en choisissant un alignement à gauche et à droite (voir Figure 6.29).

Figure 6.29

Modification des marges et de l'alignement de MaBalle2.



Recompilez votre application pour voir le résultat. Elle fonctionne mieux, mais cette solution vous oblige à positionner votre balle en haut à gauche lors de la compilation. La seconde solution – plus pratique – consiste à soustraire les marges existantes et à spécifier un alignement en haut et gauche :

```
private void PlayAnimBalle(object sender, MouseButtonEventArgs e)
{
    Point Position = e.GetPosition(null);
    InfoTxt.Text = Position.ToString();
    double NewPosY = Position.Y - MaBalle2.Margin.Top - (MaBalle2.Height/2);
    double NewPosX = Position.X - MaBalle2.Margin.Left - (MaBalle2.Width/2);
    (AnimRebondCode.Children[0] as DoubleAnimation).To = NewPosY;
    (AnimRebondCode.Children[1] as DoubleAnimation).To = NewPosX;
    AnimRebondCode.Begin();
}
```

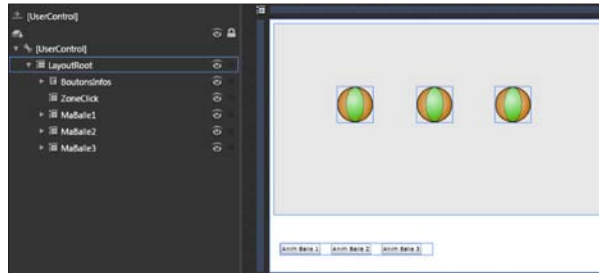
Dans le code précédent, la moitié de la largeur et de la hauteur de la balle est également soustraite. Ceci vous permet de positionner le centre de la balle exactement là où vous avez cliqué.

6.4.3 Affectation dynamique de Storyboards

Vous allez affecter la même animation à différents objets contenus dans une grille. Créez un nouveau projet nommé `AnimRebond_Dynamic`. Commencez par la mise en place du projet sous Expression Blend. Au sein du conteneur `LayoutRoot`, créez trois copies de notre balle ayant pour nom `MaBalle1`, `MaBalle2` et `MaBalle3`. En arrière-plan des balles, et toujours dans la grille `LayoutRoot`, instanciez un nouvel objet de type `Grid` et appelez-le `ZoneClick`. Cette grille doit posséder une marge basse de 110 pixels. Vous pouvez lui affecter une couleur d'arrière-plan pour la visualiser plus facilement. La marge vous permet de positionner un composant `StackPanel` horizontal entre le bas de la zone de clic et le bord inférieur de notre application. Le composant contiendra les boutons qui modifieront la cible de l'animation, ainsi que les coordonnées lors du clic de souris. Autrement dit, nous changerons dynamiquement la référence correspondant à l'objet ciblé. Cela est réalisable grâce à la méthode statique `SetTarget` de la classe `Storyboard`. Nommez le `StackPanel` `BoutonsInfos` et créez les boutons. Pour chacun d'eux, la propriété `Content` doit être affectée de la valeur `Anim Balle 1`, `Anim Balle 2` ou `Anim Balle 3` (voir Figure 6.30).

Figure 6.30

Interface pour animation dynamique.



Le code ressemble à celui de notre projet précédent, toutefois de subtiles modifications nous permettent d'améliorer son comportement. Commencez par recopier le code de la précédente animation. Il vous faut ajouter le membre de classe `BalleEnCours` (de type `FrameworkElement`). Il correspond à la référence de la balle actuellement ciblée par l'animation. Spécifiez ensuite une méthode pour l'événement `MouseLeftButtonDown` de chaque bouton. Vous pouvez utiliser le panneau des événements à cette fin. Nommez ces méthodes `SetBalle1`, `SetBalle2` et `Setballe3`. Au sein de ces fonctions, vous allez redéfinir la cible de l'animation. Ainsi, lorsque vous cliquerez sur le bouton 3, le `Storyboard` ciblera la balle 3.

Voici la totalité du code mis à jour :

```
public partial class MainPage : UserControl
{
    Storyboard AnimRebondCode;
    FrameworkElement BalleEnCours;

    public MainPage()
    {
        InitializeComponent();
        Loaded += new System.Windows.RoutedEventHandler(MainPage_Loaded);
    }

    TextBlock InfoTxt = new TextBlock();

    void MainPage_Loaded(object sender, RoutedEventArgs e)
```

```

{
    //Par défaut on anime la première balle
    BalleEnCours = MaBalle1;

    BoutonsInfos.Children.Add(InfoTxt);

    AnimRebondCode = new Storyboard();

    DoubleAnimation DaY = new DoubleAnimation();
    DoubleAnimation DaX = new DoubleAnimation();

    DaY.To = 200;

    DaY.EasingFunction = new BackEase();
    DaX.EasingFunction = new ElasticEase();

    //La classe Storyboard nous permet de modifier l'objet
    //ciblé par l'animation via la méthode statique SetTarget
    Storyboard.SetTarget(AnimRebondCode, BalleEnCours);

    Storyboard.SetTargetProperty(DaY, new PropertyPath("UIElement.
        RenderTransform).(TransformGroup.Children)[3].
        (TranslateTransform.Y)"));
    Storyboard.SetTargetProperty(DaX, new PropertyPath("UIElement.
        RenderTransform).(TransformGroup.Children)[3].(
        TranslateTransform.X)"));

    AnimRebondCode.Children.Add(DaY);
    AnimRebondCode.Children.Add(DaX);
}

private void PlayAnimBalle(object sender, MouseButtonEventArgs e)
{
    Point Position = e.GetPosition(null);
    InfoTxt.Text = "X :: " + Position.X + " - Y :: " + Position.Y;

    double NewPosY = Position.Y - BalleEnCours.Margin.Top -
        (BalleEnCours.Height / 2);

    double NewPosX = Position.X - BalleEnCours.Margin.Left -
        (BalleEnCours.Width / 2);
    (AnimRebondCode.Children[0] as DoubleAnimation).To = NewPosY;
    (AnimRebondCode.Children[1] as DoubleAnimation).To = NewPosX;
    AnimRebondCode.Begin();
}

private void SetBalle1(object sender, RoutedEventArgs e)
{
    AnimRebondCode.Stop();
    BalleEnCours = MaBalle1;
    Storyboard.SetTarget((AnimRebondCode.Children[0]), BalleEnCours);
    Storyboard.SetTarget((AnimRebondCode.Children[1]), BalleEnCours);
}

private void SetBalle2(object sender, RoutedEventArgs e)
{
    AnimRebondCode.Stop();
    BalleEnCours = MaBalle2;
    Storyboard.SetTarget((AnimRebondCode.Children[0]), BalleEnCours);
}

```

```

        Storyboard.SetTarget((AnimRebondCode.Children[1]), BalleEnCours);
    }

    private void SetBalle3(object sender, RoutedEventArgs e)
    {
        AnimRebondCode.Stop();
        BalleEnCours = MaBalle3;
        Storyboard.SetTarget((AnimRebondCode.Children[0]), BalleEnCours);
        Storyboard.SetTarget((AnimRebondCode.Children[1]), BalleEnCours);
    }
}

```

Revenons un peu sur ce code. Le moteur Silverlight ne permet pas à un Storyboard d'être utilisé plusieurs fois, tant que celui-ci est actif. Il nous faut donc libérer la ressource Storyboard en invoquant la méthode `Stop`, avant de pouvoir lui affecter une nouvelle cible :

```

private void SetBalle3(object sender, RoutedEventArgs e)
{
    AnimRebondCode.Stop();
    BalleEnCours = MaBalle3;
    Storyboard.SetTarget((AnimRebondCode.Children[0]), BalleEnCours);
    Storyboard.SetTarget((AnimRebondCode.Children[1]), BalleEnCours);
}

```

Lorsque vous relâchez le bouton gauche de la souris n'importe où sur la zone de clic, la méthode `PlayAnimBalle` est exécutée. Il est assez pratique dans notre cas d'appeler une méthode lorsque le bouton gauche de la souris est enfoncé (`MouseLeftButtonDown`), puis d'en appeler une autre quand celui-ci est relâché (`MouseLeftButtonUp`). Cela vous permet de contrôler sereinement l'enchaînement des étapes. Nous pourrions par exemple définir la nouvelle balle à animer lorsque l'utilisateur appuie sur l'une d'entre elles, puis déclencher l'animation lorsqu'il relâche le bouton. Les objets présents au sein de l'arbre visuel sont tous interactifs, cela vous évite d'instancier des composants `Button` pour tout et n'importe quoi.

6.4.4 Dupliquer un *Storyboard* créé dans Blend via C#

Pour réaliser cet exercice, il vous faudra désarchiver *MenuAnim_Dynamique.zip* du dossier *chap6* des exemples du livre. Vous pouvez ouvrir le projet dans Blend ou Visual Studio.

Ce projet est très simple, il contient plusieurs primitives `Rectangle`, mais seul le premier enfant de la grille est animé. Notre objectif va consister à récupérer l'animation générée dans Blend par le designer interactif, puis à la dupliquer dynamiquement afin de l'affecter à chaque `Rectangle` contenu dans la grille. Cloner dynamiquement un `Storyboard` n'est malheureusement pas une opération simple à réaliser, ceci pour deux raisons. La première est que la méthode `MemberwiseClone` héritée de `Object` est protégée, donc inaccessible depuis l'extérieur (de plus cette méthode n'est pas satisfaisante car elle ne clone l'objet qu'en surface). La seconde raison est que la classe `XamlWriter`, utilisée pour récupérer la chaîne de caractères XAML de toute référence sous WPF, n'est pas supportée par Silverlight et qu'elle est disponible uniquement au sein de WPF. Nous pourrions utiliser l'API de réflexion qui permet de parcourir les types, méthodes, propriétés, etc. Toutefois un tel code serait fastidieux et ne servirait pas notre propos.

Nous allons donc commencer par récupérer la chaîne de caractères correspondant au `Storyboard`. Il suffit de copier le code XAML généré dans Blend, puis de le coller directement au sein du fichier C# dans le constructeur. Une fois cette étape réalisée, supprimez toutes les propriétés `Sto-`

ryboard.TargetName="Rectangle1", les propriétés BeginTime="00:00:00", ainsi que la propriété x:Name="Anim3D". Vous obtiendrez le résultat ci-dessous :

```
<Storyboard>
  <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty=
    "(UIElement.Projection).(PlaneProjection.RotationY)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="650"/>
    <EasingDoubleKeyFrame KeyTime="00:00:01" Value="0">
      <EasingDoubleKeyFrame.EasingFunction>
        <CubicEase/>
      </EasingDoubleKeyFrame.EasingFunction>
    </EasingDoubleKeyFrame>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty=
    "(UIElement.Opacity)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
    <EasingDoubleKeyFrame KeyTime="00:00:01" Value="1"/>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty=
    "(UIElement.Projection).(PlaneProjection.GlobalOffsetZ)">
    <EasingDoubleKeyFrame KeyTime="00:00:00" Value="-700"/>
    <EasingDoubleKeyFrame KeyTime="00:00:01.5000000" Value="0">
      <EasingDoubleKeyFrame.EasingFunction>
        <CubicEase/>
      </EasingDoubleKeyFrame.EasingFunction>
    </EasingDoubleKeyFrame>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty=
    "(FrameworkElement.Width)">
    <EasingDoubleKeyFrame KeyTime="00:00:01" Value="51"/>
    <EasingDoubleKeyFrame KeyTime="00:00:01.5000000" Value="150">
      <EasingDoubleKeyFrame.EasingFunction>
        <CubicEase/>
      </EasingDoubleKeyFrame.EasingFunction>
    </EasingDoubleKeyFrame>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Visual Studio lève plusieurs erreurs, mais ce n'est que temporaire. Il va nous falloir transformer le XAML en objet de type String. C'est la partie du code un peu rébarbative, mais c'est une phase sensible car au moindre faux pas le compilateur lèvera une erreur.

Nous devons utiliser la classe `StringBuilder` pour concaténer une chaîne de caractères. Concaténer revient à mettre plusieurs chaînes de caractères à la suite. Favoriser l'utilisation de `StringBuilder` optimise grandement la mémoire allouée pour ce type d'opération. Référez l'espace de noms `System.Text` *via* l'instruction `using`. Nous allons parcourir tous les enfants de la grille `LayoutRoot` et créer un `Storyboard` pour chaque `UIElement` trouvé :

```
//espace de noms à ajouter
using System.Windows.Markup;
using System.Collections.Generic;
using System.Text;
//On crée un dictionnaire pour stocker chaque animation créée
Dictionary<UIElement, Storyboard> DicoElementsSB = new
    Dictionary<UIElement, Storyboard>();

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    CreateStoryboard();
}
```

```

    }

    private void CreateStoryboard()
    {
        foreach (UIElement Ui in LayoutRoot.Children)
        {
            //on récupère l'index de chaque objet dans la liste d'enfants
            int i = LayoutRoot.Children.IndexOf(Ui);
        }
    }
}

```

Notre objectif est d'injecter dynamiquement la chaîne de caractères XAML. Vous allez maintenant placer le code XAML récupéré au sein de la boucle `foreach`, puis utiliser la classe `StringBuilder`. Toutefois, pour que cette opération soit possible, nous devons respecter certaines règles :

- Vous devez toujours ajouter les espaces de noms XAML à l'élément le plus élevé dans la hiérarchie. Dans notre cas, il faudra donc ajouter les espaces de noms au nœud élément `Storyboard`.
- La propriété `Name` doit toujours prendre une valeur différente si vous la définissez. Cependant, il n'est pas utile de conserver et d'affecter cette propriété.
- Pour transformer le XAML en chaîne de caractères, il faut remplacer les apostrophes doubles de chaque propriété par de simples apostrophes, puis encadrer la totalité de la ligne par deux apostrophes doubles. Cela vous évitera les erreurs. Par exemple, `KeyTime="00:00:00"` deviendra `KeyTime='00:00:00'`.

Voici le début de ce que doit être votre code :

```

    foreach (UIElement Ui in LayoutRoot.Children)
    {
        int i = LayoutRoot.Children.IndexOf(Ui);

        StringBuilder sb = new StringBuilder("<Storyboard x:Name='Anim3D"+
        i + "' xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
        xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'>");
        sb.AppendLine("<DoubleAnimationUsingKeyFrames BeginTime='00:00:00.' +
        i + "' Storyboard.TargetProperty = '(UIElement.Projection).
        (PlaneProjection.RotationY)'>");
        sb.AppendLine("<EasingDoubleKeyFrame KeyTime='00:00:00'
        Value='650' />");
        ...
    }
}

```

La méthode `AppendLine` permet de concaténer chaque nouvelle ligne XAML. Pour réaliser cette opération rapidement, vous pouvez utiliser l'outil de remplacement de texte de Visual Studio ou de Blend. Une fois fait, il faut encore transformer les chaînes de caractères créées à chaque itération de la boucle en instance de type `Storyboard`. L'espace de noms `System.Windows.Markup` que nous avons référencé contient la classe `XamlReader` qui permet de transformer la chaîne de caractères en instance :

```

    foreach (UIElement Ui in LayoutRoot.Children)
    {
        int i = LayoutRoot.Children.IndexOf(Ui);

        StringBuilder sb = new StringBuilder("<Storyboard x:Name='Anim3D"+
        i + "' xmlns='http://schemas.microsoft.com/winfx/2006/xaml/

```

```

        presentation' xmlns:x='http://schemas.microsoft.com/winfx/2006/
        xaml'>");
sb.AppendLine("<DoubleAnimationUsingKeyFrames BeginTime='00:00:00.' +
    i + "' Storyboard.TargetProperty = '(UIElement.Projection).
    (PlaneProjection.RotationY)'>");
sb.AppendLine("<EasingDoubleKeyFrame KeyTime='00:00:00' Value='650' />");
...
sb.AppendLine("</Storyboard>");

//on transforme la chaîne de caractères créée
//en une instance de Storyboard
Storyboard NewClonedStoryboard = (Storyboard) XamlReader.Load
    (sb.ToString());

//on définit un décalage de temps permettant aux
//objets Storyboard de se lancer les uns après les autres
NewClonedStoryboard.BeginTime = TimeSpan.FromMilliseconds(i * 200);

//on définit la nouvelle cible à animer sur le Storyboard
//lui-même au lieu de chaque DoubleAnimation
Storyboard.SetTarget(NewClonedStoryboard, Ui);

//on ajoute le Storyboard créé au sein d'un dictionnaire,
//avec comme clé la référence de l'instance animée
DicoElementsSB.Add(Ui, NewClonedStoryboard);
}

```

Compilez l'application. Si à ce stade vous n'avez aucune erreur levée par le compilateur, c'est que tout s'est bien passé. Dans le cas contraire, l'erreur provient dans 80 % des cas de la concaténation *via* la méthode `AppendLine` ou de l'oubli des espaces de noms comme attribut de la balise `<Storyboard xmlns="..." xmlns:x="...">`. Comme vous le constatez, il est utile de conserver un accès aux instances `Storyboard` créées. La meilleure manière d'y avoir accès est d'utiliser un dictionnaire. Vous pourriez également utiliser la propriété `Resource` de votre application. Toutefois celle-ci peut contenir bien d'autres ressources que des animations. Ce choix vous appartient. Dans tous les cas, nous avons besoin de stocker les occurrences de `Storyboard` pour les réutiliser plus tard. Nous allons maintenant déclencher les animations lors de chaque clic de la souris sur la grille principale. Au sein de `Blend`, dans le panneau des événements entrez la chaîne `DeclencheAnim` pour l'événement `MouseButtonLeftButtonDown`. Dans le code C#, il suffit de parcourir le dictionnaire et d'appeler la méthode `Begin` pour chaque animation contenue :

```

private void DeclencheAnim(object sender, MouseButtonEventArgs e)
{
    foreach (Storyboard sb in DicoElementsSB.Values)
    {
        sb.Begin();
    }
}

```

Cet exercice permet de comprendre comment les développeurs et les designers interactifs peuvent travailler les uns avec les autres sans se gêner et tout en conservant le travail de chacun. Vous trouverez l'exercice finalisé dans l'archive *MenuAnim_Dynamique_Final.zip* du dossier *chap6* des exemples de l'ouvrage.

6.5 Les transformations relatives

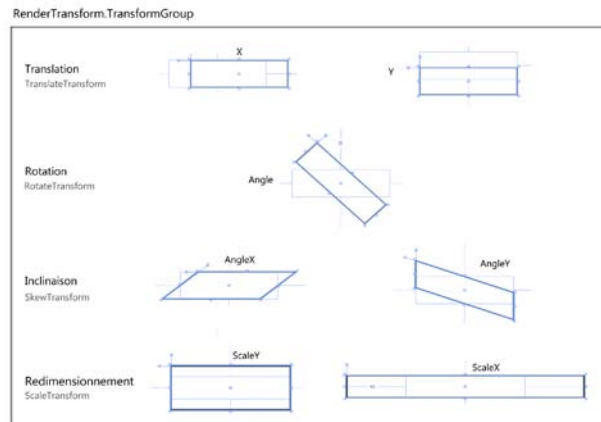
Dans cette section, nous allons aborder les transformations relatives de manière plus approfondie. Vous remarquez que lorsque vous modifiez la position d'un objet au sein d'une animation, Blend privilégie l'utilisation des transformations relatives. Nous aurions tendance à penser qu'au sein d'un composant Canvas, les propriétés `Canvas.Left` et `Canvas.Top` seraient employées pour animer la position, mais il n'en est rien. Là encore, Blend choisira d'animer `X` et `Y` du nœud `TranslateTransform` plutôt que d'interpoler les propriétés attachées par le conteneur (`Canvas.Left` et `Canvas.Top`). Les transformations relatives nous permettent d'échapper aux contraintes de mise en forme posée par le contexte conteneur. Pour cette raison, Blend les choisit en priorité. Il est donc pratique de savoir créer des instances de ce type dynamiquement *via* C#. Une autre raison plus terre à terre nous invite à générer ce type de nœud. Lorsque vous avez dupliqué la balle dans l'exercice précédent, celle-ci possédait déjà un nœud `RenderTransform.TransformGroup`, qui a donc été transmis aux copies de la balle. De ce fait, l'animation de rebond pouvait cibler les balles copiées. Toutefois, si l'affectation de cette propriété avait manqué, Blend aurait levé une erreur d'accès à la compilation. Dans certains cas, il vous faudra donc ajouter ce nœud dynamiquement.

6.5.1 Principes

Il existe quatre types de transformations relatives (voir Figure 6.31).

Figure 6.31

Les 4 types de transformations relatives.



Du point de vue d'un développeur, il est possible d'affecter de différentes manières des transformations relatives aux instances. La première consiste simplement à affecter la propriété d'une transformation de son choix :

```
TranslateTransform Tt = new TranslateTransform();
Tt.X = 200;
MonFrameworkElement.RenderTransform = Tt;
//décale MonFrameworkElement de 200 pixels vers la droite

ScaleTransform St = new ScaleTransform();
St.ScaleX = 2;
MonFrameworkElement.RenderTransform = St;
//écrase l'ancienne affectation de translation
//Multiplie par 2 la largeur de MonFrameworkElement
```

La deuxième est d'affecter plusieurs transformations relatives au même objet en les groupant dans une instance de type `TransformGroup` :

```
TranslateTransform Tt = new TranslateTransform();
Tt.X = 200;
ScaleTransform St = new ScaleTransform();
St.ScaleX = 2;

TransformGroup Tg = new TransformGroup();
Tg.Children.Add(Tt);
Tg.Children.Add(St);

MonFrameworkElement.RenderTransform = Tg;
//Affecte les deux transformations sans que l'une écrase l'autre
```

Pour un designer, cela est légèrement différent car Blend génère par défaut un nœud XAML de type `TransformGroup` qui contient les quatre types de transformation :

```
<UIElement.RenderTransform>
  <TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform/>
    <TranslateTransform/>
  </TransformGroup>
</UIElement.RenderTransform>
```

Dès lors, pour des raisons de communication et d'homogénéité, le développeur doit, dans certains cas, cibler ou affecter les transformations contenues dans un nœud XAML `TransformGroup`. Cela pour la bonne raison que le nœud XAML est utilisé par le designer interactif au sein de Blend.

6.5.2 Tester la présence d'une instance *TransformGroup*

Avant d'ajouter ou de cibler un nœud de type `TransformGroup`, la première chose à faire est de tester la propriété `RenderTransform`. Quelle que soit l'instance de `UIElement`, cette propriété n'est pas nulle car sa valeur correspond à la matrice de transformation de l'objet par défaut (`MatrixTransform`). Toutefois, vous pouvez tester si elle contient une instance de type `TransformGroup`. Si c'est le cas, cela signifie que la propriété `RenderTransform` de l'instance a été modifiée sous Blend ou qu'elle a été affectée par code. Vous devrez si possible éviter tout écrasement des transformations modifiées par le graphiste dans Blend.

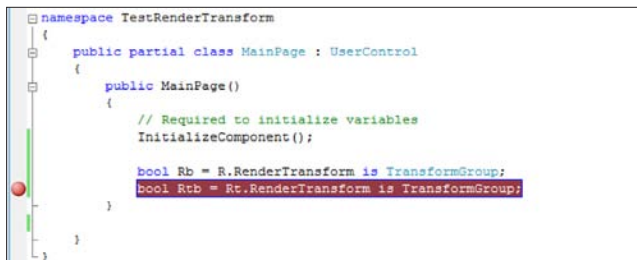
Créez un nouveau projet dans Blend et deux instances de `Rectangle` dans `LayoutRoot`. Nommez l'un R et l'autre Rt. Sur le `Rectangle` Rt, modifiez les transformations relatives *via* le panneau des propriétés. Ouvrez le projet sous Visual Studio. Nous allons utiliser cet environnement pour tester les transformations relatives de chaque `Rectangle` :

```
public MainPage()
{
    InitializeComponent();
    bool Rb = R.RenderTransform is TransformGroup;
    bool Rtb = Rt.RenderTransform is TransformGroup;
}
```


Ce code ne suffit pas, le mieux serait de poser un point d'arrêt pour vérifier quelles valeurs prennent Rb et Rtb. Cliquez à gauche de la ligne contenant la déclaration de Rtb (voir Figure 6.32).

Figure 6.32

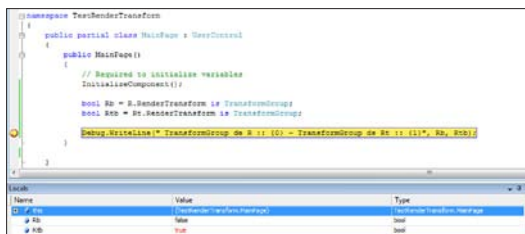
Poser un point d'arrêt dans Visual Studio.



Lancez la compilation. Elle est interrompue lorsque la méthode MainPage_Loaded est déclenchée. Pour connaître la valeur de Rb et de Rtb, il vous suffit de survoler l'opérateur is qui évalue le type. La variable Rtb n'est pas encore affectée à cet instant, c'est pourquoi le survol de la variable Rtb renvoie false. Visual Studio affiche la valeur sous chaque variable (Figure 6.33).

Figure 6.33

Afficher les valeurs à l'exécution grâce au débogueur.



Non seulement Visual Studio affiche la valeur au survol, mais il renseigne la valeur des variables accessibles depuis la méthode MainPage_Loaded au sein du panneau Locals. Pour tester une référence ou une valeur, vous pouvez également utiliser la fenêtre de sortie de Visual Studio. La classe statique Debug possède la méthode WriteLine permettant d'écrire en fenêtre de sortie. Elle se trouve dans l'espace de noms System.Diagnostics. Voici un exemple d'utilisation :

```
public MainPage()
{
    InitializeComponent();
    bool Rb = R.RenderTransform is TransformGroup;
    bool Rtb = Rt.RenderTransform is TransformGroup;

    Debug.WriteLine(" TransformGroup de R :: {0} - TransformGroup de Rt :: {1}", Rb, Rtb);
}
```

Le raccourci F5 sous Visual Studio lance automatiquement le débogueur. Tant que celui-ci est exécuté, le code logique ou déclaratif sous Visual Studio n'est pas modifiable. Désactivez le débogueur sous Visual Studio pour y accéder de nouveau *via* le raccourci Maj+F5. Cet exercice est disponible dans *TestRenderTransform.zip* du dossier *chap6*.

6.5.3 Affecter la propriété *RenderTransform*

Maintenant que nous savons tester la présence du nœud élément `TransformGroup`, nous allons le créer de toute pièce lorsqu'il est absent. Pour des raisons de standard d'écriture XAML, nous allons le créer de la même manière que le ferait Blend. L'exemple de code ci-dessous gère entièrement l'animation, la création des objets graphiques et des transformations relatives :

```
public partial class MainPage : UserControl
{
    Storyboard Sb;
    DoubleAnimation DaX;
    DoubleAnimation DaY;

    public MainPage()
    {
        // Required to initialize variables
        InitializeComponent();

        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }

    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        Ellipse Balle = CreateBalle();

        Balle = (Ellipse)CreateRenderTransformNode(Balle);

        CreateAnimationBalle(Balle);

        LayoutRoot.MouseLeftButtonUp += OnLayoutClick;
    }

    private Ellipse CreateBalle()
    {
        Ellipse MaBalle = new Ellipse();
        MaBalle.Width = 100;
        MaBalle.Height = 100;
        MaBalle.HorizontalAlignment = HorizontalAlignment.Left;
        MaBalle.VerticalAlignment = VerticalAlignment.Top;
        MaBalle.Fill = new SolidColorBrush(Colors.Gray);
        LayoutRoot.Children.Add(MaBalle);
        return MaBalle;
    }

    private UIElement CreateRenderTransformNode(UIElement balle)
    {
        //l'objectif est de standardiser l'écriture du nœud
        //RenderTransform. Pour cette raison, nous le créons
        //de la même manière que Blend
        if ( (balle.RenderTransform is TransformGroup))
        {
            TransformGroup Tg = new TransformGroup();
            Tg.Children.Add(new ScaleTransform());
            Tg.Children.Add(new SkewTransform());
            Tg.Children.Add(new RotateTransform());
            Tg.Children.Add(new TranslateTransform());
            balle.RenderTransform = Tg;
        }
        return balle;
    }
}
```

```

private void CreateAnimationBalle(Ellipse balle)
{
    Sb = new Storyboard();
    DaX = new DoubleAnimation();
    DaY = new DoubleAnimation();

    DaX.Duration = TimeSpan.FromSeconds(0.6);
    DaY.Duration = TimeSpan.FromSeconds(0.6);

    DaX.EasingFunction = new ElasticEase();
    DaY.EasingFunction = new ElasticEase();

    Storyboard.SetTarget(DaX, balle);
    Storyboard.SetTarget(DaY, balle);

    Storyboard.SetTargetProperty(DaX, new PropertyPath("UIElement.
        RenderTransform).(TransformGroup.Children)[3].
        (TranslateTransform.X)"));
    Storyboard.SetTargetProperty(DaY, new PropertyPath("UIElement.
        RenderTransform).(TransformGroup.Children)[3].
        (TranslateTransform.Y)"));

    Sb.Children.Add(DaX);
    Sb.Children.Add(DaY);
}

void OnLayoutClick (object sender, MouseButtonEventArgs e)
{
    // la balle se déplacera lorsque vous cliquerez
    // n'importe où au sein du conteneur LayoutRoot
    DaX.To = e.GetPosition(null).X - 50 ;
    DaY.To = e.GetPosition(null).Y - 50 ;
    Sb.Begin();
}

```

Compilez votre application. Lorsque vous cliquez n'importe où sur la scène, l'animation est mise à jour et la balle se déplace à l'endroit cliqué. Vous trouverez cet exercice ici : *chap6/pleinEcran_maquetteAnimee_1.zip*.

Créer un nœud élément `RenderTransform` peut se révéler assez fastidieux. De plus, son chemin d'accès n'est pas forcément simple à renseigner. À cette fin, vous pouvez soit vous aider de Blend, soit utiliser la bibliothèque `ProxyRenderTransform`. Nous allons voir son utilisation dans la prochaine section.

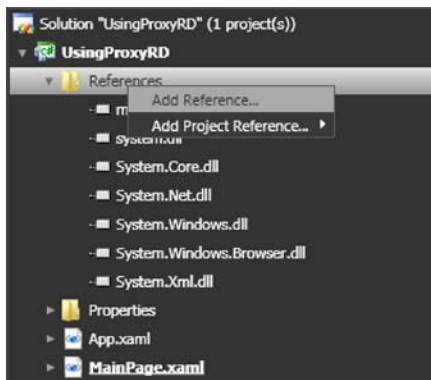
6.5.4 La bibliothèque *ProxyRenderTransform*

6.5.4.1 Principes et utilisation simple

Assurez-vous d'avoir téléchargé la bibliothèque au préalable. Elle est à votre disposition sur le portail CodePlex, à l'adresse : <http://proxyrd.codeplex.com>. Puis, référencez-la au sein d'un nouveau projet nommé `UsingProxyRD` (voir Figure 6.34).

Figure 6.34

Référencer la bibliothèque ProxyRenderTransform.



Ensuite, importez l'espace de noms au sein de votre code :

```
using ProxyRenderTransform;
```

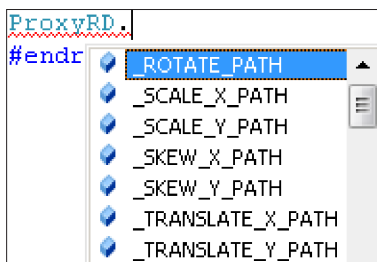
Cette bibliothèque fournit un certain nombre d'avantages. Elle peut ajouter dynamiquement un groupe de transformations relatives aux objets qui n'en possèdent pas. Elle permet également de récupérer ou d'affecter la valeur de chaque transformation *via* l'utilisation de méthodes d'extension. Vous pourrez par exemple écrire :

```
MonUIElement.SetScaleX(2);  
//ou encore  
MonUIElement.SetX(50);
```

Ces méthodes sont accessibles directement grâce à l'IntelliSense que vous fournit Visual Studio (voir Figure 6.35).

Figure 6.35

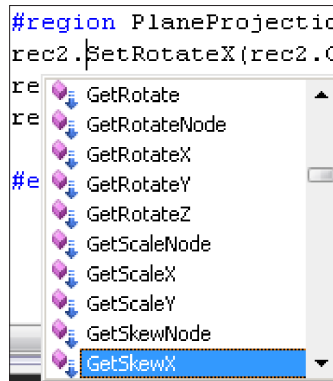
IntelliSense de la bibliothèque ProxyRenderTransform dans Visual Studio.



Comme vous pouvez le voir, cette bibliothèque donne également accès aux projections 3D (voir Chapitre 10). Si vous souhaitez récupérer la référence d'un nœud de transformation contenu dans le groupe généré par défaut sous Blend, vous pouvez appeler les méthodes GetRotateNode, GetScaleNode, etc. Pour finir, la bibliothèque vous permet de récupérer les chemins d'accès aux différents nœuds en renvoyant des objets typés PropertyPath (voir Figure 6.36).

Figure 6.36

Récupération des chemins d'accès aux transformations.



Vous pouvez désormais écrire :

```
Storyboard.SetTargetProperty(DaX, ProxyRD._TRANSLATE_X_PATH );
```

Cette écriture est non seulement simple, mais également facile d'accès *via* l'IntelliSense de Visual Studio. L'IntelliSense de Blend ne donne pas accès dynamiquement aux méthodes d'extension.

6.5.4.2 Animer avec le métronome DispatcherTimer

Nous allons faire une animation très simple pour illustrer son utilisation. Dans un nouveau projet nommé AnimProg, ajoutez la bibliothèque ProxyRenderTransform, puis créez un composant Rectangle au sein de la grille principale LayoutRoot. Placez-le en haut à gauche de la grille et nommez-le Suiveur. Dans le panneau des événements de la grille principale, entrez GetMousePos pour l'événement MouseMove. Vous allez utiliser la classe DispatcherTimer pour modifier la position du rectangle selon un intervalle de temps. Voici le code commenté permettant au rectangle de suivre constamment la souris avec un léger effet de ralenti :

```
...
using ProxyRenderTransform;
using System.Windows.Threading;
public partial class MainPage : UserControl
{
    //la destination du rectangle mis à jour lors de l'événement Tick
    //qui se déclenche selon l'intervalle de temps spécifié
    double NewDestX=0;
    double NewDestY=0;
    //Stocke les coordonnées actuelles de la souris
    double MouseX;
    double MouseY;

    public MainPage()
    {
        // Required to initialize variables
        InitializeComponent();

        DispatcherTimer dt = new DispatcherTimer();

        dt.Interval = TimeSpan.FromMilliseconds(10);

        dt.Tick += new EventHandler(dt_Tick);
    }
}
```

```

dt.Start();

}

void dt_Tick(object sender, EventArgs e)
{
    //ces expressions permettent de créer l'effet de déplacement ralenti
    //0.1 correspond au facteur de ralenti, plus ce facteur est faible
    //plus le ralentissement s'accroît
    NewDestX += ((MouseX - Suiveur.Width/2) - (double)Suiveur.GetX()) * 0.1;
    NewDestY += ((MouseY - Suiveur.Height/2) - (double)Suiveur.GetY()) * 0.1;
    Suiveur.SetX(NewDestX);
    Suiveur.SetY(NewDestY);
}

//lors du déplacement de la souris, on récupère sa position
private void GetMousePos(object sender, MouseEventArgs e)
{
    MouseX = e.GetPosition(null).X;
    MouseY = e.GetPosition(null).Y;
}
}

```

L'animation est très fluide et ne repose pas sur la classe `Storyboard`. Vous pouvez pratiquement réaliser n'importe quel type d'animation grâce à la classe `DispatcherTimer`. Dans tous les cas, l'animation – même écrite avec des nœuds éléments `Storyboard` – est basée en interne sur ce type de fonctionnement.

INFO

Il n'est pas réellement possible de récupérer les coordonnées de la souris en dehors d'un gestionnaire d'événements pour la bonne raison que la classe "Mouse" n'existe pas. Dans le cas contraire, nous n'aurions pas besoin d'écouter l'événement `MouseMove` et notre code serait plus optimisé.

Vous trouverez le projet dans l'archive *AnimProg.zip* du dossier *chap6* des exemples.

Nous allons maintenant animer avec la classe `Math`. Cette dernière centralise de nombreuses méthodes dont certaines sont liées à la trigonométrie. Cette discipline est assez intéressante car elle permet de coder des animations complexes en quelques lignes seulement. Nous allons utiliser deux méthodes, `Math.Sin` et `Math.Cos`, pour générer un mouvement circulaire. Créez un nouveau projet et nommez-le `AnimationCirculaire_Math`. Référez la bibliothèque `ProxyRender-Transform`. Créez ensuite deux instances de type `Ellipse`. La première, nommée `Axe`, symbolise l'axe de rotation. La deuxième, appelée `Satellite`, va subir une rotation autour de l'axe. Elles doivent toutes deux être alignées horizontalement et verticalement au centre de la grille. Voici le code permettant de générer une rotation *via* `DispatcherTimer` :

```

//Vitesse angulaire exprimée en radians
//elle dépend directement de l'intervalle de l'objet DispatcherTimer
double vitesseAngulaire = .1;
//angle initial
double angle = 0; //exprimé en radian
//Le rayon de rotation
double rayon = 100;
//l'axe de rotation
Point CentreRotation;

DispatcherTimer Dt = new DispatcherTimer();

```

```

public MainPage()
{
    InitializeComponent();
    Loaded += new RoutedEventHandler(MainPage_Loaded);
}

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    //Lorsque l'application est chargée, on récupère les coordonnées
    //de l'objet servant d'axe de rotation
    CentreRotation = new Point((double)Axe.GetX(), (double)Axe.GetY());
    Dt.Interval = TimeSpan.FromMilliseconds(30);
    Dt.Tick += new EventHandler(OnTick);
    Dt.Start();
}

void OnTick(object sender, EventArgs e)
{
    //On récupère les nouvelles coordonnées du satellite
    //en fonction de l'angle, du rayon et du centre de la rotation
    Satellite.SetX(CentreRotation.X + Math.Sin(angle) * rayon);
    Satellite.SetY(CentreRotation.Y + Math.Cos(angle) * rayon);

    //À chaque appel de la méthode, on redéfinit l'angle
    //en lui ajoutant la vitesse angulaire
    angle += vitesseAngulaire;
}

```

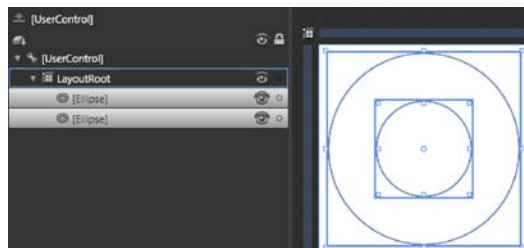
Cet exercice est disponible dans l'archive *AnimationCirculaire_Math.zip*, du dossier *chap6* des exemples de cet ouvrage.

6.5.5 Effets antagonistes et complémentaires

Ce type d'effet est très utile en matière de graphisme car il permet de créer des visuels à la fois faciles à mettre à jour et impressionnants. Ces effets reposent essentiellement sur l'imbrication, l'ordre hiérarchique des composants, ainsi que sur les transformations relatives. Nous allons simuler de la 3D sans utiliser la projection qui est disponible depuis la version 3 de Silverlight. La projection 3D est pratique, mais elle offre le désavantage d'un rendu parfois pixélisé (voir Chapitre 10). Nous n'aurons pas ce problème puisque nous utiliserons le mode vectoriel 2D uniquement. Créez un nouveau projet nommé *DisqueAnim*. Instanciez une ellipse de 200 pixels de largeur par 200 pixels de hauteur avec un fond blanc et une bordure de couleur noir. Faites-en une deuxième mais de 100 × 100 pixels. Sélectionnez-les, puis *via* le menu d'alignement centrez-les verticalement et horizontalement (voir Figure 6.37).

Figure 6.37

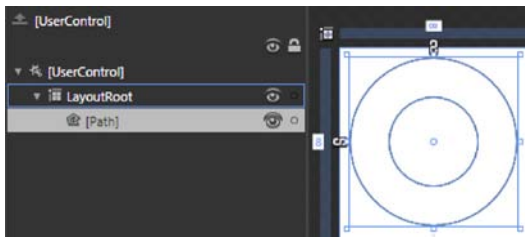
Les deux instances de Ellipse alignées.



Vous allez soustraire la plus petite à la plus grande afin de percer cette dernière. Pour cela, vous devez utiliser le menu Combiner accessible *via* un clic droit ou par le menu Objet. Sélectionnez les deux instances d'Ellipse puis, dans le menu Combiner, choisissez l'opération Soustraire. L'ordre de sélection compte. Vous devriez récupérer un tracé vectoriel de type Path ressemblant à un tore (voir Figure 6.38).

Figure 6.38

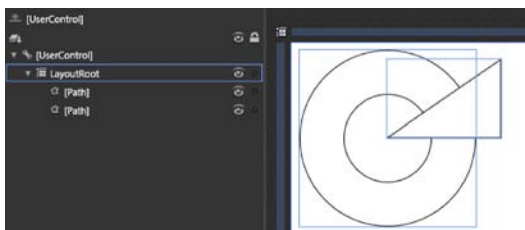
Opération de soustraction sur les Ellipses.



Créez un triangle droit grâce à l'outil Plume, celui-ci doit partir du centre du tore vers la droite (voir Figure 6.39).

Figure 6.39

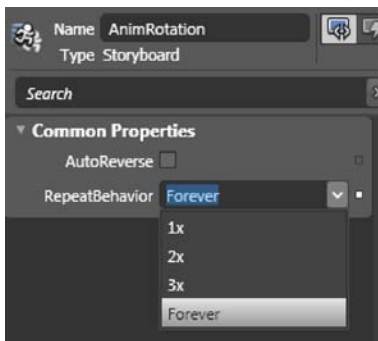
Création d'un triangle droit via l'outil Plume.



Sélectionnez le triangle, puis le tore, et répétez l'opération de soustraction. Nous allons maintenant animer le tracé obtenu avec une rotation. Créez une nouvelle animation nommée *AnimRotation*, puis ouvrez le panneau des transformations relatives. À la seconde 0, créez une clé d'animation avec une rotation de 0 degré. Positionnez la tête de lecture à la seconde 2, puis saisissez une rotation de 360 degrés. Sélectionnez le Storyboard, pour cela cliquez sur son nom au-dessus de l'arbre visuel. Au sein du panneau des propriétés, dans le champ *RepeatBehavior*, sélectionnez la valeur *Forever*. L'animation de rotation se répétera ainsi indéfiniment (voir Figure 6.40).

Figure 6.40

Lecture d'un Storyboard en boucle via la propriété RepeatBehavior.



Sélectionnez la grille principale, dans le panneau des événements, entrez OnLoaded pour Loaded. Au sein du code C#, déclenchez l'animation *via* la méthode Begin :

```
private void OnLoaded(object sender, RoutedEventArgs e)
{
    AnimRotation.Begin();
}
```

Le disque est maintenant animé dès le chargement de l'application. Vous allez créer un effet antagoniste. Pour cela, groupez le disque au sein d'un Canvas. Sélectionnez ce dernier, puis au sein de l'onglet des transformations relatives, entrez la valeur 0,25 pour l'échelle ScaleY. Recompilez ; cette fois, le disque subit toujours la rotation, mais la totalité de cette dernière est aplatie, simulant un effet de perspective.

Si vous aviez modifié l'échelle directement au sein du disque, vous n'auriez pas eu ce type de résultat car vous auriez aplati le disque et non l'interpolation elle-même (voir Figures 6.41 et 6.42).

Ce n'est qu'un simple exemple et vous pouvez imaginer un grand nombre d'utilisations différentes de ce concept. Vous trouverez le projet final, *AnimProg.zip*, dans le dossier *chap6* des exemples du livre.

Figure 6.41

Modification de l'échelle directement sur le tracé.

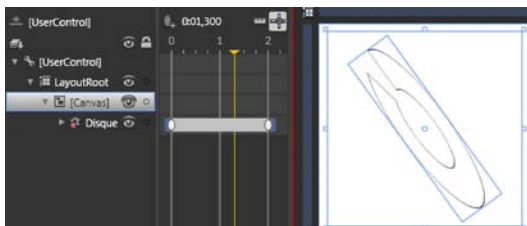
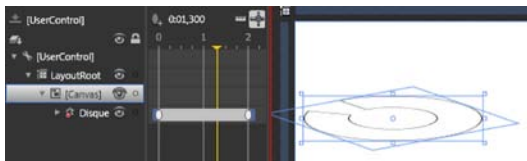


Figure 6.42

Effet complémentaire suite à la modification de l'échelle sur le conteneur Canvas.



6.6 Animer des particules

Si un type d'animation passe pour être rébarbatif et difficile auprès des animateurs, il s'agit bien de l'animation des particules. Aucun outil dédié n'est présent au sein de Blender à cette fin. De plus, une particule possède des comportements soit aléatoires soit dictés par des ensembles complexes. Par exemple, la poussière, la pluie, les oiseaux et même les humains peuvent être considérés comme des particules dès lors qu'on les examine de loin. Ainsi, un animateur traditionnel, dont le métier est avant tout d'animer un nombre restreint d'objets, est dépassé par la somme colossale de travail à réaliser pour ce genre de visuel.

Toutefois, cela n'est pas inaccessible : les mouvements de foule du film *Akira* ont été réalisés par des dizaines d'animateurs et sans développement informatique. Cela reste un cas particulier. Dans le film d'animation du *Bossu de Notre Dame*, la foule regroupée autour de Notre Dame est

quant à elle générée par ordinateur. L'idée est d'associer différents mouvements, comportements, couleurs et formes à chaque individu présent sur la place. Cela confère au dessin animé beaucoup de réalisme, mais ne requiert pas pour autant une armée d'animateurs. Par la suite, un algorithme informatique associe aléatoirement chaque composant constituant un individu, afin de lui donner un caractère unique au sein de la foule. C'est exactement ce que nous allons faire dans cette section pour donner l'illusion de la diversité.

6.6.1 Exemples d'un fond sous-marin

Téléchargez le projet *chap6/BubbleParticule.zip*. Décompressez le fichier, puis chargez le projet dans Expression Blend. Vous avez un aperçu du visuel final de cet exercice à la Figure 6.43.

Figure 6.43

Visuel du projet finalisé.



6.6.1.1 Mise en place

Au sein du panneau des projets, vous constatez la présence d'un composant personnalisé nommé *Bulle*. Vous n'avez pas besoin de savoir pour l'instant comment celui-ci a été réalisé (vous pourrez vous en faire une idée au Chapitre 12).

Un composant personnalisé peut être instancié de la même manière que n'importe quel autre *via* le code C# ou la bibliothèque de composants accessible dans Blend. La grille principale du projet contient deux composants dont seul le premier nous intéresse : le composant *Canvas* nommé *Emetteur*. C'est ce composant qui va contenir nos particules. Celles-ci seront en fait des bulles qui remonteront vers la surface en zigzagant. Pour déplacer les bulles, nous n'utiliserons pas les *RenderTransform* car cela engendrerait la création d'une transformation relative de type *TranslateTransform* et ajouterait une charge processeur supplémentaire.

INFO

Il est parfois utile d'éviter l'utilisation des nœuds *RenderTransform*. Vous pourrez vous en passer dans deux cas précis : lorsque vous souhaitez déplacer dans l'espace un objet vectoriel ou le redimensionner. Les opérations de rotation (*RotateTransform*) ou d'inclinaison (*SkewTransform*) ne sont en revanche pas réalisables sans l'utilisation des transformations relatives ou d'une matrice de transformation. Pour déplacer simplement un contrôle, il vous suffit d'utiliser le composant *Canvas*. Ce composant permet le placement des objets sans contrainte de repositionnement. Cela est très pratique si vous souhaitez vous écarter au maximum d'une mise en forme traditionnelle.

Pour redimensionner un composant sans sa propriété `RenderTransform`, il vous faut utiliser le composant `ViewBox`. Vous trouverez ce contrôle dans la bibliothèque `Silverlight Toolkit` à l'adresse : <http://www.codeplex.com/Silverlight>. Lorsque vous changez la largeur (`Width`) et la hauteur (`Height`) de ce conteneur à enfant unique, il redimensionne son contenu comme si celui-ci avait eu sa propriété `RenderTransform` affectée d'une instance `ScaleTransform` modifiée.

Vous constatez également la présence d'une animation si vous ouvrez la liste des objets de type `Storyboard` dans `Blend`. Vous pouvez modifier cette liste à loisir pour donner plus de réalisme à la scène. Ouvrez le fichier `MainPage.xaml.cs`. Vous constatez que l'on commence par démarrer l'animation des rais de lumière. Vous remarquez également que celle-ci est cinq fois moins rapide que l'aperçu dans `Blend` car sa propriété `SpeedRatio` est affectée à `0.2` :

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    //initialisation de l'animation des rais de lumière
    RaiesLumieres.SpeedRatio = 0.2;
    RaiesLumieres.Begin();
}
```

Nous allons maintenant créer des instances du composant personnalisé `Bulle` selon un laps de temps.

6.6.1.2 Créer les particules

Il faudra limiter le nombre de particules. Pour cela, il nous suffit de définir une constante représentant le nombre maximum de bulles générées :

```
public partial class MainPage : UserControl
{
    //On définit un nombre de bulles maximum
    private const int NOMBRE_BULLE = 50;
    ...
}
```

Chaque bulle de notre visuel est conçue à un instant différent. Pour générer les bulles, le mieux est d'utiliser la classe `DispatcherTimer` contenue dans l'espace de noms `System.Windows.Threading`. Nous pouvons ajouter une instance de cette classe comme membre de la classe principale, puis la configurer dans notre méthode `Loaded` :

```
public partial class MainPage : UserControl
{
    //On définit un nombre de bulles maximum
    private const int NOMBRE_BULLE = 50;

    //on crée une instance de l'objet DispatcherTimer
    //qui va nous servir à créer des bulles selon un intervalle de temps
    DispatcherTimer monTimer = new DispatcherTimer();

    public MainPage()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainPage_Loaded);
    }

    void MainPage_Loaded(object sender, RoutedEventArgs e)
```

```

{
    //initialisation de l'animation des rais de lumière
    RaiesLumieres.SpeedRatio = 0.2;
    RaiesLumieres.Begin();

    //Intervalle de temps entre la création de chaque bulle
    monTimer.Interval = TimeSpan.FromMilliseconds(200);

    //Souscription de l'écouteur monTimer_Tick à l'événement Tick
    monTimer.Tick += new EventHandler(monTimer_Tick);

    //Démarrage du métronome
    monTimer.Start();
}

```

Il nous faut un objet de type Random. Celui-ci possède plusieurs méthodes capables de calculer un nombre aléatoire. C'est donc grâce à lui que nous positionnerons les bulles aléatoirement. Sous la déclaration de l'instance DispatcherTimer, créez une instance de l'objet Random. En le déclarant comme membre de classe, nous le rendons accessible depuis n'importe quelle méthode de la classe MainPage :

```

public partial class MainPage : UserControl
{
    //On définit un nombre de bulles maximum
    private const int NOMBRE_BULLE = 50;

    //on crée une instance de l'objet DispatcherTimer
    //Il va nous servir à créer des bulles selon un intervalle de temps
    DispatcherTimer monTimer = new DispatcherTimer();

    //On initialise une instance Random dont on va se servir
    //pour gérer les mouvements aléatoires
    private Random rnd = new Random();

    ...
}

```

Il ne nous reste plus qu'à instancier puis à positionner aléatoirement les bulles au sein de la méthode monTimer_Tick. Chacune des bulles sera un enfant du Canvas Emetteur.

```

void monTimer_Tick(object sender, EventArgs e)
{
    //On commence par vérifier que l'on n'a pas atteint le nombre maximum
    //de bulles générées.
    if (Emetteur.Children.Count==NOMBRE_BULLE)
    {
        //Si c'est le cas, on arrête le métronome
        monTimer.Stop();
        //et on sort de la méthode
        return;
    }

    //dans le cas contraire, on crée une nouvelle bulle
    Bulle MaBulle = new Bulle();

    //Pour donner l'impression de la diversité
    //on affecte la largeur et la hauteur de la bulle avec
    //une valeur aléatoire entre 10 et 20
    MaBulle.Width = MaBulle.Height = rnd.Next(10, 20);

    //on modifie l'opacité par défaut aléatoirement entre 0.2 et 0.7
}

```

```

//soit entre 20 % et 70 %
MaBulle.Opacity = (rnd.NextDouble()*0.5) + 0.2D;

//on crée deux valeurs aléatoires
double PosX = rnd.NextDouble() * Emetteur.ActualWidth;
double PosY = rnd.NextDouble() * Emetteur.ActualHeight;

//La première est affectée à la propriété Canvas.Left
Canvas.SetLeft(MaBulle, PosX);
//la seconde à la propriété Canvas.Top de chaque bulle
Canvas.SetTop(MaBulle, PosY);

//On ajoute la bulle créée comme enfant du Canvas Emetteur
Emetteur.Children.Add(MaBulle);

//Pour finir, on appelle la méthode qui va créer une animation
//propre à la bulle créée
CreerAnimationAleatoire(MaBulle);
}

```

Nous allons voir en détail certaines parties de ce code. La première concerne l'affectation de la hauteur et de la largeur de chaque objet Bulle. La méthode `Next` de l'objet `Random` renvoie des entiers. Lorsque vous passez deux arguments, vous demandez en fait un entier aléatoire dans une plage de valeurs. Les propriétés `Width` et `Height` étant typée `Double`, nous procédons ainsi à un transtypage implicite. La valeur entière récupérée est convertie en type `Double`. Sur cette ligne de code, nous effectuons également une triple égalité. Cela nous permet d'avoir des dimensions dont le rapport initial est conservé. Concernant l'opacité, elle est obtenue grâce à la méthode `NextDouble` de l'objet `Random`. Cette méthode renvoie un nombre de type `Double` aléatoire situé entre 0 et 1. Multiplier le retour de cette méthode par 0.5 revient à demander un nombre entre 0 et 0.5. Ajouter 0.2D signifie que vous ajoutez la valeur 0.2 de type `Double`. Nous obtenons donc une opacité située entre 0.2 et 0.7. Cela donne un effet de profondeur à la scène. De la même manière, pour positionner la bulle aléatoirement, nous utilisons la méthode `NextDouble`. Les propriétés `ActualWidth` et `ActualHeight` renvoient en fait la largeur et la hauteur mises à jour selon les contraintes de redimensionnement du site. Ces valeurs sont réellement utiles lorsque votre conteneur possède des propriétés `Width` et `Height` en mode `Auto`. C'est notre cas, la largeur (`Width`) du `Canvas Emetteur` est constamment mise à jour selon le redimensionnement de la fenêtre de navigation. Les propriétés `Canvas.Left` et `Canvas.Top` sont des propriétés attachées aux objets contenus dans un conteneur de type `Canvas`. C'est le cas des instances de `Bulle` que nous créons. Il existe deux manières d'affecter des propriétés attachées, les voici dans le cas de `Left` et `Top` :

```

//1 - on utilise les méthodes statiques de la classe Canvas

Canvas.SetTop(MonInstance, 100);
//positionne mon instance à 100 pixels du bord haut
Canvas.SetLeft(MonInstance, 100);
//positionne mon instance à 100 pixels du bord gauche

//2 - on utilise la méthode SetValue propre aux instances de
// DependencyObject

MonInstance.SetValue(Canvas.TopProperty, 100);
//on obtient le même résultat
MonInstance.SetValue(Canvas.LeftProperty, 100);

```

Compilez votre projet pour voir le résultat. Les bulles sont créées aléatoirement au sein du conteneur Canvas (Figure 6.44).

Figure 6.44

Création des bulles aléatoirement dans le composant Emetteur.



6.6.2 Créer l'animation des bulles

Nous allons maintenant nous concentrer sur l'animation de chaque bulle. Le code sera centralisé dans la méthode `CreerAnimationAleatoire` :

```
private void CreerAnimationAleatoire(Bulle maBulle)
{
    //on commence par instancier une nouvelle ressource Storyboard
    Storyboard AnimBulle = new Storyboard();
    ...
}
```

Cette méthode reçoit en paramètre la nouvelle bulle créée, puis lui affecte deux animations uniques. La première animation cible l'axe verticale Y représenté par la propriété `Canvas.Top` de chaque Bulle. La seconde interpole la position des particules sur l'axe X, défini par la propriété `Canvas.Left`.

6.6.1.1 Animation verticale

Pour l'axe vertical, cela est assez simple car l'animation générée avec C# est semblable à celle créée à la section 6.4.2.1. Pourtant elle diffère en plusieurs points. Tout d'abord l'animation débute à partir de l'endroit où la bulle a été créée aléatoirement. Pour récupérer la position d'un objet au sein d'un Canvas, il vous suffit d'utiliser ses méthodes statiques :

```
Canvas.GetTop(MonObjetContenu);
//renvoie la position de l'objet sur l'axe vertical si celui-ci
//est contenu dans un Canvas
Canvas.GetLeft(MonObjetContenu);
//renvoie la position de l'objet sur l'axe horizontal si celui-ci est
//contenu dans un Canvas
//Dans les deux cas, il est inutile de préciser
//la référence du conteneur car celui-ci est implicitement
//le parent de l'instance MonObjetContenu
```

Une fois récupérée, nous pouvons affecter cette valeur à la propriété `From` de notre animation. Nous définissons l'opposé de la hauteur de la page comme valeur de destination. La deuxième particularité de cette animation est qu'elle est jouée en boucle. Ceci se fait en affectant la valeur `Forever` à la propriété `RepeatBehavior`. Cette propriété est héritée de la classe abstraite `Time-`

Line. Concrètement, lorsqu'une bulle a terminé son parcours, elle revient à son point de départ et rejoue son déplacement. Cela évite de générer constamment de nouvelles bulles et allège la charge processeur en réutilisant celles qui sont déjà instanciées :

```
//Cette animation se joue en boucle indéfiniment sur l'axe vertical
//Une fois arrivée à sa destination, la bulle recommence son trajet
DaY.RepeatBehavior = RepeatBehavior.Forever;
```

La dernière particularité est de définir une durée aléatoire pour chaque animation verticale *via* l'objet Random :

```
//L'animation durera entre 8 et 14 secondes maximum
//grâce à l'utilisation d'une valeur aléatoire
DaY.Duration = TimeSpan.FromSeconds((rnd.NextDouble() * 6) + 8);
```

Voici le code complet générant cette animation :

```
//on crée une sous-séquence d'animation
DoubleAnimation DaY = new DoubleAnimation();

//Celle-ci cible l'axe Y grâce à la propriété Canvas.Top de chaque Bulle
Storyboard.SetTargetProperty(DaY, new PropertyPath(Canvas.TopProperty));

//Cette seconde sous-séquence cible également l'instance maBulle
Storyboard.SetTarget(DaY, maBulle);

DaY.RepeatBehavior = RepeatBehavior.Forever;

DaY.From = Canvas.GetTop(maBulle);

DaY.To = -LayoutRoot.ActualHeight - 50;

DaY.Duration = TimeSpan.FromSeconds((rnd.NextDouble() * 6) + 8);
```

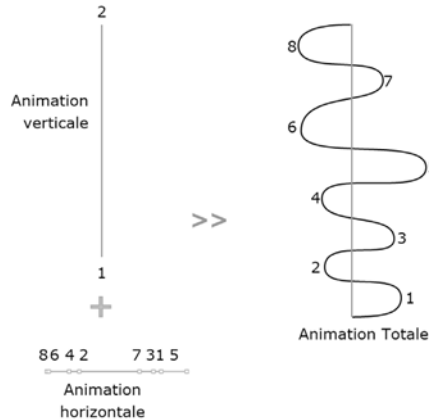
Cette animation est assez simple, ce n'est pas vraiment elle qui donnera son caractère à l'animation car l'objet Random est très peu utilisé. Nous allons maintenant voir comment déplacer aléatoirement chaque bulle sur l'axe X.

6.6.1.2 Animation horizontale

Cette animation est un peu plus compliquée. Nous allons tout d'abord essayer de comprendre l'interpolation dont nous avons besoin. La bulle doit remonter lentement tout en zigzagant à gauche et à droite. Si elle possède une amplitude constante à gauche et droite, nous aurons un mouvement trop mécanique et donc pas assez aléatoire et chaotique. Il faut donc que celle-ci possède une amplitude différente lorsqu'elle va de droite à gauche. Si l'on additionne l'animation verticale à une animation horizontale de gauche à droite, on obtient aisément l'effet recherché (voir Figure 6.45).

Figure 6.45

Conjugaison de l'animation verticale et horizontale.



Notre objectif consiste à créer des clés d'animation qui prendront une valeur aléatoire autour de la position d'origine (`Canvas.Left`) de la bulle. Nous avons besoin de créer une animation un peu plus complexe qu'une `DoubleAnimation` puisque celle-ci doit contenir des clés d'animation. Comme vu à la section 6.1, il nous faut instancier une animation de type `DoubleAnimationUsingKeyFrames`. De plus, celle-ci doit être un mouvement en boucle sans à-coup. Il suffit de préciser qu'elle joue à l'envers une fois arrivée à son terme, puis qu'elle joue en boucle. Voici le code permettant de la générer :

```
//On va créer une animation contenant des clés d'animation, l'avantage
//est de pouvoir générer une animation plus complexe et chaotique
DoubleAnimationUsingKeyFrames DaX = new DoubleAnimationUsingKeyFrames();

//Cette seconde sous-séquence cible la propriété Canvas.Left
//pour déplacer les bulles sur l'axe X
Storyboard.SetTargetProperty(DaX, new PropertyPath(Canvas.LeftProperty));

//Cette seconde sous-séquence cible l'instance maBulle
Storyboard.SetTarget(DaX, maBulle);

//l'animation sur cet axe doit être lue en boucle
DaX.RepeatBehavior = RepeatBehavior.Forever;

//Elle doit également se jouer dans les deux sens
DaX.AutoReverse = true;

//on crée 5 clés d'animation avec une boucle
for (int i = 0 ; i < 5; i++)
{
    //Pour lisser le mouvement, on utilise une clé d'animation
    //de type accélération
    EasingDoubleKeyFrame dkx = new EasingDoubleKeyFrame();

    //on définit donc un type de easing
    ExponentialEase Ee = new ExponentialEase();

    //afin de lisser l'animation de droite à gauche
    //l'accélération doit être la même au départ comme
    //à l'arrivée de l'interpolation
    Ee.EasingMode = EasingMode.EaseInOut;
```



```
//on affecte cette équation à la propriété EasingFunction
//de chaque clé
dkx.EasingFunction = Ee;

//Chaque clé doit posséder une valeur qui affectera
//la propriété Canvas.Left de la bulle
dkx.Value = (double)Canvas.GetLeft(maBulle)+(rnd.NextDouble()*120) - 60;

dkx.KeyTime = TimeSpan.FromMilliseconds((rnd.NextDouble()*1500) +
                                          ( (i+1) * 1500));
DaX.KeyFrames.Add(dkx);
}
```

Comme vous le constatez, les animations utilisant des clés d'animation possèdent la propriété `KeyFrames`, qui est en fait une collection de `DoubleKeyFrame`. Chaque clé d'animation fait donc partie d'un type précis d'accélération.

Vous aurez le choix entre `Easing`, représentant les équations d'accélération, `Spline` pour les courbes d'accélération (voir section 6.3.2.3), `Linear` permettant les interpolations linéaires et `Discrete` pour des clés d'animation sans interpolation. L'un des aspects importants consiste à éviter des mouvements trop saccadés sur l'axe horizontal. Pour cela, il faudra éviter de positionner trop de clés dans l'animation horizontale. Il nous reste maintenant à ajouter les deux animations à l'instance de `Storyboard` et à la lire :

```
//On ajoute la sous-séquence d'animation comme enfant du Storyboard
AnimBulle.Children.Add(DaY);
AnimBulle.Children.Add(DaX);

//Puis on demande à l'animation de se jouer
AnimBulle.Begin();
```

Finalement, on pourrait encore créer deux animations de fondu pour l'apparition progressive d'une bulle et pour sa disparition. Il suffit pour cela de stocker la durée aléatoire de l'animation verticale et de jouer une animation d'opacité 2 secondes avant sa fin. Vous trouverez ce projet finalisé dans l'archive *BubbleParticule_final.zip* du dossier *chap6* des exemples.

Au Chapitre 7, nous reviendrons sur des notions d'animation dans un contexte complètement différent puisqu'elles seront conçues *via* le gestionnaire d'états visuels. Nous aborderons la création de boutons personnalisés à travers la conception d'un lecteur vidéo simple.

Boutons personnalisés

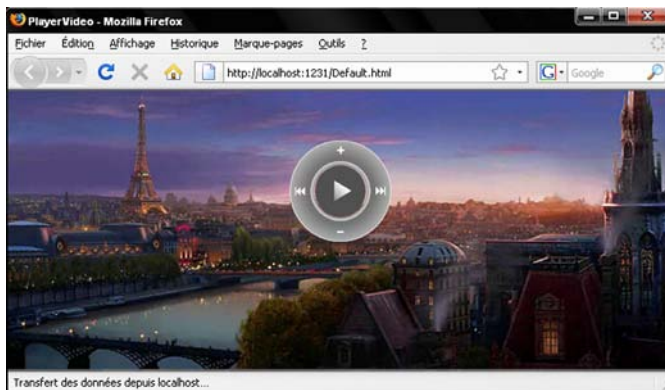
Tout au long de ce chapitre, vous concevrez le design d'un lecteur vidéo simple. À travers cet exemple, vous allez créer des composants de type `ButtonBase` entièrement personnalisés. C'est une première étape facilitant l'apprentissage des notions propres à la personnalisation de tous types de contrôles. Ainsi, vous connaîtrez la différence entre style et modèle de composant, au sein de la plateforme Silverlight. Vous étudierez la liaison de modèles qui assouplit et facilite la maintenance des styles et des modèles tout en évitant leur multiplication. Dans un second temps, vous découvrirez le gestionnaire d'états visuels qui permet aux designers de gérer les transitions aussi bien au sein des contrôles qu'au niveau de l'application elle-même. Pour finir, vous apprendrez à créer un bouton interrupteur à deux états et utiliserez le système d'agencement fluide pour faciliter les transitions entre états.

7.1 Créer un lecteur vidéo

Nous allons concevoir le lecteur vidéo correspondant à la Figure 7.1. Dans un premier temps l'idée est de mettre en forme ce projet d'un point de vue design.

Figure 7.1

Lecteur vidéo finalisé.

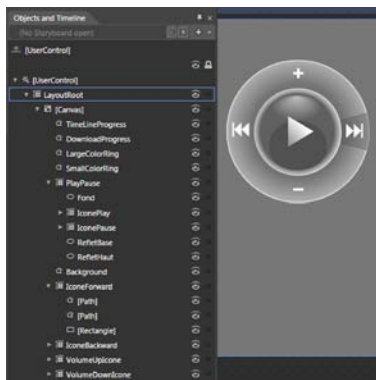


7.1.1 Mettre en place le projet

Afin de simplifier les opérations, vous pouvez télécharger le projet du dossier : *chap7/Player-Video.zip*. L'application se redimensionne automatiquement en fonction du navigateur. Vous trouverez plusieurs objets, dans des grilles nommées, répartis dans l'arbre visuel. Il s'agit en réalité d'une base visuelle pour les futurs contrôles du lecteur (voir Figure 7.2).

Figure 7.2

Arbre visuel du projet
PlayerVideo.



Comme vous le constatez, le lecteur vidéo est centralisé dans un Canvas. Le contrôle Media-Element contenant le fichier vidéo n'est pas encore présent (nous l'étudierons au Chapitre 13). Cela est assez logique car il ne sera pas redimensionné par l'utilisateur. De plus, l'agencement de ses composants est facilité par le placement libre que propose ce type de conteneur. Le premier de ses enfants, nommé *TimeLineProgress*, permettra à l'utilisateur de déplacer la tête de lecture à un instant précis. Le second, *DownloadProgress*, est un tracé qui affichera l'état de téléchargement des vidéos en cours de lecture. Les troisième et quatrième enfants, correspondants à *LargeColorRing* et *SmallColorRing*, sont des indicateurs de couleur recouvrant pour l'instant complètement les précédents. Lorsqu'un des boutons sera survolé, ces deux disques changeront de couleur. Au centre du lecteur, un interrupteur nommé *PlayPause* permettra de lire la vidéo ou de la mettre en pause et sera de type *ToggleButton*. Il affichera alternativement l'icône de lecture ou l'icône de pause. Nous aborderons son aspect visuel plus tard au sein de ce chapitre. Juste en dessous de l'interrupteur, un tracé nommé *Background* ainsi qu'une grille (*Grid*), *IconForward*, constitueront notre bouton d'avance rapide. Les trois derniers enfants du Canvas sont en fait les icônes que nous utiliserons pour les prochains boutons.

7.1.2 Insérer une image d'arrière-plan

Avant de créer les boutons, nous allons afficher une image à l'arrière-plan du contrôleur vidéo. Sélectionnez la grille principale *LayoutRoot* comme conteneur contexte. Au sein du panneau Project, ouvrez le répertoire *bitmaps* et double-cliquez sur l'image nommée *back.jpg*. Celle-ci prend place au sein d'un conteneur spécialisé de type *Image*.

INFO

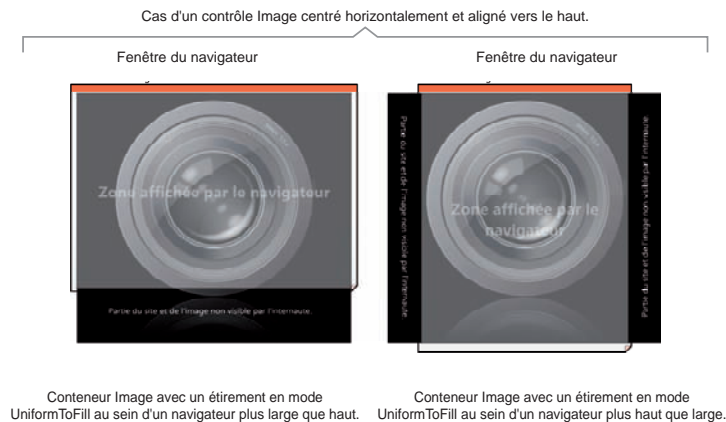
Vous pouvez cliquer droit sur l'image au sein du projet et choisir l'option *Insert* pour réaliser cette opération. Il est également possible de rechercher le composant *Image* dans la bibliothèque de composants *Silverlight* pour l'instancier directement sur la scène. Ensuite, vous devrez lui définir

un chemin d'accès pointant vers l'image que vous souhaitez afficher *via* sa propriété `Source`. Une image bitmap ne peut être affichée que de deux manières. La première consiste à utiliser un conteneur `Image`. La seconde est d'utiliser un pinceau de type `ImageBrush` (voir Chapitre 9).

Sélectionnez le composant `Image` nouvellement créé. Placez-le à l'index 0 de la liste d'affichage de `LayoutRoot`. Définissez une opacité de 20 % pour le composant `Image`. Centrez-le horizontalement et alignez-le en haut. L'idéal serait d'empêcher la déformation de l'image bitmap. Pour cela, dans les options propres au composant `Image`, choisissez un étirement de type `UniformToFill`. Ce mode de remplissage permet d'éviter la déformation de l'image lorsque le composant est étiré tout en évitant les zones de remplissage vides (voir Figure 7.3).

Figure 7.3

*Mode d'étirement
UniformToFill.*



Pour configurer ce type de remplissage en C#, il suffit de cibler la propriété `Stretch` du composant `Image` comme suit :

```
//Instanciation d'un composant Image
Image myImage = new Image();
//On définit son mode de redimensionnement
myImage.Stretch = Stretch.UniformToFill;

Uri adresseImage = new Uri("fond.png",UriKind.Relative);

BitmapImage bi = new BitmapImage( adresseImage );
myImage.Source = bi;
```

Nous ne nous attarderons pas sur ce code car nous aborderons le chargement dynamique de médias au Chapitre 10.

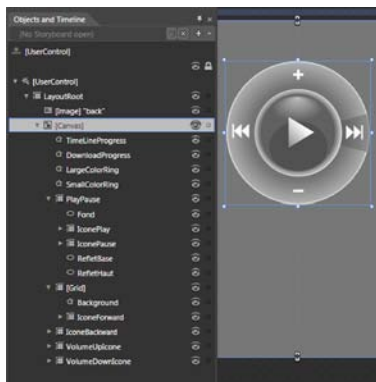
7.1.3 Le rôle du composant *Grid*

La grille est le conteneur à enfants multiples sur lequel repose l'architecture de la majorité des composants Silverlight. Lorsque vous souhaitez créer un composant personnalisé, vous devrez souvent centraliser les objets graphiques au sein d'une grille avant de créer le composant. Celle-ci est très performante et offre de nombreux avantages exploités par l'équipe de développeurs chargée de créer la bibliothèque de composants. C'est pour cette raison que le contrôle `Grid`, nommé `PlayPause` dans l'arbre visuel de notre application, contient tous les éléments qui composeront le

futur composant `ToggleButton`. Vous allez commencer par créer le bouton d'avance rapide. Pour cela, sélectionnez le tracé nommé `Background` ainsi que le conteneur `IconeForward`, puis utilisez le raccourci `Ctrl+G` pour les imbriquer au sein d'une nouvelle grille (voir Figure 7.4).

Figure 7.4

Arbre visuel de la maquette finalisée.



Vous trouverez la maquette finalisée dans l'archive *PlayerVideo_Maquette.zip* du dossier *chap7* des exemples du livre. Maintenant que vous avez centralisé tous les éléments graphiques, il est temps de créer un bouton personnalisé.

7.2 Style visuel

Personnaliser l'affichage d'un composant visuel consiste souvent à définir un nouveau style à ce type d'objet. On peut donc se demander ce que représente un style. C'est assez simple, un style est un ensemble de propriétés prédéfinies propres à un type de composant. Concrètement, vous pourriez vouloir que tous vos boutons contiennent un texte par défaut et qu'ils aient une largeur de 150 pixels. Ce paramétrage prédéfini de propriétés d'objet est appelé `Style`. Le principe des feuilles de style pour le contenu HTML repose sur cette définition.

La définition d'un style peut également être abordée d'un point de vue purement technique. Tous les composants héritant de `Control` possèdent la propriété `Style`. C'est le cas de nombreux composants tels que `Button`, `RadioButton` ou `Slider`. Ils ont la même architecture que notre application. Ils sont constitués d'un fichier XAML déclaratif ainsi que d'un fichier C# contenant le code logique du composant. Toutefois, ils ne sont ouverts au changement que du point de vue visuel. Autrement dit, leur code logique est fermé à la modification, à l'opposé de leur code déclaratif qui est accessible et facilement modifiable. La raison en est simple : la fonctionnalité (donc le code logique d'une barre de défilement ou d'un bouton) ne changera jamais alors que son apparence dépendra grandement de la charte graphique. Pour affecter ou atteindre le code XAML permettant de personnaliser un composant de type `Control`, on utilisera sa propriété `Style`.

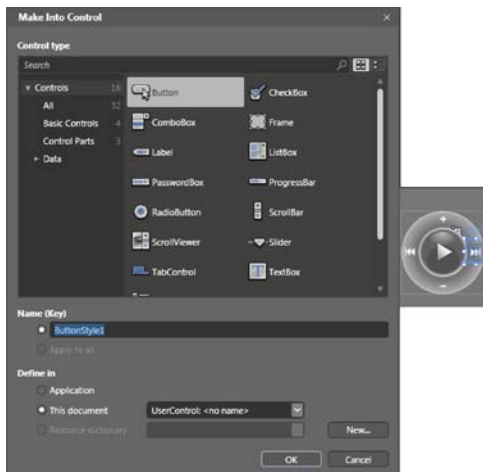
7.2.1 Créer un style personnalisé

Sélectionnez la grille contenant les objets graphiques d'avance rapide et dans le menu `Tools`, cliquez sur le menu `Make Into Control...` Une boîte de dialogue correspondant à la Figure 7.5 s'affiche. Elle vous permet de créer un nouveau composant personnalisé à partir du conteneur sélectionné et de ses enfants. Comme vous pouvez le constater, la liste de composants possibles est

longue. Dans certains cas, cette tâche demande un minimum d'expérience, nous verrons comment aborder cet apprentissage au Chapitre 9.

Figure 7.5

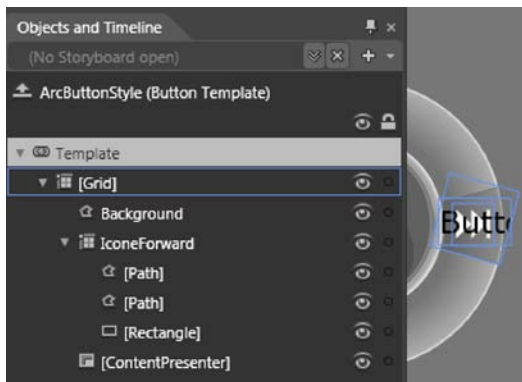
Création d'un style.




Nommez le style `ArcButtonStyle1`. En dessous de *Define in*, laissez l'option *This document* cochée. Cette option place le style au sein de la balise `UserControl1` correspondant à notre composant racine. Ainsi, tous les composants de type `Button` qui sont dans le composant `UserControl1` auront la possibilité de recevoir ce style. Ceux qui seront situés à l'extérieur du `UserControl1` n'auront pas accès au style que nous avons créé. Cette série d'options définit la portée d'accès au style généré (voir Chapitre 11). Cliquez sur *OK*. Blend vous place par défaut et immédiatement au sein du modèle `Button` généré, et non au niveau du style lui-même. Seul l'arbre visuel du composant `Button` apparaît (voir Figure 7.6).

Figure 7.6

Arbre visuel du bouton personnalisé.



Comme vous le constatez, la liste d'affichage se différencie de l'arbre visuel de `MainPage.xaml` par de nombreux points. Tout d'abord, l'élément le plus haut n'est plus `UserControl1`, mais l'élément `Template` qui signifie modèle. Le modèle d'un composant constitue son arbre visuel et logique. C'est l'une des propriétés prédéfinies qu'un style peut posséder. En second lieu, nous savons que nous sommes dans un modèle propre aux composants de type bouton car vous pouvez lire `Button Template` à droite du nom du style `ArcButtonStyle1`. Il peut arriver de se perdre un

peu dans l'interface lorsque l'on débute avec Blend car elle est entièrement contextuelle. Veillez bien à conserver des repères visuels tels que le nom de l'objet le plus haut dans la hiérarchie. Vous remarquerez, à gauche de `ArcButtonStyle`, l'icône  qui est devenue active. Elle ne l'était pas au sein de l'arbre visuel principal de l'application. Si vous la cliquez, vous pourrez revenir sur l'arbre visuel principal. C'est l'une des manières de naviguer entre différents niveaux de composants.

Pour finir, vous trouverez en bas de l'arbre visuel un composant de type `ContentPresenter` généré lors de la création du style. L'explication en est simple : la classe `Button` hérite elle-même de `ContentControl`. Cette dernière ajoute un comportement d'imbrication par défaut à toutes ses classes héritées. Grâce à la propriété `Content`, vous pouvez afficher n'importe quel type de contenu au sein du bouton en le glissant directement sur le bouton dans l'arbre visuel principal de l'application. Le composant `ContentPresenter` assure cette fonction. Sélectionnez-le et passez sa propriété `Visibility` à `Collapsed` pour désactiver momentanément cette fonctionnalité. La chaîne de caractères `Button` affichée par défaut disparaît.

7.2.2 Naviguer entre style, modèle et application principale

La navigation entre chaque niveau d'imbrication représente l'une des difficultés rencontrées lors de la prise en main de Blend. Vous aurez souvent besoin de corriger ou de mettre à jour un style personnalisé. Mais nous allons commencer par analyser le code XAML généré afin de mieux comprendre le fonctionnement de la navigation dans Blend. Ouvrez le mode d'édition XAML et placez-vous au niveau de la balise `<UserControl.Resources>`. Vous devriez visualiser l'équivalent non abrégé du code XAML montré ci-dessous :

```
<UserControl.Resources>
  <Style x:Key="ArcButtonStyle" TargetType="Button">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          <Grid>
            <VisualStateManager.VisualStateGroups>
              ...
            </VisualStateManager.VisualStateGroups>
            <Path x:Name="Background" ...
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </UserControl.Resources>
```

Lorsque le style est généré dans un `UserControl`, il fait partie de sa propriété `Resources`. Cette dernière représente un ensemble de ressources de n'importe quel type. L'attribut `x:Key` de la balise `Style` désigne le nom du style. Un style est une ressource en tant que telle, on utilise une clé de ressource plutôt qu'un nom de référence (voir Chapitre 9). La propriété `TargetType` spécifie le type d'objet ciblé par ce style. La balise `Setter` définit une propriété spécifique au sein du style. Dans notre cas, lorsque le `Style` sera appliqué à un bouton, la propriété `Template` de ce bouton sera affectée indirectement par notre `Style` nommé `ArcButtonStyle`. Le modèle n'est donc qu'une propriété parmi d'autres que l'on peut trouver au sein d'un style. Cette importante notion conditionne la navigation au sein de l'interface d'Expression Blend.

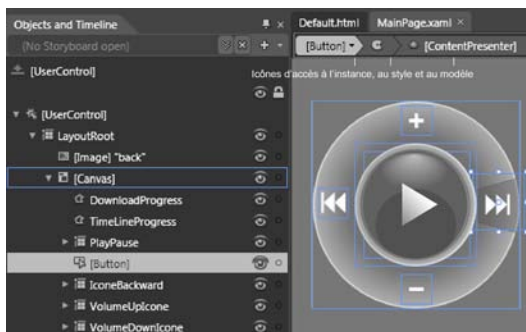
Passez en mode création, puis cliquez sur l'icône , située à gauche du nom `ArcButtonStyle`. Blend affiche à nouveau l'arbre visuel principal. La grille qui contenait les éléments visuels du


bouton a été remplacée par un composant `Button`. Nous avons, jusqu'à présent, accédé au modèle mais pas au style. Il existe trois méthodes différentes pour accéder à un style :

- La première méthode est la plus simple. En haut de la fenêtre de design, vous apercevez trois icônes vous permettant d'accéder, soit à l'instance du bouton, soit au style, soit directement au modèle du bouton sélectionné (voir Figure 7.7). Attention toutefois au fait que ces icônes sont visibles uniquement si le `Style` ou le `Template` ont déjà été modifiés ou du moins atteints dans Blend.

Figure 7.7

Icônes d'accès au style, au modèle et nouvel arbre visuel du lecteur vidéo.



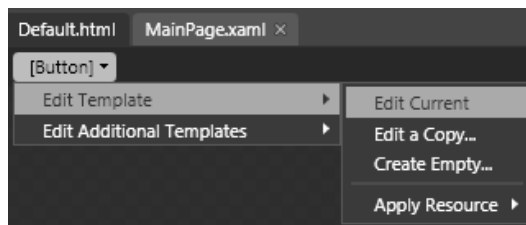
L'icône contenant `[Button]` représente l'instance du bouton actuellement sélectionné au sein du `Canvas`. Si vous nommez votre bouton, c'est le nom qui apparaîtra. Le pictogramme en forme de palette de peintre () représente, quant à lui, un raccourci vers le style. Il suffit donc de cliquer dessus pour y accéder. La dernière icône, `[ContentPresenter]`, correspond au dernier élément sélectionné dans le modèle et donne accès à celui-ci.

INFO

Lorsque vous venez d'ouvrir un projet, seule la première des trois icônes est présente en haut de la fenêtre de design. Ce n'est pas un problème. Il suffit de cliquer sur cette icône, puis de sélectionner le menu `Edit Template > Edit Current` (voir Figure 7.8). Toutefois, vous accédez alors directement au modèle et non au style. Vous devrez donc remonter d'un niveau pour afficher le style.

Figure 7.8

Accéder au modèle d'un composant.

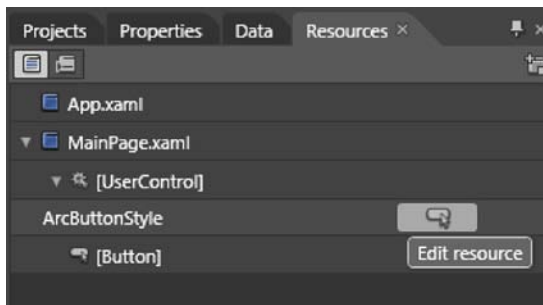


- La seconde possibilité est d'utiliser le menu `Object > Edit Style`, puis de cliquer sur `Edit Current` (voir Chapitre 11 pour les options de création de style).

- La dernière méthode consiste à utiliser le panneau Resources situé à droite, à déplier l'arborescence du UserControl et à cliquer sur l'icône du style auquel vous souhaitez accéder (voir Figure 7.9). Vous pouvez également utiliser le panneau des propriétés, cliquer sur le bouton des options avancées situé au niveau de la propriété Style et cliquer sur Edit Resource.

Figure 7.9

Icônes d'accès au style, au modèle et nouvel arbre visuel du lecteur vidéo.



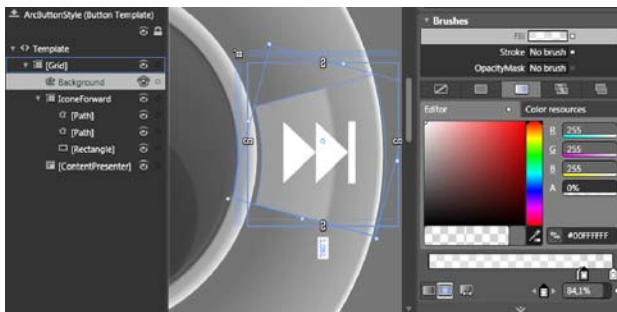
Une fois au sein du style, vous constatez qu'aucun arbre visuel n'est représenté. C'est tout à fait normal car nous ne sommes pas encore au sein du modèle de notre bouton, mais simplement au niveau des propriétés propres à la classe Button. Modifiez la propriété Cursor en sélectionnant la valeur Hand. De cette manière, lorsque le bouton sera survolé, le curseur de la souris sera remplacé par une main. Compilez le projet si nécessaire pour le vérifier. Étudiez maintenant le code XAML généré :

```
<UserControl.Resources>
  <Style x:Key="ArcButtonStyle" TargetType="Button">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="Button">
          <Grid>
            <VisualStateManager.VisualStateGroups>
              ...
            </VisualStateManager.VisualStateGroups>
            <Path x:Name="Background" ...
          </ControlTemplate>
        </Setter.Value>
      </Setter>
      <Setter Property="Cursor" Value="Hand"/>
    </Style>
  </UserControl.Resources>
```

Une nouvelle balise Setter est contenue dans la balise Style. Elle indique que la propriété Cursor doit prendre Hand pour valeur par défaut lorsque le style est appliqué. Cela montre à quel point le modèle, représenté par la propriété Template des objets de type Control, n'est qu'une propriété parmi d'autres. Vous allez maintenant modifier le modèle en créant un léger reflet. Définissez un dégradé transparent pour le remplissage du tracé nommé Background. Supprimez également le contour du tracé (voir Figure 7.10).

Figure 7.10

Bouton avec reflet.



Vous avez finalisé la première étape de création du bouton. Il serait possible de créer autant de styles que nécessaire pour chaque bouton présent dans le lecteur. Vous allez toutefois utiliser le style actuel pour créer d'autres boutons ce qui est bien plus optimisé en termes de production.

7.3 Bouton générique

Les quatre boutons principaux présents sur le disque du lecteur partagent de nombreuses similitudes. Par exemple, afin de respecter la charte graphique, ils doivent avoir le même comportement au survol de la souris. De plus, leur structure interne est identique et seul le pictogramme diffère. Créer autant de styles que de boutons serait inutile en plus d'être fastidieux. Il existe trois méthodes différentes pour résoudre ce type de problématique et minimiser le nombre de styles créés.

- La première est d'utiliser du code logique, c'est un peu ennuyeux dans notre situation. Bien que cette solution soit envisageable, cela serait un peu prématuré d'utiliser du code logique pour résoudre cette problématique. Lorsque le nombre d'opérations de ce type est limité, il est préférable que le graphiste garde cette tâche pour lui et ne dépende pas du planning d'un développeur. Toutefois, cela est salutaire et inévitable dans certains cas lorsque le nombre de boutons à gérer est trop important. De manière générale, plus une tâche est rébarbative, plus il est souhaitable qu'elle soit automatisée *via* le code logique.
- La seconde manière de procéder consiste à utiliser la propriété `Content` propre aux classes héritant de `ContentControl`.
- La troisième possibilité est d'utiliser la liaison de modèles que nous aborderons en détail au Chapitre 11. Nous choisirons donc la deuxième solution.

7.3.1 La propriété *Content*

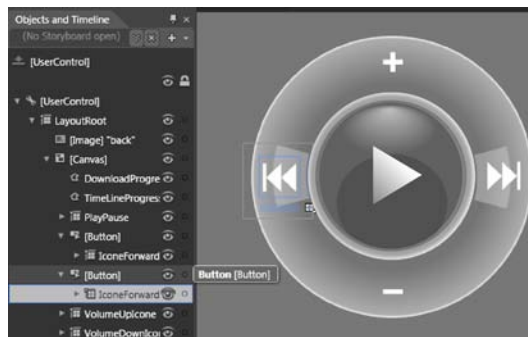
Finalement la seule différence entre chacun de nos boutons est l'icône elle-même. La propriété `Content`, héritée de la classe `ContentControl`, permet d'afficher n'importe quelle instance d'objet. Si nous externalisons l'icône présente dans le modèle de bouton, en l'affectant à la propriété `Content` des instances, chaque exemplaire de `Button` possèdera sa propre icône tout en ayant un visuel homogène car affecté du même style.

Une fois au sein du modèle, sélectionnez la grille nommée `IconForward`, puis faites un couper-coller *via* le raccourci `Ctrl+X`. Ensuite, sélectionnez le composant `ContentPresenter` et passez sa propriété `Visibility` à `Visible`. Revenez au niveau de l'application et sélectionnez le bouton comme contexte conteneur si ce n'est pas le cas. Ensuite, utilisez le raccourci `CTRL+V` pour im-

briquer l'icône à l'intérieur du bouton. Cela revient à affecter la grille à la propriété Content du bouton. Vous pouvez recentrer l'icône à l'intérieur du bouton facilement en utilisant la propriété Margin de la grille IconeForward. Créez les trois autres boutons à partir de celui existant. Après l'avoir dupliqué trois fois, il vous reste à positionner ses copies avec une simple rotation *via* l'onglet RenderTransform du panneau des propriétés. Vous pouvez vous faciliter la tâche en déplaçant le centre de transformation du bouton au centre du cercle. Ainsi, chaque copie du bouton sera placée correctement *via* une rotation adéquate (voir Figure 7.11).

Figure 7.11

*Copie du bouton
généré avec centre de
transformation déplacé.*



Vous pouvez supprimer la grille nommée IconeBackward dans ce cas précis. La rotation du bouton d'avance rapide correspond en effet au visuel du bouton de retour rapide. Ensuite, il reste à renommer la grille contenue dans le nouveau bouton par IconeBackward. Pour les deux autres copies, glissez les icônes correspondant à la gestion du volume en utilisant l'arbre visuel et logique.

INFO

Utiliser une rotation fonctionne car tous nos boutons sont symétriques sur leur axe horizontal. Cette manipulation n'est donc pas valable dans tous les cas de figure. La propriété Content, bien que très utile, ne répond pas à toutes les problématiques. Toutefois, son utilisation repose sur un concept plus large que nous allons maintenant étudier. Il est également possible d'opérer une symétrie horizontale.

7.3.2 Liaison de modèles

La liaison de modèles est un concept provenant de WPF et peut s'appliquer à diverses propriétés. L'instance ContentPresenter au sein de tout modèle d'objet de type ContentControl repose sur ce principe. Ainsi, sans nous en rendre compte, la liaison de modèles nous a permis de créer quatre boutons différents en apparence, mais ayant pourtant le même style.

7.3.2.1 Principes

Pour mieux le constater, il vous suffit d'aller dans le code XAML du modèle :

```
<ControlTemplate ... >
    ...
    <Grid>
        <ContentPresenter HorizontalAlignment="
            {TemplateBinding HorizontalContentAlignment}"
```

```

        VerticalAlignment="{TemplateBinding VerticalContentAlignment}" />
    </Grid>
</ControlTemplate>

```

Comme vous le remarquez, les deux propriétés d'alignement sont affectées d'une valeur entre accolades. Cela signifie que ces propriétés ne sont pas affectées d'une valeur définie en dur, mais d'une expression dont le résultat peut évoluer. Dans notre cas, c'est même un peu plus compliqué car l'expression entre accolades commence par le mot-clé `TemplateBinding`. Il signifie que les valeurs d'alignement de l'instance `ContentPresenter` ne sont pas définies directement au sein du modèle, mais au niveau de l'exemplaire du bouton. C'est exactement ce qui permet d'utiliser la propriété `Content` de chaque exemplaire afin de spécifier une icône différente à afficher.

Par défaut, une instance `ContentPresenter` possède une propriété `Content` liée à la valeur affectée sur la propriété `Content` de l'exemplaire. Celle-ci n'est donc pas précisée dans le code XAML du `ContentPresenter`. Vous pourriez donc écrire le code XAML suivant, le résultat serait exactement le même :

```

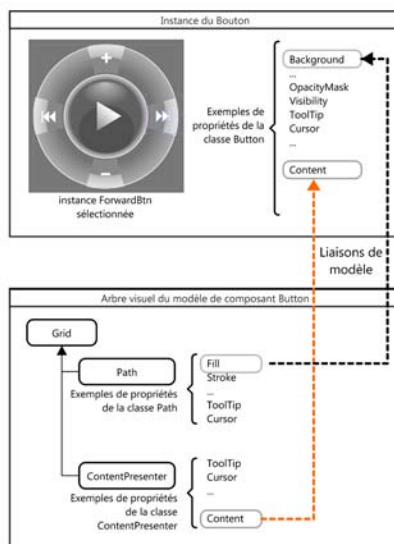
<ControlTemplate ... >
    ...
    <Grid>
        ...
        <ContentPresenter HorizontalAlignment=
            "{TemplateBinding HorizontalContentAlignment}"
            VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
            Content="{TemplateBinding Content}" />
    </Grid>
</ControlTemplate>

```

Vous pouvez consulter la Figure 7.12 qui décrit le schéma de deux liaisons de modèles. La première correspond à la liaison de modèles propre à la propriété `Content`. Elle est créée par défaut. La seconde expose une liaison de modèles entre la propriété `Fill`, du tracé présent à l'intérieur du modèle, et la propriété `Background` de chaque exemplaire de `Button`.

Figure 7.12

Le principe de la liaison de modèles.



En tant que designer interactif, vous pouvez définir une liaison de modèles directement *via* l'interface d'Expression Blend. Nous allons permettre à chaque exemplaire de bouton, au sein de notre lecteur vidéo, d'afficher une couleur de remplissage différente.

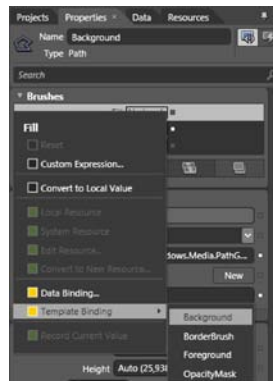
7.3.2.2 Créer une liaison de modèles

Afin de mieux comprendre l'intérêt de ce type de liaison, sélectionnez n'importe lequel des boutons créés. Modifiez sa propriété Background en choisissant n'importe quelle couleur *via* le nuancier. Comme vous le constatez, modifier la couleur d'arrière-plan des boutons ne change en rien leur affichage. C'est tout à fait normal car les propriétés de couleurs ne sont reliées à aucun objet présent dans le modèle de nos boutons. C'est cette problématique que nous allons maintenant traiter.

Sélectionnez le bouton d'avance rapide, puis affichez l'arbre visuel de son modèle. Sélectionnez le tracé nommé Background. Il possède un remplissage que nous avons défini en dur. Cela signifie que si vous modifiez ce tracé directement à l'intérieur du modèle, tous les exemplaires du bouton seront affectés de ces modifications. Cliquez maintenant sur l'icône carrée, située à droite de la propriété Fill du tracé. Ensuite, sélectionnez le menu Template Binding, puis la propriété Background (voir Figure 7.13).

Figure 7.13

Définir une liaison de modèles au sein d'Expression Blend.



Tous les boutons du lecteur possédant ce modèle affichent désormais la couleur définie sur leur propriété Background. Revenez au niveau de l'application principale, puis définissez une couleur d'arrière-plan différente pour chaque exemplaire. Cela peut-être un dégradé, une couleur pleine ou n'importe quelle instance de type Brush. Choisissez des couleurs vives afin de donner un peu plus de vie à notre lecteur. Des couleurs bonbon fluo correctement dosées seraient les bienvenues (voir Figure 7.14).

Voici le code XAML généré dans Blend :

```
<ControlTemplate x:Key="ArcButtonStyle" TargetType="Button">
    ...
    <Grid>
        <Path Fill="{TemplateBinding Background}" Data="..." .../>
        ...
    </Grid>
</ControlTemplate>
```

En quelques opérations assez simples, nous avons créé une impression de richesse visuelle alors qu'un seul style est utilisé.

Figure 7.14

Chaque exemplaire de bouton possède désormais sa propre couleur.



Décliner plusieurs visuels à partir d'un unique modèle est une technique de marketing utilisée depuis bien longtemps et toujours d'actualité (voir Figure 7.15). Il est donc normal que le langage XAML fournisse les moyens aux designers de réaliser simplement cette tâche.

Figure 7.15

Déclinaisons à partir d'un unique objet.



Vous pouvez télécharger le projet finalisé, *PlayerVideo_ButtonsGenerique.zip*, dans le dossier *chap7* des exemples.

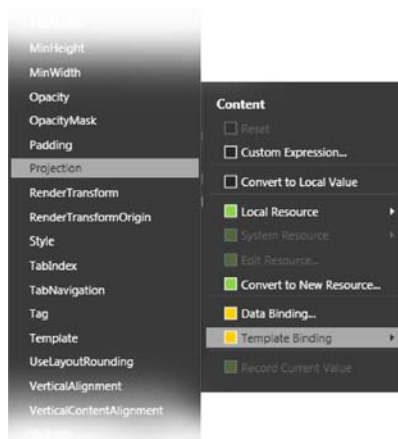
7.3.2.3 Contraintes

La liaison de modèles est une fonctionnalité accessible pour la grande majorité des propriétés d'objets présents au sein des modèles. Comme nous l'étudierons aux Chapitres 11 et 12, toute propriété de type `DependencyProperty` peut être la cible d'une liaison de modèles. La seule condition de fonctionnement est que la propriété de l'objet présent dans le modèle ne peut définir de liaison que vers une propriété de même type. Ainsi, lorsque vous avez affiché la liste des propriétés acceptant une liaison pour `Fill`, seules quatre propriétés de `Button` ont été disponibles : `Background`, `BorderBrush`, `Foreground` et `OpacityMask`. Elles sont disponibles car elles acceptent des valeurs de type `Brush` au même titre que `Fill` ou `Stroke` (qui sont propres aux objets héritant de `Shape`). Lorsqu'un transtypage implicite est possible, la liaison de modèles est également accessible. Par exemple, si vous essayez de définir une liaison de modèles sur la propriété `Content` d'un `ContentPresenter`, toutes les propriétés de l'instance seront accessibles. Ceci est normal

puisque par définition la propriété `Content` est typée `Object` et que toutes les classes héritent de `Object`. `Content` peut donc être liée à n'importe quelle propriété (voir Figure 7.16).

Figure 7.16

Liste des propriétés disponibles pour une liaison de modèles à la propriété `Content`.



D'un point de vue logique, on pourrait considérer cette fonctionnalité comme une béquille permettant de résoudre certaines problématiques de conception. En effet, nous avons lié la propriété `Fill` à `Background` car elles ont des rôles très proches. Toutefois, rien ne nous empêcherait de lier `Fill` à `OpacityMask`. Cela n'est pas très logique, mais la liaison de modèles nous l'autorise. Il faudra parfois passer par des contorsions de ce type pour concevoir un composant générique et facile à maintenir. Le désavantage de ces contorsions est qu'il faut se souvenir d'une liaison de modèles qui n'est pas forcément naturelle lorsque vous ouvrez à nouveau le projet après un long moment.

INFO

Dans certains cas, vous ne souhaitez pas récupérer directement la valeur en provenance de la propriété d'instance mais la convertir à la volée pour en avoir une représentation légèrement différente au sein du modèle. Dans ce cas, vous utiliserez une classe implémentant l'interface `IValueConverter`. Cela peut-être très pratique et se révèle simple à coder. Nous en étudierons un exemple concret lorsque nous aborderons la création de modèles personnalisés plus complexes (`ListBox`, `Slider`, `ProgressBar`).

La dernière contrainte concerne en particulier les designers. Les propriétés liées ne sont plus modifiables au sein du modèle car elles récupèrent leur valeur dynamiquement. Il est donc tout à fait logique de ne plus accéder à la modification de ces propriétés au sein du modèle. Lorsqu'une propriété est liée dans un modèle d'objet, elle est entourée d'un liseré interdisant toute manipulation (voir Figure 7.17).

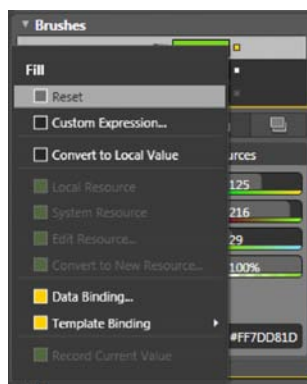
Dans la grande majorité des situations, vous pouvez toutefois réinitialiser la valeur de la propriété en cliquant sur l'icône carrée située à sa droite et en sélectionnant l'option `Reset` (voir Figure 7.18).

Figure 7.17

Affichage d'une propriété liée au sein d'Expression Blend.

**Figure 7.18**

Supprimer une liaison de modèles.



Cette dernière contrainte paraît évidente, mais il faut compter avec, surtout lorsque vous souhaitez animer la propriété en question. Pour l'instant notre bouton est fonctionnel, mais il ne réagit pas aux interactions utilisateur les plus simples. Vous allez y remédier en utilisant le gestionnaire d'états visuels abordé dans la prochaine section.

7.4 Le gestionnaire d'états visuels

Le gestionnaire d'états visuels est un module introduit depuis Silverlight 2. Il est représenté par une API dont la classe principale est `VisualStateManager`. Celle-ci a pour objectif de gérer les transitions entre chaque interface utilisateur au sein d'une application. Cette démarche et le concept de gestion d'états visuels représentent l'un des axes majeurs de réflexion pour les ergonomes et les designers interactifs. Si l'esthétique et l'ergonomie des interfaces visuelles composant une application furent considérées comme importantes au milieu des années 90, les transitions qui les mettent en scène ne furent quant à elles prises en compte que bien plus tard. Aujourd'hui encore, les transitions restent négligées au sein de nombreux développements applicatifs. Pour les développements de type Windows, celles-ci ne furent considérées d'un point de vue technique qu'à partir de .Net 3, c'est-à-dire lors de l'apparition de Windows Presentation Foundation début 2007.

7.4.1 Évolution de l'expérience utilisateur

Pour mieux comprendre l'importance des transitions, nous allons faire un rapide retour sur l'évolution des interfaces utilisateurs. Ainsi, nous comprendrons mieux le positionnement du lecteur Silverlight dans cette évolution.

7.4.1.1 Les différentes générations d'interfaces

Trois grandes familles d'interfaces cohabitent aujourd'hui. Elles furent successivement découvertes depuis la première interface informatique homme-machine (voir Figure 7.19).

Figure 7.19

Les trois grands types d'interfaces utilisateur.



Le type CLI (*Command Line Interface*) fut la première famille d'interface réellement disponible. Encore aujourd'hui, Unix, Linux, FreeBSD, MS-DOS, le terminal sous Mac OS X et de nombreux autres environnements reposent sur cette conception de l'accès aux fonctionnalités. Concrètement, l'utilisateur doit connaître le langage propre au système d'exploitation pour l'utiliser. Ainsi, lister un répertoire sous MS-DOS ou Linux Debian est réalisé en exécutant une commande prédéfinie, immuable et propre au système d'exploitation. La grammaire et le vocabulaire de ce langage étant tous deux propres à chaque système, cela force l'utilisateur à se spécialiser ou à ingurgiter un nombre de connaissances important et anecdotique tant leur utilisation peut s'avérer ponctuelle. L'utilisation du clavier comme médium permettant la communication homme-machine est obligatoire. La souris n'a simplement aucun intérêt car naviguer dans l'interface est impossible.

GUI est l'acronyme de *Graphic User Interface*. Une équipe de Xerox PARC, composée entre autres de Alan Kay et Douglas Engelbar, a créé, à la toute fin des années 70, la première interface utilisateur graphique. Par la suite, Steve Jobs a poursuivi ce travail à Apple, aidé de nombreux anciens de Xerox. Parallèlement à Apple et PARC, X Window est développé au MIT sous le nom de Projet Athena et devient accessible dès 1983. Windows 1 représentant l'interface graphique de MS-DOS devient disponible à partir de 1985. Au sein d'une interface GUI, on privilégie l'exploration de l'utilisateur ainsi qu'un apprentissage empirique (la pédagogie par l'erreur ou l'expérience, au choix). Cela est réalisé à travers un mode fenêtré grâce auquel l'utilisateur peut exécuter des programmes, trouver ses fichiers et cliquer sur des icônes. Windows, Mac OS X, X-Window sont tous basés sur ce principe. Celui-ci n'a pas réellement évolué depuis 1983 et Vista, Windows 7, Snow Leopard ainsi que la grande majorité des systèmes d'exploitations sont basés sur ce principe. La souris devient dès ce moment un instrument incontournable pour utiliser ce type d'interfaces. Comme pour les interfaces de type CLI de nombreuses différences existent encore

entre chaque système d'exploitation. Cela repose constamment la problématique de formation pour la prise en main et l'administration du système.

Le type NUI (*Natural User Interface*) est aujourd'hui visible sur de très rares périphériques tels que les tables de type Surface, l'iPhone – plus proche du grand public – ou encore le projet NATAL de Microsoft. L'objectif est simple : éviter à l'utilisateur l'apprentissage d'une interface en se reposant sur l'expérience déjà acquise dans son environnement naturel. Ainsi, pour un jeu de voiture sur l'iPhone, tourner le volant sera accompli en inclinant l'iPhone. L'utilisateur par sa propre nature possède tous les outils et instincts nécessaires pour interagir avec l'interface. La voix, le toucher, la reconnaissance visuelle des objets, la géo-localisation, la détection du mouvement ainsi que sa direction, tous ces éléments sont directement considérés comme des stimuli au même titre que le clic de la souris dans un environnement de type GUI.

L'interface NUI y réagit directement. L'utilisateur n'est donc que très peu limité par son environnement social, son âge ou son investissement dans l'apprentissage de telle ou telle technologie. La table Surface est à ce jour le périphérique supportant le mieux ce type d'interfaces. De plus, développer pour Surface est assez simple car son framework repose sur la technologie WPF. Tous les périphériques supportant les interfaces NUI possèdent un point commun : ils s'accordent et se marie à l'environnement physique. Poser un objet physique sur une table Surface, se retrouver entre amis autour, collaborer, jouer à plusieurs est non seulement possible mais souhaitable. Surface est un environnement massivement multi-touch à 360 degrés. De nombreuses autres technologies de ce type existent dont certaines ont été développées par des universitaires et sont complètement libre d'accès moyennant un peu de bricolage fait maison. Dans ce type d'interfaces, les transitions sont extrêmement importantes car aucune fenêtre d'exploration n'est présente. En effet, les interfaces utilisateur NUI étant entièrement contextuelles, c'est-à-dire liées au contexte d'utilisation, il devient crucial que l'utilisateur connaisse sa position au sein du processus d'utilisation. Il doit également savoir son point de départ et ses destinations possibles.

Si vous souhaitez plus d'informations sur les interfaces de type NUI, vous pouvez consulter le blog de Dr Neil, très investi dans la technologie Surface de Microsoft, à cette adresse : <http://blogs.msdn.com/surface/archive/2009/03/13/friday-afternoon-chat-with-dr-neil-on-education.aspx>, ou encore le site de Douglas Edric Stanley concernant l'hypertable et ses expérimentations : <http://www.abstractmachine.net/blog/>.

Nous allons maintenant essayer de comprendre comment se positionne Silverlight dans cette évolution.

7.4.1.2 L'apport des moteurs vectoriels

Bien que les moteurs vectoriels, comme Flash ou Silverlight, soient contenus par défaut au sein d'un navigateur web (donc d'une interface GUI), ils proposent des interfaces utilisateur novatrices échappant complètement aux contraintes GUI. Le fichier compilé peut en effet proposer n'importe quel type d'interfaces utilisateur puisqu'il est complètement indépendant du système d'exploitation. Cela est possible car les designers d'interfaces sont partie prenante dans la conception de l'application, mais également parce que chaque développement est unique et industrialisable au prix de nombreux efforts. Contrairement à une application Windows Forms, limitant grandement la manière de présenter les informations mais facile à développer, une application WPF est entièrement personnalisable mais plus difficile à concevoir (car intégrant de nouveaux profils métier). Flash est sans doute, jusqu'à présent, le moteur vectoriel qui a le plus bénéficié de la communauté

des designers. Cela a permis de s'affranchir des modes de réflexion et des standards d'ergonomie liés au système d'exploitation. L'exemple de l'application de réservation en ligne du célèbre hôtel Broadmoor au Colorado est la preuve de l'efficacité d'une bonne ergonomie (voir Figure 7.20). Bien que l'interface (voir Figure 7.20) fût développée il y a maintenant sept ans, elle n'a été modifiée que très récemment. Nous pourrions procéder à un lifting complet d'un point de vue esthétique, mais la manière dont elle est envisagée répond complètement aux attentes du client. Lors de sa mise en ligne, les réservations de l'hôtel ont simplement triplé.

Figure 7.20

L'interface de réservation du Broadmoor avec ses 3 étapes de réservation.

The screenshot displays the Broadmoor Colorado Springs reservation system. It features a calendar for selecting check-in and check-out dates, a list of available room types with their prices, and a form for providing guest details and payment information. The interface is clean and functional, with clear instructions and a logical flow for the reservation process.

Ainsi les applications vectorielles *Cross Platform* bénéficient d'un statut intermédiaire puisqu'elles sont conçues indépendamment du système ou du navigateur qui les affiche. L'application de réservation en est un exemple criant et les transitions qu'elle propose à l'utilisateur entre chaque étape du processus sont pertinentes encore aujourd'hui. Celles-ci bénéficient, en outre, de nombreux comportements asynchrones tels que la connexion socket XML ou binaire, le peer to peer, l'appel de services distants, le streaming vidéo live ou enregistré, l'affichage dynamique de médias. Ces fonctionnalités favorisent grandement l'essor d'applications à caractère social et la diffusion de contenus riches, deux thèmes majeurs du NUI. La grande limite reste donc la souris et le clavier comme medium incontournable de l'expérience utilisateur pour ce type d'environnement.

7.4.1.3 L'importance des transitions

Les transitions sont considérées comme aussi importantes que les interfaces applicatives. Chaque interface visuelle au sein d'une application peut être considérée comme une fonctionnalité de cette dernière, une brique applicative. Les transitions jouent le rôle de mortier entre chaque brique fonctionnelle. Autrement dit, elles font le lien entre deux interfaces visuelles. Elles possèdent de ce fait trois rôles importants :

- **Esthétisme et ludique.** Elles doivent être esthétiques et amusantes, car ce qui est ludique est directement connecté à l'affect de l'utilisateur. Le plaisir ressenti à l'utilisation est aujourd'hui un domaine qui n'est plus exclusivement relié aux jeux vidéo. L'expérience utilisateur lorsqu'elle est agréable peut faire la différence commercialement.

- **Sens.** Bien que ludiques, les transitions ne sont pas pour autant gratuites. Elles ont donc pour mission de renforcer le sens donné à chaque interface visuelle. Nous pourrions par exemple utiliser des transitions 3D sur la profondeur des objets pour permettre à l'utilisateur de ressentir leur importance à un instant donné du processus tout en évitant une navigation trop linéaire.
- **Localisation.** Elles ont également pour objectif de faciliter l'immersion de l'utilisateur en améliorant l'ergonomie. Afin d'atteindre ces objectifs, elles doivent fournir une indication supplémentaire sur l'emplacement, la position ou l'étape du processus dans lequel est situé l'utilisateur. Il ne suffit pas de savoir ce que l'on fait, il faut également savoir d'où l'on vient et où l'on va. Cela permet à l'utilisateur de se repérer et d'avoir une idée claire des tâches restant à accomplir ou les fonctionnalités accessibles. Cette notion est très importante pour les interfaces de type naturelle non fenêtrée pour des périphériques comme Surface, l'iPhone, etc. Pour ces interfaces de nouvelle génération, il faut éviter autant que possible à l'utilisateur d'avoir à découvrir l'application par une exploration systématique ou par l'expérimentation empirique. L'expérience des interactions dans la vie réelle est déjà accomplie, le designer d'expérience utilisateur doit récupérer ce savoir acquis pour accompagner la démarche d'utilisation.

Pour finir sur ce sujet, il est important de concevoir les transitions le plus tôt possible dans le processus de conception, mais également d'équilibrer les trois composantes décrites plus haut car privilégier trop l'une au dépend des autres engendre des comportements non souhaités. Une application trop agréable, mais dans laquelle l'utilisateur ne sait pas se situer, est intéressante dans une démarche artistique, mais n'est pas satisfaisante d'un point de vue commercial. À l'inverse, savoir se situer, mais n'avoir aucune surprise ou plaisir dans la navigation, peut provoquer une désaffection et de l'ennui. Sur Internet, l'ennui est le grand ennemi des Webmasters. Nous allons maintenant aborder la conception des transitions.

7.4.2 Au sein d'un *control*

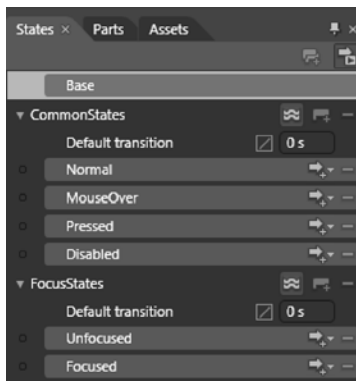
Gérer les transitions est réalisable à deux niveaux différents. Vous pouvez gérer celles appartenant à l'application elle-même, ou celles prédéfinies dans les composants personnalisables tels que `Button`. D'un point de vue développeur, il n'y a pas réellement de différences entre ces deux niveaux. L'application peut, en effet, être considérée comme un composant géant très spécifique car son architecture est similaire aux composants standard déjà fournis. Toutefois d'un point de vue organisationnel et utilisateur final, la différence est flagrante. Nous allons commencer par gérer les états d'un des boutons personnalisés pour mieux comprendre l'utilisation des transitions au sein de Silverlight.

7.4.2.1 Principes

Sélectionnez le bouton d'avance rapide, cliquez-le droit et sélectionnez successivement les menus `Edit > Template > Edit Current`. Vous atteignez une nouvelle fois le modèle du bouton. C'est à ce niveau que les transitions sont gérées pour tous les composants et non au niveau du style. Ouvrez le panneau contenant les états visuels `States`. Les objets de type `Button` possèdent tous les mêmes états (voir Figure 7.21).

Figure 7.21

Les états visuels au sein d'un composant Button.



Tout d'abord, les états visuels sont contenus au sein de groupes d'états visuels représentés par la classe `VisualStateManager`. Le code XAML, formalisant nos états visuels, est situé dans la balise `VisualStateManager.VisualStateGroups` :

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="FocusStates">
    <VisualState x:Name="Focused"/>
    <VisualState x:Name="Unfocused"/>
  </VisualStateGroup>
  <VisualStateGroup x:Name="CommonStates">
    <VisualState x:Name="Normal"/>
    <VisualState x:Name="MouseOver"/>
    <VisualState x:Name="Pressed"/>
    <VisualState x:Name="Disabled"/>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Les groupes d'états visuels sont répartis par fonctionnalités. Dans le code XAML ci-dessus, le premier groupe gère le visuel selon le focus utilisateur, le deuxième en fonction de ses états d'interactions utilisateur. Ainsi, un bouton peut ne pas posséder l'intérêt utilisateur tout en étant survolé. Il peut donc afficher deux états visuels à la fois, mais un seul par groupe d'états. Il peut avoir une opacité différente pour le survol de la souris et posséder une petite ombre pour indiquer le fait qu'il possède le focus utilisateur. Deux états centralisés dans des groupes différents seront indépendants.

INFO

Un bouton possédant le focus utilisateur est cliquable par défaut *via* la barre d'espace ou la touche Entrée. C'est assez pratique dans certains cas d'utilisation, notamment pour les formulaires d'inscription ou pour les fenêtres de connexion. Vous devrez toutefois gérer l'ordre de tabulation pour éviter au focus utilisateur de se placer sur n'importe quel composant interactif.

Avant d'aller plus loin, sachez que `Base` fait référence au visuel du bouton lorsque celui-ci ne possède aucun état visuel actif. Lorsque vous sélectionnez `Base`, puis que vous modifiez une propriété d'objet, tous les états visuels seront, par défaut, affectés de cette modification. Voici la description des états interactifs du groupe `CommonStates` :

- **Normal**. L'état normal est toujours actif par défaut, il représente l'état d'interaction normal du bouton. C'est lui qui est, en réalité, affiché lorsque vous êtes à l'extérieur du modèle.
- **MouseOver**. Représente le visuel du bouton lorsque celui-ci est survolé par la souris.
- **Pressed**. Affiche l'état du bouton lorsque l'utilisateur clique dessus en appuyant sur le bouton gauche de la souris.
- **Disabled**. Définit le visuel affiché lorsque le bouton n'est pas actif. Vous pourriez désactiver un bouton de souscription tant qu'un formulaire n'est pas rempli de manière conforme par exemple. Vous pouvez directement l'activer en modifiant la propriété `Enabled` d'un exemple de composant.

Considérez ces états comme une liste de choix. Vous ne pourrez en afficher qu'un seul à la fois par groupe d'états. Nous abordons cette contrainte à la section 7.4.3.2. Chaque type de composant possède sa propre liste d'états visuels. Vous trouverez pourtant certains points communs entre ceux-ci. Les principes que nous exposons ici sont, pour leur part, communs à tous composants

7.4.2.2 Modifier les états visuels

Modifier un état est assez simple. Cliquez sur l'état `MouseOver`, vous passez en mode enregistrement d'état et un contour rouge vous le signale. Vous pourriez être tenté de modifier la couleur à ce stade. Ainsi, le bouton changerait de couleur lors du survol. Toutefois cela est impossible car la propriété `Fill` est liée à la propriété `Background` de l'instance. Celle-ci est donc fermée à la modification au sein du modèle. Ce n'est pas grave, nous allons passer outre cette contrainte en utilisant l'état `Normal`. Sélectionnez-le, puis cliquez sur l'objet représentant le fond de couleur. Affectez la propriété `Opacity` du tracé de la valeur 0. Comme vous le constatez, tous les boutons perdent leur couleur d'arrière-plan à ce stade. C'est tout à fait logique puisque l'état `Normal` est affiché par défaut. Les autres états visuels utilisent les propriétés définies par `Base`, la couleur est donc visible au survol (`MouseOver`), sur le clic de l'utilisateur (`Pressed`) et lorsque le bouton est inactif (`Disabled`).

Vous allez maintenant agrandir de 20 % les dimensions de l'icône lors du survol de la souris. Comme l'icône n'est pas contenue au sein du modèle, mais affectée à la propriété `Content` des boutons, il suffit d'augmenter pour cela les dimensions du composant `ContentPresenter`. C'est ce composant qui affiche la valeur de la propriété `Content` de chaque exemplaire. Il est nécessaire d'utiliser les transformations relatives afin de s'affranchir des contraintes de positionnement liées au conteneur `Grid`. Sélectionnez l'état `MouseOver`, puis `ContentPresenter`. Ouvrez l'onglet des transformations relatives et modifiez l'échelle en x (`ScaleX`) et en y (`ScaleY`) en affectant à chacune d'elles la valeur 1.2. L'icône est dorénavant agrandie lorsque vous passez de l'état `Normal` à l'état `MouseOver` (voir Figure 7.22).

Figure 7.22

Modification de l'état `MouseOver`.



Lorsque vous testez votre application après l'avoir compilée, vous remarquez que tout correspond à nos actions. Toutefois lorsque vous cliquez sur les boutons, l'icône retrouve une taille normale car vous passez implicitement à l'état Pressed. Cela serait plus élégant si nous n'avions pas de différences marquées entre Pressed et MouseOver. Revenez dans Blend et cliquez-droit sur l'état MouseOver, puis cliquez sur l'option Copy to Pressed State. Toutes les modifications réalisées sur l'état MouseOver viennent d'être directement affectées aux objets sur Pressed. Ce menu caché vous évitera des manipulations rébarbatives dans de nombreux cas.

Tous les états sont maintenant configurés, il ne nous reste plus qu'à les animer.

INFO

Vous pouvez visualiser un état visuel sans pour autant le sélectionner. Pour cela, cliquez sur Base, puis sur le cercle noir présent à gauche de l'état dont vous souhaitez avoir un aperçu (voir Figure 7.23). Une fois activé, l'icône d'un œil apparaît (👁). Cela ne fonctionnera toutefois pas forcément lorsque vous utiliserez des animations personnalisées.

Figure 7.23

Affichage d'états visuels sans sélection.

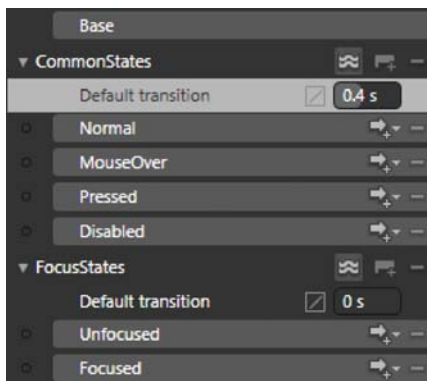


7.4.2.3 Animer les états visuels

Pour l'instant, chaque transition est brutale donc peu élégante. Pour y remédier, il faut définir des interpolations pour chacune. Il existe trois possibilités *via* l'interface de Blend pour créer des transitions animées. Le code XAML est assez proche pour chacune d'elles, mais elles apportent une finesse de personnalisation différente. La première méthodologie consiste à définir une durée d'animation générique qui sera propre à l'ensemble des états visuels contenus dans un groupe. C'est très pratique en terme de productivité. Toujours au sein du modèle dans le panneau States, dans le groupe d'états visuels CommonStates, vous trouverez l'option Default transition. Dans le champ de saisie situé à droite et indiquant 0s, définissez une durée de 0.4s (voir Figure 7.24).

Figure 7.24

Création d'une transition générique de groupe.



Vous venez en une seule manipulation de définir trois transitions différentes. Quel que soit l'état interactif actuel du bouton, une transition de 4/10 de seconde sera par défaut jouée afin d'atteindre n'importe quel autre état du groupe CommonStates. Voici le code XAML généré :

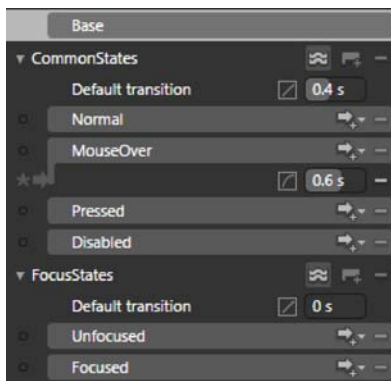
```
<VisualStateManager x:Name="CommonStates">
  <VisualStateManager.Transitions>
    <VisualTransition GeneratedDuration="00:00:00.4"/>
  </VisualStateManager.Transitions>
  <VisualState x:Name="Normal">
    ...
  </VisualState>
</VisualStateManager>
```

Nous allons maintenant outrepasser l'animation de base. Utiliser une seule interpolation générique par groupe d'états est très bien pour un prototype rapide, mais cela est parfois un peu trop sobre ou standard pour une version finale de l'application. Vous aurez besoin de marquer la différence d'importance ou de définir une accélération propre à chaque transition. L'esthétique entre également en ligne de compte. Normaliser ou industrialiser trop les transitions, bien qu'efficace en terme de temps, n'est pas une bonne pratique. Il faut travailler chaque transition indépendamment les unes des autres. Cela ne signifie pas pour autant que vous devez trop en faire, bien au contraire. Les transitions les moins importantes prendront moins de temps. Les plus importantes sont situées au niveau de l'application et permettent de valider les différentes étapes d'utilisation. Celles-ci doivent être plus travaillées.

La deuxième méthodologie permet de définir une durée et une équation d'accélération propre à chaque état. Cliquez sur l'icône représentant une flèche et un signe plus (➡+) située à droite de l'état MouseOver. Une autre série d'icônes vous permet de définir une transition spécifique. Choisissez celle définie par une étoile suivie de MouseOver. Celle-ci définit une transition unique depuis n'importe quel état d'origine à destination de MouseOver. Comme vous le constatez, la valeur par défaut correspond à celle que vous avez définie pour la transition par défaut, soit 0.4. Spécifiez un délai de 0.6s, ainsi qu'une équation d'accélération de type CubicOut (voir Figure 7.25).

Figure 7.25

Création d'une transition d'état.



Les spécificités de chaque transition sont contenues dans la balise `VisualStateGroup.Transitions`. Vous le constatez dans le code XAML ci-dessous :

```
<VisualStateGroup.Transitions>
  <VisualTransition GeneratedDuration="00:00:00.4" />
  <VisualTransition GeneratedDuration="00:00:00.6" To="MouseOver">
    <VisualTransition.GeneratedEasingFunction>
      <CubicEase EasingMode="EaseOut" />
    </VisualTransition.GeneratedEasingFunction>
  </VisualTransition>
</VisualStateGroup.Transitions>
```

Lorsque les états de départ ou de destination, au sein de la balise `VisualTransition`, ne sont pas précisés, cette balise concerne toutes les transitions d'état à état quel que soit leur nombre. Les transitions dont vous souhaitez différencier le comportement passeront outre le réglage de base.

INFO

Vous pouvez dès à présent tester ces transitions en utilisant le mode de prévisualisation. Cliquez sur l'icône contenant une flèche et l'icône de lecture pour activer ce mode (🔍). Celle-ci est située tout en haut du panneau des états. Sélectionnez l'état `Normal`, puis `MouseOver`, la transition est automatiquement jouée de manière optimisée. Cette fonctionnalité est vraiment très utile et évite de compiler systématiquement l'application pour tester les transitions.

La dernière manière de gérer les transitions consiste à créer une animation personnalisée à l'aide d'un objet `Storyboard` personnalisé. Cette manière de procéder offre bien plus de possibilités mais il est moins facile de la maintenir sur le temps. Toutefois c'est également celle qui vous permettra de créer des transitions plus rythmées et plus originales. Sélectionnez l'état `MouseOver` et cliquez sur l'icône permettant d'afficher le scénario : 📄. Vous remarquez qu'une clé d'animation est présente pour l'objet `ContentPresenter` à la seconde 0. C'est le comportement par défaut adopté par le mode d'enregistrement d'état. Passez ensuite en mode d'édition XAML, Le code ci-dessous montre clairement l'intégration de l'objet `Storyboard` :

```
<VisualStateGroup x:Name="CommonStates">
  <VisualStateGroup.Transitions>
    <VisualTransition GeneratedDuration="00:00:00.4" />
    <VisualTransition GeneratedDuration="00:00:00.6" To="MouseOver">
```

```

        <VisualTransition.GeneratedEasingFunction>
            <CubicEase EasingMode="EaseOut" />
        </VisualTransition.GeneratedEasingFunction>
    </VisualTransition>
</VisualStateGroup.Transitions>
...
<VisualState x:Name="MouseOver">
    <Storyboard>
        <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
            Duration="00:00:00.001" Storyboard.
            TargetName="contentPresenter"
            Storyboard.TargetProperty="(UIElement.RenderTransform).
            (TransformGroup.Children)[0].(ScaleTransform.ScaleX)">
            <EasingDoubleKeyFrame KeyTime="00:00:00" Value="1.4" />
        </DoubleAnimationUsingKeyFrames>
        <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
            Duration="00:00:00.001" Storyboard.
            TargetName="contentPresenter" Storyboard.
            TargetProperty="(UIElement.RenderTransform).
            (TransformGroup.Children)[0].(ScaleTransform.ScaleY)">
            <EasingDoubleKeyFrame KeyTime="00:00:00" Value="1.4" />
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
...
</VisualStateGroup>

```

La structure de chaque état est simple, les transitions sont par défaut centralisées dans la balise `VisualStateGroup.Transitions`. La définition de chaque état contenant les propriétés modifiées est stockée, quant à elle, dans une balise `VisualState`. Cette balise contient au moins l'attribut `x:Name` correspondant au nom de la transition. Chaque objet de type `VisualState` contient un `Storyboard`. Cette animation est déclenchée chaque fois que l'état doit être affiché, mais par défaut les clés d'animation se situent toutes à la seconde 0. Autrement dit, la durée du `Storyboard` lui-même est par défaut de 0 seconde. La durée de la transition, qui est différente, est gérée au niveau des transitions, *via* la propriété `GeneratedDuration`, qui est configurable *via* le panneau `States` au sein de `Blend`. Supprimez la transition que nous avons définie pour l'état `MouseOver` en passant, soit par le code XAML, soit par l'interface de `Blend`. Pour le faire au sein de `Blend`, il vous suffit de cliquer sur l'icône moins (–). Déplacez ensuite la clé d'animation à la seconde 0.6. Pour finir, définissez une équation d'accélération de type `CubicEase` sur celle-ci. Vous obtenez le code XAML ci-dessous :

```

<VisualStateGroup x:Name="CommonStates">
    <VisualStateGroup.Transitions>
        <VisualTransition GeneratedDuration="00:00:00.4" />
    </VisualStateGroup.Transitions>
    ...
    <VisualState x:Name="MouseOver">
        <Storyboard>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
                Storyboard.TargetName="contentPresenter" Storyboard.
                TargetProperty="(UIElement.RenderTransform).
                (TransformGroup.Children)[0].(ScaleTransform.ScaleX)">
                <EasingDoubleKeyFrame KeyTime="00:00:00.6" Value="1.4">
                    <EasingDoubleKeyFrame.EasingFunction>
                        <CubicEase EasingMode="EaseOut" />
                    </EasingDoubleKeyFrame.EasingFunction>
                </EasingDoubleKeyFrame>
            </DoubleAnimationUsingKeyFrames>

```

```

<DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.
    TargetName="contentPresenter" Storyboard.
    TargetProperty="(UIElement.RenderTransform).
        (TransformGroup.Children)[0].(ScaleTransform.ScaleY)">
    <EasingDoubleKeyFrame KeyTime="00:00:00.6" Value="1.4">
        <EasingDoubleKeyFrame.EasingFunction>
            <CubicEase EasingMode="EaseOut"/>
        </EasingDoubleKeyFrame.EasingFunction>
    </EasingDoubleKeyFrame>
</DoubleAnimationUsingKeyFrames>
</Storyboard>
</VisualState>
...
</VisualStateGroup>

```

Maintenant que vous connaissez toutes les manières de créer des transitions, finalisez le design de chaque état interactif au sein du groupe `CommonStates`. Vous avez le choix : soit différencier l'état `Pressed` de l'état `MouseOver`, soit les cloner. Dupliquer l'état, par copier-coller du code XAML déjà généré, reste le plus simple.

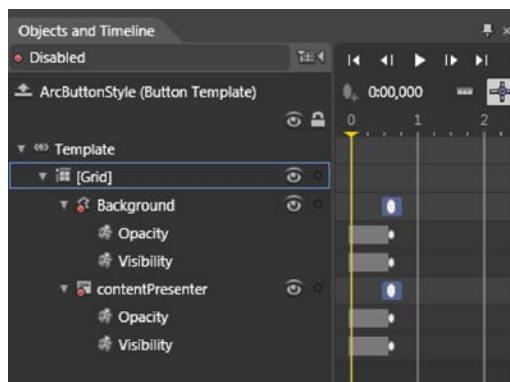
INFO

Si vous créez un Storyboard personnalisé comme réalisé pour l'état `MouseOver`, vous ne pourrez pas avoir d'aperçu de cet état en utilisant l'icône de l'œil. C'est tout à fait normal puisque cet œil ne montre que la seconde 0 du Storyboard (🕒). L'idéal serait évidemment d'afficher la clé d'animation finale.

Pour l'état `Disabled`, vous pouvez passer l'opacité de l'arrière-plan de couleur à 0 et passer sa propriété `Visibility` à `Collapsed`. Cela ne suffit pas pour le différencier de l'état `Normal`. Affectez l'opacité et la visibilité du `ContentPresenter` de la même manière. Ainsi lorsque le bouton est désactivé, il est complètement caché. Dépliez la ligne du temps ou scénario de cet état. Décalez ensuite toutes les clés d'animation de 6/10 de seconde. La propriété `Visibility` est animée grâce à une clé de type `Discrete`. Cela signifie qu'elle ne prendra la valeur `Collapsed` que lorsque la tête de lecture atteindra la clé. L'opacité sera, quant à elle, interpolée car la plage de valeurs admises est de type `Double` (voir Figure 7.26).

Figure 7.26

Scénario de l'état `Disabled`.

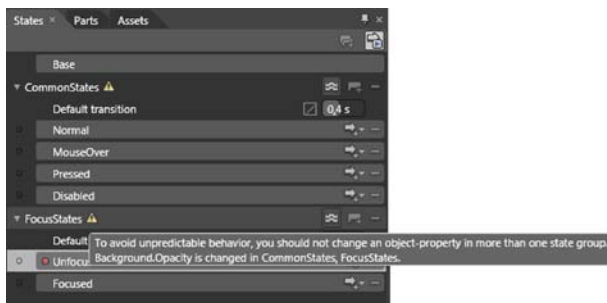


7.4.2.4 Contraintes logiques

Vous pouvez décider de personnaliser les états Focused et Unfocused. Si tel est le cas, sachez que vous ne pourrez pas réutiliser des propriétés déjà modifiées dans le groupe d'états CommonStates. Pour vous en rendre compte, sélectionnez l'état Unfocused et modifiez l'opacité du tracé vectoriel assurant le fond de couleur. Au sein du panneau des états, une nouvelle icône apparaît. Elle vous indique une utilisation illogique des états (voir Figure 7.27).

Figure 7.27

Erreur logique d'enregistrement d'état.

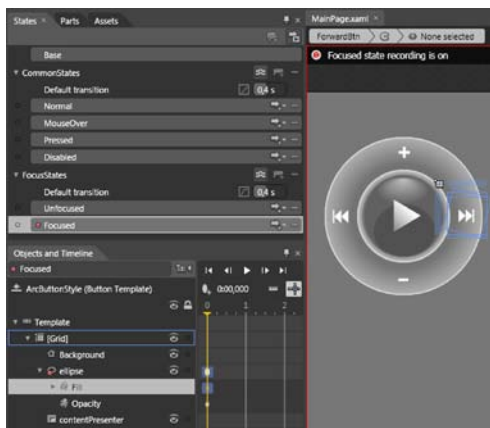


Le dilemme est simple, les groupes d'états sont indépendants les uns des autres du point de vue de leur affichage. Le bouton peut posséder le focus ou non, et être survolé par la souris en même temps. Dès lors, les deux états vont être tous deux affichés. Toutefois, comme la propriété `Opacity` du tracé `Background` a été affectée dans chaque état, le gestionnaire d'états visuels ne saura pas laquelle des deux affectations choisir. L'affectation est due au fait que le bouton n'a pas le focus ou elle est générée par le survol de la souris. C'est un choix impossible à résoudre de manière élégante et cela est tout à fait logique d'un point de vue conception. Il n'y a aucune manière simple de résoudre ce type de conflit car c'est le raisonnement à la source qui est faux.

Avoir le focus utilisateur n'est pas la même chose que survoler le bouton. Le visuel doit donc être différent. Supprimez les modifications apportées à l'état `Unfocused`. Le mieux serait de rajouter une forme de type `Ellipse` sous le `ContentPresenter` avec un léger dégradé du blanc vers le blanc transparent pour simuler une lueur. Lorsque le bouton n'a pas l'intérêt utilisateur, cette lueur est désactivée en passant son opacité à 0. Compilez et testez votre application en utilisant la touche `Tabulation` (voir Figure 7.28).

Figure 7.28

Gestion visuelle du focus utilisateur.



INFO

Pour voir l’affichage du focus sans l’arrière-plan de couleur, vous pouvez sélectionner l’état `Focused`, puis cliquez sur l’icône d’aperçu à gauche de l’état `Normal` (voir Figure 7.28). De cette manière, vous pouvez tester le visuel de l’état `Focused` avec n’importe quel autre état interactif si celui-ci n’appartient pas au groupe `FocusStates`.

Vous trouverez cet exercice corrigé dans : *chap7/PlayerVideo_ButtonsAnime.zip*.

7.4.3 Au niveau de l’application

Nous allons maintenant créer des états au niveau de l’application elle-même. Les états que nous avons configurés jusqu’à présent étaient propres à la classe `Button`. Une application `Silverlight` ne possède pas d’états visuels par défaut, il vous faudra donc les créer manuellement. Avant de commencer, il importe de faire un rapide tour des différents affichages que nous pourrions envisager.

7.4.3.1 Découper l’application

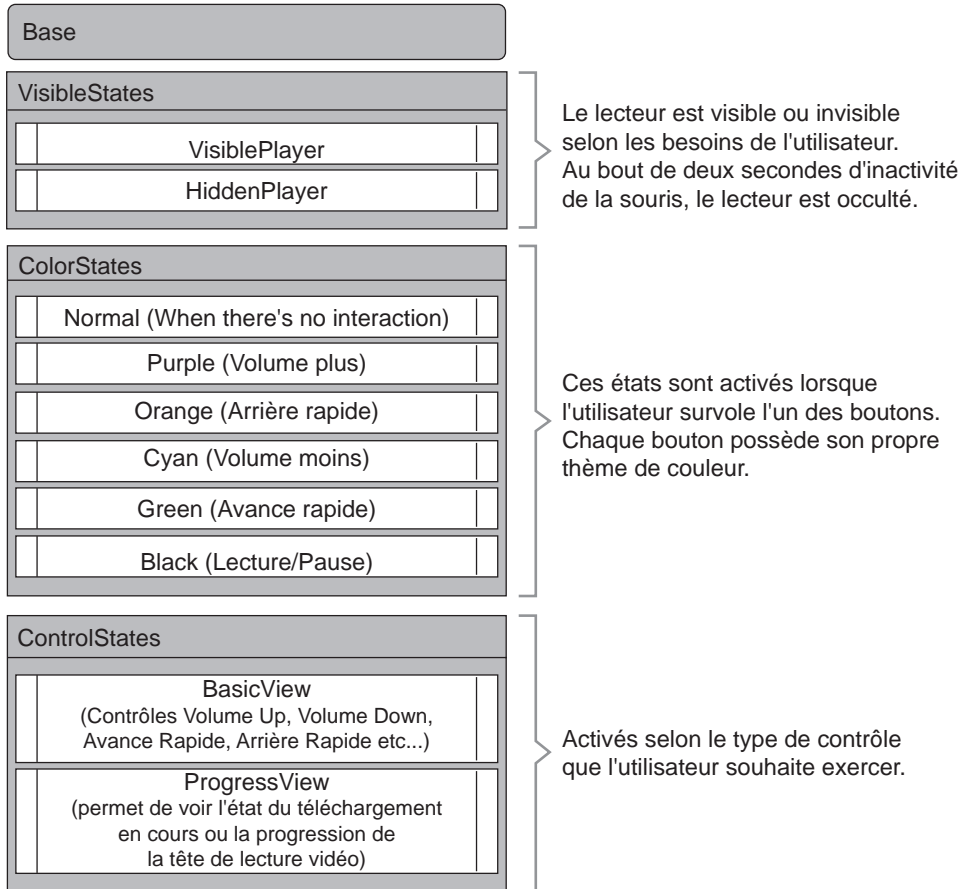
Nous partons du principe que le lecteur vidéo s’occultera de lui-même après un certain temps d’inactivité de la souris. Il sera donc parfois visible et parfois invisible. Cette fonctionnalité peut faire l’objet d’un groupe d’états nommé `VisibleStates`. Pour l’instant le lecteur vidéo ne semble présenter que des fonctionnalités sommaires, toutefois on pourrait imaginer plusieurs affichages différents.

Par exemple, à chaque fois que la souris survolerait l’un des boutons de contrôle, le disque se teinterait partiellement pour illustrer l’action en cours. Pour créer ce type d’interaction, il nous faudra créer un groupe d’états nommé `ColorStates`. Nous pourrions également proposer deux modes d’affichage. L’utilisateur pourrait soit utiliser les contrôles de base que nous avons déjà réalisés, soit consulter l’état de téléchargement et naviguer grâce à une tête de lecture directement sur une ligne de temps représentée par le disque. Au final, cette fonctionnalité constituerait également un groupe d’états visuels nommé `ControlStates`.

Nous aurons donc trois groupes d’états indépendants qui géreront chacun l’affichage de l’une de ces fonctionnalités (voir Figure 7.29).

Figure 7.29

Schéma des groupes d'états visuels propres à notre application.



7.4.3.2 Créer des états visuels

Positionnez-vous au niveau de l'application et affichez le panneau States. Celui-ci est pour l'instant vide. Créez trois groupes *via* l'icône d'ajout de groupe (📁). Nommez-les respectivement `VisibleStates`, `ColorStates` et `ControlStates`. Vous allez créer tous les états décrits à la Figure 7.29. À droite du groupe `VisibleStates`, cliquez sur l'icône d'ajout d'état (📄). Nommez le nouvel état créé `VisiblePlayer` et répétez l'opération autant de fois que nécessaire pour tous les groupes. Vous allez commencer par gérer les transitions du groupe `ColorStates`. Pour celui-ci, définissez une transition générique de 4/10 de seconde ainsi qu'une équation d'accélération de votre choix. Vous n'avez pas réellement besoin d'animations personnalisées pour l'interpolation de couleur. Utiliser une animation générique est dans ce cas le meilleur moyen de gagner du temps (voir Figure 7.30).

Figure 7.30

Groupes d'états visuels du lecteur vidéo.



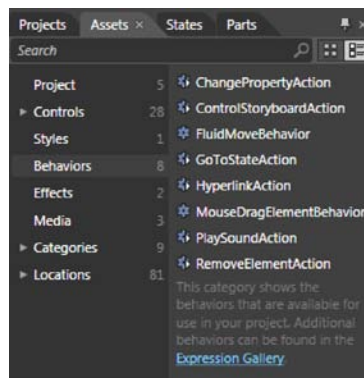
Il ne vous reste plus qu'à sélectionner chaque état au sein de ce groupe pour modifier la couleur du dégradé du disque situé sous les contrôles utilisateur. Pour récupérer les bonnes nuances de couleur, vous pouvez temporairement modifier l'état `Normal` du bouton générique. En passant la valeur d'opacité du bouton à 100 %, vous faites apparaître les couleurs de chacun des contrôles. Ensuite, sélectionnez l'état `Purple`, puis dans l'arbre visuel de l'application, l'objet `LargeColorRing`. C'est en fait l'un des disques que vous allez colorer. Modifiez son remplissage pour un dégradé du rose vers le blanc. Vous pouvez vous baser sur la couleur du bouton de volume `VolumeUpBtn`. Sélectionnez ensuite le tracé `SmallColorRing` et recommencez le processus. Procédez de même pour chaque état de couleur en utilisant les couleurs de base de chaque bouton. Retournez dans le modèle de bouton générique, au sein de l'état `Normal`, et redéfinissez un arrière-plan à 100 % d'opacité. Avant de piloter le gestionnaire d'états visuels, vous pouvez créer une ou deux animations personnalisées pour les états du groupe `VisibleStates`. Ne vous occupez pas pour l'instant du groupe `ControlStates`.

7.4.3.3 Gérer les transitions grâce aux comportements interactifs

Les comportements ou *Behaviors* au sein de Blend et de l'API XAML permettent aux designers interactifs de gérer par eux-mêmes les interactions simples. Ouvrez le panneau **Assets** et le menu **Behaviors**, la liste des comportements accessibles par défaut apparaît (voir Figure 7.31).

Figure 7.31

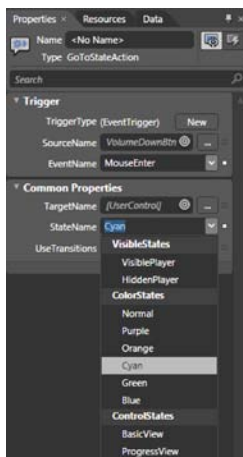
Liste des comportements disponibles.



Vous les utiliserez tous tôt ou tard dans vos développements, ainsi que dans ce livre. Nous les aborderons plus en détail au Chapitre 8. Glissez-déposez le comportement nommé `GoToStateAction` sur chacun des quatre contrôles utilisateur déjà réalisés. Ils permettent d'accéder à un état visuel lors de l'interaction utilisateur de votre choix. Dans le panneau des propriétés, configurez-les pour qu'ils réagissent au survol de la souris (`MouseEnter`) et ciblent l'état de couleur correspondant au sein du groupe `ColorStates` (voir Figure 7.32).

Figure 7.32

Configuration du comportement `GoToStateAction`.

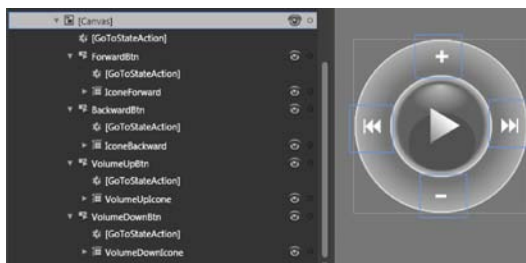


Compilez et testez le projet. Vous remarquez que dès que vous survolez un bouton, l'application change d'état visuel. La couleur des deux disques se met à jour dynamiquement. Toutefois, l'application ne réaffiche pas l'état `Normal`. Il faudrait pour cela définir un nouveau comportement ciblant cet état lorsque la souris quitte le survol des boutons. C'est assez simple à réaliser.

Afin de vous éviter un travail fastidieux, groupez tous les contrôles au sein d'un conteneur de type `Canvas`. Déposez le comportement `GoToStateAction` sur ce dernier. Ensuite, configurez-le afin qu'il se déclenche lorsque la souris quitte son survol et qu'il cible l'état `Normal`. Cette manière de procéder vous évite de recréer le même comportement quatre fois (une fois par contrôle). L'arbre visuel de notre application s'est légèrement étoffé, vous pouvez voir la portion correspondant aux comportements ajoutés sur la Figure 7.33.

Figure 7.33

Partie de l'arbre visuel avec comportements.



Testez et compilez votre application : celle-ci réagit de manière fluide lors du survol de chaque contrôle.

7.4.3.4 Piloter le gestionnaire d'états visuels avec C#

Piloter et gérer le déclenchement des transitions en C# est assez simple. Nous devons pour cela utiliser la classe `VisualStateManager`. Nous allons faire en sorte que tant que la souris bouge, l'état `VisiblePlayer` soit l'état de destination. Au bout de deux secondes d'inactivité, l'état ciblé sera `HiddenPlayer`. Le code ci-dessous constitue la première étape :

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();

        //Par défaut le lecteur est caché
        VisualStateManager.GoToState(this, "HiddenPlayer", true);

        //on écoute l'événement MouseMove
        LayoutRoot.MouseMove += LayoutRoot_MouseMove;
    }

    //Chaque fois que celui-ci est diffusé
    //donc lorsque la souris bouge
    void LayoutRoot_MouseMove(object sender, MouseEventArgs e)
    {
        //on utilise le gestionnaire d'états visuels
        //afin d'afficher l'état VisiblePlayer
        VisualStateManager.GoToState(this, "VisiblePlayer", true);
    }
}
```

Dans le code ci-dessus, les deux lignes importantes concernent la classe `VisualStateManager`. Elle possède la méthode statique `GoToState` qui permet d'afficher l'état visuel de n'importe quel contrôle. Le premier argument est le contrôle dont vous souhaitez afficher l'état. Dans notre cas, nous souhaitons atteindre l'état du `UserControl` racine, soit `this`, l'instance de la classe `MainPage`. Le second paramètre est la chaîne de caractères correspondant au nom de l'état. Le dernier argument permet de spécifier si nous souhaitons utiliser ou non une transition animée. Toutefois, cette option ne concerne pas les transitions reposant sur un objet `Storyboard` spécifique. Une transition personnalisée sera jouée quelle que soit la valeur du dernier argument car ce n'est pas l'objet `State` qui définit la durée d'animation (*via* la propriété `GeneratedDuration`) mais, dans ce cas, le `Storyboard`. Pour finaliser l'interaction utilisateur, il va falloir créer un compte à rebours. Ce dernier sera géré par une instance d'objet de type `DispatcherTimer`. L'idée consiste à réinitialiser le compte à rebours à chaque fois que la souris bouge. Lorsque la souris ne bouge plus, le compte à rebours égrène les secondes. Au bout de deux secondes d'inactivité, on déclenche l'animation vers l'état `HiddenPlayer`. On occulte donc celui-ci.

```
//Ajouter l'espace de noms Threading
//pour instancier DispatcherTimer
using System.Windows.Threading;

namespace PlayerVideo
{
    public partial class MainPage : UserControl
    {
        //on crée un métronome
        DispatcherTimer Dt = new DispatcherTimer();
```

```

//on lui définit un intervalle de temps de deux secondes
Dt.Interval=TimeSpan.FromSeconds(2);

public MainPage()
{
    InitializeComponent();

    //Par défaut le lecteur est caché
    VisualStateManager.GoToState(this, "HiddenPlayer", true);

    //on écoute l'événement MouseMove
    LayoutRoot.MouseMove += LayoutRoot_MouseMove;

    //on écoute le métronome (DispatcherTimer)
    Dt.Tick += Dt_Tick;
}

//Chaque fois que celui-ci est diffusé
//donc lorsque la souris bouge
void LayoutRoot_MouseMove(object sender, MouseEventArgs e)
{
    //on utilise le gestionnaire d'états visuels
    //afin d'afficher l'état VisiblePlayer
    VisualStateManager.GoToState(this, "VisiblePlayer", true);

    //puis on réinitialise le compte à rebours à deux secondes
    Dt.Start();
}

//lorsque les deux secondes sont atteintes
//c'est-à-dire lorsque la souris
//ne bouge plus depuis deux secondes
void Dt_Tick(object sender, EventArgs e)
{
    //on cache le lecteur vidéo grâce à la classe
    //VisualStateManager
    VisualStateManager.GoToState(this, "HiddenPlayer", true);

    //on stoppe le métronome pour optimiser les performances
    Dt.Stop();
}
}...

```

Testez et compilez l'application, les transitions sont fluides et agréables. Les contrôles du lecteur vidéo ne sont accessibles que lorsque l'utilisateur en a réellement besoin. Le lecteur est donc contextuel, ce qui est encore la meilleure manière de concevoir l'expérience utilisateur. Vous trouverez cet exercice dans : *chap7/PlayerVideo_AppAnime.zip*.

7.5 Le bouton interrupteur ou *ToggleButton*

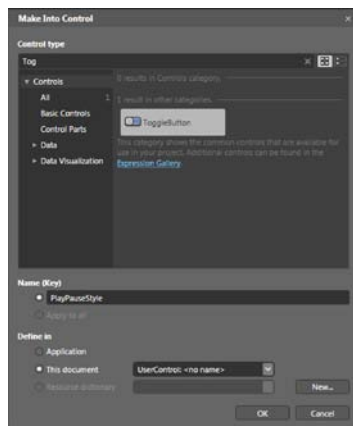
Nous allons maintenant personnaliser un bouton de type interrupteur (*ToggleButton*). Vous pouvez considérer ce type de bouton comme une case à cocher (*CheckBox*) car seule leur forme diffère. Ces deux composants possèdent trois états visuels supplémentaires par rapport à un bouton traditionnel : *Checked*, *Unchecked* et *Indeterminate*. Habituellement, on trouve ces boutons dans les formulaires. Grâce au XAML et à la personnalisation de contrôle utilisateur, nous utiliserons le comportement on/off de ce type de bouton pour alterner entre la lecture de la vidéo et sa mise en pause.

7.5.1 Créer le bouton

Dans l'arbre visuel et logique de l'application, sélectionnez la grille nommée `PlayPause`, cliquez-droit dessus et choisissez l'option `Make Into Control...` Dans la liste affichée, sélectionnez `ToggleButton`, nommez le style `PlayPauseStyle`. Vous pouvez vous aider du champ de recherche pour afficher le bouton de type `ToggleButton` (voir Figure 7.34). Confirmez ensuite la création du style en cliquant sur `OK`.

Figure 7.34

*Création du style
PlayPause.*

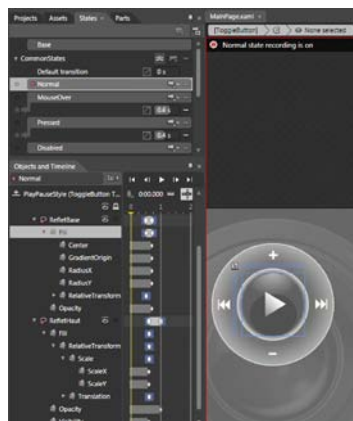


Commencez par cacher le composant `ContentPresenter` en passant sa propriété `Visibility` sur `Collapsed`. Vous utiliserez ce composant ultérieurement pour indiquer une information à l'utilisateur. Vous allez faire apparaître le reflet lors du survol de la souris uniquement. Pour cela, sélectionnez l'état `Normal` et créez une animation de votre choix pour faire disparaître le reflet de manière élégante. Vous pouvez, par exemple, concevoir une animation personnalisée sur les dégradés simulant l'impact lumineux et le reflet. Ceux-ci correspondent respectivement aux valeurs de la propriété `Fill` pour les composants `Ellipse RefletHaut` et `RefletBase`.

Créer une animation personnalisée pour l'état `Normal` vous simplifiera grandement la tâche. Il suffit de faire disparaître les reflets déjà présents. Pour les autres états, vous pouvez utiliser des transitions génériques afin d'alléger au maximum votre travail (voir Figure 7.35). Testez et compilez le projet.

Figure 7.35

*Exemple d'animation
de disparition pour
l'état Normal.*




7.5.2 Le système d'agencement fluide

Les trois états supplémentaires, propres au `ToggleButton` et déjà évoqués, sont présents au sein d'un groupe d'états nommé `CheckStates`. Nous n'avons pas besoin d'utiliser l'état `Indeterminate` car nous avons besoin d'un interrupteur à deux états. Lorsque le bouton sera coché, nous ferons apparaître la grille nommée `IconePause`, indiquant ainsi à l'utilisateur que la vidéo est en cours de lecture et qu'il doit appuyer sur le bouton pour la mettre en pause. À l'opposé, lorsque le bouton sera décoché, la vidéo sera sur pause et la grille `IconePlay` sera visible, indiquant à l'utilisateur qu'il peut la remettre en lecture à tout instant.

Sélectionnez l'état `Checked` et affectez la propriété `Visibility` de `IconePause` à `Visible`, ensuite passez la propriété `Visibility` de la grille `IconePlay` à `Collapsed`. Compilez et testez votre projet, les icônes apparaissent alternativement l'une et l'autre lors de chaque clic gauche de la souris.

Toutefois, il n'y a pas de transitions fluides entre les états coché et non coché. Cela est très simple à résoudre grâce à une fonctionnalité présente depuis Silverlight 3, nommée système d'agencement fluide ou *Fluid Layout System*. Les propriétés qui ne sont pas de type `Point`, `Double` ou `Color` ne sont pas interpolées et subissent des animations de type `Discreet`. Le système d'agencement fluide résout en partie cette limitation en générant une animation de ce que pourrait être l'interpolation, la plus logique, de telles propriétés. Par exemple, vous pourriez décider qu'un des enfants d'une grille soit aligné à gauche dans un état visuel, puis à droite dans un autre état visuel. Habituellement, une transition fluide serait impossible car l'alignement est géré par la propriété `HorizontalAlignment` qui est une énumération. Toutefois, le système d'agencement fluide permet de simuler ce que serait l'animation la plus logique entre les deux positions d'alignement.

Commencez par définir une transition générique d'une durée de 6/10 de seconde pour le groupe d'état `CheckStates`. Cliquez sur l'icône représentant des vaguelettes () située à droite de l'état `CheckStates` afin d'activer le système d'agencement fluide. Compilez et testez à nouveau le bouton de lecture/pause. Cette fois une animation est jouée ; elle n'était pas vraiment facile à prévoir, on a l'impression d'un redimensionnement des icônes permettant de les faire apparaître ou disparaître. En un seul clic, nous avons permis de simuler l'animation de la propriété `Visibility` qui n'est pourtant pas interpolable. Du côté XAML, le code est assez simple :

```
<VisualStateManager x:Name="CheckStates"
    ic:ExtendedVisualStateManager.UseFluidLayout="True">
    <VisualStateManager.Transitions>
        <VisualTransition GeneratedDuration="00:00:00.6" />
    </VisualStateManager.Transitions>
    <VisualState x:Name="Unchecked" />
    <VisualState x:Name="Indeterminate" />
    <VisualState x:Name="Checked">
        <Storyboard>
            <ObjectAnimationUsingKeyFrames BeginTime="00:00:00"
                Duration="00:00:00.001" Storyboard.TargetName="IconePause"
                Storyboard.TargetProperty="(UIElement.Visibility)">
                <DiscreteObjectKeyFrame KeyTime="00:00:00">
                    <DiscreteObjectKeyFrame.Value>
                        <Visibility>Visible</Visibility>
                    </DiscreteObjectKeyFrame.Value>
                </DiscreteObjectKeyFrame>
            </ObjectAnimationUsingKeyFrames>
            <ObjectAnimationUsingKeyFrames BeginTime="00:00:00"
                Duration="00:00:00.001" Storyboard.TargetName="IconePlay">
```

```

        Storyboard.TargetProperty="(UIElement.Visibility)">
        <DiscreteObjectKeyFrame KeyTime="00:00:00">
        <DiscreteObjectKeyFrame.Value>
        <Visibility>Collapsed</Visibility>
        </DiscreteObjectKeyFrame.Value>
        </DiscreteObjectKeyFrame>
    </ObjectAnimationUsingKeyFrames>
</Storyboard>
</VisualState>
</VisualStateGroup>

```

La propriété `UseFluidLayout` possède la valeur `true`, c'est celle-ci qui permet d'activer le gestionnaire d'états visuels. Vous remarquez cependant que la propriété `Visibility` n'est pas interpolée car elle subit des clés d'animation de type `DiscreteObjectKeyFrame`. L'animation réelle ressemblant à un redimensionnement (`RenderTransform`) nous est complètement occultée. C'est le gestionnaire d'états qui gère cette animation en arrière-plan. Il ne vous reste plus qu'à créer un nouveau comportement pour le bouton de lecture, qui indiquera à l'application de naviguer vers l'état `Black` lors de son survol (voir section 7.4.3.3).

D'un point de vue design, le lecteur vidéo est terminé, vous finaliserez son développement au Chapitre 12, dédié aux composants personnalisés. Vous trouverez le lecteur vidéo dans : *chap7/PlayerVideo_ToggleButton.zip*.

Nous ne finalisons pas le lecteur vidéo dans ce chapitre pour diverses raisons. L'une d'elles est que la création du code logique pour ce type d'application repose grandement sur la notion de programmation événementielle. Depuis le début du livre, nous avons évoqué à de nombreuses reprises ce concept sans vraiment l'étudier. Cette notion est très importante au sein des langages de haut niveau tels que C#. Nous allons donc l'aborder au Chapitre 8 et voir comment une utilisation adéquate de ses concepts peut nous faciliter grandement la vie.

Interactivité et modèle événementiel

Le concept d'interactivité repose sur la notion de stimulus, et d'une ou plusieurs actions invoquées en réponse à celui-ci. Ce principe, pour les applications, est essentiellement lié à l'utilisateur, il fait appel à la logique événementielle. Dans ce chapitre, vous étudierez en profondeur les mécanismes du modèle événementiel dans Silverlight, pour le langage C#. Ainsi, vous apprendrez la diffusion, la souscription et le désabonnement d'événements. Puis, vous verrez comment optimiser votre code grâce au couplage faible et à la propagation des événements. Pour finir, vous aborderez la création de comportements interactifs personnalisés afin de faciliter et d'améliorer le flux de production entre designers et développeurs.

8.1 Les fondements du modèle événementiel

Le modèle événementiel, à la base de tout développement orienté utilisateur, est souvent relié aux langages de haut niveau, tels que C#. Il occupe également une place importante au sein des technologies asynchrones comme Silverlight ou Ajax. Il correspond pleinement à la notion de connexion distante sur laquelle repose Internet ; sur le Web, rien n'est synchrone. La durée écoulée entre l'instant où vous cliquez sur un lien et le moment où la nouvelle page s'affiche varie de manière impossible à prévoir. Cette variation s'explique assez bien : les données sont distantes et doivent donc être téléchargées avant d'être affichées. Le débit des lignes Internet, la fréquentation d'un site ou l'occupation mémoire comme processeur d'un serveur à un instant t ne sont jamais identiques. La logique événementielle prend tout son sens sur ce type de réseau. Il en va de même pour les interactions utilisateur.

Le développeur ne peut jamais prévoir quand un utilisateur va cliquer sur un bouton ou quand il va bouger la souris. Il peut, en revanche, décider d'un comportement à adopter lorsque cela arrivera. Pour cela, il utilise les outils fournis par le modèle événementiel, propres à chaque langage. Nous allons maintenant essayer de comprendre l'origine et les mécanismes du modèle événementiel.

8.1.1 Le pattern Observateur

Les modèles de conception (ou *design patterns*) sont nés à la fin des années 80 avec l'avènement de la programmation orientée objet (POO). À cette époque, la majeure partie, soit 99,99 %, des problèmes de conception avait déjà été résolue par les développeurs en matière de POO et n'avait plus rien d'originale. Les patterns sont nés du besoin de formaliser des solutions génériques répondant aux différentes problématiques rencontrées. Dès 1991, quatre développeurs, connus sous le nom de "Gang of Four", écrivent le premier ouvrage de référence sur ce sujet : *Design Patterns: Elements of Reusable ObjectOriented Software*. Il s'agit d'Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Ces concepteurs formalisent ainsi les bonnes pratiques de conception orientée objet répondant à la majorité des problématiques. Le modèle Observateur, ou *Observer* en anglais est l'un d'entre eux, il fait partie des patterns liés à la gestion du comportement des objets. Il définit une relation de un à plusieurs objets, et permet lorsqu'un objet change d'état, de le notifier à plusieurs autres qui en dépendent. Ceux-ci peuvent ainsi réagir ou être mis à jour automatiquement. Pour que les objets soient notifiés ou informés des changements, ils doivent souscrire auprès de l'objet qui diffuse les notifications. Ce modèle de conception repose donc sur deux notions essentielles : la souscription et la diffusion de notifications. C'est exactement le principe de la programmation événementielle. Dans ce contexte, une notification est appelée événement. Prenons un exemple de la vie courante :

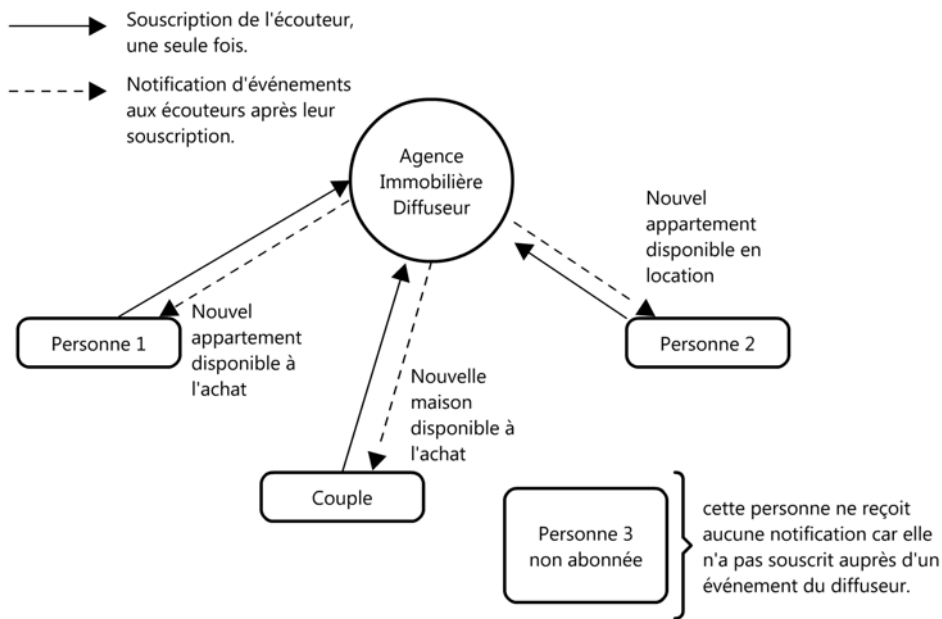
1. Vous désirez acheter ou louer un bien immobilier. Vous souhaitez être informé lorsqu'un nouvel appartement ou qu'une nouvelle maison est disponible à l'achat ou à la location pour ne pas vous déplacer pour rien.
2. Vous souscrivez aux lettres d'informations de plusieurs agences immobilières. Vous vous abonnez en fait aux événements "nouvel appartement à louer" et "nouvel appartement à acheter" diffusés par les agences immobilières.
3. Lorsqu'une agence reçoit une nouvelle offre de location ou d'achat, elle vous en fait part. Pourquoi ? Simplement parce que vous êtes abonné. Les non abonnés ne sont pas notifiés de l'événement.
4. Vous pouvez réagir différemment en fonction des caractéristiques du bien immobilier. Le descriptif de celui-ci est contenu le plus souvent dans un objet événementiel, par exemple, la lettre d'information.
5. Lorsque vous avez trouvé le bien adéquat et que vous l'avez acquis, vous vous désabonnez car il n'est plus nécessaire de chercher.

Nous venons de décrire exactement le principe du modèle événementiel. Chaque agence immobilière possède plusieurs abonnés et diffuse des événements de type "nouveau bien disponible" à chacun d'eux. Le diffuseur est également appelé *sujet*, les abonnés sont appelés *écouteurs* (voir Figure 8.1).

Nous allons maintenant démontrer en quoi ces principes sont souples et faciles à maintenir à travers une introduction au couplage faible.

Figure 8.1

Principes du pattern Observateur à travers l'exemple d'annonces immobilières.



8.1.2 Introduction au couplage faible

Lorsque deux objets n'ont pas besoin de se connaître pour collaborer ensemble, on dit qu'ils sont faiblement couplés, c'est-à-dire qu'ils n'entretiennent pas de relations spécifiques fortes. Une relation forte signifie qu'un objet fait explicitement référence à un autre objet pour être fonctionnel. Le couplage faible représente exactement la nature des relations entretenues par les écouteurs et les diffuseurs d'événements :

- Le sujet (ou diffuseur) n'a pas besoin de connaître les objets qui écoutent ses événements diffusés. Ainsi, il est possible de supprimer des écouteurs sans que le diffuseur ne soit dérangé dans son fonctionnement.
- De même, il est possible d'ajouter des écouteurs à tout instant en cours d'exécution à n'importe quel diffuseur.
- Les écouteurs ou les diffuseurs peuvent avoir d'autres activités que la souscription ou la diffusion d'événements de manière totalement indépendante.
- Modifier un objet abonné ou diffuseur ne change pas les relations ou le fonctionnement de chacun des acteurs du processus.

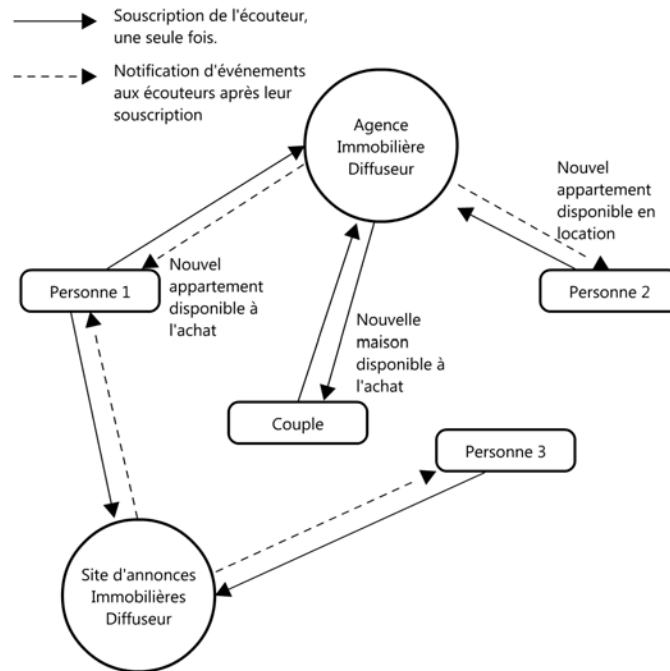
L'avantage principal du couplage faible est de permettre des conceptions souples et faciles à maintenir. Il est possible de concevoir des combinaisons d'écouteurs-diffuseurs plus complexes, sans

mettre en danger le fonctionnement ou la cohérence des liaisons. Par exemple, vous pouvez vous abonner aux lettres d'informations de plusieurs agences immobilières (voir Figure 8.2).

De plus, la suppression d'un écouteur ou d'un sujet n'affecte en rien le fonctionnement de l'un ou de l'autre. Nous verrons à la section 8.2 comment désabonner un écouteur à l'exécution.

Figure 8.2

Un schéma plus complexe.



8.1.3 Souscrire à un événement en C#

Dans la majorité des conceptions, l'écouteur est une méthode. Ainsi, définir un écouteur d'événements consiste à définir une méthode s'exécutant à chaque diffusion de l'événement. Toutefois, nous allons voir que ces méthodes sont particulières. Si vous développez pour le Web en JavaScript ou ActionScript, vous avez pu rencontrer l'écriture suivante :

```
uneInstance.addEventListener("événement", écouteur ) ;
```

L'écriture en C# est assez comparable. Le code générique ci-dessous permet d'écouter un événement en langage C# :

```
UneInstance.Evenement += Ecouteur ; //l'écouteur est une méthode
...
void Ecouteur ( Diffuseur, ObjetEvenementiel)
{
    //Fait quelque chose
}
```

L'opérateur d'affectation += permet d'ajouter la méthode à la liste des écouteurs. En pratique, déclencher la méthode ClickButton chaque fois que l'utilisateur clique sur un bouton nommé MonBouton est réalisé comme ci-dessous :

```
MonBouton.Click += new RoutedEventHandler(ClickButton);
...
void ClickButton ( object sender, RoutedEventArgs e)
{
    //Fait quelque chose
}
```

Voici la traduction en "bon français" de ce qui est réalisé : l'écouteur, représenté par la méthode ClickButton, souscrit à l'événement Click de MonBouton. Lorsque l'utilisateur clique sur le bouton durant l'exécution, ce dernier diffuse l'événement Click. La méthode ClickButton est alors invoquée car elle fait partie des abonnés à l'événement Click de MonBouton.

Nous allons maintenant mettre en pratique cet exemple.

8.1.4 Cas pratique

Ouvrez le projet nommé PlayerVideo_ToggleButton réalisé au Chapitre 7. Vous pouvez également le trouver dans l'archive *PlayerVideo_ToggleButton.zip* du dossier du *chap8*. Sauvegardez-en une copie nommée PlayerVideo_Evenement via le menu File d'Expression Blend, puis ouvrez cette copie ; cela vous évite de modifier le projet original. Nous allons remplacer les comportements (ou Behaviors) définis dans Expression Blend, pour naviguer d'un état visuel de l'application à un autre, par du code C#. Commencez par supprimer tous les comportements que vous trouverez sur les boutons (dont le Toggle-Button). Nommez le ToggleButton de lecture/pause PlayPauseBtn et le conteneur Canvas des contrôles ControlsRingArea. Ensuite, dans le code C#, ajoutons l'équivalent de l'interactivité utilisateur produite par les comportements. Pour cela, il faut écouter chaque bouton de contrôle du lecteur vidéo séparément comme montré dans le code ci-dessous :

```
public MainPage()
{
    InitializeComponent();
    //Par défaut le lecteur est caché
    VisualStateManager.GoToState(this, "HiddenPlayer", true);

    //on écoute l'événement MouseMove
    LayoutRoot.MouseMove += new MouseEventHandler(LayoutRoot_MouseMove);

    Dt.Tick += new EventHandler(Dt_Tick);

    #region gestion des transitions vers les états de
    #couleurs ColorStates Group
    ForwardBtn.MouseEnter += new MouseEventHandler(ForwardBtn_MouseEnter);

    BackwardBtn.MouseEnter += new MouseEventHandler(BackwardBtn_MouseEnter);

    VolumeDownBtn.MouseEnter += new MouseEventHandler
        (VolumeDownBtn_MouseEnter);

    VolumeUpBtn.MouseEnter += new MouseEventHandler(VolumeUpBtn_MouseEnter);
```

```

        PlayPauseBtn.MouseEnter += new MouseEventHandler
                                   (PlayPauseBtn_MouseEnter);

        ControlsRingArea.MouseLeave += new MouseEventHandler
                                   (ControlsRingArea_MouseLeave);
    #endregion
}

...

void ControlsRingArea_MouseLeave(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Normal", true);
}

void VolumeUpBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Purple", true);
}

void VolumeDownBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Cyan", true);
}

void BackwardBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Orange", true);
}

void ForwardBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Green", true);
}

void PlayPauseBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Black", true);
}

```

INFO

Le code généré est un peu verbeux, vous pourriez l'éviter en définissant l'écoute de l'événement en langage déclaratif XAML *via* le panneau Properties. Toutefois, cela peut se révéler plus problématique qu'autre chose. Le développeur n'ira pas naturellement vers cette solution propre aux designers interactifs. Si l'écoute est définie dans le XAML, elle est partiellement cachée au développeur car un fichier déclaratif peut être fastidieux à explorer. Cela pourrait engendrer des pertes de performance ou des conflits si le développeur n'a pas pris connaissance de l'écoute d'un événement.

Testez et compilez le lecteur vidéo, vous obtenez exactement la même interactivité qu'avec les comportements. Vous trouverez plusieurs pistes à la section 8.5 afin de choisir entre comportements interactifs et programmation événementielle C#. Ce choix sera fonction de la situation et du contexte de production que vous rencontrerez. Il est tout de même utile de préciser que l'avantage de la programmation événementielle reste le contrôle total des performances par le développeur. En effet, ce dernier peut, à tout moment, ajouter ou supprimer l'écoute d'un événement à l'exécu-

tion. Ainsi, l'application ne consomme que la mémoire et la ressource processeur qui lui est utile à un instant donné. Nous allons maintenant étudier plusieurs techniques d'optimisation.

8.2 Supprimer l'écoute d'un événement

La première technique consiste tout simplement à arrêter l'écoute d'un événement lorsqu'il n'est plus utile d'y réagir.

8.2.1 Principe

Dans la vie réelle, c'est une tâche courante et saine. Dans l'exemple de l'agence immobilière, lorsque l'un des abonnés aux lettres d'informations a trouvé un appartement ou une maison, il ne souhaite plus être informé des nouvelles offres. Traduire cette notion en langage C# est simple : à l'opposé de l'opérateur d'affectation `+=` qui signifie ajouter, supprimer se traduit par l'opérateur `-=`. Le code ci-dessous montre l'écriture complète :

```
UneInstance.Evenement -= Ecouteur ;
```

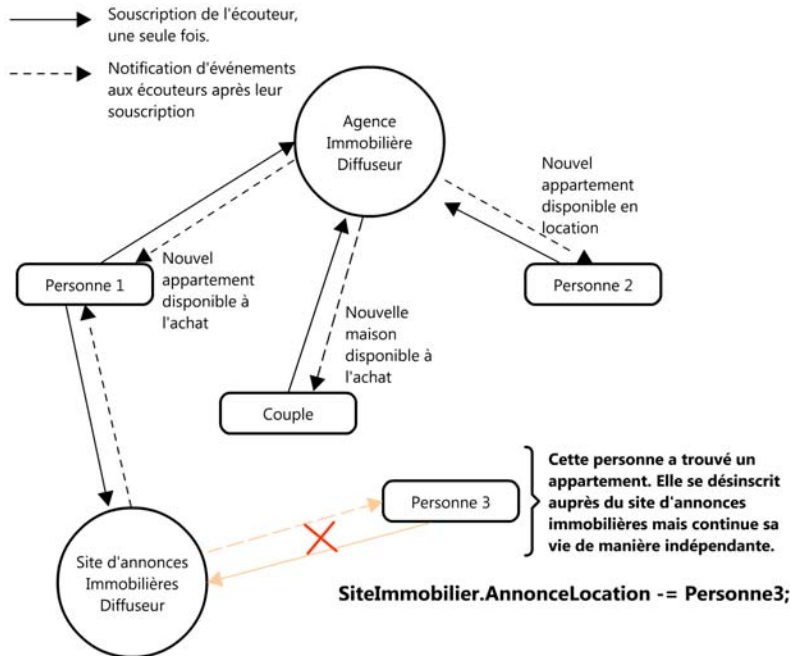
Ainsi, pour supprimer l'écoute d'un bouton, vous pouvez écrire :

```
MonBouton.Click -= ClickButton;
```

Pour éviter toute confusion, vous pouvez traduire ce code en français par : l'écouteur `ClickButton` se désabonne de l'événement `Click` diffusé par `MonBouton`.

Il ne faut pas vous imaginer que vous supprimez la méthode `ClickButton` de cette manière. Dans les faits, vous supprimez juste la référence `ClickButton` dans la liste des écouteurs de l'événement `Click` (voir Figure 8.3).

Vous pourrez donc souscrire à nouveau `ClickButton` à cet événement plus tard si besoin. Si la souscription auprès d'événements diffusés est importante, le désabonnement l'est tout autant pour diverses raisons. La première est de contrôler l'interactivité. La seconde raison est liée à la gestion des ressources. Écouter un événement consomme de la mémoire, parfois inutilement. Par exemple, écouter le déplacement de la souris occupe beaucoup de ressources processeur. Ainsi, il est fortement conseillé de supprimer l'écoute de l'événement `MouseMove` quand vous n'en avez plus besoin. Le langage C# est dit géré car l'allocation des ressources mémoire n'est pas directement gérée par le développeur, mais par le ramasse-miettes ou *Garbage Collector*. Toutefois, pour que ce dernier libère les ressources le plus tôt possible, il est important de lui présenter des objets qui ne sont plus référencés par d'autres, lorsque vous n'en avez plus besoin. Arrêter la souscription d'un événement entre dans ce cadre.

Figure 8.3*Suppression d'un événement.*

8.2.2 Un cas concret d'utilisation

Dans cet exemple, nous allons utiliser le projet nommé `PlayerVideo_Optimisations`. Il se trouve dans l'archive `PlayerVideo_Optimisations.zip` du dossier *chap8* des exemples. L'idée de ce projet est simple : deux nouveaux états permettent de positionner le lecteur vidéo soit au centre de la fenêtre soit en bas. Pour changer de position dynamiquement, il suffit de cliquer alternativement sur le bouton interrupteur, nommé `FixedPositionBtn`. Lorsque les contrôles utilisateur ne sont pas placés au centre, il n'y a pas d'effet de disparition au bout de deux secondes. Pour réaliser cette tâche, le développeur a écouté les événements `Checked` et `Unchecked` du bouton `FixedPositionBtn`. Ainsi, il peut à la fois arrêter le compte à rebours de disparition et déplacer les contrôles utilisateurs vers le bas ou le centre de l'application. De même, il teste lors du déplacement de la souris (événement `MouseMove`) si le bouton est coché ou décoché. Ainsi, il peut également arrêter ou lancer le compte à rebours en fonction du `ToggleButton` nommé `FixePositionBtn`. Voici le code logique correspondant à ce développement :

```

public MainPage()
{
    ...
    LayoutRoot.MouseMove += new MouseEventHandler(LayoutRoot_MouseMove);
    ...
    FixePositionBtn.Checked += new RoutedEventHandler
        (FixePositionBtn_Checked);
    FixePositionBtn.Unchecked += new RoutedEventHandler
        (FixePositionBtn_Unchecked);
  }

```

```

    }

    void FixePositionBtn_Unchecked(object sender, RoutedEventArgs e)
    {
        VisualStateManager.GoToState(this, "Center", true);
        //la position est centrée car le bouton est décoché,
        //on cache donc automatiquement les contrôles utilisateur
        //au bout de deux secondes
        Dt.Start();
    }

    void FixePositionBtn_Checked(object sender, RoutedEventArgs e)
    {
        VisualStateManager.GoToState(this, "Bottom", true);
        //le bouton fixe position est coché,
        //on ne fait donc pas disparaître le lecteur
        //en stoppant le compte à rebours de disparition
        Dt.Stop();
    }
    ...
    //lorsque la souris bouge
    void LayoutRoot_MouseMove(object sender, MouseEventArgs e)
    {
        //on teste la position centrée ou basse des contrôles
        //utilisateur, si la position est basse on arrête
        //le compte à rebours sinon on le relance
        if ((bool)FixePositionBtn.IsChecked) Dt.Stop();
        else Dt.Start();

        //on utilise le gestionnaire d'état visuel
        //afin d'afficher l'état VisiblePlayer
        VisualStateManager.GoToState(this, "VisiblePlayer", true);
    }
}

```

Ce code présente plusieurs problématiques. Tout d'abord, la logique concernant l'arrêt du compte à rebours de disparition est répartie dans trois écouteurs différents. La logique de la méthode `LayoutRoot_MouseMove` ne devrait pas tester l'état du bouton `FixePositionBtn` en dur. Après tout, les instructions ont toutes le même rôle : relancer le compte à rebours de disparition chaque fois que la souris bouge, et effectuer une transition vers l'état `VisiblePlayer` si besoin. Cela empêche le fondu des contrôles lorsque l'utilisateur a une quelconque activité.

La seconde problématique concerne les performances. Le fait que l'événement `MouseMove` de `LayoutRoot` continue d'être écouté lorsque nous positionnons le lecteur en bas de la fenêtre consomme inutilement des ressources processeur et mémoire. Comme nous ne souhaitons pas d'effet de disparition dans l'état `Bottom`, l'écoute de `MouseMove` qui gère cet effet devient superflue sur cet état. Autant supprimer cette écoute lorsque le bouton `FixePositionBtn` est coché, puis se réabonner à l'événement uniquement lorsque le bouton est décoché (indiquant l'état centré). La désinscription et la souscription de l'événement `MouseMove` au bon moment génèrent un code plus propre et moins gourmand en ressources :

```

public MainPage()
{
    ...
    LayoutRoot.MouseMove += new MouseEventHandler(LayoutRoot_MouseMove);
    ...
    FixePositionBtn.Checked += new RoutedEventArgsHandler(
        FixePositionBtn_Checked);
}

```

```

        FixePositionBtn.Unchecked += new RoutedEventHandler
                                   (FixePositionBtn_Unchecked);
    }

    void FixePositionBtn_Unchecked(object sender, RoutedEventArgs e)
    {
        VisualStateManager.GoToState(this, "Center", true);
        //la position est centrée car le bouton est décoché,
        //on cache donc automatiquement les contrôles utilisateur
        //au bout de deux secondes
        Dt.Start();
        //On écoute l'événement MouseMove que lorsque c'est nécessaire
        LayoutRoot.MouseMove += LayoutRoot_MouseMove;
    }

    void FixePositionBtn_Checked(object sender, RoutedEventArgs e)
    {
        VisualStateManager.GoToState(this, "Bottom", true);
        //le bouton fixe position est coché,
        //on ne fait donc pas disparaître le lecteur
        //en stoppant le compte à rebours de disparition
        Dt.Stop();
        //On supprime la souscription de l'écouteur lorsqu'il n'est pas
        //nécessaire de faire disparaître les contrôles
        LayoutRoot.MouseMove -= LayoutRoot_MouseMove;
    }

    ...
    //lorsque la souris bouge
    void LayoutRoot_MouseMove(object sender, MouseEventArgs e)
    {
        //Il n'y a plus besoin de tester l'état du bouton FixePositionBtn
        //ici car l'écoute de l'événement MouseMove est désactivée avant

        //on utilise le gestionnaire d'états visuels
        //afin d'afficher l'état VisiblePlayer
        VisualStateManager.GoToState(this, "VisiblePlayer", true);
        Dt.Start();
    }
}

```

Testez en compilant votre projet. Pour mettre en valeur l'optimisation apportée par ce code, vous pouvez utiliser le mode "debug" de Visual Studio. Vous pourrez ainsi écrire un message en fenêtre de sortie chaque fois que la souris se déplace :

```

Using System.Diagnostics;

Int nMessage = 0;

//lorsque la souris bouge
void LayoutRoot_MouseMove(object sender, MouseEventArgs e)
{
    //Il n'y a plus besoin de tester l'état du bouton FixePositionBtn ici
    //car l'écoute de l'événement MouseMove est désactivée avant
    //On écrit un message lorsque la souris bouge
    Debug.WriteLine("la souris bouge pour la {0}ème fois", ++nMessage);

    //on utilise le gestionnaire d'état visuel
    //afin d'afficher l'état VisiblePlayer
}

```

```

        VisualStateManager.GoToState(this, "VisiblePlayer", true);
        Dt.Start();
    }

```

Vous constaterez qu'en position basse, le lecteur ne consomme plus du tout de ressources. L'événement `MouseMove` n'est plus capté par l'écouteur `LayoutRoot_MouseMove`. Les messages en fenêtre de sortie ne sont donc plus incrémentés. Nous pouvons encore améliorer notre code et optimiser notre application grâce au couplage faible.

8.3 Le couplage faible en pratique

Comme nous l'avons vu précédemment, le couplage faible est avant tout le résultat d'une bonne pratique du développement orienté objet. C'est une manière de créer des liens faibles entre les objets : la modification et la suppression d'objets collaborent de concert et n'impactent pas les objectifs de fonctionnement propres à chacun.

8.3.1 Principe

Pour le modèle événementiel, le couplage faible est réalisé grâce aux deux paramètres récupérés par les méthodes d'écoute. Le premier argument représente une référence au diffuseur de l'événement, il est de type `object` car on ne peut savoir à l'avance quel est le type de l'objet diffuseur. Ceci pour la bonne raison que des objets de types différents peuvent diffuser le même événement. Par exemple, l'événement `Click` peut être diffusé par les composants héritant de la classe `ButtonBase`, soit `RadioButton`, `CheckBox`, `ToggleButton` ou `Button`. L'événement `Loaded` est diffusable, quant à lui, par toutes les classes héritant de `UIElement`. Le fait de typer `object` nous permet donc de nous affranchir du type diffuseur puisque tous les types ont pour origine `object`. Le deuxième argument est nommé objet événementiel. Il donne des informations complémentaires sur l'événement diffusé comme la position à l'instant précis du clic de la souris. Grâce à ces deux paramètres, il n'est nul besoin de référencer l'objet diffuseur en dur. Dans le code ci-dessous, vous trouverez deux manières de réaliser la même tâche. Voici la mauvaise manière de récupérer la valeur d'un `Slider` dont la valeur vient de changer :

```

//On diffuse l'événement ValueChanged chaque fois que l'utilisateur
//modifie la position du curseur
MonSlider.ValueChanged += new RoutedPropertyChangedEventHandler<double>
    (MonSlider_ValueChanged);

//la méthode d'écoute se déclenche chaque fois que l'utilisateur
//modifie la position du curseur
void MonSlider_ValueChanged(object sender, RoutedPropertyChangedEventArgs
    <double> e)
{
    //lorsque c'est le cas on trace en fenêtre
    //de sortie la nouvelle valeur
    Debug.WriteLine("nouvelle valeur de MonCurseur : "+MonSlider.Value);
}

```

Ce n'est pas la bonne manière de procéder car `MonSlider` est référencé fortement dans la méthode d'écoute. Voici une autre manière de procéder qui présente l'avantage de ne pas faire directement référence au `Slider` :


```

void MonSlider_ValueChanged(object sender, RoutedEventArgs
                           <double> e)
{
    if ( (sender as Slider)!=null) Debug.WriteLine("nouvelle valeur de
                                                MonCurseur : "+ (sender as Slider).Value);
}

```

Avec cette méthode, le développeur utilise l'argument nommé `sender` faisant référence à l'objet diffuseur. Comme `sender` est de type `object`, il nous faut transformer son type grâce au mot-clé `as`. Ce mot-clé permet de renvoyer `null` si la conversion n'est pas possible. Avec une conversion de type explicite notée `(Slider)sender`, nous serions susceptible d'avoir une erreur levée à l'exécution et nous serions obligés d'utiliser une instruction `try / catch` consommant plus de ressources en cas d'exception levée. Une seconde manière de procéder consiste à utiliser l'objet événementiel pour récupérer la nouvelle valeur ainsi que l'ancienne :

```

void MonSlider_ValueChanged(object sender, RoutedEventArgs
                           <double> e)
{
    Debug.WriteLine("ancienne valeur de MonCurseur : "+e.OldValue);
    Debug.WriteLine("nouvelle valeur de MonCurseur : "+e.NewValue);
}

```

Ces deux méthodes ne font pas référence au `Slider` directement. Ainsi, la méthode écouteur `MonSlider_ValueChanged` pourrait souscrire à l'événement `ValueChanged` diffusé par plusieurs objets `Slider` :

```

//On crée plusieurs contrôles Slider
Slider MonSlider1 = new Slider(){Name="Slider1"};
Slider MonSlider2 = new Slider(){Name="Slider2"};
Slider MonSlider3 = new Slider(){Name="Slider3"};
...
//On définit le même écouteur pour chacun d'eux
MonSlider1.ValueChanged += new RoutedEventArgsEventHandler<double>
                           (MonSlider_ValueChanged);

MonSlider2.ValueChanged += new
RoutedEventArgsEventHandler<double>(MonSlider_ValueChanged);

MonSlider3.ValueChanged += new RoutedEventArgsEventHandler<double>
                           (MonSlider_ValueChanged);

...
void MonSlider_ValueChanged(object sender, RoutedEventArgs
                           <double> e)
{
    Debug.WriteLine("ancienne valeur de "+(sender as Slider).Name+" :
                                                "+e.OldValue);
    Debug.WriteLine("nouvelle valeur de "+(sender as Slider).Name+" :
                                                "+e.NewValue);
}

```

Dans le code ci-dessus, vous récupérez le nom du `Slider` dont la valeur a été modifiée par l'utilisateur et cela de manière complètement dynamique. Bien sûr, l'intérêt d'un tel code est relatif à la tâche de chaque `Slider`. Si ceux-ci font sensiblement la même chose, le couplage faible apporte un réel confort de conception et factorise votre code.

8.3.2 Simplifier le code du lecteur vidéo

Nous allons appliquer ce concept à notre lecteur vidéo. Vous le trouverez dans les exemples du livre : *chap8/PlayerVideo_Couplage.zip*. Ouvrez le projet au sein de Blend et de Visual Studio. Notre code est pour l'instant redondant, peu optimisé et inélégant. En voici un exemple criant : si vous voyez un jour ce type de code, c'est qu'il y a une erreur de conception :

```
//Gère les transitions vers les états de couleurs ColorStates Group
ForwardBtn.MouseEnter += new MouseEventHandler(ForwardBtn_MouseEnter);

BackwardBtn.MouseEnter += new MouseEventHandler(BackwardBtn_MouseEnter);

VolumeDownBtn.MouseEnter += new MouseEventHandler(VolumeDownBtn_MouseEnter);

VolumeUpBtn.MouseEnter += new MouseEventHandler(VolumeUpBtn_MouseEnter);

PlayPauseBtn.MouseEnter += new MouseEventHandler(PlayPauseBtn_MouseEnter);

ControlsRingArea.MouseLeave += new MouseEventHandler
    (ControlsRingArea_MouseLeave);

...
void ControlsRingArea_MouseLeave (object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Normal", true);
}

void VolumeUpBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Purple", true);
}

void VolumeDownBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Cyan", true);
}

void BackwardBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Orange", true);
}

void ForwardBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Green", true);
}

void PlayPauseBtn_MouseEnter(object sender, MouseEventArgs e)
{
    VisualStateManager.GoToState(this, "Black", true);
}
```

Comme vous le constatez, chaque écouteur fait exactement la même chose. Lorsqu'un objet est survolé, les écouteurs exécutent une transition vers un état de couleur spécifique *via* le gestionnaire d'états visuels. Un seul écouteur pourrait réaliser tout ce travail simplement. Si les boutons survolés contenaient chacun un indice de l'état de couleur ciblé, il deviendrait simple de référencer le diffuseur et de pointer vers le bon état de couleur. Nous pourrions renommer chaque bouton d'une autre manière pour faire référence à l'état ciblé.

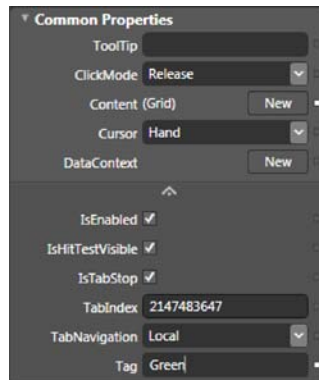
Toutefois, ce n'est pas une bonne pratique car le nom d'un objet indique avant tout sa fonctionnalité et non la forme ou la transition à laquelle il fait référence. Le mieux est d'utiliser la propriété Tag (de type Object) propre à tout objet FrameworkElement et d'y stocker le nom de l'état de couleur ciblé.

Dans Blend, sélectionnez chaque bouton les uns après les autres et entrez le nom de l'état de couleur ciblé pour chacun. Par exemple, pour le bouton ForwardBtn, déployez complètement le panneau Common Properties, vous pouvez entrer la chaîne de caractères Green dans le champ de saisie Tag (voir Figure 8.4).

N'oubliez pas de modifier également la propriété Tag de l'objet ControlsRingArea qui permettra de revenir à l'état visuel Normal.

Figure 8.4

Configuration de la propriété Tag.



Une fois que vous avez finalisé cette partie, vous pouvez simplifier le code existant en ne définissant qu'une seule méthode d'écoute :

```
//Gère les transitions vers les états de couleurs ColorStates Group
ForwardBtn.MouseEnter += new MouseEventHandler(GotoColorState);

BackwardBtn.MouseEnter += new MouseEventHandler(GotoColorState);

VolumeDownBtn.MouseEnter += new MouseEventHandler(GotoColorState);

VolumeUpBtn.MouseEnter += new MouseEventHandler(GotoColorState);

PlayPauseBtn.MouseEnter += new MouseEventHandler(GotoColorState);

PlayPauseBtn.MouseEnter += new MouseEventHandler(GotoColorState);

ControlsRingArea.MouseLeave += new MouseEventHandler(GotoColorState);
...
void GotoColorState(object sender, MouseEventArgs e)
{
    //on vérifie que l'objet cliqué est de type ButtonBase
    if (sender as FrameworkElement != null)
    {
        var TargetedState = (sender as FrameworkElement).Tag;
        VisualStateManager.GoToState(this, TargetedState.ToString(), true);
    }
}
```

Transtyper un objet vers `FrameworkElement` permet de récupérer la propriété `Tag` pour n'importe quel type de contrôles Silverlight. Comme l'événement est diffusé par trois types différents d'objets (`Canvas`, `Button`, `ToggleButton`), utiliser `FrameworkElement` – dont ces objets héritent – est pratique.

Dans le code précédent, chaque événement est écouté par la même méthode nommée `Goto-ColorState`. Grâce au couplage faible entretenu entre l'écouteur et le diffuseur, nous sommes capables de récupérer dynamiquement la valeur de la propriété `Tag`. Celle-ci étant typée `Object`, il nous faut utiliser la méthode `ToString` pour récupérer la valeur sous forme de chaîne de caractères.

INFO

La propriété `Tag` est de type `Object`, elle peut donc recevoir n'importe quel type d'objet. Comme nous ne sommes pas dans un langage dynamique, tel que JavaScript ou ActionScript, il n'est pas possible d'ajouter dynamiquement des propriétés aux objets d'affichage du framework. La propriété `Tag` permet de contourner cette limitation tout en centralisant le contenu que vous souhaitez ajouter aux objets. Selon les écoles, et notamment pour les adeptes de la POO, ajouter dynamiquement des propriétés aux objets natifs n'est pas une bonne pratique de conception. Cela génère un code souvent difficile à maintenir même si cela permet parfois plus de souplesse de développement dans certaines situations.

Vous pouvez trouver le lecteur finalisé dans les exemples du livre : *chap8/PlayerVideo_Couplage-Final.zip*.

8.3.3 Affectation dynamique d'animations

Nous allons voir comment réaffecter une animation dynamiquement en fonction de l'objet diffuseur. Ouvrez le projet nommé `AnimRebond_CouplageDynamic ()`. Trois balles sont présentes sur `LayoutRoot`. La première est ciblée par un `Storyboard` nommé `AnimRebond`. Lors de l'exécution, si vous cliquez sur celle-ci, l'animation est jouée grâce au code ci-dessous :

```
public MainPage()
{
    InitializeComponent();

    MaBalle1.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle1_MouseLeftButtonUp);
}

void MaBalle1_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    AnimRebond.Begin();
}
```

Nous pourrions créer autant d'animations que de balles, mais nous allons profiter du couplage faible pour réassigner dynamiquement l'animation à la balle cliquée. Nous pouvons ainsi écrire :

```
public MainPage()
{
    InitializeComponent();

    MaBalle1.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle1_MouseLeftButtonUp);
```

```

        MaBalle2.MouseLeftButtonUp += new MouseButtonEventHandler
            (MaBalle_MouseLeftButtonUp);
        MaBalle3.MouseLeftButtonUp += new MouseButtonEventHandler
            (MaBalle_MouseLeftButtonUp);
    }

    void MaBalle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
    {
        Storyboard.SetTarget(AnimRebond, (sender as Grid));
        AnimRebond.Begin();
    }

```

Toutefois, si vous faites le test, vous remarquez que l'assignation dynamique ne fonctionne qu'une seule fois. Ceci est dû à la gestion en mémoire des instances de Storyboard par Silverlight. Les ressources de type Storyboard doivent tout d'abord être libérées avant d'être accessibles à la modification. Il suffit d'invoquer la méthode Stop sur le Storyboard, puis de lui réassigner une nouvelle cible :

```

    void MaBalle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
    {
        AnimRebond.Stop();
        Storyboard.SetTarget(AnimRebond, (sender as Grid));
        AnimRebond.Begin();
    }

```

Cette fois, vous pouvez cliquer sur chaque balle pour réaffecter la cible et jouer l'animation. L'ancienne balle animée revient à sa position de départ. Ce principe est intéressant à plus d'un titre puisque cela évite aux designers de créer autant d'animations que d'objets. Pour éviter le côté saccadé dû à l'appel de la méthode Stop, vous pouvez également affecter une animation de retour à l'ancien objet ciblé. Cela serait particulièrement utile pour un grand nombre d'objets à animer. Pour éviter un appel injustifié de la méthode Stop, vous pouvez également tester l'état actuel de l'animation :

```

    void MaBalle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
    {
        if (AnimRebond.GetCurrentState() != ClockState.Stopped)
            AnimRebond.Stop();
        Storyboard.SetTarget(AnimRebond, (sender as Grid));
        AnimRebond.Begin();
    }

```

Le principal avantage réside au niveau de la rapidité de production entre développeur et designer. La maintenance est également plus facile puisqu'il suffit au designer de modifier une seule animation pour mettre à jour l'application. Le développeur, quant à lui, ne fait référence qu'à une seule animation pour un groupe d'objet ayant une fonctionnalité similaire.

8.4 Propagation événementielle

La propagation événementielle est propre aux objets graphiques imbriqués. Cette logique événementielle ne s'applique donc qu'aux objets héritant de la classe UIElement. Cette spécificité du modèle événementiel permet d'améliorer les performances et épurer le code logique produit.

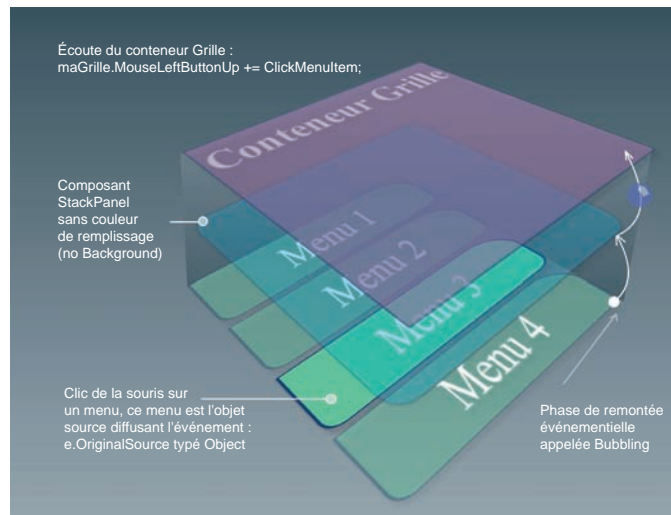
8.4.1 Principe

Le principe est simple : un événement, lorsqu'il est diffusé, parcourt l'arborescence des objets graphiques de manière verticale. Au sein de Silverlight, une seule direction existe. L'événement part de l'élément diffuseur le plus bas dans la hiérarchie et se dirige vers son parent le plus haut. Durant ce trajet, chaque parent de l'objet source peut, à son tour, diffuser cet événement. C'est la phase de remontée événementielle, ou *Bubbling*. On considère que les événements sont comme des bulles d'air remontant à la surface de l'eau. Vous pouvez écouter l'événement sur chaque parent de l'objet qui est à l'origine de l'événement. Cette notion explique le sens de nombreux comportements.

Par exemple, lorsque vous placez des formes au sein d'une grille, vous pouvez écouter l'événement `MouseLeftButtonUp` sur cette grille même si elle ne possède pas de remplissage. Pourquoi ? Simplement parce que l'événement est diffusé par l'une des formes contenues et que celui-ci remonte vers son conteneur parent qui le diffuse à son tour. En l'occurrence, le conteneur parent en question est la grille (voir Figure 8.5).

Figure 8.5

Principe de la propagation événementielle.



Comme vous le constatez à la Figure 8.5, l'avantage principal réside dans le fait qu'il suffit au développeur d'écouter un événement sur un conteneur plutôt que sur chaque objet contenu. Il est possible de connaître l'objet diffuseur originel *via* la propriété `OriginalSource` de l'objet événementiel. Attention toutefois au fait que seuls certains des événements acheminés (dits routés et typés `RoutedEvent`) utilisent la propagation événementielle. En voici une liste :

- `MouseLeftButtonUp` est un événement diffusé lors du relâché du bouton gauche de la souris.
- `MouseLeftButtonDown` est notifié lors de l'appui sur le bouton gauche de la souris.
- `MouseMove` est déclenché lorsque la souris bouge.
- `KeyUp` est diffusé lorsqu'une touche du clavier est relâchée.
- `KeyDown` est diffusé lors de l'appui sur une touche du clavier.
- `GotFocus` est un événement diffusé lorsqu'un composant bénéficie de l'intérêt utilisateur ou focus.

- `LostFocus` est diffusé lorsqu'un composant perd le focus utilisateur.
- `MouseWheel` est un événement diffusé lors de l'utilisation de la molette de la souris.
- `BindingValidationError` est un événement diffusé lorsqu'une erreur est levée lors de l'assignation d'une valeur non conforme.

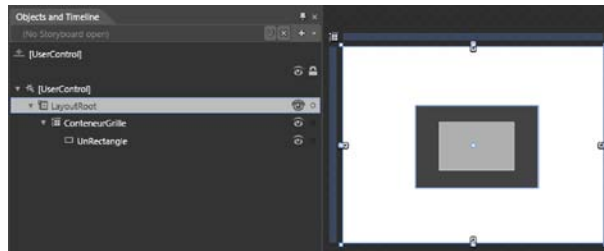
Chacun des événements ci-dessus est une interaction fondamentale de l'utilisateur. Ainsi, l'événement `Click` n'est qu'un événement spécifique propre aux composants orientés utilisateur, alors que `MouseLeftButtonUp` est un événement diffusable par toutes les classes dérivées de `UIElement`.

8.4.2 Un exemple simple de propagation

Nous allons mettre en pratique les notions que nous venons d'apprendre. Téléchargez le projet nommé "Propagation" depuis les exemples du livre : *chap8/Propagation.zip*. Ce projet constitue un cas très simple de propagation événementielle. Un conteneur de type `Grid` contient une forme de type `Rectangle`. Le premier est rempli d'une couleur gris foncé ; le rectangle, quant à lui, affiche une couleur de fond gris clair (voir Figure 8.6).

Figure 8.6

Visuel du projet *Propagation*.



Ajoutez un champ `TextBlock` nommé `Resultat` en haut à gauche au sein du conteneur `LayoutRoot`. Écoutez ensuite l'événement `MouseLeftButtonUp` sur chacun des deux objets (`ConteneurGrille` et `UnRectangle`) de cette manière :

```
public MainPage()
{
    InitializeComponent();
    UnRectangle.MouseLeftButtonUp += new MouseButtonEventHandler
        (MouseLeftButtonUp_generique);
    ConteneurGrille.MouseLeftButtonUp += new MouseButtonEventHandler
        (MouseLeftButtonUp_generique);
}

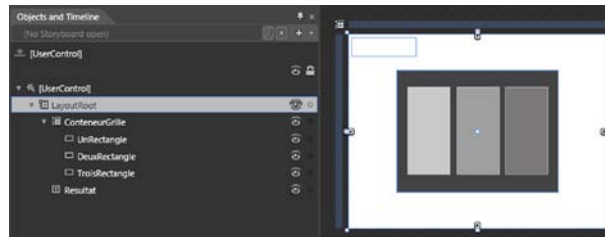
void MouseLeftButtonUp_generique(object sender, MouseButtonEventArgs e)
{
    string NomObjet = (sender as FrameworkElement).Name;
    Resultat.Text += "Diffuseur :: " + NomObjet + "\n";
}
```

Lorsque vous cliquez sur le rectangle gris clair, le champ texte vous confirme que les deux composants ont diffusé le relâché de la souris. Lorsque vous cliquez sur la grille sombre en revanche, seule celle-ci diffuse l'événement. Grâce à la propagation événementielle, vous n'êtes pas obligé d'écouter l'événement deux fois. Vous pouvez utiliser la propriété `OriginalSource` pour

déterminer quel objet a été cliqué au sein de la grille. Ajoutez deux autres rectangles nommés DeuxRectangle et TroisRectangle au sein de la grille et répartissez-les équitablement (voir Figure 8.7).

Figure 8.7

Mise en place du projet propagation.



Vous pouvez optimiser le code en ne définissant l'écouteur que sur la grille nommée Conteneur-Grille :

```
public MainPage()
{
    InitializeComponent();
    ConteneurGrille.MouseLeftButtonUp += new MouseButtonEventHandler
        (MouseLeftButtonUp_generique);
}
void MouseLeftButtonUp_generique(object sender, MouseButtonEventArgs e)
{
    string NomObjet = (sender as FrameworkElement).Name;
    Resultat.Text = " Objet écouté en dur :: " + NomObjet+"\n";
    Resultat.Text += " Objet diffuseur originel de l'événement ::
        " + (e.OriginalSource as FrameworkElement).Name + "\n";
}
```

Le code en gras permet de tracer l'objet à la source de la diffusion. Celui-ci peut être différent de l'objet écouté directement dans votre code. Compilez et testez votre application en cliquant sur chacun des rectangles puis sur ConteneurGrille. Chaque rectangle peut être responsable de la diffusion de l'événement MouseLeftButtonUp, vous pouvez récupérer la référence de l'objet diffuseur source grâce à la propriété OriginalSource de l'objet événementiel. Le code est ainsi beaucoup plus simple à maintenir.

8.4.3 Éviter la diffusion d'événements

Parfois, vous devrez empêcher les événements d'être diffusés. Reprenons l'exercice des balles, nous pourrions être tenté d'améliorer le code en gras ci-dessous grâce à la propagation événementielle :

```
public MainPage()
{
    InitializeComponent();

    MaBalle1.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle_MouseLeftButtonUp);
    MaBalle2.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle_MouseLeftButtonUp);
    MaBalle3.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle_MouseLeftButtonUp);
}
```


Il suffit pour cela d'écouter le conteneur parent commun à chaque balle. Dans ce cas, il correspond à la grille principale `LayoutRoot` :

```
public MainPage()
{
    InitializeComponent();

    LayoutRoot.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle_MouseLeftButtonUp);
}

void MaBalle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    if (AnimRebond.GetCurrentState() != ClockState.Stopped)
        AnimRebond.Stop();
    Storyboard.SetTarget(AnimRebond, (e.OriginalSource as Grid));
    AnimRebond.Begin();
}
```

Si vous testez ce code, vous vous apercevrez qu'il ne fonctionne pas pour diverses raisons. La première étant que le premier objet à l'origine de la diffusion de l'événement est forcément de type `Ellipse`. Les balles contiennent chacune trois ellipses qui diffuseront l'événement en premier. L'affectation de cible ne fonctionnera pas. La seconde raison est du même ordre. L'événement aura tendance à être diffusé par `LayoutRoot` lui-même. Lorsque vous cliquerez sur la grille `LayoutRoot`, l'animation ne sera pas jouée car elle ne contient pas le nœud `RenderTransform` lui permettant d'être ciblée par l'animation. Il nous faut donc réaliser deux tâches différentes qui ont toutes deux pour objectif d'éviter la diffusion d'un événement. Tout d'abord, il faut que les objets de type `Ellipse` ne puissent pas diffuser l'événement `MouseLeftButtonUp`. Pour cela, nous pouvons passer leur propriété `IsHitTestVisible` à `false`, nous y aurons accès en dépliant complètement l'onglet `Common Properties`, situé au sein du panneau `Properties`. Ensuite, il est nécessaire de tester le nom de la référence renvoyée par `e.OriginalSource`. Si cette référence est `LayoutRoot`, on sort de la méthode sans rien faire *via* l'instruction `return`. Voici le code définitif :

```
public MainPage()
{
    InitializeComponent();

    LayoutRoot.MouseLeftButtonUp += new MouseButtonEventHandler
        (MaBalle_MouseLeftButtonUp);
}

void MaBalle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    //on teste le nom de l'objet diffuseur
    if ((e.OriginalSource as Grid).Name == "LayoutRoot") return;
    //on teste ensuite l'état actuel de l'animation
    if (AnimRebond.GetCurrentState() != ClockState.Stopped)
        AnimRebond.Stop();
    Storyboard.SetTarget(AnimRebond, (e.OriginalSource as Grid));
    AnimRebond.Begin();
}
```

Testez et compilez, les balles sont cette fois réactives au clic gauche de la souris. Les instances d'`Ellipse` ne diffusent plus l'événement `MouseLeftButtonUp`, laissant ce rôle aux grilles `MaBalle1`, `MaBalle2` et `MaBalle3`. Gardez cependant à l'esprit que cela est possible parce que `MaBalle1`, `MaBalle2` et `MaBalle3` possèdent un fond transparent. Leur propriété `Fill` possède donc

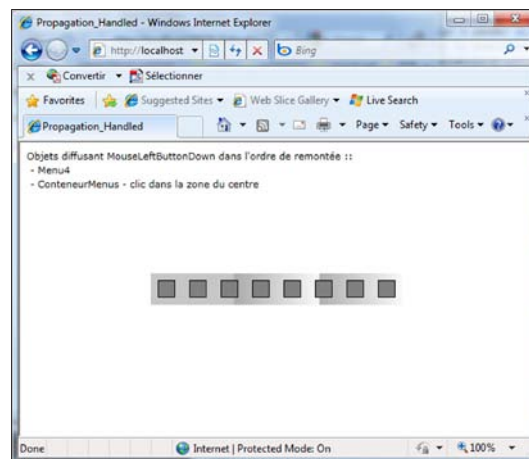
une valeur hexadécimale de type `0x00xxxxxx`. Une grille qui ne possède pas de remplissage ne peut diffuser les événements que grâce aux objets qu'elle contient. Comme les ellipses au sein des balles ne sont plus cliquables, il est obligatoire que les grilles `MaBalle1`, `MaBalle2` et `MaBalle3` possèdent un fond. Toutefois, cette solution n'est pas entièrement satisfaisante dans notre cas. La surface cliquable par l'utilisateur correspond en fait aux dimensions rectangulaires de chaque grille. L'utilisateur pourra donc cliquer dans l'espace invisible situé entre la surface de la balle et la surface restante de la grille. Nous allons maintenant étudier une manière d'empêcher la diffusion d'événements propagés.

8.4.4 Arrêter la propagation événementielle

Stopper la propagation événementielle est pratique dans le cas où vous ne souhaitez pas qu'un objet conteneur diffuse un événement déjà écouté par ses enfants. Par exemple, vous pourriez avoir besoin d'écouter le même événement sur un conteneur et sur ses enfants tout en invoquant deux méthodes d'écoutes différentes. L'une concernerait le conteneur, l'autre se déclencherait lorsque l'un des enfants diffuse l'événement. Vous pourriez décider que l'une ou l'autre méthode d'écoute se déclenche, mais pas les deux en même temps. C'est exactement le cas dans le projet que nous allons voir maintenant. Ouvrez le projet nommé `Propagation_Handled` (*chap8/Propagation_Handled.zip*). Compilez et testez l'application. En haut à gauche du conteneur principal, vous apercevez un composant `TextBlock` nommé `TestTxt`. Il va nous permettre d'afficher toutes les méthodes d'écoute déclenchées dans l'ordre de diffusion. Au centre de `LayoutRoot` se trouve un conteneur `StackPanel`. Il contient plusieurs rectangles gris qui représentent chacun un voyant (voir Figure 8.8).

Figure 8.8

Visuel du projet
`Propagation_Handled`.



Au sein de ce projet, deux méthodes d'écoute sont déclenchées pour le même événement `MouseLeftButtonDown` diffusé :

- La première méthode, `ConteneurMenus_MouseLeftButtonDown`, est diffusée par le `StackPanel`. Ce dernier possède un remplissage sous forme de dégradé linéaire représentant trois zones horizontales. Lorsque l'utilisateur clique sur ce conteneur, nous testons l'endroit où l'utilisateur a cliqué (événement `MouseLeftButtonDown`), puis nous repositionnons le conteneur dans l'application en fonction de la zone de clic. Cette tâche est réalisée grâce au gestion-

naire d'agencement fluide vu au Chapitre 7. Par exemple, si l'utilisateur clique dans la partie gauche du `StackPanel`, il s'alignera à gauche de l'application. Voici le code permettant de gérer ce comportement :

```
public MainPage()
{
    // Required to initialize variables
    InitializeComponent();

    ConteneurMenus.MouseLeftButtonDown +=new MouseButtonEventHandler
        (ConteneurMenus_MouseLeftButtonDown);
    ...
}

private void ConteneurMenus_MouseLeftButtonDown(object sender,
        MouseButtonEventArgs e)
{
    StackPanel MonConteneur = sender as StackPanel;

    //À chaque fois que l'on clique sur le conteneur
    //on passe la couleur d'arrière-plan de tous ses enfants en gris
    foreach (Rectangle rec in MonConteneur.Children)
    {
        rec.Fill = new SolidColorBrush (Colors.Gray);
    }

    //on teste l'endroit qui a été cliqué
    //et qui est relatif aux conteneurs
    double XClic = e.GetPosition(MonConteneur).X;
    string pos="";

    if ( XClic >= MonConteneur.ActualWidth*2/3)
    {
        VisualStateManager.GoToState(this,"RightPosition",true);
        pos = " - clic dans la zone droite";
    }
    else if (XClic < MonConteneur.ActualWidth*2/3 && XClic >=
        MonConteneur.ActualWidth/3)
    {
        VisualStateManager.GoToState(this,"CenterPosition",true);
        pos = " - clic dans la zone du centre";
    }
    else if (XClic < MonConteneur.ActualWidth/3)
    {
        VisualStateManager.GoToState(this,"LeftPosition",true);
        pos = " - clic dans la zone gauche";
    }

    //On finit en incrémentant le champ TextBlock
    //afin d'avertir de la diffusion de l'événement
    TestTxt.Text += "\n - "+MonConteneur.Name + pos;
}
```

- La seconde méthode déclenchée est commune à tous les objets de type `Rectangle` qui sont contenus au sein du `StackPanel` (soit `Menu1`, `Menu2`, etc.). L'objectif est simple : nous souhaitons leur donner un rôle d'indicateur visuel en changeant leur couleur de remplissage par du blanc. Cela est réalisé lors de chaque clic de la souris sur l'un des rectangles (`MouseLeftButtonDown`). Gardez également à l'esprit que lorsque l'utilisateur clique sur leur conteneur `StackPanel`, leur couleur est réinitialisée. Cette fonctionnalité est gérée par le code en

gras ci-dessus dans la méthode `ConteneurMenus_MouseLeftButtonDown`. Voici l'écouteur déclenché lorsque les menus diffusent l'événement `MouseLeftButtonDown` :

```
public MainPage()
{
    InitializeComponent();

    ConteneurMenus.MouseLeftButtonDown += ...

    //À chaque fois que l'on trouve un enfant au sein du conteneur on
    //souscrit la méthode d'écoute Menu_MouseLeftButtonDown
    foreach (UIElement e in ConteneurMenus.Children)
    {
        e.MouseLeftButtonDown += new MouseButtonEventHandler
            (Menu_MouseLeftButtonDown);
    }
}

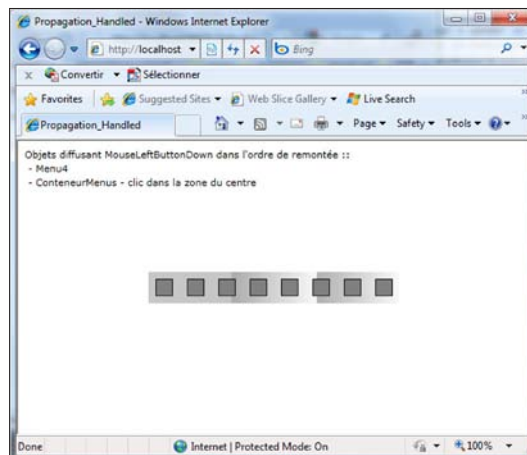
private void Menu_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Rectangle MenuClic = (sender as Rectangle);
    MenuClic.Fill = new SolidColorBrush(Colors.White);

    TestTxt.Text = "Objets diffusant MouseLeftButtonDown dans l'ordre de
        remontée :: ";
    TestTxt.Text += "\n - " + (sender as FrameworkElement).Name;
}
```

Si vous compilez et cliquez sur chaque menu, le résultat attendu par ce code ne fonctionne pas. La couleur d'arrière plan des rectangles ne devient pas blanche. C'est en fait très logique. La méthode appelée par le conteneur est déclenchée après celle exécutée par les menus. La couleur de chaque menu est réellement modifiée, mais ensuite la méthode `ConteneurMenus_MouseLeftButtonDown` la réinitialise. Le champ texte en haut à gauche le prouve, les rectangles diffusent l'événement avant le conteneur (voir Figure 8.9).

Figure 8.9

Ordre de diffusion mis en valeur par le champ TestTxt.



Il est tout à fait possible d'écrire ce code sans connaître le concept de propagation événementielle. Il devient alors difficile de diagnostiquer certains comportements comme celui que nous avons rencontré à l'instant. Pour remédier à ce problème, nous devons arrêter la propagation de l'évé-

nement après que les enfants du conteneur l'aient diffusé. La propriété `Handled` de l'objet événementiel, lorsqu'elle est passée à `true`, permet d'attraper l'événement et l'empêche de remonter au niveau supérieur pour être diffusé par le parent.

Voici le code mis à jour pour la méthode d'écoute de chaque rectangle :

```
private void Menu_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Rectangle MenuClic = (sender as Rectangle);
    MenuClic.Fill = new SolidColorBrush(Colors.White);

    TestTxt.Text = "Objets diffusant MouseLeftButtonDown
                  dans l'ordre de remontée :: ";

    TestTxt.Text += "\n - " + (sender as FrameworkElement).Name;

    //On stoppe la propagation
    e.Handled = true;
}
```

Cette fois-ci, tous les rectangles changent de couleur. Si vous cliquez directement sur l'arrière-plan du conteneur, il se repositionne en fonction de la zone cliquée, et réinitialise les couleurs de chaque enfant en gris. Vous pouvez ainsi définir des interactions à la fois différentes et complémentaires pour un conteneur et ses enfants à partir d'un unique événement.

8.4.5 Glisser-déposer et capture des événements souris

Les opérations de glisser-déposer sont aujourd'hui communes et n'étonnent personne au sein des interfaces riches. Ce n'est pas un effet de mode, puisque cela existe depuis les premières interfaces graphiques. C'est un type d'interaction largement adopté et offrant une ergonomie performante et accentuant l'impression de liberté d'action. Ce type d'interaction, au sein de Silverlight, n'est pas si simple à réaliser du fait de la nature même de l'événement `MouseMove`. Cet événement est assez particulier : il subit la propagation événementielle sans pour autant permettre l'arrêt de sa propagation *via* la propriété `Handled`. L'explication en est simple : l'objet événementiel, qui est de type `MouseEventArgs`, ne possède pas cette propriété. De plus, le type de propagation est légèrement différent, l'événement n'effectue pas vraiment la phase de remontée. En réalité, chaque parent diffuse l'un après l'autre l'événement par lui-même. Il est ainsi impossible d'arrêter cette pseudo propagation.

Nous allons maintenant générer cette interaction en connaissance de cause. Créez un projet nommé `ClickAndDrag`. Vous utiliserez les transformations relatives de translation pour gérer le glisser-déposer. À cette fin, il est préférable de télécharger la bibliothèque `ProxyRenderTransform` que je mets à disposition à cette adresse : <http://proxyrd.codeplex.com/>. L'avantage des `RenderTransform` est de pouvoir déplacer un objet quelque soit le type de contrainte imposée par le conteneur parent de l'objet à déplacer. Référez le fichier dll que vous avez téléchargé en cliquant droit sur le répertoire `Reference`, puis en choisissant le menu `Add Reference`. Attention ensuite à bien faire référence à la bibliothèque au sein du code C# *via* l'instruction `using ProxyRenderTransform`. Créez un rectangle nommé `MonRectangle` sur la grille principale `LayoutRoot`. Ouvrez maintenant le fichier `MainPage.xaml.cs` au sein de `Blend` ou de `Visual Studio`.

Si vous décomposez l'interaction de glisser-déposer, vous remarquez que trois événements sont nécessaires à son fonctionnement :

- L'événement `MouseDown` sur l'objet à déplacer permettra de lier l'objet aux coordonnées de la souris.
- L'événement `MouseUp`, correspondant au relâché de la souris diffusé par l'objet, aura la charge de libérer l'objet du déplacement de la souris.
- Pour finir, `MouseMove` – se déclenchant à chaque déplacement de la souris – gèrera le déplacement de l'objet.

Attention toutefois au fait que `MouseMove` est coûteux en termes de performance. L'idéal serait de ne le diffuser que lorsque c'est nécessaire. C'est tout à fait possible de limiter sa diffusion. Dans la vie courante, vous ne dessinez sur le papier que lorsque la mine de votre crayon est appuyée et qu'elle se déplace sur la feuille mais pas lorsqu'elle est au-dessus. Cela paraît évident et c'est tant mieux car le comportement de glisser-déposer est similaire. Vous ne déplacez l'objet que lorsque vous cliquez dessus tout en laissant le bouton appuyé. Ainsi, vous n'avez pas besoin de diffuser l'événement `MouseMove` de manière permanente, mais uniquement quand le bouton gauche de la souris reste appuyé. Vous pouvez commencer par définir l'écoute des événements `MouseDown` et `MouseUp` dans le constructeur de `MainPage` :

```
public MainPage()
{
    InitializeComponent();

    MonRectangle.MouseDown += new MouseButtonEventHandler
        (MonRectangle_MouseLeftButtonDown);

    MonRectangle.MouseUp += new MouseButtonEventHandler
        (MonRectangle_MouseLeftButtonUp);
}
```

Comme vous l'avez vu plus haut, vous pouvez définir l'écoute de l'événement `MouseMove` seulement quand l'utilisateur appuie sur l'objet. Il faut également arrêter l'abonnement de l'écouteur au relâché du bouton. Ainsi vous pouvez déjà écrire les méthodes d'écoute correspondantes :

```
void MonRectangle_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    //Lorsqu'on appuie sur le rectangle
    //on écoute le déplacement de la souris
    MonRectangle.MouseMove += new MouseEventHandler(MonRectangle_MouseMove);
}

void MonRectangle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    MonRectangle.MouseMove -= new MouseEventHandler(MonRectangle_MouseMove);
}
```

Pour vous persuader de l'efficacité en termes de performance, vous pouvez tracer une variable incrémentée chaque fois que `MouseMove` est diffusée. Vous verrez que celle-ci ne s'incrémente que lorsque vous déplacez la souris au-dessus du rectangle après avoir appuyé dessus :

```
int i = 0;
void MonRectangle_MouseMove(object sender, MouseEventArgs e)
{
    //On incrémente une variable
}
```

ATTENTION

Il faut relâcher la souris au-dessus du rectangle pour arrêter la souscription à l'événement `MouseMove`. Il y a en effet une différence entre relâcher au-dessus de l'objet et relâcher à l'extérieur. Dans le premier cas, l'événement `MouseLeftButtonUp` sera bien diffusé, pas dans le second.

À ce stade, il nous reste à définir la méthode d'écoute `MonRectangle_MouseMove` et à créer l'algorithme de déplacement. Voici le code complété, notez qu'il utilise la position de la souris ainsi que les transformations relatives sans aucun besoin des marges de l'objet dans la grille. Il est beaucoup plus simple de procéder ainsi :

```
void MonRectangle_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    MonRectangle.MouseMove -= new MouseEventHandler(MonRectangle_MouseMove);
}

void MonRectangle_MouseMove(object sender, MouseEventArgs e)
{
    //On redéfinit les nouvelles transformations relatives
    //par rapport au repère d'origine
    Point p = e.GetPosition(null);
    MonRectangle.SetX(p.X - coordX);
    MonRectangle.SetY(p.Y - coordY);
}

void MonRectangle_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    //Lorsqu'on appuie sur le rectangle, on écoute le déplacement
    //de la souris
    MonRectangle.MouseMove += new MouseEventHandler(MonRectangle_MouseMove);

    //On récupère la différence entre les coordonnées de la souris
    //et les transformations relatives de l'objet.
    coordX = e.GetPosition(null).X - (double)MonRectangle.GetX();
    coordY = e.GetPosition(null).Y - (double)MonRectangle.GetY();
}
```

Testez et compilez le projet. Si vous déplacez lentement l'objet `MonRectangle`, tout fonctionne bien. Toutefois, dès que la souris va trop vite et sort des limites du rectangle, plus rien ne va. Les bogues qui apparaissent ont deux raisons similaires. Tout d'abord, `MouseMove` n'est diffusé que lorsque la souris est dans les limites imparties par le rectangle. Or, si vous allez un peu trop vite dans les déplacements de la souris, celle-ci sort des limites de l'enveloppe et l'objet n'est plus déplacé. En second lieu, si vous relâchez le bouton de la souris en dehors de l'enveloppe, encore une fois parce que vous allez un peu trop vite, l'événement `MonRectangle_MouseLeftButtonUp` n'est pas diffusé. La conséquence directe de ce comportement, c'est que vous allez souscrire une seconde fois à l'événement `MouseLeftButtonDown`, lorsque vous allez le re cliquer. Il devient alors impossible d'arrêter l'écoute de l'événement `MouseMove`.

Vous allez régler ces deux problèmes en une seule fois en capturant tous les événements liés à la souris. Ceci est réalisé grâce aux méthodes `CaptureMouse` et `ReleaseMouseCapture` propres aux objets héritant de la classe `UIElement`. Ainsi, l'événement `MouseMove` continuera d'être diffusé même lorsque la souris sortira des dimensions du rectangle. Pour la même raison, l'événement `MouseLeftButtonUp` sera diffusé même si l'utilisateur relâche la souris en dehors du rectangle. Voici le code finalisé de l'application :

```

using ...
using ProxyRenderTransform;

namespace DragAndDropObject
{
    public partial class MainPage : UserControl
    {
        double coordX;
        double coordY;

        public MainPage()
        {
            InitializeComponent();

            MonRectangle.MouseLeftButtonDown += new MouseButtonEventHandler
                (MonRectangle_MouseLeftButtonDown);

            MonRectangle.MouseLeftButtonUp += new MouseButtonEventHandler
                (MonRectangle_MouseLeftButtonUp);
        }

        void MonRectangle_MouseLeftButtonUp(object sender,
            MouseButtonEventArgs e)
        {
            //on arrête la capture des événements propres à la souris
            MonRectangle.ReleaseMouseCapture();
            MonRectangle.MouseMove -= new MouseEventHandler
                (MonRectangle_MouseMove);
        }

        void MonRectangle_MouseMove(object sender, MouseEventArgs e)
        {
            MonRectangle.SetX(e.GetPosition(null).X - coordX);
            MonRectangle.SetY(e.GetPosition(null).Y - coordY);
        }

        void MonRectangle_MouseLeftButtonDown(object sender,
            MouseButtonEventArgs e)
        {
            //MonRectangle capture tous les événements
            //diffusés par la souris
            MonRectangle.CaptureMouse();

            //Lorsqu'on appuie sur le rectangle on écoute
            //le déplacement de la souris
            MonRectangle.MouseMove += new MouseEventHandler
                (MonRectangle_MouseMove);

            //On récupère les coordonnées de la souris sur LayoutRoot
            coordX = e.GetPosition(null).X - (double)MonRectangle.GetX();
            coordY = e.GetPosition(null).Y - (double)MonRectangle.GetY();
        }
    }
}

```

Testez et compilez votre projet, vous verrez que cette fois, tout se déroule comme prévu. Vous trouverez cet exercice corrigé dans : *chap8/DragAndDropObject.zip*.

8.5 Les comportements

Le concept de comportement repose sur le pattern Décoration ou *Decorator*. Ce modèle de conception permet d'attacher des fonctionnalités ou des caractéristiques non natives aux objets de manière dynamique et non intrusive. Décorer un objet ne modifie pas son code natif, mais étend ses possibilités. Cela permet des architectures objet bien plus facile à produire et à maintenir que l'héritage. Ajouter des comportements à un objet va dans ce sens. Utiliser des comportements ou ajouter des fonctionnalités *via* un langage logique sont deux manières équivalentes d'arriver à un même résultat, l'interactivité. Pourtant ces deux méthodologies se différencient sur plusieurs points : la souplesse d'utilisation selon les cas de figure, l'optimisation des performances et la philosophie du flux de production entre designers et développeurs. Il est également possible de vouloir ajouter des capacités à un objet sans pour autant que celles-ci soient liées à une quelconque interaction utilisateur.

8.5.1 Comportements *versus* programmation événementielle

Au sein de Silverlight, le mot comportement est relié à l'interface de Blend et aux designers. Les comportements sont représentés sous forme de petites icônes que vous pouvez placer sur un objet. Utiliser des comportements se révèle efficace dans deux situations. Dans le premier cas, le designer souhaite atteindre un objectif rapidement sans coder pour produire une maquette semi-fonctionnelle. Il peut alors utiliser les comportements de navigation fournis en standard pour déclencher un Storyboard, affecter une propriété ou accéder à un état visuel. La seconde situation apparaît lorsque l'objectif à atteindre est très complexe. Par exemple, vous souhaitez ajouter des interactions physiques à vos objets comme la gravité ou les collisions, ou capturer les mouvements de la souris. C'est typiquement le genre d'interactivité difficile à mettre en place, mais qui pourrait être paramétrée par un designer. L'apport d'un développeur est alors crucial. Celui-ci peut fournir un jeu de comportements que le designer pourra librement affecter aux objets de son choix et paramétrer à loisir dans Blend.

Coder les événements est pourtant plus souple d'un autre point de vue. Le comportement est instancié dans le projet par le designer dans la plupart des cas. Le développeur pourrait donc ne pas être informé de leur existence puisqu'ils apparaissent au sein du code déclaratif. Cela peut générer des quiproquos ou des conflits d'interactivité si la communication intermétiers n'est pas correctement établie. La deuxième raison, qui pousse les développeurs à gérer eux-mêmes l'interactivité, est la centralisation du code logique et l'optimisation des performances de l'application. S'il est possible pour un graphiste d'ajouter des comportements au sein de Blend à n'importe quel élément visuel, il est en revanche impossible pour lui de le faire durant l'exécution de l'application. De même, supprimer des comportements dynamiquement n'est pas réalisable pour lui sans code. Il devra donc faire appel à un développeur pour ces tâches. La programmation événementielle reste donc utilisée dans 90 % des cas – ce qui est heureux. Il faut maintenir un équilibre constant en fonction des équipes, des compétences de chacun et surtout justifier l'utilisation des comportements soit par une phase de production rapide, soit par des besoins spécifiques.

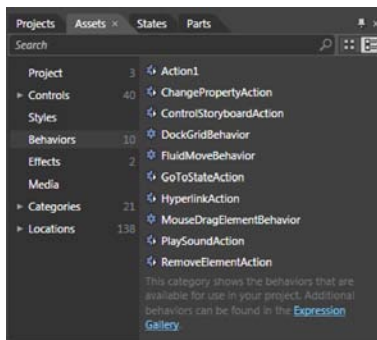
8.5.2 Les différents types de comportements

On distingue deux grandes familles de comportements : les comportements simples héritant de la classe *Behavior* et les comportements d'action déclenchée qui sont typés *TriggerAction* et

TargetedTriggerAction. Au sein de Blend, ces deux familles sont évoquées par des icônes différentes que vous pouvez voir dans le panneau Assets. Les comportements simples sont représentés par un pictogramme en forme d'engrenage et leur nom se termine par Behavior. Les comportements d'action ciblés, ou non, ont quant à eux une icône composée d'un engrenage sur lequel s'ajoute une flèche de lecture. Ils sont suffixés du mot Action (voir Figure 8.10).

Figure 8.10

Les deux familles de comportements.

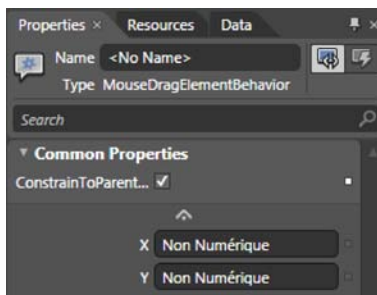


8.5.2.1 Principe des comportements simples

Le but d'un comportement simple est d'ajouter des fonctionnalités. Il contient à cette fin des gestionnaires d'événements. Toutefois, la logique événementielle permettant ce type de comportements n'est pas accessible au designer sous Blend. Celle-ci est fermée à la modification. C'est exactement le cas du comportement MouseDragElementBehavior. Ouvrez un nouveau projet dans Blend, créez un simple rectangle dans LayoutRoot, puis glissez-déposez le comportement MouseDragElementBehavior sur celui-ci. Dans le panneau des propriétés, vous pouvez voir les propriétés paramétrables du comportement simple (voir Figure 8.11).

Figure 8.11

Propriétés du comportement simple MouseDragElementBehavior.



Comme vous pouvez le voir, il n'est pas possible d'accéder à la logique événementielle, mais seulement à des propriétés qui vont influencer la manière dont le comportement s'exécutera. Dans le cas présent, vous pouvez choisir si l'objet à déplacer restera ou non dans les limites imparties par le conteneur.

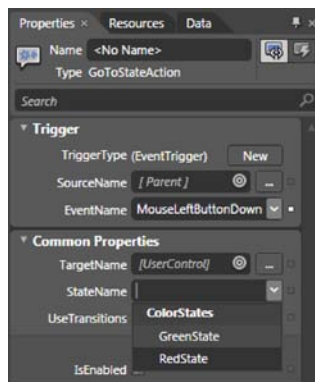
8.5.2.2 Les comportements d'action déclenchée

Les comportements simples affectent les fonctionnalités de l'objet auquel ils sont attachés, une fois pour toute à la compilation. Les comportements d'action sont plus évolués car ils vous per-

mettent de choisir l'événement qui déclenchera leur exécution ainsi que l'objet diffuseur. Cela peut-être très utile pour un designer car il n'aura pas à coder en C# la logique événementielle, mais simplement à paramétrer les propriétés du comportement. C'est exactement le cas de `GoToStateAction`. Créez deux états visuels, nommez le premier `RedState` et le second `GreenState`. Sélectionnez l'état `RedState`, puis assignez une couleur rouge pâle au remplissage de `LayoutRoot` ; procédez de même pour l'état `GreenState` en lui assignant une couleur verte. Placez ensuite deux boutons sur `LayoutRoot` et glissez le comportement `GoToStateAction` sur les deux boutons. Vous pouvez maintenant permettre à l'utilisateur de passer d'un état à un autre en paramétrant les deux comportements (voir Figure 8.12).

Figure 8.12

Propriétés du comportement d'action ciblé `GoToStateAction`.



Les comportements d'action reproduisent complètement la logique événementielle et donnent accès à son paramétrage. L'onglet Trigger (déclencheur) centralise les propriétés gérant la diffusion et permet notamment de choisir l'objet diffuseur ainsi que l'événement à écouter. Par défaut, l'objet diffuseur est l'objet auquel le comportement a été attaché. Vous pouvez ensuite décider de l'événement déclencheur. Dans ce projet, l'événement à choisir est `Click` car les boutons ne diffusent pas `MouseDown` par défaut. Vous avez la possibilité de choisir un objet cible ainsi que l'état à afficher et l'utilisation ou non d'une transition animée. Le paramétrage est au final très simple et ne demande aucune ligne de code. Le comportement `GoToStateAction` est particulier car il vous permet de choisir une instance de `Control`. Pour cette raison, il est de type `TargetedTriggerAction`.

8.5.3 Créer des comportements personnalisés

8.5.3.1 Ajouter la fonctionnalité filigrane aux champs de saisie

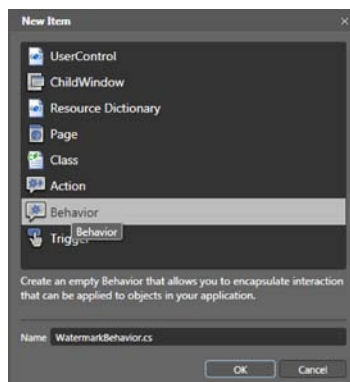
Nous allons maintenant créer un comportement simple personnalisé. Son objectif est simple, nous le déposerons sur un champ de saisie. Tant qu'il n'aura pas été rempli par l'utilisateur ou n'aura pas le focus utilisateur, il affichera une chaîne de caractères en gris clair permettant d'indiquer le type d'information à saisir. Nous allons créer, de cette manière, un champ de saisie avec filigrane, ou `WatermarkTextBox`. Créer le comportement est relativement aisé puisque Blend et Visual Studio génèrent une partie du code initial.

Créez un nouveau projet Silverlight nommé `YourBehaviors`. Au sein du panneau Projects, cliquez-droit sur le projet et choisissez le menu Add New Item... Dans la boîte de dialogue qui s'aff-

fiche, vous avez le choix entre un comportement simple (Behavior) et un comportement d'action (TiggerAction). Sélectionnez Behavior, puis nommez la classe WatermarkBehavior.cs (voir Figure 8.13).

Figure 8.13

*Création du
comportement
WatermarkBehavior.*



Comme vous le constatez, dans le code logique du comportement par défaut, la classe est paramétrée avec `DependencyObject` par défaut. Cette classe est la plus générique possible car elle est parente de `UIElement`. Cela n'est pas pratique car ce comportement n'a de sens que s'il est affecté sur des instances de `TextBox`, il nous faut donc définir le type d'objet paramétré comme étant de type `TextBox` :

```
public class WatermarkBehavior : Behavior<TextBox>
{
    public WatermarkBehavior()
    {
        ...
    }
    ...
}
```

Outre le constructeur de la classe `WatermarkBehavior`, deux méthodes ont été générées : `OnAttached` et `OnDetaching`. Bien que visible dans l'arbre visuel et logique sous Blend, un comportement n'est pas un objet visuel mais purement logique. Il ne possède pas la méthode d'initialisation `Loaded` propre aux objets d'affichage, mais `OnAttached` la remplace. Celle-ci se déclenche à la compilation ou à l'exécution lorsque le comportement est attaché à un objet. La méthode `OnDetaching` est invoquée lorsque le comportement est supprimé ou détaché de l'objet à l'exécution. Pour être fonctionnel, notre comportement doit posséder au moins une propriété permettant au designer de choisir le message à afficher lorsque le champ de saisie n'est pas rempli. Nous pouvons commencer par la créer. Voici la propriété créée en C# au sein de la classe :

```
public class WatermarkBehavior : Behavior<TextBox>
{
    private string watermark = "Veuillez saisir une information";
    public string Watermark
    {
        get { return watermark; }
        set { watermark = value; }
    }
    ...
}
```

Le scénario d'utilisation est simple : lorsque le champ de saisie sera initialisé ou perdra le focus utilisateur, un test sera réalisé sur le contenu de sa propriété `Text`. Si la valeur de `Text` est vide, alors le champ affichera la chaîne de caractères, contenue par la propriété `WatermarkText`, en grisé. Lorsque le champ texte obtiendra le focus utilisateur, si sa propriété `Text` contient une chaîne de caractères correspondant à la propriété `WatermarkText`, alors il se videra pour laisser l'utilisateur saisir les informations personnalisées nécessaires. Tout se joue donc sur les événements propres à la gestion du focus utilisateur : `LostFocus` et `GotFocus`. Toutefois, dans notre cas, le sujet (ou diffuseur de l'événement) est en fait le champ de saisie sur lequel vous glisserez le comportement. Pour récupérer une référence du futur champ de saisie attachant le comportement, vous pouvez utiliser la propriété héritée de la classe `Behavior` : `AssociatedObject`. Attention toutefois au fait que `AssociatedObject` n'est affectée de l'instance de l'objet attaché qu'à l'instant où celui-ci l'est effectivement dans l'interface de Blend. Autrement dit, au sein du constructeur de la classe, la propriété `AssociatedObject` ne contient aucune instance de `TextBox`. L'initialisation des écouteurs doit donc être centralisée dans la méthode `OnAttached`. Voici le code qui contient la gestion des événements nécessaires :

```
public class WatermarkBehavior : Behavior<TextBox>
{
    private string watermarkText = "Please enter information here";
    public string WatermarkText
    {
        get { return watermarkText; }
        set { watermarkText = value; }
    }

    public WatermarkBehavior()
    {
    }

    protected override void OnAttached()
    {
        base.OnAttached();
        //on teste par défaut le champ de saisie pour afficher le filigrane
        //si le champ est vide
        AssociatedObject.LostFocus(this.AssociatedObject, null);

        AssociatedObject.GotFocus += new RoutedEventHandler
            (AssociatedObject_GotFocus);
        AssociatedObject.LostFocus += new RoutedEventHandler
            (AssociatedObject_LostFocus);
    }

    //lorsque le champ de saisie perd le focus
    void AssociatedObject_LostFocus(object sender, RoutedEventArgs e)
    {
        //on teste le nombre de caractères contenus si rien n'est
        //rempli alors on affecte le champ texte de la valeur en filigrane
        if (AssociatedObject.Text.Length == 0)
        {
            AssociatedObject.Text = WatermarkText;
        }
    }

    //lorsque le champ de saisie obtient le focus
    void AssociatedObject_GotFocus(object sender, RoutedEventArgs e)
    {
    }
}
```

```

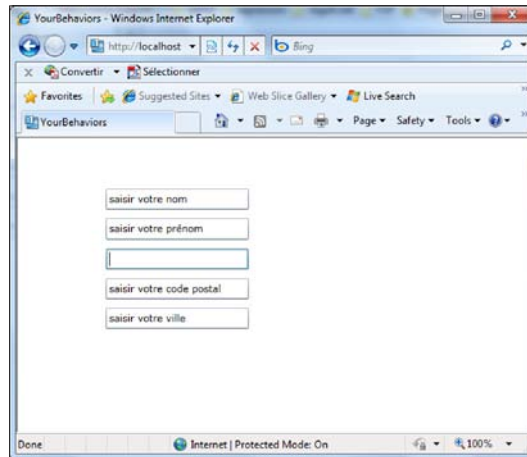
//on regarde si la valeur de la propriété Text correspond
//au filigrane. Si elle correspond, on vide le champ texte
//pour que l'utilisateur puisse saisir les informations
if (AssociatedObject.Text.Contains(WatermarkText))
{
    AssociatedObject.Text = "";
}
}
...
}

```

Compilez le projet. Pour le tester efficacement, vous devez créer plusieurs champs de saisie sur LayoutRoot – créer cinq champs serait idéal. Au sein du panneau Assets, sélectionnez l'onglet Project. Vous faites ainsi apparaître tous les composants créés pour le projet. Le comportement WatermarkBehavior est donc présent dans la liste. Déposez-le sur chacun des champs de saisie. Supprimez leur contenu texte, puis dans les paramètres de chaque comportement, définissez respectivement les chaînes de caractères : "saisir votre nom", "saisir votre prénom", "saisir votre adresse", "saisir votre code postal", "saisir votre ville". Vous pouvez également répartir ces champs au sein d'un StackPanel. Une fois cette tâche réalisée, il est plus facile de tester le projet. Au sein du navigateur, utilisez la touche Tabulation afin de passer d'un champ à l'autre (voir Figure 8.14).

Figure 8.14

Test des champs de saisie au sein du navigateur.



Pour parfaire le comportement, il faudrait griser le texte lorsque le filigrane est affiché. Rien de plus simple, il suffit d'affecter la propriété Foreground du champ de saisie. Toutefois, il est obligatoire de sauvegarder au sein d'une variable la couleur d'origine. Ceci nous permettra d'afficher le texte de manière normale lorsque les informations sont saisies correctement. Il serait également avantageux de laisser le soin au designer de choisir la couleur du filigrane afin de respecter la charte graphique. Voici le code complet du comportement :

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;

```

```

using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Interactivity;

//permet entre autres choses de gérer la répartition des
//propriétés dans Blend
using System.ComponentModel;

namespace YourBehaviors
{
    public class WatermarkBehavior : Behavior<TextBox>
    {
        private string watermarkText ;
        [Category("Watermark Properties")]
        [DefaultValue("saisir un texte")]
        public string WatermarkText
        {
            get { return watermarkText; }
            set { watermarkText = value; }
        }

        //couleur d'origine
        private SolidColorBrush originColor;

        //couleur du filigrane
        private SolidColorBrush watermarkForeground =
            new SolidColorBrush(Colors.Gray);
        public SolidColorBrush WatermarkForeground
        {
            get { return watermarkForeground; }
            set { watermarkForeground = value; }
        }

        public WatermarkBehavior(){ }

        protected override void OnAttached()
        {
            base.OnAttached();

            //on commence par sauvegarder la couleur d'origine
            originColor = (SolidColorBrush)AssociatedObject.Foreground;

            //on initialise par défaut le champ
            AssociatedObject.LostFocus(this.AssociatedObject, null);

            //on écoute les événements
            AssociatedObject.GotFocus += new RoutedEventHandler
                (AssociatedObject_GotFocus);
            AssociatedObject.LostFocus += new RoutedEventHandler
                (AssociatedObject_LostFocus);
        }

        protected override void OnDetaching()
        {
            base.OnDetaching();
            //on reste fidèle aux bonnes pratiques en supprimant
            //l'écoute des événements dont on n'a plus besoin
            AssociatedObject.GotFocus -= AssociatedObject_GotFocus;
            AssociatedObject.LostFocus -= AssociatedObject_LostFocus;
        }
    }
}

```

```

void AssociatedObject_LostFocus(object sender, RoutedEventArgs e)
{
    //on teste le nombre de caractères contenus
    //si rien n'est rempli alors on affecte le champ texte
    //de la valeur en filigrane
    if (AssociatedObject.Text.Length == 0)
    {
        AssociatedObject.Text = WatermarkText;
        AssociatedObject.Foreground = watermarkForeground;
    }
    else
    {
        AssociatedObject.Foreground = originColor;
    }
}

//on regarde si la valeur de la propriété Text correspond
//au filigrane. Si elle correspond, on vide le champ texte
//pour que l'utilisateur puisse saisir les informations
void AssociatedObject_GotFocus(object sender, RoutedEventArgs e)
{
    AssociatedObject.Foreground = originColor;
    if (AssociatedObject.Text.Contains(WatermarkText))
    {
        AssociatedObject.Text = "";
    }
}
}
}

```

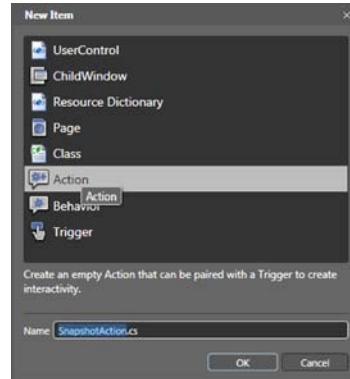
Nous pourrions encore perfectionner ce comportement en affectant, par exemple, le `ToolTip` du champ de saisie par la valeur de l'indication. Nous pourrions également prévoir tous les cas de mauvaise utilisation. Que se passe-t-il si l'objet auquel on attache le comportement n'est pas un champ de saisie de type `TextBox` ? Il faudrait pouvoir le gérer directement au sein de `Blend`. Une autre problématique est que nous utilisons la propriété `Text` du champ de saisie, ainsi si le formulaire ne vérifie que la présence ou non d'une chaîne de caractères, l'indication sera envoyée en lieu et place d'une saisie correcte de l'utilisateur. Il faudra donc également vérifier que la valeur du champ ne correspond pas à la valeur de la propriété `WatermarkText` et lever une erreur si tel est le cas. Nous allons maintenant créer un autre comportement, plus évolué par certains côtés puisqu'il rend disponible la gestion des événements.

8.5.3.2 Recopie bitmap d'un composant ou comment simuler les pinceaux visuels WPF

Depuis la version 3 de `Silverlight`, il est possible de générer une image bitmap à partir de n'importe quelle arborescence au sein de l'arbre visuel. Notre comportement aura pour vocation de créer un cliché bitmap d'un composant auquel il est attaché et de l'affecter, soit à la propriété `Fill` d'une forme (`Shape`), soit à la propriété `Background` d'une instance de la classe `Control`. Au sein du panneau `Projects`, cliquez-droit sur le projet et choisissez le menu `Add New Item...` Dans la boîte de dialogue qui s'affiche, optez pour un comportement d'action (`TriggerAction`), puis nommez la classe `SnapshotAction.cs` (voir Figure 8.15).

Figure 8.15

Création d'un comportement d'action.



Blend ouvre automatiquement le fichier `SnapshotAction.cs`, affichant ainsi le code généré. Supprimez les commentaires générés si leur présence vous gêne :

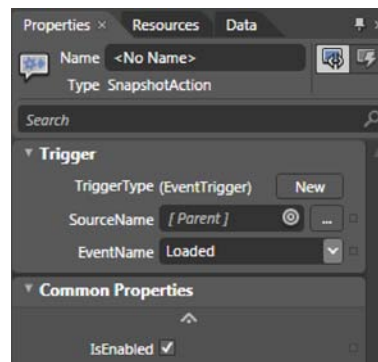
```
namespace YourBehaviors
{
    public class SnapshotAction : TriggerAction<DependencyObject>
    {
        public SnapshotAction()
        {
        }

        protected override void Invoke(object o)
        {
        }
    }
}
```

La méthode `Invoke` est déclenchée lorsque l'événement que le designer a choisi sera diffusé. Pour mieux comprendre ce concept, compilez votre projet puis créez un rectangle sur `LayoutRoot` et glissez le comportement d'action `SnapshotAction` dessus. Le panneau des propriétés met en évidence les propriétés paramétrables par défaut (voir Figure 8.16).

Figure 8.16

Propriété d'un comportement d'action standard.



Comme vous le constatez, le designer peut définir l'écoute d'un événement diffusé par le sujet de son choix. Par défaut, l'objet diffuseur de l'événement est l'objet attaché. Dans notre cas, il s'agit du rectangle. Le comportement est légèrement plus avancé puisque nous désirons également pouvoir définir une cible afin d'en créer un cliché bitmap. Nous pourrions, par exemple, vouloir cibler le conteneur `StackPanel` contenant les champs de saisie. Pour arriver à nos fins, il nous faut un comportement de type `TargetedTriggerAction`. Il suffit d'étendre cette classe au lieu de `TriggerAction`. Nous allons cibler des composants utilisateur. Vous pouvez également modifier le type paramétré comme suit :

```
namespace YourBehaviors
{
    public class SnapshotAction :
        TargetedTriggerAction<UIElement>
    {
        public SnapshotAction()
        {
        }

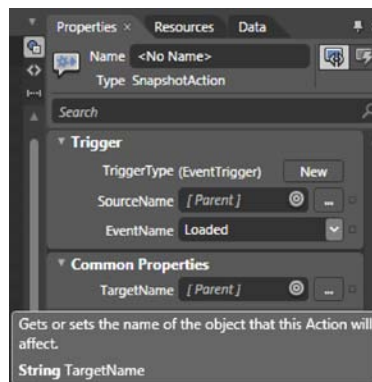
        protected override void Invoke(object o)
        {
        }
    }
}
```

Puis, compilez à nouveau le projet pour voir les changements apportés dans le panneau des propriétés du comportement (voir Figure 8.17).

La nouvelle propriété `TargetName` a été ajoutée. Il est maintenant possible de sélectionner l'objet dont nous souhaitons créer un cliché. Le code en lui-même est assez simple, il faut tout d'abord vérifier que l'espace de noms `System.Windows.Media.Imaging` est référencé.

Figure 8.17

Propriété d'un comportement d'action ciblée.



Nous allons utiliser la classe `WritableBitmap` qui permet de faire l'instantané de n'importe quel contrôle :

```
protected override void Invoke(object o)
{
    //on déclare une nouvelle image bitmap
```

```

WriteableBitmap Wb;

//on l'instancie en lui passant l'objet
//dont elle doit créer un instantané
Wb = new WriteableBitmap(this.Target, null);
//on crée un nouveau pinceau d'image
ImageBrush Ib = new ImageBrush();

Ib.Stretch = Stretch.None;
//on affecte l'image binaire de l'instantané à
//la propriété ImageSource du pinceau d'image
Ib.ImageSource = Wb;

//Ensuite on teste le type et on affecte
//la bonne propriété en fonction de ce dernier
if (AssociatedObject is Shape)
    ((Shape)AssociatedObject).Fill = Ib;

else if (AssociatedObject is Panel)
    ((Panel)AssociatedObject).Background = Ib;

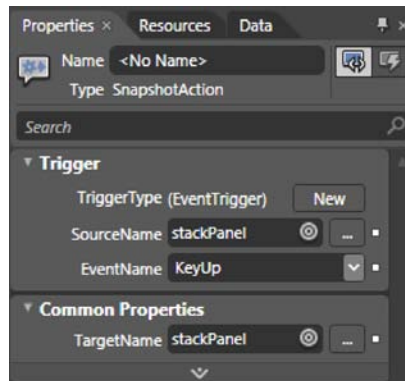
else if (AssociatedObject is Control)
    ((Control)AssociatedObject).Background = Ib;
}

```

Il n'y a rien d'autre à faire, mis à part définir les événements qui invoqueront la méthode (voir Figure 8.18).

Figure 8.18

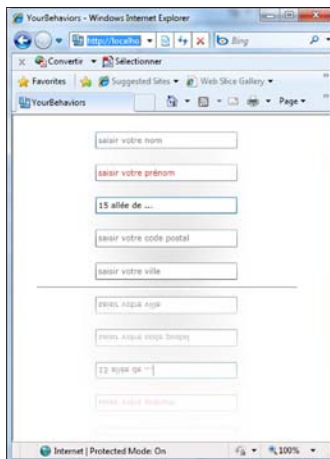
Paramétrage du comportement d'action ciblé SnapshotAction.



Voici un exemple simple du formulaire avec un effet de reflet créé à l'aide du comportement.

Figure 8.19

Effet de reflet d'un formulaire en ligne via le comportement d'action SnapshotAction.



Nous pourrions, encore une fois, améliorer grandement ce comportement de plusieurs manières. L'utilisation de conditions `if / else if` est un peu lourde et maladroite. Nous avons la possibilité de récupérer le type de l'objet associé dynamiquement et de rechercher si ce dernier possède la propriété `Fill` ou `Background`. Nous pouvons également automatiser la création du reflet grâce à une instance de `DispatcherTimer`. Elle permettrait de rafraîchir le cliché bitmap selon un laps de temps indiqué par le designer ou le développeur et plus seulement *via* les événements proposés. Nous n'irons cependant pas plus loin dans ce chapitre car nous avons abordé l'ensemble des notions indispensables permettant d'élaborer une interactivité performante. Vous trouverez ce projet dans les exemples de ce livre : *chap8/YourBehaviors.zip*.

Dans le prochain chapitre, nous allons apprendre les principes de base de la projection 3D. Nous pourrions ainsi améliorer le lecteur vidéo que nous avons conçu auparavant. Nous nous servirons de la projection 3D pour prototyper nos applications au sein de Blend avec le nouvel outil révolutionnaire qu'est SketchFlow (Chapitre 10). Il était important d'apprendre les bases du modèle événementiel avant de nous y plonger et c'est maintenant chose faite.

Les bases de la projection 3D

Dans ce chapitre, vous allez découvrir les bases de la 3D adaptées à la plateforme Silverlight. Vous aborderez les propriétés 3D d'objets et leur fonctionnement. Vous apprendrez à cibler et à manipuler ces propriétés, au sein de l'interface graphique Expression Blend, mais également dans Visual Studio *via* le code logique C#. Pour finir, vous mettrez en pratique les connaissances acquises pour réaliser deux exercices dont l'un vous permettra d'améliorer le lecteur vidéo réalisé dans le chapitre précédent. Ainsi, vous vous confronterez aux problématiques de production les plus courantes lorsqu'il s'agit de créer des interfaces en trois dimensions.

9.1 L'environnement 3D

L'image en trois dimensions, ou 3D, est apparue très tôt dans les Beaux-Arts. Les peintres Piero Della Francesca et Filippo Brunelleschi furent parmi les premiers à utiliser et à concevoir les techniques de perspective. Leurs œuvres, bien qu'assez proches de celles de leurs contemporains, se démarquèrent par l'utilisation de la perspective. Piero della Francesca diffuse, à l'époque, plusieurs ouvrages traitant directement de la géométrie dans l'espace. Ainsi, il est considéré de nos jours comme l'un des pionniers du dessin d'environnement réaliste et de la renaissance artistique. L'utilisation de la perspective et par conséquent la représentation en trois dimensions d'une scène mythologique ou religieuse apporte à la peinture une nouvelle vision. Si le point de fuite au sein d'une perspective conique représente l'infini, un nouvel acteur invisible est mis en valeur : le spectateur. Ce dernier, sans faire explicitement partie de la toile, participe à la scène peinte ou dessinée. C'est de son point de vue que celle-ci est représentée.

Ainsi, l'imagerie 3D, qu'elle soit de synthèse, de peinture ou de dessin, replace l'être humain au centre de l'action. Elle enflamme notre imagination et nous immerge directement dans l'action. Au sein des interfaces riches, elle n'est pas une fin en soi, mais un moyen d'expression au même titre qu'un autre. L'utilisation d'une interface en pure 3D se révélerait assez catastrophique d'un point de vue utilisateur. Très peu de personnes sont aujourd'hui capables de se localiser dans un environnement de ce type (les pilotes d'avions et les joueurs invétérés en font par exemple partie).

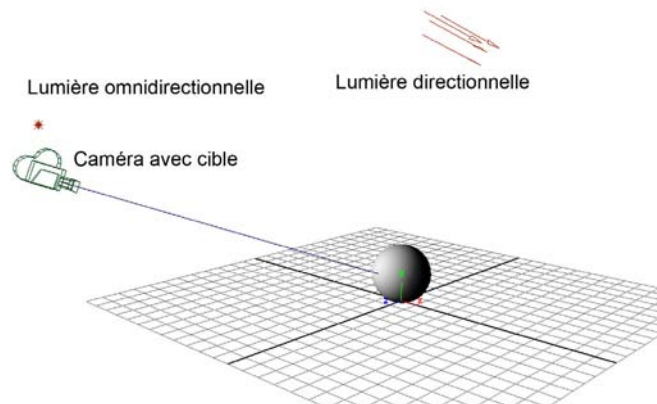
N'en abusez donc pas dans vos interfaces et faites en sorte de toujours mettre la 3D au service de l'expérience utilisateur. L'objectif final étant que ce dernier s'approprie et se plonge dans l'application. Nous allons maintenant étudier son fonctionnement et son intégration dans la plateforme Silverlight.

9.1.1 Introduction

Au sein des moteurs de rendu, on distingue deux grandes catégories. Les moteurs 3D temps réel (comme en possèdent souvent les jeux vidéo) et les moteurs d'images 3D précalculées. Silverlight étant un moteur vectoriel interactif, il se positionne d'emblée dans la première catégorie. Il vous permet donc, par exemple, de déplacer des objets en redéfinissant leur trajectoire à l'exécution. Toutefois, il ne déplacera pas réellement d'objets 3D, mais simplement des projections planaires de ces derniers. Nous allons maintenant aborder ce principe. Des logiciels comme Maya, 3D Studio Max, Blender, Lightwave ou Softimage font partie des deux mondes. D'un côté, ils génèrent des images précalculées, de l'autre, ils donnent accès à une interface d'édition élaborée, en 3D temps réel. Alors qu'un environnement en 2D permet aux graphistes de créer ou modifier les objets selon un axe horizontal et vertical, notés respectivement x et y, les environnements 3D gèrent également l'axe z représentant la profondeur. Par défaut, quatre vues sont donc proposées à l'utilisateur dans les logiciels que nous avons évoqués : la vue de dessus, de côté, de face et une perspective avec point de fuite. Chacune de ces vues permet à l'infographiste de se repérer dans l'espace pour mieux concevoir les objets et agencer l'espace tridimensionnel. Ces vues sont en réalité des caméras particulières car, mis à part la vue de perspective, elles sont isométriques, donc sans point de fuite. Une scène 3D standard possède toujours au moins une caméra, une lumière ainsi qu'un objet 3D ou 2D à mettre en scène (voir Figure 9.1).

Figure 9.1

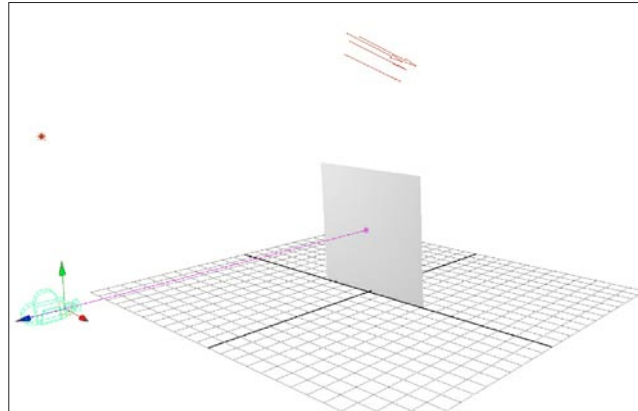
*Exemple
d'une scène en trois
dimensions.*



Avec ce type de logiciels, le designer conçoit et interagit avec des objets en vraie 3D. Cela signifie que ceux-ci possèdent une hauteur, une largeur et une profondeur. Il est toutefois possible de mettre en scène des objets en deux dimensions comme un disque ou un plan (voir Figure 9.2).

Figure 9.2

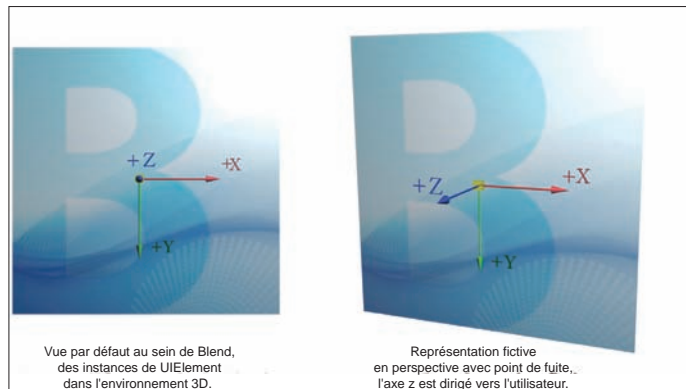
Exemple d'une scène en trois dimensions avec plan 2D.



Comme dans la réalité, les caméras 3D possèdent un angle d'ouverture ainsi qu'une longueur de focale dont les valeurs sont directement liées l'une à l'autre. Modifier la focale revient à modifier l'angle d'ouverture et inversement. Pour finir, la caméra est plus ou moins distante des objets qu'elle encadre. Au sein de Silverlight, le principe est exactement le même, à la différence près que les caméras ne sont pas des objets directement manipulables par le designer interactif, mais un point de vue abstrait, implicite et non modifiable. Nous aborderons la notion de caméra de manière plus approfondie plus loin dans ce chapitre. Comme vous le constatez aux Figures 9.1 et 9.2, l'axe des y est orienté vers le haut. Ce n'est pas le cas au sein de Silverlight dont les objectifs de production sont différents. L'axe 3D des ordonnées est orienté vers le bas, l'axe z des profondeurs est orienté vers le spectateur (voir Figure 9.3).

Figure 9.3

Orientation des axes x, y et z au sein de Silverlight et de l'interface Blend.



9.1.2 Le plan de projection

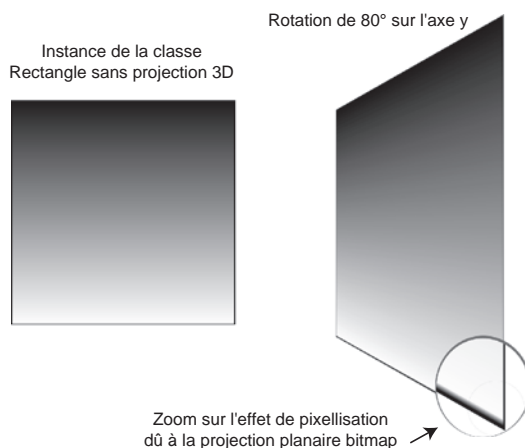
Depuis sa version 3, Silverlight permet d'afficher des objets dans un espace en trois dimensions. Toutefois, Silverlight étant à l'origine un moteur vectoriel temps réel, il n'affiche en premier lieu que des objets en deux dimensions. L'affichage des objets vectoriels 2D est calculé par le processeur de l'ordinateur. Tous les contrôles utilisateur héritant de `UIElement` ont la capacité d'évoluer dans l'espace 3D. Pour des raisons de performances d'affichage, lorsque les contrôles utilisateur

sont définis dans l'espace en trois dimensions, ils sont rendus sous forme d'images bitmap projetées sur un plan. Cela évite un recalcul constant et coûteux au processeur. Les objets restent interactifs et peuvent, si besoin, changer de visuel.

Ainsi, un bouton peut toujours afficher un état différent au survol ou au clic de la souris. Le fait de rendre les vecteurs sous forme bitmap limite grandement le nombre de calculs nécessaires à l'affichage ; les images sont placées en mémoire vive et mises à jour uniquement lorsqu'une interaction survient. Pour placer une instance de `UIElement` dans l'espace en trois dimensions de Silverlight, il suffit de lui appliquer une transformation 3D. La conversion des vecteurs en images implique une très légère dégradation de l'affichage des objets. C'est tout à fait logique et ne se remarque que lorsque vous effectuez un zoom au sein de Blend ou que vous appliquez des transformations 3D de manière extrême. Cela est dû au fait que les vecteurs sont rendus sous forme d'images dont les pixels sont définis une fois pour toute (voir Figures 9.4 et 9.5).

Figure 9.4

Effet de pixellisation suite à la rotation d'un rectangle sur l'axe y.



Comme vous le constatez, les bords sont lissés : pour éviter un effet de crénelage, le rendu des images est automatiquement lissé. Les contrôles utilisateur, quels qu'ils soient, peuvent également subir une transformation 3D (voir Figure 9.5).

Figure 9.5

Effet de pixellisation suite au déplacement d'une CheckBox sur l'axe z de 500 pixels.



Vous remarquez l'effet pixellisé généré lors du déplacement sur l'axe des profondeurs. Ceci est un effet direct de la projection bitmap. Même si les objets ont la capacité d'évoluer dans un espace en trois dimensions, ils ne possèdent pas de réelle profondeur. Pour cette raison, on utilise le terme de 2,5D lorsque l'on évoque la 3D dans Silverlight. L'espace est en 3D, mais les objets restent en 2D.

9.2 Les propriétés 3D

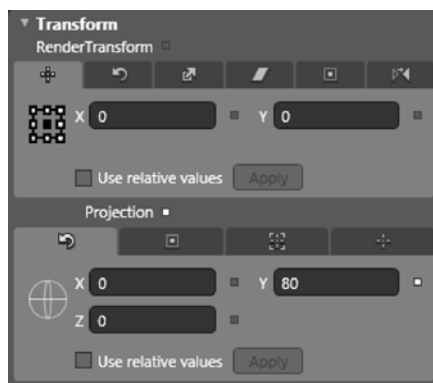
Nous allons maintenant étudier les propriétés 3D accessibles au sein de Silverlight et voir comment elles nous permettent de manipuler les objets de type `UIElement`.

9.2.1 Rotation 3D

Les rotations représentent les transformations 3D par excellence car elles mettent directement le ou les points de fuite en valeur. Créez un nouveau projet nommé "Test3D". Créez ensuite un rectangle de 200 pixels de large par 200 pixels de haut. Puis, assignez-lui un dégradé, en tant que remplissage. Vous pourriez également utiliser un composant `Image` afin de mieux visualiser les effets de perspective. Pour accéder aux projections 3D, sélectionnez le rectangle ou le contrôle `Image`, puis, dans le panneau `Properties`, dépliez l'onglet `Transform` en dessous de la partie dédiée aux transformations relatives : vous trouverez le menu `Projection` composé de quatre panneaux. Chacun d'eux représente un type de propriété 3D (voir Figure 9.6).

Figure 9.6

Menu des projections 3D.



Le premier onglet concerne les rotations. Changez la valeur de chaque propriété de rotation. Procédez par étape afin de visualiser leur effet de manière séparée (voir Figure 9.7). Vous constatez qu'un manipulateur interactif, situé à gauche des champs de saisie, vous permet également de modifier la rotation. Toutefois, cet outil, bien qu'intuitif, est au final assez peu pratique car trop imprécis.

Figure 9.7

Rotation 3D.



Rotation sur l'axe Y

Rotation sur l'axe X

Rotation sur l'axe Z

Le fait de définir des propriétés de projection indépendamment des transformations relatives est assez révélateur. Comme les `RenderTransform`, les propriétés de projections 3D n'appartiennent pas directement aux objets, mais leurs sont attachées lorsque cela est nécessaire. Un nœud est créé à cette fin, voici le code XAML y faisant référence :

```
<Rectangle HorizontalAlignment="Left" Width="200" Height="200"
            RenderTransformOrigin="0.5,0.5">
    <Rectangle.Projection>
        <PlaneProjection RotationY="60" />
    </Rectangle.Projection>
    <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="Black" Offset="0" />
            <GradientStop Color="White" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Comme nous l'avons déjà précisé, tout `UIElement` possède la propriété `Projection`, qui accepte les instances de type `PlaneProjection` ou `Matrix3DProjection`. Comme la gestion de la 3D est centralisée par cette propriété, vous pouvez dès lors considérer qu'il est possible de mélanger les transformations relatives et les projections 3D. Ces deux types de propriétés sont cousines et se ressemblent donc en de nombreux points. Le code XAML ci-dessous est donc non seulement réalisable mais réaliste en termes de production :

```
<Rectangle HorizontalAlignment="Left" Width="200" Height="200"
            RenderTransformOrigin="0.5,0.5">
    <Rectangle.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="2" ScaleY="2" />
            <SkewTransform />
            <RotateTransform />
            <TranslateTransform />
        </TransformGroup>
    </Rectangle.RenderTransform>
    <Rectangle.Projection>
        <PlaneProjection RotationY="60" />
    </Rectangle.Projection>
    ...
</Rectangle>
```

Attention, toutefois, à ne pas vous perdre car il n'est pas toujours facile de conceptualiser à l'avance le résultat visuel émanant de différents types de transformations. Choisissez les unes ou les autres en fonction de vos besoins. Lorsqu'il est possible de simuler, de manière efficace, une projection 3D en utilisant une ou plusieurs transformations relatives, privilégiez toujours ce choix car les transformations relatives sont plus simples à gérer.

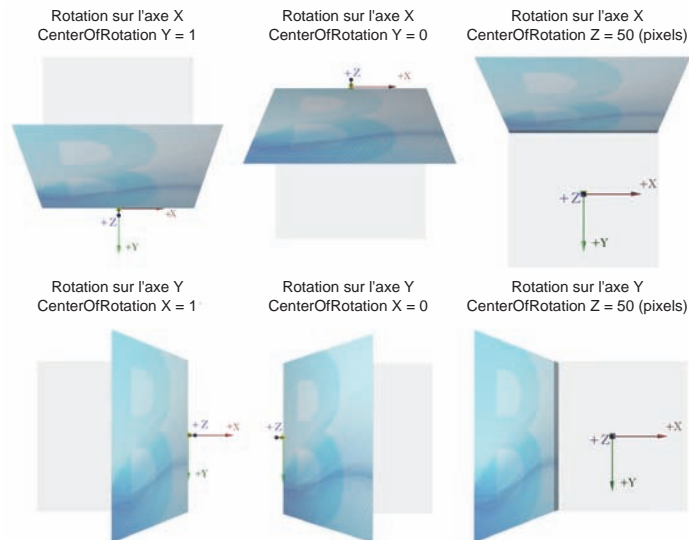
9.2.2 L'axe de rotation

La rotation 3D que vous avez testée a mis en évidence un élément important : l'axe de rotation. Sa position est déterminée par trois propriétés : `CenterOfRotationX`, `CenterOfRotationY` et `CenterOfRotationZ`. Le deuxième panneau, au sein du menu `Projection`, permet de modifier les coordonnées de cet axe afin d'avoir plus de contrôle sur les rotations obtenues (voir Figure 9.8).

Comme vous le constatez à la Figure 9.8, les valeurs admises sont relatives sur les axes x et y, il n'est donc pas facile de manipuler ces propriétés de manière intuitive. Sur l'axe x, la valeur 1 correspond à 100 % de la largeur du rectangle, sur l'axe y cette valeur correspond à 100 % de la hauteur de ce dernier. Par défaut, l'axe gère les rotations de manière symétrique car il est situé à 50 % en x et y de l'instance de UIElement, soit une valeur de 0.5. Ces valeurs n'apparaissent pas dans l'interface de Blend à moins de les modifier. Comme les objets n'ont pas d'épaisseur, la valeur de UIElement sur l'axe des profondeurs (z) est égale à 0. Ainsi, les instances d'objets graphiques subissant une rotation sur les axes x et y ne se sont pas déplacées sur l'axe des profondeurs z. De plus, sur l'axe z, la plage de valeurs admises n'est pas exprimée en pourcentage, mais en pixels, du fait que l'épaisseur est égale à zéro. Cela n'aurait, en effet, aucun sens de demander 200 % de 0 pixel d'épaisseur. Vous pouvez voir, sur la droite de la Figure 9.8, deux vues engendrées par le repositionnement de l'axe de rotation de 50 pixels vers l'avant. On obtient des résultats équivalents en déplaçant le centre sur les axes x ou y. Toutefois, utiliser l'axe z offre plus de souplesse et se révèle pratique pour manipuler par la suite les objets en 3D.

Figure 9.8

Rotations avec repositionnement de l'axe de rotation 3D.

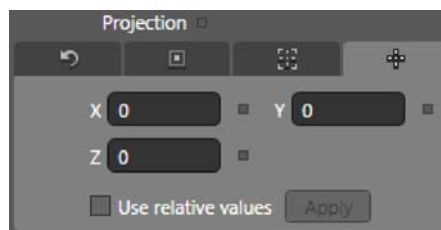


9.2.3 Les axes de translation

Nous allons maintenant nous concentrer sur les axes de translation. Au sein de Silverlight, deux types de translations cohabitent : les translations locales et globales. Lorsque vous avez effectué une rotation 3D du rectangle, vous avez également modifié la rotation de son axe. Le dernier des quatre onglets vous permet de déplacer le rectangle de manière locale (voir Figure 9.9).

Figure 9.9

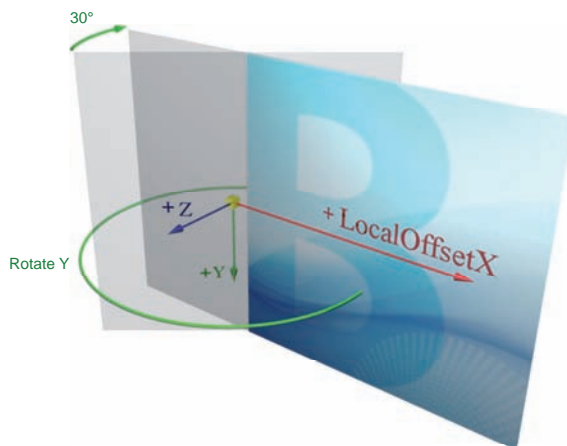
Inspecteur de propriétés des translations 3D locales.



Concrètement, le déplacement local d'un composant UIElement sera soumis à l'orientation du centre de rotation. Quel que soit l'ordre de vos opérations entre rotation et déplacement local, cela revient à faire pivoter l'objet, puis à le déplacer en suivant l'axe de rotation (voir Figure 9.10).

Figure 9.10

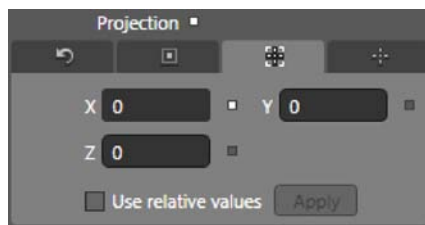
Déplacement du rectangle le long de l'axe de rotation local après une rotation effectuée sur l'axe Y.



Vous pouvez également choisir de déplacer un objet en utilisant l'axe de translation global à l'aide du troisième onglet (voir Figure 9.11).

Figure 9.11

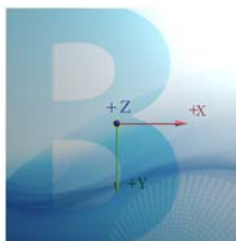
Inspecteur de propriétés des translations 3D globales.



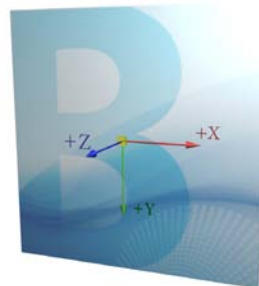
Ce repère 3D global est invisible, non modifiable et fait référence aux coordonnées propres à l'écran lui-même (Figure 9.12).

Figure 9.12

L'axe global de l'écran.



Vue par défaut au sein de Blender, des instances de UIElement dans l'environnement 3D.

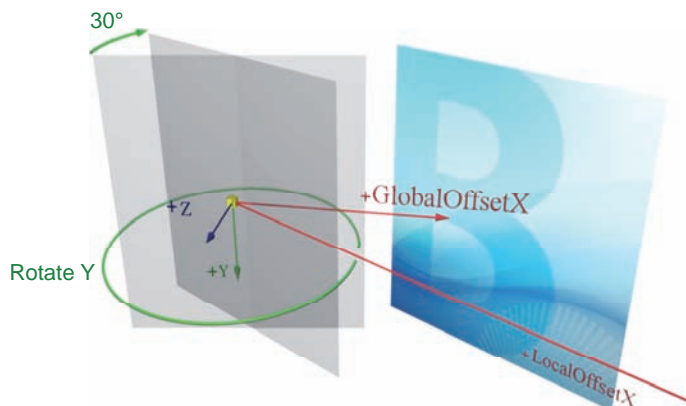


Représentation fictive en perspective avec point de fuite, l'axe z est dirigé vers l'utilisateur.

La translation globale d'instances de `UIElement` sera soumise à l'orientation figée de cet axe 3D immuable, quelle que soit la rotation de l'objet autour de son axe (voir Figure 9.13).

Figure 9.13

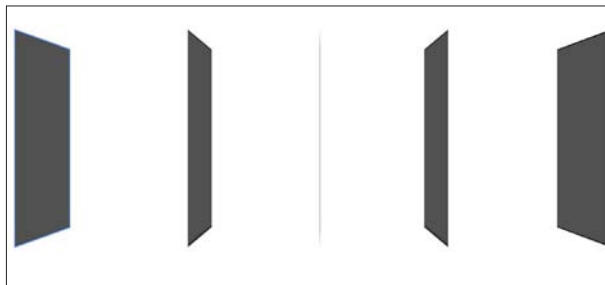
Rotations avec repositionnement de l'axe de rotation 3D.



Quel que soit le type de translations utilisées, elles seront soumises au point de fuite de la caméra. C'est un assez bon moyen de mettre en évidence la perspective et les points de fuite. Pour le constater, créez cinq carrés, puis faites une rotation de 90 degrés sur l'axe y pour chacun d'eux. Ensuite déplacez-les sur l'axe global x afin de mettre en valeur la perspective (voir Figure 9.14).

Figure 9.14

Rotation de 90° sur l'axe y, puis translation sur l'axe global x de chaque rectangle.



Il est avantageux de déplacer les rectangles en utilisant l'axe global car obtenir le même résultat avec l'axe local demande un effort de visualisation en trois dimensions. Il nous faudrait en effet déplacer l'objet sur l'axe de translation local z, ce qui peut devenir rapidement difficile à gérer et assez perturbant.

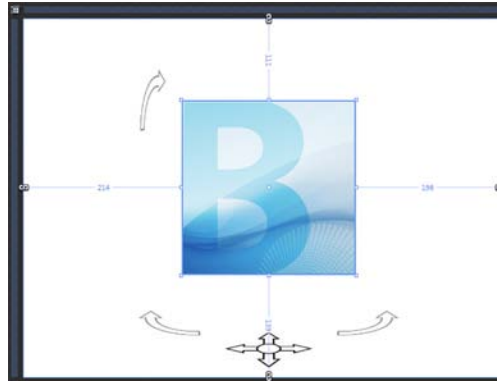
9.2.4 Accès et modification via C#

Les propriétés de projection 3D sont bien plus faciles à lire et à modifier que les transformations relatives. Toutefois, vous aurez le même type de problématique. Les propriétés de projection 3D sont attachées, donc absentes par défaut, et la propriété `Projection` de `UIElement` accepte plusieurs types dont `PlaneProjection`. Elle est nulle par défaut. Ainsi, vous devrez obligatoirement affecter une instance de `PlaneProjection` pour cibler les propriétés auxquelles vous avez accès par nature dans l'interface de Blend. La conséquence de cette architecture au sein de Blend est directe, vous ne pouvez pas cibler une propriété de `PlaneProjection`, au sein d'un `Storyboard`,

si le nœud XAML n'est pas présent en dur. Téléchargez le projet : *chap9/ProjectionPlan.zip*. Ce projet va nous permettre d'apprendre à cibler les projections *via* C#. Il contient huit boutons dont l'objectif est de contrôler le composant Image en trois dimensions (voir Figure 9.15).

Figure 9.15

*Rotation de 90°
sur l'axe y, puis
translation sur
l'axe global x de
chaque rectangle.*



Le composant Image ne possède pas de nœud élément XAML `PlaneProjection`. Le pavé de flèches directionnelles va nous permettre de déplacer l'image sur l'axe global en x et z. Les flèches à gauche et à droite joueront une animation de rotation sur y, et celle en haut à gauche, un pivotement de l'image sur x. Le mieux est de stocker l'instance de `PlaneProjection` comme membre de la classe `MainPage`. Vous pouvez ainsi cibler les propriétés de manière très simple :

```
public partial class MainPage : UserControl
{
    PlaneProjection pp = new PlaneProjection();

    public MainPage()
    {
        InitializeComponent();

        Logo.Projection = pp;

        MouseLeftButtonUp += new MouseButtonEventHandler(MainPage_
                                                                MouseLeftButtonUp);
    }

    void MainPage_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
    {
        pp.RotationY += 10;
    }
}
```

Bien que très simple, cette méthode possède un inconvénient. Si le designer, sous Blend, a déjà modifié certaines propriétés de projection, vous risquez d'écraser leur affectation. Il convient donc de tester la valeur de la propriété `Projection`. Si celle-ci est nulle, on peut lui associer une nouvelle instance ; si celle-ci possède déjà une valeur, il convient de vérifier également son type. Le code logique C#, en gras ci-dessous, est responsable du test de la propriété `Projection` :

```
public partial class MainPage : UserControl
{
    PlaneProjection pp = new PlaneProjection();
```

```

public MainPage()
{
    InitializeComponent();

    if (Logo.Projection == null)
        Logo.Projection = pp;
    else if (Logo.Projection != null && Logo.Projection is
        PlaneProjection)
        pp = (PlaneProjection)Logo.Projection;
    else
        throw new NotImplementedException("la projection est de type
            Matrix3DProjection");

    MouseLeftButtonUp += new MouseButtonEventHandler(MainPage_
        MouseLeftButtonUp);
}

void MainPage_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    pp.RotationY += 10;
}
}

```

Vous pouvez également utiliser les méthodes héritées de la classe `DependencyObject`, `SetValue` et `GetValue`. Toutefois, le code nécessaire est plus verbeux :

```

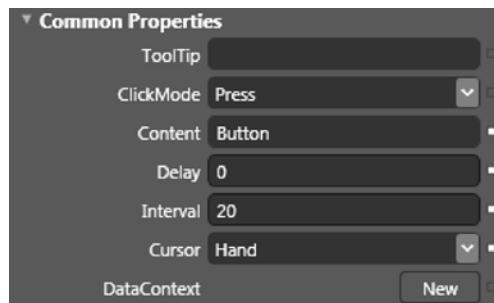
void MainPage_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    //pp.RotationY = pp.RotationY + 10;
    double ActualRYValue = (double) pp.GetValue (PlaneProjection.
        RotationYProperty );
    pp.SetValue(PlaneProjection.RotationYProperty, ActualRYValue + 10);
}

```

Nous allons animer le composant `Image` de manière très simple. Lorsque l'une des flèches sera pressée, le composant se déplacera ou pivotera. Quand vous relâcherez le bouton de la souris, l'animation de déplacement ou de rotation sera stoppée. Nous aurions pu utiliser une instance de la classe `DispatcherTimer`, mais cela aurait engendré plus de code C#. Afin de simplifier au maximum le code logique, les contrôles utilisateur sont en majorité des instances de `RepeatButton`. Seul le bouton `ResetPosition` est de type `Button`. L'avantage d'un `RepeatButton` est d'embarquer en son sein le comportement de répétition. Sélectionnez le bouton nommé `RotateY-Left`. Dans le panneau `Common Properties`, vous constatez que celui-ci possède des capacités supplémentaires (voir Figure 9.16).

Figure 9.16

Propriétés Delay et Interval de la classe RepeatButton.



La propriété `Delay` accepte une valeur entière (`Int32`) et permet de spécifier le nombre de millisecondes le temps d'attente en position appuyée avant que le comportement de répétition ne débute. Le laps de temps entre chaque répétition, lui aussi exprimé en milli-secondes, est défini grâce à la propriété `Interval` (`Int32`). Le code finalisé de l'application devient trivial :

```
namespace ProjectionCSharp
{
    public partial class MainPage : UserControl
    {
        PlaneProjection pp = new PlaneProjection();

        public MainPage()
        {
            InitializeComponent();

            if (Logo.Projection == null)
                Logo.Projection = pp;
            else if (Logo.Projection != null && Logo.Projection is
                PlaneProjection)
                pp = (PlaneProjection)Logo.Projection;
            else
                throw new NotImplementedException("la projection est de type
                    Matrix3DProjection");

            RotateXBack.Click += new RoutedEventHandler(RotateXBack_Click);
            RotateYLeft.Click += RotateYLeft_Click;
            RotateYRight.Click += RotateYRight_Click;
            GoForward.Click += GoFrontward_Click;
            GoBackward.Click += GoBackward_Click;
            GoLeft.Click += GoLeft_Click;
            GoRight.Click += GoRight_Click;
            ResetPosition.Click += ResetPosition_Click;
        }

        void ResetPosition_Click(object sender, RoutedEventArgs e)
        {
            //on utilise une triple égalité pour réinitialiser
            //les projections
            Logo.Projection = pp = new PlaneProjection();
        }

        void RotateXBack_Click(object sender, RoutedEventArgs e)
        {
            pp.RotationX -= 5;
        }

        void GoLeft_Click(object sender, RoutedEventArgs e)
        {
            pp.GlobalOffsetX -= 10;
        }

        void GoRight_Click(object sender, RoutedEventArgs e)
        {
            pp.GlobalOffsetX += 10;
        }

        void GoBackward_Click(object sender, RoutedEventArgs e)
        {
            pp.GlobalOffsetZ -= 10;
        }
    }
}
```

```

void GoForward_Click(object sender, RoutedEventArgs e)
{
    pp.GlobalOffsetZ += 10;
}

void RotateYLeft_Click(object sender, RoutedEventArgs e)
{
    pp.RotationY += 5;
}

void RotateYRight_Click(object sender, RoutedEventArgs e)
{
    pp.RotationY -= 5;
}
}
}

```

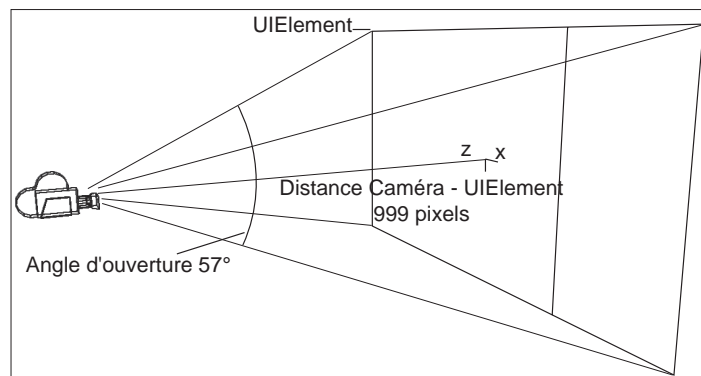
Comme vous le constatez, ces propriétés sont facilement accessibles. Avec un peu plus de code, nous pourrions envisager des interactions 3D performantes sans recourir à des notions de mathématiques complexes. Toutefois, si vous êtes férus de géométrie ou d'arithmétique, vous pouvez également utiliser la classe `Matrix3DProjection` afin d'améliorer les performances et le code logique de vos interfaces. Vous pouvez télécharger le projet : *chap9/ProjectionCSharp.zip*.

9.3 La caméra

Nous allons maintenant approfondir les notions de conteneur 3D et de caméra tout en mettant en pratique ce que nous avons déjà appris. Au sein de Silverlight, la caméra n'est pas vraiment manipulable en tant que telle. Toutefois, elle est bien présente. La meilleure preuve de son existence est l'effet de perspective dès que vous déplacez un objet dans l'espace 3D. La déformation des objets engendrée par la perspective est directement dépendante de différentes propriétés de la caméra. Trois facteurs sont importants. Les deux premiers, directement liés, représentent la longueur de la focale et le champ de vision (ou *field of view*) qui correspond à l'angle d'ouverture verticale de la caméra. Le dernier facteur représente la distance entre la lentille de la caméra et l'objet ciblé (voir Figure 9.17).

Figure 9.17

Principe de la caméra.



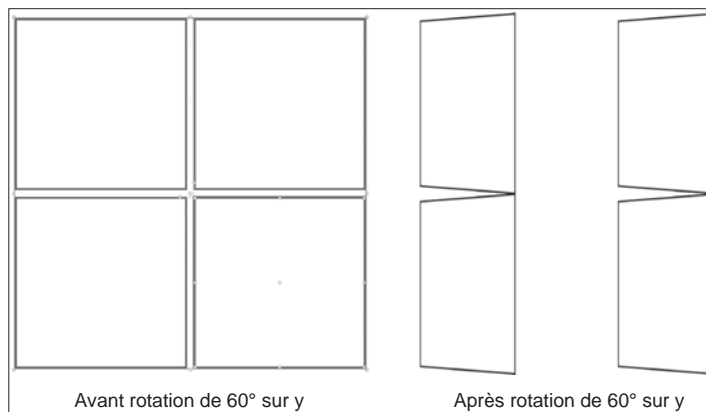
9.3.1 Position de la caméra dans l'espace de projection

Chaque instance de la classe `UIElement` possède la capacité d'être projetée dans un espace indépendant. Cela signifie que chaque objet possède sa propre caméra de projection 3D. Par défaut, la caméra possède une position égale à `ActualWidth/2` et `ActualHeight/2` pour les axes `x` et `y`, et cela par rapport à l'origine 2D de l'objet, qui est située en haut à gauche. Les propriétés `ActualWidth` et `ActualHeight` représentent la valeur réelle de la largeur ou de la hauteur d'un objet quel que soit son mode de redimensionnement. Si nous déplaçons un objet *via* les propriétés de mise en forme standard, la caméra de projection accompagnera l'objet dans son déplacement. Nous allons illustrer ce comportement.

Créez une nouvelle application Silverlight. Instanciez quatre exemplaires de `Rectangle` au sein de `LayoutRoot` et affectez leur largeur et hauteur d'une valeur de 100 pixels. Alignez-les au sein de la grille de manière à créer un damier et appliquez leur une rotation 3D de 60° sur l'axe `y` (voir Figure 9.18).

Figure 9.18

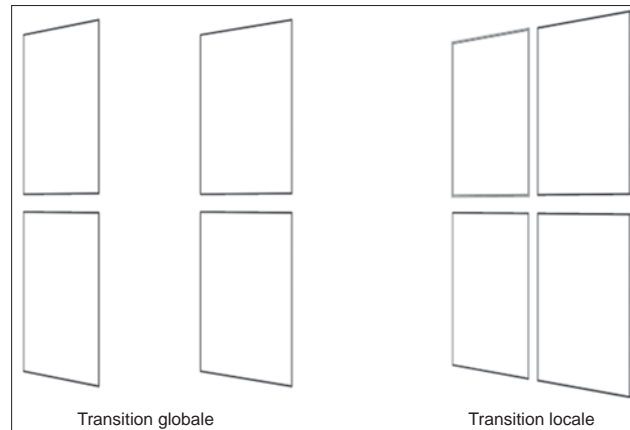
Principe des caméras de projection indépendante.



Comme vous le constatez, les rectangles ne sont pas mis en perspective les uns par rapport aux autres car les propriétés de mise en forme standard ne font pas partie de l'espace projeté. Ainsi leur caméra est déplacée avec eux. Pour que ces rectangles bénéficient de la même perspective, il faut que leur caméra respective soit placée exactement au même endroit les uns par rapport aux autres. Il ne faut donc pas les aligner grâce aux propriétés de mise en pages standard. Nous devons, à la place, utiliser les propriétés de projections 3D. Supprimez tous ces rectangles et recréez-en un de 100 pixels de large par 100 pixels de haut. Ensuite, faites-en trois copies. Les rectangles sont maintenant superposés. Utilisez les translations globales `x` et `y` afin de les aligner pour en faire un damier, ensuite affectez à chacun d'eux une rotation de 60° sur l'axe `y`. Cette fois, vous remarquez que leur position dans l'espace 3D est cohérente. Vous pouvez également tester en les alignant grâce aux translations locales, puis réaffectez une rotation de 60°. Le résultat est très différent selon le type de translations employées (voir Figure 9.19).

Figure 9.19

Principe des caméras de projection indépendantes et alignement grâce aux translations globales.



Les deux types de visuels sont valables car les caméras de projection propres à chaque rectangle sont exactement placées au même endroit. Nous venons de déterminer un fait essentiel : les caméras de projection appartiennent à chaque objet et sont indépendantes les unes des autres.

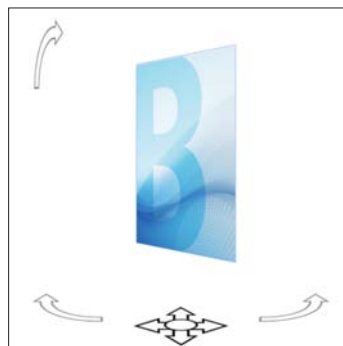
Il nous reste à déterminer la distance par défaut entre un `UIElement` et la caméra sur l'axe z . Celle-ci est assez facile à trouver, bien que non documentée. Il vous suffit de déplacer un objet de 1 000 pixels en z pour ne plus voir cet objet. Il n'est plus visible car il est passé derrière l'objectif de la caméra. Il ne fait plus partie de son champ de vision. Blend vous donne tout de même un aperçu de son enveloppe selon la distance dont l'objet dépasse la caméra. La caméra est donc située à 999 pixels de l'instance `UIElement` associée. Cette distance n'est modifiable qu'en déplaçant l'objet. Aucune propriété ne permet actuellement de modifier la position de la caméra. La distance en z a été choisie de manière à ce que les objets s'affichent de manière 2D plane face à la caméra et à 100 % de leur dimension réelle lorsque ceux-ci n'ont pas subis de projection 3D planaire.

9.3.2 L'angle d'ouverture

L'angle d'ouverture dans Silverlight est de 57 degrés par défaut. Cette valeur n'est pas modifiable, elle est définie en dur dans le code source et conditionne directement la longueur de focale fixée à 33 mm. Il n'est donc pas possible de modifier la déformation de perspective de manière aisée. Ouvrez le projet nommé `ProjectionPlan`, que nous avons enrichi à la section 9.2.4. Compilez-le via le raccourci F5 et faites une rotation sur l'axe y du contrôle Image (voir Figure 9.20).

Figure 9.20

Rotation simple de 60° sur l'axe y , la caméra est à une distance de 999 pixels en z .

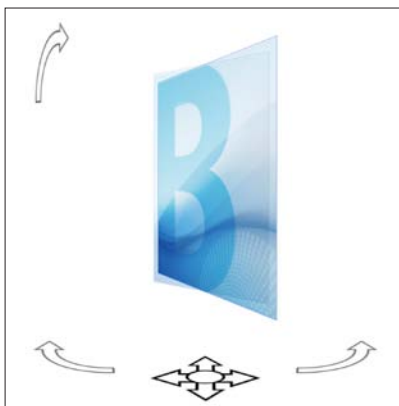


L'effet de perspective est directement dépendant de l'angle d'ouverture et de la focale de la caméra. Il est possible d'accentuer cet effet avec deux transformations complémentaires. Revenez sous Blend et définissez une translation de +500 pixels sur l'axe z du composant Image. Modifiez ensuite les transformations relatives du composant Image pour diviser par deux l'échelle de l'objet. Pour cela, affectez les propriétés `ScaleX` et `ScaleY` chacune de la valeur 0.5. Ainsi, nous aurons l'impression que l'objet est à la même distance de la caméra, mais l'effet de perspective dû à la rotation est très accentué (voir Figure 9.21).

C'est un moyen assez simple permettant de simuler un grand angle de prise de vue. Toutefois cette astuce est peu précise et vous oblige à mélanger les deux types de transformations.

Figure 9.21

Rotation simple de 60° sur l'axe y, la caméra est à une distance de 499 pixels en z, mais l'échelle de l'objet est divisée par deux.



INFO

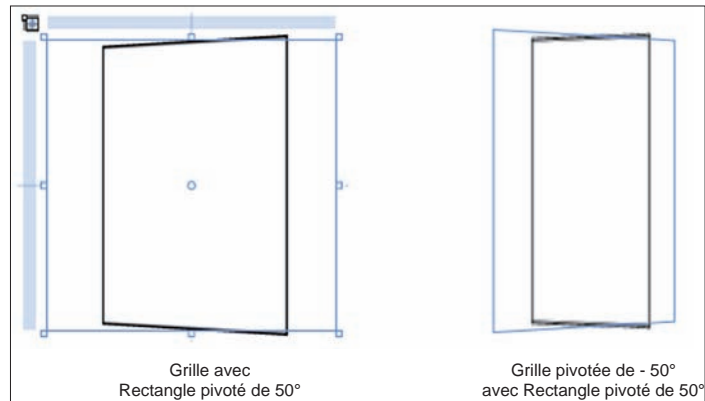
Bien que l'angle d'ouverture soit de 57°, les objets dépassant le cône représentant le champ de vision de 57° ne sont pas masqués. C'est une particularité du modèle de projection qui est avant tout une représentation mathématique simplifiée d'un environnement 3D. Dans la vie réelle, une caméra ne peut simplement pas filmer les objets qui ne sont pas dans son champ de vision. Ce n'est pas le cas dans Silverlight puisque la caméra n'existe pas réellement.

9.3.3 Les conteneurs 3D

Comme nous l'avons vu précédemment, chaque instance de `UIElement` possède son propre environnement de projection 3D et donc sa propre caméra. Cette particularité a une conséquence directe sur la manière dont est gérée la projection. Les conteneurs à plusieurs enfants, lorsqu'ils subissent une projection, n'affectent pas cette projection dynamiquement à chacun de leurs enfants. Autrement dit, les conteneurs de projection 3D n'existent pas par défaut dans Silverlight. Le résultat visuel de comportement est assez simple à démontrer. Dans un nouveau projet, créez une grille, puis un rectangle. Affectez sur ce dernier une rotation de 50° sur l'axe y. Pour finir, faites une rotation sur l'axe y de -50° sur la grille parente du rectangle. Si la grille possédait le comportement d'un conteneur 3D, le rectangle en son sein serait affiché comme s'il ne possédait pas de rotation car la rotation de la grille l'annulerait. Or ce n'est pas le cas, les projections sont calculées séparément et sont appliquées l'une sur l'autre (voir Figure 9.22).

Figure 9.22

Projections 3D sur un conteneur de type Panel et son enfant de type Rectangle.

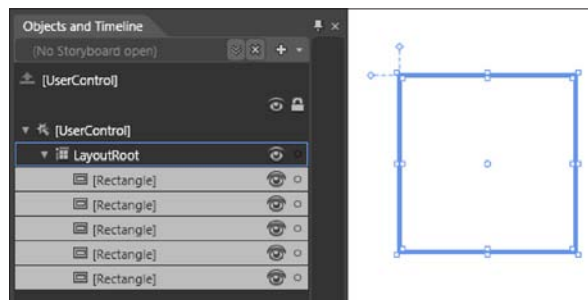


Simuler le comportement d'un conteneur 3D est assez simple mais fastidieux. L'avantage principal de ce type de conteneur résiderait dans sa capacité à affecter les propriétés de projection de tous ses enfants lorsque l'on modifierait les siennes. Faire une rotation de ce conteneur appliquerait une rotation équivalente à tous ces enfants. Pour reproduire ce comportement, il nous faudrait créer les enfants exactement au même endroit dans un conteneur de type Panel, puis les déplacer de manière locale. Ainsi l'axe de rotation serait commun à tous les enfants.

Au sein d'un nouveau projet, créez cinq exemplaires de Rectangle au même endroit dans la grille principale. Ceux-ci doivent donc être centrés les uns par rapport aux autres (voir Figure 9.23).

Figure 9.23

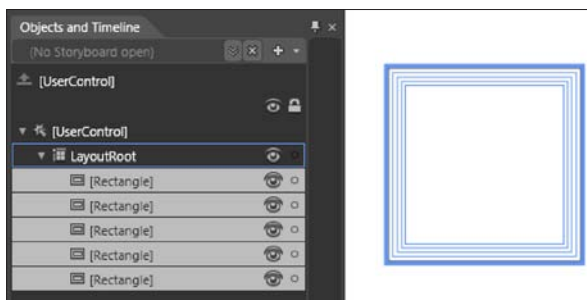
Rectangles centrés au sein de LayoutRoot.



Le fait de centrer les rectangles les uns par rapport aux autres permet de définir une caméra commune. Espacez-les de 50 pixels sur l'axe local z. Le rectangle le plus éloigné doit ainsi posséder une position de -100 pixels en z et le plus proche une position de +100 pixels sur ce même axe. Comme vous utilisez les propriétés de projection, la caméra reste commune à tous (voir Figure 9.24).

Figure 9.24

Rectangles décalés via les axes x et y de translations locales de projection 3D.

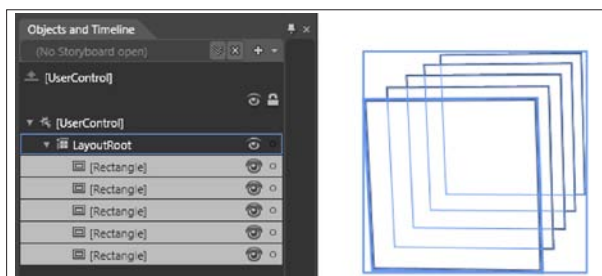


Pour finir, sélectionnez-les tous en même temps, puis affectez-leur une rotation afin d'obtenir un effet de perspective (voir Figure 9.25).

Nous simulons de cette manière le comportement d'un conteneur 3D. Plusieurs autres techniques existent. Vous pourriez, par exemple, grouper les objets au sein de conteneurs de type Canvas de mêmes dimensions et créés aux mêmes coordonnées. Lorsque vous affecteriez une projection 3D à ces conteneurs, les objets qui y sont contenus seraient transformés dans un environnement 3D homogène.

Figure 9.25

Rotation 3D des rectangles.



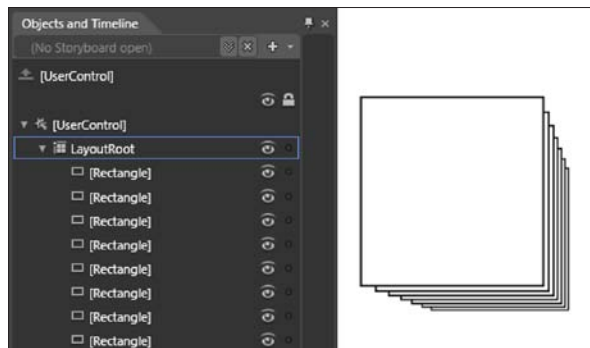
9.3.4 Le point de fuite

Le point de fuite principal, également appelé centre de projection, correspond par défaut aux coordonnées de la caméra en x et y, à l'instant de la création d'un objet. Pour positionner le point de fuite, il vous suffit de créer l'objet à un endroit précis, puis de le déplacer sur l'axe de translation global ou local. Toutefois, vous n'êtes pas obligé de reconstruire un projet depuis le début. Il vous suffit de modifier l'alignement des objets à mettre en perspective dans l'environnement 2D habituel, puis de redéfinir une translation locale ou globale dans l'espace 3D.

Sur l'exemple précédent, supprimez les rotations 3D affectées aux rectangles ; vous obtenez un visuel équivalent à celui de la Figure 9.24. Sélectionnez-les tous et modifiez leurs options d'alignement pour les fixer en bas à droite de la grille LayoutRoot. Pour finir, déplacez-les sur l'axe global de -100 en x et de -100 en y. Vous obtenez automatiquement des lignes de fuite dirigées vers le coin en bas à gauche de la grille, soit le point précis sur lequel ces rectangles étaient alignés dans l'espace 2D (voir Figure 9.26).

Figure 9.26

Effet de perspective 3D par simple déplacement autour du point de fuite.



Pour que des objets partagent le même point de fuite, il vous faudra simplement les créer aux mêmes coordonnées.

INFO

Jusqu'à maintenant, nous avons vu deux moyens différents de gérer l'ordre de superposition des objets. L'ordre des enfants d'un conteneur de type `Panel` affecte directement leur superposition. L'affectation de la propriété `ZIndex` permettait d'affiner cette gestion en passant outre l'ordre d'affichage de l'index de l'enfant. Un troisième moyen existe, il n'est toutefois activé que lorsque la propriété `Projection` des instances d'`UIElement` possède une valeur, donc uniquement lorsque les enfants sont projetés dans un espace 3D. La distance de l'objet à la caméra va directement conditionner l'ordre de superposition. Au sein de Silverlight, le tri sur l'axe z, donc sur l'axe caméra objet, est géré sans aucun besoin d'un quelconque calcul de votre part. Ce tri est automatique. Vous pouvez le voir à l'œuvre sur la Figure 9.26. De la même manière qu'avec le tri par index d'enfants, vous pouvez outrepasser le tri sur l'axe z via la propriété 2D `ZIndex`.

9.4 Introduction aux matrices

Jusqu'à maintenant, nous avons abordé les transformations 3D du point de vue d'un graphiste. Bien que nous puissions affecter des transformations 3D par C# en utilisant la classe `PlaneProjection`, celle-ci apparaîtra limitée lorsque vous aurez besoin d'appliquer des projections 3D plus complexes aux instances d'`UIElement`. La compréhension des matrices permet de s'affranchir des limitations de la classe `PlaneProjection`, puisque vous pourrez utiliser `Matrix3DProjection` à sa place. Attention, cette section est une introduction aux matrices, le but est de vous familiariser avec ce concept. Toutefois si vous n'êtes pas développeur, vous pouvez complètement occulter cette section et utiliser la classe simplifiée `PlaneProjection` qui répond à 60 % des cas. Nous mettrons également en parallèle l'API 3D fournie par Silverlight et celle fournie par DirectX.

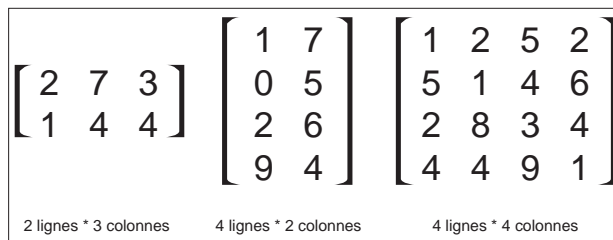
9.4.1 Définition et principes

Une matrice est une notation mathématique permettant de représenter des équations, des expressions ou des données, de manière à simplifier les calculs complexes. Elles sont particulièrement utiles aux développeurs d'environnement 3D, mais également aux graphistes, qui les emploient sans forcément le savoir (via les outils de modification d'objets). Elles permettent, entre autres, d'appliquer des transformations aux objets dans un environnement 2D ou 3D. Concrètement, une

matrice contient une série de nombres (ou d'expressions renvoyant des nombres) formalisée sous forme de n lignes et de n colonnes encadrées par des crochets (voir Figure 9.27).

Figure 9.27

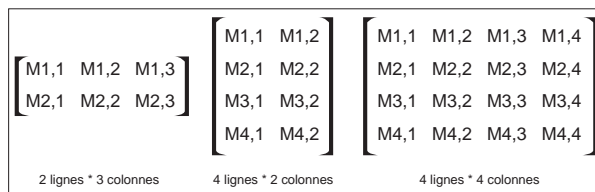
Différents exemples de matrices.



Les indices de ligne et de colonne permettent aux valeurs contenues dans une matrice d'être accessibles. Ils constituent un système de coordonnées sous forme de numéros de ligne et de numéro de colonne. Par convention, ces coordonnées sont notées $MnLigne, nColonne$ (voir Figure 9.28).

Figure 9.28

Coordonnées de membres de matrices.

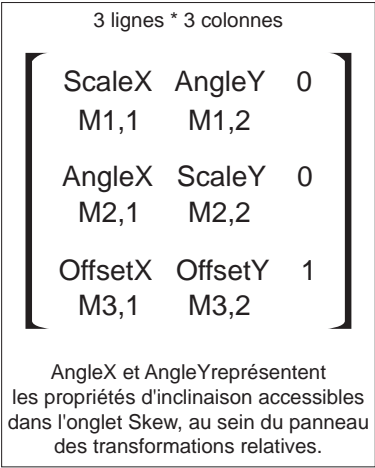


Nous avons déjà vu les matrices de transformations dans ce livre, même si nous ne l'avons pas formalisé, notamment lorsque nous abordé les transformations relatives (voir Chapitre 5). Le nœud `TransformGroup` contenant tous les types de transformation est en fait la représentation simplifiée d'une matrice de transformation 2D :

```
<Rectangle.RenderTransform>
  <TransformGroup>
    <ScaleTransform/>
    <SkewTransform/>
    <RotateTransform Angle="15" />
    <TranslateTransform X="60" Y="40" />
  </TransformGroup>
</Rectangle.RenderTransform>
```

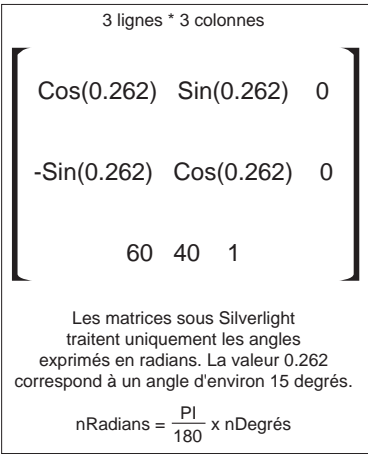
Son principal avantage réside dans sa facilité d'utilisation. Il permet, par exemple, aux designers de faire des rotations 2D, ce qui n'est pas aussi simple qu'il y paraît. Comme nous l'avons déjà évoqué, la propriété `RenderTransform` des instances d'`UIElement` accepte également une instance de la classe `MatrixTransform`. Celle-ci possède la propriété `Matrix` acceptant une valeur (structure) de type `Matrix`. Cette dernière est la version brute d'une matrice mathématique constituée de trois colonnes et de trois lignes. Elle permet de contrôler entièrement les déformations de manière colinéaire (voir Figure 9.29).

Figure 9.29
Représentation de la
matrice de transformation
vectorielle 2D de Silverlight.



La matrice correspondant aux transformations du nœud `RenderTransform` (celui que nous avons décrit en XAML et qui affecte au rectangle +15 degrés pour la rotation, 60 pixels de déplacement en X et 40 en Y) est décrite à la Figure 9.30.

Figure 9.30
Représentation de la
matrice de transformation
correspondant au nœud
`RenderTransform`.



Au sein d'une matrice de transformation, la propriété exprimant la rotation n'existe pas réellement. Elle est en réalité générée par la combinaison de l'inclinaison et du redimensionnement de l'objet. Voici la traduction de notre matrice côté XAML :

```
<Rectangle.RenderTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix OffsetY="60" OffsetY="40" M11="0.966" M22="0.966"
        M12="0.259" M21="-0.259" />
    </MatrixTransform.Matrix>
  </MatrixTransform>
</Rectangle.RenderTransform>
```

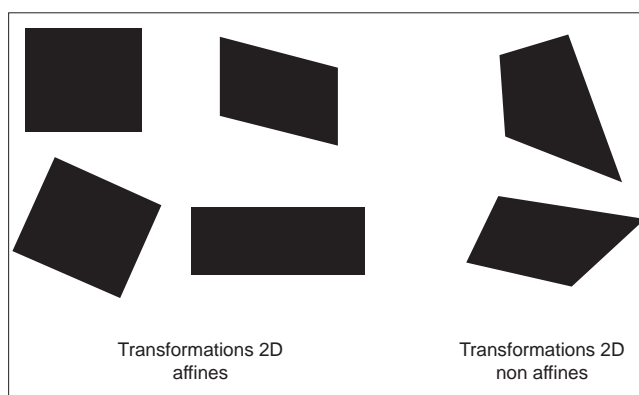
INFO

Vous remarquez que les indices aux coordonnées M31 et M32 sont directement notés OffsetX et OffsetY au sein de la structure `Matrix`. Les autres propriétés sont accessibles par la notation `Mn,m`. Cela permet à un développeur d'identifier et d'utiliser simplement les propriétés de la matrice qui correspondent aux translations en x et y.

La matrice de transformation 2D possède trois colonnes. La troisième colonne contient $M1,3=0$, $M2,3=0$ et $M3,3=1$. Ces valeurs ne sont pas modifiables. Sans rentrer dans une explication mathématique complexe, cela permet d'éviter les déformations ou les distorsions des objets 2D lorsque ceux-ci subissent des rotations, des mises à l'échelle ou des inclinaisons. La matrice 2D de Silverlight gère ainsi des transformations dites affines, également appelées transformations colinéaires (voir Figure 9.31).

Figure 9.31

Transformations affines et non affines.



Du côté C#, une instance de `MatrixTransform` est affectée à la propriété `RenderTransform` d'un objet héritant de `UIElement`. La classe `MatrixTransform` possède la propriété `Matrix` qui contient réellement les valeurs entre crochets étudiées plus haut :

```
MatrixTransform Mt = new MatrixTransform();
MonUIElement.RenderTransform = Mt;

/*
 * Renvoie Identity qui fait référence
 * à la matrice d'identité par défaut :
 * 1,0,0
 * 0,1,0
 * 0,0,1
 */
Debug.WriteLine(Mt.Matrix.ToString());
```

Dans l'exemple ci-dessus, l'objet n'est pas transformé. Lorsque vous instanciez un nouvel objet `MatrixTransform`, sa propriété `Matrix` correspond par défaut à une matrice d'identité. Une matrice d'identité est neutre et n'applique aucune transformation à l'objet. Elle représente l'état de la matrice lorsque l'objet n'a subi aucune transformation. Vous pouvez également instancier la structure `Matrix` afin d'appliquer une transformation personnalisée à l'objet de type `UIElement` lors de l'initialisation :

```

Matrix M = new Matrix(1,1,
                     0,1,
                     0,0);

MatrixTransform Mt = new MatrixTransform();

Mt.Matrix = M;

bt.RenderTransform = Mt;

/*
 * Renvoie la matrice affectée :
 * 1,1,
 * 0,1,
 * 0,0
 */
Debug.WriteLine(Mt.Matrix.ToString());

```

Comme vous le constatez, la matrice ne renvoie pas la dernière colonne, celle-ci n'est simplement pas accessible ou modifiable.

9.4.2 Les matrices 3D

Au sein de Silverlight, les matrices de transformations 3D suivent les mêmes principes que ceux évoqués ci-dessus. Elles gèrent toutefois les transformations non affines car elles ont pour but d'afficher des objets en perspective 3D, ce qui déforme les objets dans une certaine mesure. Elles sont constituées de quatre lignes et de quatre colonnes et possèdent une notation par ligne équivalente à celle que propose DirectX. À l'opposé, les matrices 3D fournies par OpenGL et Flash ont une notation par colonne. Vous pouvez facilement identifier le type d'écriture utilisé en fonction du positionnement des valeurs exprimant le déplacement (voir Figure 9.32).

Figure 9.32

*Représentation
des matrices 3D selon
les plateformes.*

$\begin{bmatrix} M1,1 & M1,2 & M1,3 & M1,4 \\ M2,1 & M2,2 & M2,3 & M2,4 \\ M3,1 & M3,2 & M3,3 & M3,4 \\ \text{OffsetX} & \text{OffsetY} & \text{OffsetZ} & M4,4 \end{bmatrix}$	$\begin{bmatrix} M1,1 & M1,2 & M1,3 & Tx \\ M2,1 & M2,2 & M2,3 & Ty \\ M3,1 & M3,2 & M3,3 & Tz \\ M4,1 & M4,2 & M4,3 & M4,4 \end{bmatrix}$
Notation par ligne DirectX adoptée par Silverlight	Notation par colonne OpenGL adoptée par Flash

INFO

Il est également intéressant de noter que l'axe z n'est pas dirigé vers l'utilisateur, mais vers l'horizon pour OpenGL et Flash, à l'opposé de DirectX et Silverlight. On évoque à ce sujet la notion de repère main droite ou gauche : DirectX utilise le repère main gauche alors qu'OpenGL est basé sur un repère main droite. Il faut également remarquer que même si les matrices au sein de Silverlight héritent de DirectX, l'axe y ne suit pas la même direction. Au sein du lecteur Silverlight, il est dirigé vers le bas et sous DirectX vers le haut, mais cela ne change rien aux valeurs contenues par la matrice.

La classe `Matrix3DProjection` est l'équivalent de la classe `MatrixTransform` qui permet les transformations 2D. Les instances de `Matrix3DProjection` ne sont pas affectées à la propriété `RenderTransform` des exemplaires d'`UIElement`, mais à leur propriété `Projection`. La structure

Matrix3D est homologue à Matrix. Il est donc possible d'affecter des matrices 2D et 3D en même temps. Toutefois, cela est déconseillé dans la majorité des situations car le résultat devient très vite difficile à prédire. Dans tous les cas de projection, il faudra utiliser la propriété ProjectionMatrix de la classe Matrix3DProjection et lui affecter une instance de la structure Matrix3D. L'écriture en XAML est assez similaire à celle que nous avons déjà étudiée plus haut, mais un peu plus verbeuse :

```
<Rectangle.Projection>
  <Matrix3DProjection>
    <Matrix3DProjection.ProjectionMatrix>
      <Matrix3D
        M11="1" M12="0" M13="0" M14="0"
        M21="0" M22="1" M23="0" M24="0"
        M31="0" M32="0" M33="1" M34="0"
        OffsetX="50" OffsetY="0" OffsetZ="0" M44="1" />
      </Matrix3DProjection.ProjectionMatrix>
    </Matrix3DProjection>
  </Rectangle.Projection>
```

Vous pouvez également utiliser cette écriture simplifiée :

```
<Rectangle.Projection>
  <Matrix3DProjection
    ProjectionMatrix="1, 0, 0, 0,
                      0, 1, 0, 0,
                      0, 0, 1, 0,
                      50, 0, 0, 1"
  />
</Rectangle.Projection>
```

En voici une version C# :

```
//on crée une nouvelle instance de matrice
Matrix3D MyMatrix = new Matrix3D();

//On effectue une translation de 50 pixels sur l'axe x.
MyMatrix.M11=1; MyMatrix.M12=0; MyMatrix.M13=0; MyMatrix.M14=0;
MyMatrix.M21=0; MyMatrix.M22=1; MyMatrix.M23=0; MyMatrix.M24=0;
MyMatrix.M31=0; MyMatrix.M32=0; MyMatrix.M33=1; MyMatrix.M34=0;
MyMatrix.OffsetX=50; MyMatrix.OffsetY=0; MyMatrix.OffsetZ=0; MyMatrix.M44=1;

//on crée une nouvelle instance de la classe
//enveloppante Matrix3DProjection
Matrix3DProjection MyMatrix3DProjection = new Matrix3DProjection();

//on lui affecte MyMatrix créée auparavant
MyMatrix3DProjection.ProjectionMatrix = MyMatrix;

//on affecte la propriété Projection du rectangle
MonRectangle.Projection = m3dProjection;
```

Comme nous l'avons dit plus haut, la notation apportée par les matrices présente l'avantage de faciliter les calculs mathématiques complexes. L'opération la plus courante est la multiplication, nous allons l'aborder dans la prochaine section.

9.4.3 Opérations sur les matrices

À l'origine, les matrices de transformation possèdent de nombreuses méthodes facilitant leur utilisation. L'API 3D de Silverlight étant toute récente, la plupart des méthodes que vous trouverez sous DirectX ne sont pas intégrées nativement dans le lecteur. Microsoft en fournit toutefois un certain nombre au sein de la documentation de Silverlight. Vous pourrez les utiliser pour bénéficier de plus de capacités.

En premier lieu, il nous faut comprendre certains concepts de base permettant de manipuler des matrices de transformations 3D. Si vous possédez un niveau équivalent à un DEUG de Maths de première année, vous connaissez sans doute les matrices et pouvez passer votre chemin. Dans ce cas, vous pourriez éventuellement vous lancer dans le portage d'une partie de l'API fournie dans DirectX vers la plateforme Silverlight.

Pour appliquer une transformation à un objet, nous devons utiliser une matrice de transformation. Celle-ci décrit précisément ce que vous souhaitez modifier. Si vous voulez, par exemple, changer l'échelle d'une instance de 200 % en y, il faut commencer par créer la matrice de transformation décrivant le redimensionnement, puis multiplier celle de l'objet par cette dernière. L'opération de multiplication est la plus utilisée car elle permet d'ajouter de nouvelles transformations aux objets. Elle obéit à des règles précises (voir Figure 9.33)

Figure 9.33

Multiplication de matrices.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} Ta & Tb \\ Tc & Td \end{bmatrix} = \begin{bmatrix} a \times Ta + b \times Tc & a \times Tb + b \times Td \\ c \times Ta + d \times Tc & c \times Tb + d \times Td \end{bmatrix}$$

Matrice de l'objet source à modifier.
Matrice de transformation à appliquer.
Matrice finale de l'objet une fois transformé.

Comme vous le constatez, nous multiplions les valeurs contenues dans les colonnes par les valeurs présentes au sein des lignes. Vous pouvez multiplier une matrice A par une matrice B à partir de l'instant où le nombre de colonnes de A est égal au nombre de lignes de B. Il est ainsi possible de multiplier une matrice définie sur une ligne et quatre colonnes, par une autre de quatre lignes et de quatre colonnes.

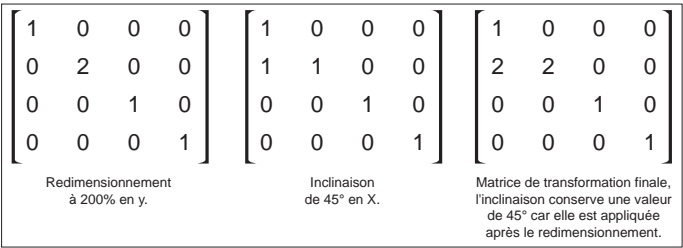
INFO

Du fait de la notation par ligne utilisée par Silverlight et héritée de DirectX, l'ordre de la multiplication est spécifique. Dans notre cas, la première matrice symbolise l'objet à transformer alors que la seconde désigne la matrice de transformation. À l'opposé, dans l'environnement de développement OpenGL et Flash, la première des deux matrices représente la matrice de transformation, la seconde est la matrice de l'objet à modifier.

Il est peut-être utile de multiplier deux matrices de transformation entre elles. Par exemple, l'une pourrait modifier l'échelle en y, et l'autre affecterait l'inclinaison en x (voir Figure 9.34).

Figure 9.34

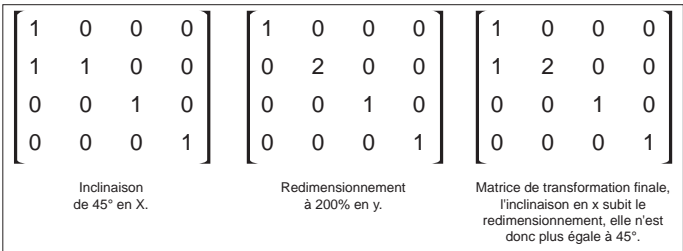
*Multiplication
de matrices de
transformation.*



Il reste à multiplier la matrice générée par celle de l'objet pour appliquer les deux transformations en une seule fois. Il est aisé de comprendre pourquoi la multiplication de matrice n'est pas commutative, c'est-à-dire que l'opération $\text{Matrice1} \times \text{Matrice2}$ est différente de $\text{Matrice2} \times \text{Matrice1}$. Si nous reprenons le même exemple en inversant l'ordre des matrices, nous n'obtenons pas le même résultat (voir Figure 9.35).

Figure 9.35

*Ordre inversé de la
multiplication
de matrice initiale.*



Visuellement, la différence est flagrante entre les deux multiplications (voir Figure 9.36).

Figure 9.36

*Deux résultats
visuels différents
selon l'ordre de
multiplication.*



Comme nous l'avons déjà évoqué plus haut, la rotation n'est en fait qu'une combinaison d'inclinaisons et de redimensionnements.

INFO

Pour le vérifier, vous pouvez prendre n'importe quelle instance de la classe `UIElement`, puis modifier ses transformations relatives `AngleX` et `AngleY` et les affecter respectivement d'un angle et de son opposé. Si vous utilisez les valeurs 30 et -30, vous remarquez que l'objet subit une rotation ainsi qu'un agrandissement engendré par les deux inclinaisons combinées. Pour palier à ce redimensionnement, il faut également affecter l'échelle en x et y (propriétés `ScaleX` et `ScaleY`) par le cosinus de l'angle d'inclinaison.

C'est exactement le même principe pour les matrices 3D sauf que nous devons prendre en compte l'axe z. Il faut en fait réfléchir par plan. La rotation s'effectue dans tous les cas sur un plan à deux axes. C'est également le cas pour les matrices de transformation 2D permettant les rotations sur le plan xy car la rotation s'effectue autour d'un axe z abstrait. Nous ne rentrerons pas dans l'explication des formules mathématiques car cela sort du cadre de ce livre. Toutefois, il suffit d'assimiler la rotation dans un environnement 2D pour comprendre ces dernières. Nous allons identifier leur emplacement (voir Figure 9.37).

Figure 9.37

Identification des opérations de rotation d'angle a dans la matrice 3D Silverlight.

$\begin{bmatrix} \cos(a) & -\sin(a) & 0 & 0 \\ \sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \text{OffsetX} & \text{OffsetY} & \text{OffsetZ} & 1 \end{bmatrix}$ <p>Rotation sur le plan xy autour de l'axe z.</p>	$\begin{bmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 1 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ \text{OffsetX} & \text{OffsetY} & \text{OffsetZ} & 1 \end{bmatrix}$ <p>Rotation sur le plan xz autour de l'axe y.</p>	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ \text{OffsetX} & \text{OffsetY} & \text{OffsetZ} & 1 \end{bmatrix}$ <p>Rotation sur le plan yz autour de l'axe x.</p>
--	--	--

Le sens de rotation est directement lié au fait que les inclinaisons x et y possèdent des valeurs opposées. Pour l'inverser, vous pouvez affecter la valeur $-\sin(a)$ à M21 et $\sin(a)$ à M12. Dans ce cas, le sens de rotation est l'opposé de celui généré par la matrice montrée à la Figure 9.37. Il serait pratique de centraliser les méthodes qui facilitent la manipulation de matrice 3D. À cette fin, nous pouvons créer une classe statique `Matrix3DUtils` contenant, entre autres, des méthodes statiques de rotation, de translation ou de redimensionnement :

```
public static Matrix3D RotateZTransform ( double a )
{
    //l'angle a est exprimé en radians
    double sin = Math.Sin(a);
    double cos = Math.Cos(a);

    Matrix3D m = new Matrix3D();

    m.M11 =cos; m.M12 =sin; m.M13 = 0; m.M14 = 0.0;
    m.M21 =-sin; m.M22 =cos; m.M23 = 0.0; m.M24 = 0.0;
    m.M31 = 0; m.M32 = 0.0; m.M33 = 0.0; m.M34 = 0.0;
    m.OffsetX = 0.0; m.OffsetY = 0.0; m.OffsetZ = 0.0; m.M44 = 1.0;

    //on retourne la matrice permettant d'appliquer la rotation
    return m;
}
```

Son utilisation faciliterait grandement la transformation d'instances de type `UIElement` :

```
Matrix3DProjection M3P = new Matrix3DProjection();

Matrix3D M3D = new Matrix3D();

void MainPage_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    //on applique une rotation de 15° lors du clic gauche de la souris
    M3D *= Matrice3DUtils.RotateZTransform(15*Math.PI/180);

    M3P.ProjectionMatrix = M3D;

    MonRectangle.Projection = M3P;
}
```


9.4.4 Initialiser une vue perspective

La classe `Matrix3D` ne fournit pas de perspective par défaut. Microsoft a volontairement fourni les briques les plus élémentaires afin que les développeurs puissent réaliser leur propre moteur 3D. Cela permet d'ouvrir le champ des possibilités mais demande quelques connaissances en mathématique. Vous devez donc passer par plusieurs étapes pour simuler un espace en 3D :

1. Choisir la valeur de l'angle d'ouverture de la caméra factice (nommé *Field Of View* ou FOV en anglais) dont nous avons évoqué le rôle à la section 9.3.2.
2. Créer une matrice définissant le centre de transformation 3D de l'objet projeté.
3. Positionner l'objet dans l'espace 3D à l'initialisation.
4. Générer une matrice simulant l'effet de perspective en fonction de l'angle d'ouverture.
5. Définir les limites du champ d'ouverture.

Toutes ces initialisations importantes sont fournies aux designers par la classe `PlaneProjection`. Cette classe est malgré tout limitée et ne permet pas de modifier la focale, la direction de la caméra ou un redimensionnement 3D de l'objet projeté. Pour bénéficier de ce type de contrôle personnalisé, vous devrez générer toutes les matrices décrites par les étapes que nous venons de lister. Heureusement, Microsoft les propose dans la documentation de `Silverlight`. Toutefois le code fourni n'est pas commenté, et les valeurs passées dans l'appel de certaines de ces fonctions ainsi que la définition de l'une d'entre elles méritent quelques réajustements afin d'être utilisées facilement. Gardez à l'esprit que l'ordre dans lequel sont affectées les matrices est réellement important. Voici le code modifié et commenté de l'appel des méthodes générant chaque matrice ainsi que leur affectation à la propriété `Projection` d'une instance :

```
private void AppliquerScene3D(object sender, MouseButtonEventArgs e)
{
    //on définit un angle d'ouverture de 60°
    double FovY = 60*Math.PI / 180;

    //On crée une matrice déplaçant l'objet afin de
    //repositionner au milieu son centre de transformation
    //Par défaut le centre de transformation est au milieu
    Matrix3D RecentrerPivotDeRotation = TranslationTransform
        (-BeachImage.ActualWidth / 2.0,
        -BeachImage.ActualHeight / 2.0,
        0);

    //angle de rotation
    double Angle = 60.0 * Math.PI / 180.0;

    //diverses matrices de transformation
    Matrix3D RotationAutourDeX = RotateXTransform(Angle);
    Matrix3D TranslationSurX = TranslationTransform(200,0,0);
    Matrix3D RotationAutourDeY = RotateYTransform(Angle);
    Matrix3D RotationAutourDeZ = RotateZTransform(Angle);

    //Translation de l'objet afin qu'il prenne
    //par défaut la largeur et la hauteur correspondant
    //à sa représentation 2D.
    double TranslationZ = -(LayoutRoot.ActualHeight/2)/Math.Tan(FovY/2) ;
    //on génère une matrice qui éloigne l'objet de la caméra sur z
    Matrix3D EloignerObjetDeLaCamera = TranslationTransform
        (0, 0, TranslationZ);
```

```

//Création de la matrice simulant la perspective
//Param 1 : Angle d'ouverture
//Param 2 : Rapport largeur/hauteur
//Param 3 : Le plan de visibilité le plus proche
//Param 4 : Le plan de visibilité le plus éloigné
Matrix3D EffetDePerspective = PerspectiveTransformFovRH
    (fovY, LayoutRoot.ActualWidth / LayoutRoot.ActualHeight,
     1.0, 2000.0);

//Matrice décrivant la fenêtre d'ouverture
Matrix3D CadreVisible = ViewportTransform
    (LayoutRoot.ActualWidth, LayoutRoot.ActualHeight,
     BeachImage.ActualWidth, BeachImage.ActualHeight);

//1 - on remplace l'objet par rapport
//à son centre de transformation
Matrix3D M = RecentrerPivotDeRotation;

//2 - on lui applique toutes les transformations que l'on souhaite,
//une rotation ainsi qu'un déplacement sur x dans le cas suivant
M = M * RotationAutourDeX;
M = M * TranslationSurX;
//M = M * RotationAutourDeY;
//M = M * RotationAutourDeZ;

//3 - on le positionne à distance de la caméra
M = M * EloignerObjetDeLaCamera;

//4 - on lui applique l'effet de perspective
M = M * EffetDePerspective;

//5 - on l'affiche dans un espace délimité
M = M * CadreVisible;

//6 - affectation de la matrice finalisée m
Matrix3DProjection M3DProjection = new Matrix3DProjection();
M3DProjection.ProjectionMatrix = M;
MonUIElement.Projection = M3DProjection;
}

```

Vous remarquez que les transformations à appliquer à un objet doivent être affectées avant que les matrices simulant la perspective soient apposées à ce dernier. Vous pourriez ainsi stocker dans une matrice unique le positionnement de l'objet, la perspective et les limites du cadre de visualisation. De cette manière, vous finiriez toujours par appliquer cette matrice en dernier lieu après avoir modifié l'objet projeté. Voici les définitions de méthodes générant les matrices de transformation 3D simples :

```

private Matrix3D TranslationTransform(double tx, double ty, double tz)
{
    Matrix3D m = new Matrix3D();

    m.M11 = 1.0; m.M12 = 0.0; m.M13 = 0.0; m.M14 = 0.0;
    m.M21 = 0.0; m.M22 = 1.0; m.M23 = 0.0; m.M24 = 0.0;
    m.M31 = 0.0; m.M32 = 0.0; m.M33 = 1.0; m.M34 = 0.0;
    m.OffsetX = tx; m.OffsetY = ty; m.OffsetZ = tz; m.M44 = 1.0;
    return m;
}

private Matrix3D ScaleTransform(double sx, double sy, double sz)

```

```

{
    Matrix3D m = new Matrix3D();

    m.M11 = sx; m.M12 = 0.0; m.M13 = 0.0; m.M14 = 0.0;
    m.M21 = 0.0; m.M22 = sy; m.M23 = 0.0; m.M24 = 0.0;
    m.M31 = 0.0; m.M32 = 0.0; m.M33 = sz; m.M34 = 0.0;
    m.OffsetX = 0.0; m.OffsetY = 0.0; m.OffsetZ = 0.0; m.M44 = 1.0;
    return m;
}

private Matrix3D RotateYTransform(double theta)
{
    double sin = Math.Sin(theta);
    double cos = Math.Cos(theta);

    Matrix3D m = new Matrix3D();
    m.M11 = cos; m.M12 = 0.0; m.M13 = sin; m.M14 = 0.0;
    m.M21 = 0.0; m.M22 = 1.0; m.M23 = 0.0; m.M24 = 0.0;
    m.M31 = -sin; m.M32 = 0.0; m.M33 = cos; m.M34 = 0.0;
    m.OffsetX = 0.0; m.OffsetY = 0.0; m.OffsetZ = 0.0; m.M44 = 1.0;
    return m;
}

private Matrix3D RotateZTransform(double theta)
{
    double cos = Math.Cos(theta);
    double sin = Math.Sin(theta);

    Matrix3D m = new Matrix3D();
    m.M11 = cos; m.M12 = sin; m.M13 = 0.0; m.M14 = 0.0;
    m.M21 = -sin; m.M22 = cos; m.M23 = 0.0; m.M24 = 0.0;
    m.M31 = 0.0; m.M32 = 0.0; m.M33 = 1.0; m.M34 = 0.0;
    m.OffsetX = 0.0; m.OffsetY = 0.0; m.OffsetZ = 0.0; m.M44 = 1.0;
    return m;
}

private Matrix3D RotateXTransform(double theta)
{
    double cos = Math.Cos(theta);
    double sin = Math.Sin(theta);

    Matrix3D m = new Matrix3D();
    m.M11 = 1.0; m.M12 = 0.0; m.M13 = 0.0; m.M14 = 0.0;
    m.M21 = 0.0; m.M22 = cos; m.M23 = sin; m.M24 = 0.0;
    m.M31 = 0.0; m.M32 = -sin; m.M33 = cos; m.M34 = 0.0;
    m.OffsetX = 0.0; m.OffsetY = 0.0; m.OffsetZ = 0.0; m.M44 = 1.0;
    return m;
}

```

Voici les méthodes qui gèrent les effets de perspective, ainsi que le positionnement de ce dernier au sein d'une fenêtre :

```

private Matrix3D PerspectiveTransformFovRH
    (double fieldOfViewY, double aspectRatio,
     double zNearPlane, double zFarPlane)
{
    double height = 1.0 / Math.Tan(fieldOfViewY / 2.0);
    double width = height / aspectRatio;
    double d = zNearPlane - zFarPlane;
    Matrix3D m = new Matrix3D();
    m.M11 = width; m.M12 = 0; m.M13 = 0; m.M14 = 0;
    m.M21 = 0; m.M22 = height; m.M23 = 0; m.M24 = 0;

```

```

        m.M31 = 0; m.M32 = 0; m.M33 = zFarPlane / d; m.M34 = -1;
        m.OffsetX = 0; m.OffsetY = 0;
        m.OffsetZ = zNearPlane * zFarPlane / d;
        m.M44 = 0;
        return m;
    }

    private Matrix3D ViewportTransform
        (double viewPortWidth, double viewPortHeight,
         double projectedObjectWidth,
         double projectedObjectHeight)
    {
        Matrix3D m = new Matrix3D();
        m.M11 = viewPortWidth / 2.0; m.M12 = 0.0; m.M13 = 0.0; m.M14 = 0.0;
        m.M21 = 0.0; m.M22 = viewPortHeight / 2.0; m.M23 = 0.0; m.M24 = 0.0;
        m.M31 = 0.0; m.M32 = 0.0; m.M33 = 1.0; m.M34 = 0.0;
        m.OffsetX = projectedObjectWidth / 2.0;
        m.OffsetY = projectedObjectHeight / 2.0;
        m.OffsetZ = 0.0;
        m.M44 = 1.0;
        return m;
    }
}

```

Dans tous les cas, nous ne faisons que simuler un environnement 3D à travers la déformation de l'objet lui-même. Les méthodes de perspective et de cadrage sont une combinaison de redimensionnements, d'inclinaisons et de translations de l'objet. Vous pouvez maintenant vous amuser à modifier la valeur de l'angle d'ouverture pour obtenir des effets de caméra intéressants (voir Figure 9.38).

Figure 9.38

Différentes longueurs de focale.



Le projet finalisé est dans l'archive des exemples du livre : *chap9/Simple3DScene.zip*. Vous pouvez améliorer le code proposé ci-dessus en créant, par exemple, des méthodes ou des propriétés de translations locales et globales équivalentes à celles fournies par la classe simple `PlaneProjection`. Vous pourriez également afficher les deux faces d'un objet de manière distincte et ainsi commencer à développer votre propre minimoteur de projection 3D pour Silverlight. Dans cette optique, je mets à disposition une bibliothèque gratuite nommée `3DLightEngine`. Vous la trouverez sur mon blog : <http://www.tweened.org>. Son code est complètement ouvert à la modification et vous pouvez l'adapter selon vos besoins.

Dans le prochain chapitre, nous aborderons le prototypage d'applications riches *via* la technologie SketchFlow. La 3D jouera sans aucun doute un rôle prépondérant en matière d'expérience utilisateur dans les prochaines années. Au Chapitre 10, nous utiliserons nos connaissances acquises en 3D pour ajouter une touche de profondeur aux différents panneaux de notre application, et améliorer ainsi son ergonomie.

Prototypage dynamique avec SketchFlow

Dans ce chapitre, vous découvrirez un nouvel outil de prototypage nommé SketchFlow. Bien qu'il soit intégré à Expression Blend, vous n'êtes pas obligé d'avoir lu la totalité des chapitres précédents ou d'avoir une connaissance approfondie de Blend pour utiliser SketchFlow. Sa facilité d'utilisation est l'un de ses grands avantages. Que vous soyez responsable de production, directeur technique ou artistique ou encore designer, ce chapitre vous concerne pleinement. Nous y aborderons le prototypage et nous créerons un premier prototype grâce à SketchFlow. L'un des objectifs de SketchFlow est de faciliter la communication entre les différents acteurs du développement afin d'arriver à un consensus. Dans cette optique, nous listerons les moyens mis à disposition par SketchFlow pour œuvrer en ce sens. Pour finir, nous étudierons les outils facilitant l'interactivité utilisateur, ainsi que les différentes approches existantes pour la conception d'interfaces utilisateur.

10.1 L'environnement

Si SketchFlow est un outil de prototypage, la notion même de prototype revêt des formes nuancées ainsi qu'une mise en pratique différente selon les environnements de production rencontrés. Dans cette section, nous allons définir ce terme et étudier l'intégration de SketchFlow au sein du flux de production. Pour finir, nous listerons les principes et moyens mis en œuvre par ce dernier lui permettant de se positionner comme outil de prototypage dynamique.

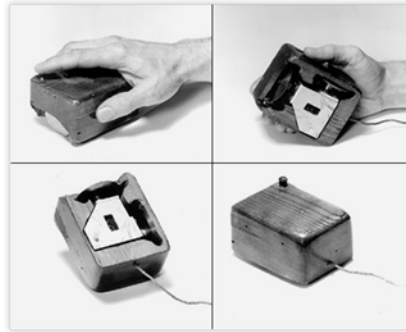
10.1.1 Le prototypage

Le prototype, même s'il n'a pas toujours été formalisé comme tel, est sans doute l'une des notions les plus vieilles de l'histoire de l'humanité. Il y a toujours une première fois. La première massue, le premier vélo, le premier avion, et même la première souris (voir Figure 10.1) ont tous été inventés et rapidement mis à l'épreuve un beau jour. Même s'ils n'ont pas été dès le départ de

francs succès techniques ou populaires, ils sont devenus des objets indispensables que l'homme n'a jamais cessé d'améliorer par la suite.

Figure 10.1

*L'une des premières
souris, créée en 1967
(source WikiPedia).*



En industrie, un prototype est le premier ou l'un des premiers exemplaires fonctionnels d'un produit industriel (que l'on peut produire en grande quantité de manière rentable). Aujourd'hui, le prototypage est très étroitement lié au design. Un objet "design" peut se définir par trois termes : industrialisable, esthétique et ergonomique. Le but des écoles de design est concentré exclusivement autour de la création de prototypes industriels qui doivent réunir ces trois qualités. Si aujourd'hui ce concept est totalement intégré à l'industrie, cela n'est pas forcément de mise concernant le développement informatique. L'industrialisation du développement informatique n'en est aujourd'hui qu'à ses balbutiements, n'oublions pas que l'informatique est un domaine très jeune comparé aux industries traditionnelles. Il est de plus assez difficile d'appliquer les recettes de succès industriels dans la conception informatique. En premier lieu, une application est théoriquement unique. Son déploiement ou encore les processus de production peuvent être industrialisés, mais il ne sert à rien de reproduire la même application 250 fois. Lorsque l'on parle d'industrialisation en matière de développement informatique, on évoque plus généralement la standardisation des procédés et des étapes de développement en équipe. En second lieu, une agence web, un studio de création ou une société de développement informatique (dans une moindre mesure) doivent produire un site ou une application interactive riche dans des délais parfois très courts. Cette phase est donc très souvent occultée ou alors complètement fusionnée à la phase de conception graphique. Ce n'est pas le cas de produits hardware industriels qui peuvent bénéficier de plusieurs années de production.

Il existe toutefois deux phases essentielles, communes au développement informatique et à l'industrie : une première phase de croquis et une seconde de prototypage, qui lancera ou non la mise en production. La moindre bouteille d'eau en plastique est le résultat d'un prototype, lui-même étant le résultat d'un schéma technique ou d'un croquis. Certains logiciels, tels que Catia de Dassault industrie ou Alias Design de la société Autodesk, permettent aux créatifs de s'exprimer tout en avançant les phases de prototypage et d'industrialisation. La souris Arc Mouse de Microsoft a suivi ces étapes avant de pouvoir être produite en masse et commercialisée (voir Figure 10.2).

Figure 10.2

Prototype de la souris
Arc Mouse (source : site
Microsoft Expression).



De nos jours, les phases de croquis et de prototypage sont très rapprochées. Elles sont parfois fondues en une seule grâce à l'outil informatique (cette pratique est encore souvent contestée). L'un des buts poursuivis par SketchFlow est de réduire le temps de conception global. Il entre donc précisément dans la catégorie d'outils liant ces deux phases, mais il permet également de se rapprocher de l'application finale. Jusqu'à quel point le prototype doit se rapprocher ou être transformer en application est une question actuellement en débat. Il facilite ainsi la communication entre les créatifs et les ingénieurs. Quel que soit l'environnement, plusieurs objectifs sont atteints grâce au prototype :

- Il permet de valider les choix de recherche, d'approche et de conception.
- Il valide le bon fonctionnement, l'ergonomie et peut mettre en valeur d'éventuelles intentions de couleurs.
- Il facilite la correction des erreurs de conception.

Tous ces objectifs sont réalisés à travers la communication et l'échange d'idées entre les différentes parties impliquées. Le prototype permet donc d'atteindre un consensus nécessaire à la mise en production. Il faut parfois beaucoup de prototypes avant d'arriver à la version industrialisable d'un objet.

Quelles que soient les parties impliquées dans sa création, financière, technique, artistique, etc., le prototype permet à chacun d'échanger et de communiquer ses impressions, et c'est là sa grande force. Nous allons maintenant voir comment SketchFlow reprend cette philosophie et engendre plus de productivité et des développements de meilleure qualité.

10.1.2 Qu'est-ce que SketchFlow ?

SketchFlow n'est pas un, il est en réalité constitué d'un ensemble d'éléments complémentaires qui ont été introduits dans la version 3 d'Expression Blend. Il est tout d'abord représenté par un type de projet spécifique généré *via* Expression Blend. Visual Studio n'est pas l'outil de prédilection pour prototyper. Cela est logique : la notion de prototype est inhérente à celle de design et Blend est avant tout un outil de designer. De plus, l'objectif est de mettre un outil de prototypage à disposition des profils créatifs et techniques. Le logiciel Visual Studio n'est pas approprié car il demande beaucoup d'expérience et fait avant tout appel à des compétences de codeur. Son interface en termes de design est limitée, comparée à celle de Blend.

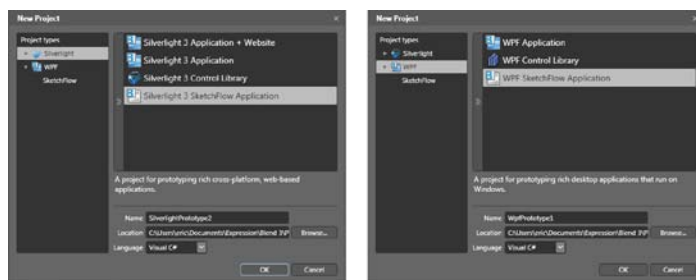
INFO

SketchFlow se veut simple et accessible afin de fédérer les acteurs d'une production autour d'un outil accessible à tous. Vous pouvez également le percevoir comme un enjeu social. Au sein de ce livre, nous avons souvent évoqué le flux de production créatif et technique. SketchFlow est l'un des outils permettant de rassembler les deux mondes trop longtemps séparés dans l'histoire du développement informatique.

Les projets SketchFlow permettent d'accéder à un ensemble de panneaux dédiés. Ils sont inaccessibles au sein de l'interface de Blend dans le cadre de projets standard. Nous verrons leur utilisation tout au long de ce chapitre. Vous pourrez générer des prototypes basés sur SketchFlow aussi bien pour WPF que pour Silverlight (voir Figure 10.3). De ce point de view, il n'y a pas de différences entre les fonctionnalités proposées sur l'une ou l'autre de ces plateformes.

Figure 10.3

Les projets SketchFlow sous Expression Blend pour Silverlight et WPF.



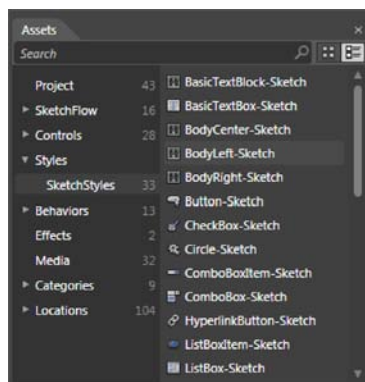
Le prototypage *via* SketchFlow consiste avant tout à concevoir la navigation, l'ergonomie, les transitions et l'expérience utilisateur de manière globale. Les projets SketchFlow contiennent par défaut le style Sketch (croquis) pour les contrôles utilisateur. Ainsi, l'on s'affranchit totalement du design de la charte graphique, qui vient après ces étapes. Cela peut paraître inutile puisque les composants par défaut connaissent les mêmes fonctionnalités. Ce style est pourtant l'un des éléments indispensables constituant SketchFlow. Il permet aux non initiés d'Expression Blend de prendre en main rapidement un projet avec une utilisation minimale du panneau des propriétés.

L'objectif est également de placer l'utilisateur dans une réflexion sur l'ergonomie, l'expérience utilisateur, en mode croquis rapide. L'erreur serait de se focaliser sur la production graphique directe, il vaut mieux rendre cette phase abstraite et moins importante dans un premier temps. Vous trouverez souvent plusieurs styles Sketch pour un même composant. Le contrôle `TextBlock` existe en version aligné à droite, à gauche, titre ou bloc de texte, etc. Ainsi, les adeptes de Blend ne sont plus les seuls à pouvoir mettre en forme une application. Vous pouvez accéder aux styles *via* le panneau Assets (voir Figure 10.4).

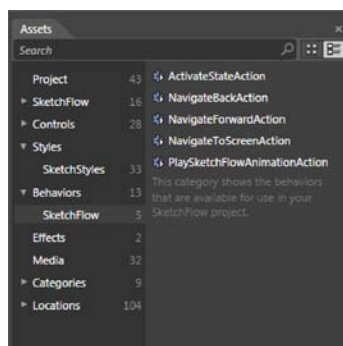
Encore une fois, le but est de permettre au plus grand nombre d'accéder à la création de prototypes. Le code logique est donc par défaut à éviter car seuls les développeurs ou les designers interactifs confirmés peuvent le créer ou le modifier. Un jeu de comportements interactifs spécifiques (Behaviors) est fourni à cette fin (voir Figure 10.5). Ils sont accessibles *via* le panneau Assets et sont utilisables de manière simplifiée grâce à de nouveaux menus contextuels. Ceux-ci permettent de les utiliser sans avoir à les déposer sur un contrôle et à les configurer. La création des prototypes en est grandement accélérée.

Figure 10.4

Le panneau Assets sous Blend donnant, entre autres, accès aux styles.

**Figure 10.5**

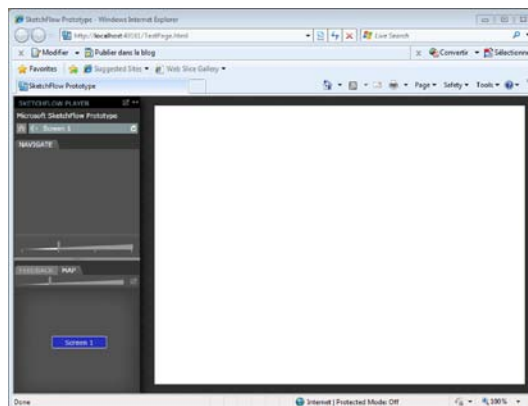
Les comportements spécifiques à SketchFlow.



Pour finir, nous avons évoqué depuis le début la collaboration intermétiers, mais cela ne serait pas possible sans un outil centralisant et permettant à chacun de fournir ses propres impressions. C'est le rôle du lecteur SketchFlow. Lorsque vous testez un projet dans le navigateur, le prototype est affiché au sein de ce lecteur (voir Figure 10.6). Ce dernier donne accès à un certain nombre de fonctionnalités aux utilisateurs de l'application, quels qu'ils soient. Nous étudierons son utilisation à la section 10.3.

Figure 10.6

Le lecteur SketchFlow visible lors de la première compilation.



Vous pouvez finalement remarquer que la compilation ne donne pas accès à l'application finalisée. Produire une application finalisée et optimisée n'est évidemment pas le but de SketchFlow. Vous pourrez toutefois détacher le cœur de votre application grâce aux compétences de développeurs et de designers interactifs, mais ce n'est pas toujours conseillé. Plus l'application que vous développerez sera complexe et moins cette manière de procéder sera viable.

10.1.3 Le flux de production

SketchFlow est utilisable à deux niveaux. Dans un premier temps, les projets sont créés par les acteurs directs de la production, qu'ils soient concepteurs techniques ou artistiques :

- Comme nous l'avons évoqué, prototype et design sont liés. SketchFlow s'adresse donc avant tout aux designers dans un sens large. Les designers d'expérience utilisateur (*UX Designers*) sont la cible la plus évidente. Ces derniers ont à charge de proposer des interfaces intuitives et ergonomiques sans pour autant s'attarder sur le graphisme proprement dit. En second lieu, viennent les designers interactifs qui possèdent une connaissance de Blend et occupent un rôle central dans le flux de production. Ils sont à même d'établir une communication entre les profils techniques et artistiques. Si vous êtes graphiste ou directeur artistique, la prise en main technique simplifiée de SketchFlow a été conçue afin de vous donner un maximum de confort et de productivité.
- Les développeurs d'applications clientes Winforms ou WPF et de RIA pour Silverlight utiliseront rapidement SketchFlow dans leurs futurs projets afin de concevoir les arborescences et écrans de leurs applications. Ils pourront de cette manière acquérir de nouvelles compétences, de prime abord inhérentes aux designers, tout au long des projets.
- Les architectes d'applications ou d'informations, dont le rôle est de proposer des structures organisées et performantes, peuvent également détourner SketchFlow afin de créer des arborescences complexes, faciles à maintenir et à partager.

Sur un tout autre plan, SketchFlow est utilisable par les responsables :

- Les responsables de projets, les directeurs artistiques, les responsables de production peuvent participer pleinement au projet par le biais de leur *feedback*. Pour cela, ils peuvent utiliser le lecteur SketchFlow qui permet de communiquer sur le prototype durant sa création.
- Le client fait partie du processus de création. Avec SketchFlow, il devient facile de trouver un consensus, de proposer plusieurs maquettes fonctionnelles ainsi qu'une direction avant de se lancer dans la production pure et dure.

Vous l'aurez compris, SketchFlow est un outil de prototypage dynamique et transverse. Dynamique, du fait de la souplesse et de l'efficacité qu'il propose pour passer d'une idée à une autre et donner un aperçu en temps réel des décisions prises. Transverse car il est facile d'accès et qu'il favorise la communication intermétiers au sein d'une production. En bref, le prototype est fait pour être testé et approuvé, SketchFlow est donc l'affaire de tous. Attention toutefois à un axiome bien connu des designers : le cœur d'une idée est émis par une personne ou deux, celle-ci peut-être étoffée, mais il faut une direction à tous projets. Quand une application doit plaire à de nombreuses parties et que ces dernières sont décideuses, le risque de perdre la force de l'idée originale est élevé. Il faudra souvent soumettre plusieurs prototypes avant de trouver le juste milieu, celui-ci pourrait toutefois ne pas être si enthousiasmant que cela.

Pour finir, SketchFlow impose d'abord une méthodologie efficace dans le flux de production. C'est un lien direct avec le client. La relation doit être gagnant / gagnant entre prestataire de services et client. Il faut commencer par proposer des maquettes fonctionnelles et ainsi faire vivre le projet. Ce qui avant était figé est maintenant évolutif.

Auparavant plusieurs storyboards étaient dessinés et représentaient le scénario interactif de l'utilisateur. Le client en choisissait un ou deux, émettait des commentaires et on se lançait dans une sorte de "proof of concept" qui prenait non seulement du temps et de l'argent, mais qui en plus n'était peut-être pas validée en fin de parcours. Aujourd'hui, à travers SketchFlow, nous nous appuyons sur les croquis et storyboards pour créer un prototype dont le scénario, la mise en forme et l'interactivité peuvent évoluer dans le temps en un minimum de temps et d'efforts. Le prototype fourni fait en quelque sorte figure de cahier de recette et de contrat autour duquel clients et prestataires peuvent s'accorder.

10.2 Prototype simple

Nous allons maintenant créer un premier prototype simple que nous étofferons au fur et à mesure des notions abordées. Pour réaliser les exercices qui vont suivre, décompressez les fichiers : *chap10/Assets.zip*.

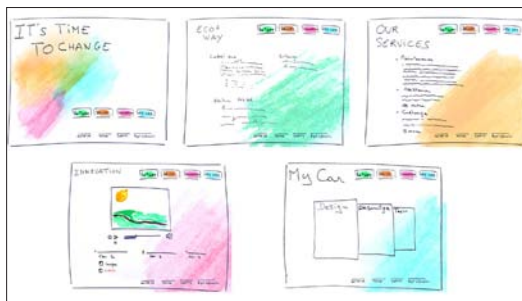
10.2.1 Problématique cliente

Dans tous projets, il faudra proposer des croquis de différentes maquettes répondant à la problématique du client. Ce dernier peut même vous fournir lui-même des croquis des différentes pages de l'application. Nous allons prendre l'exemple d'un concessionnaire de véhicule automobile. Celui-ci souhaite un site vitrine afin de mettre en valeur son savoir-faire et la qualité des voitures vendues. Faisant partie d'un secteur très concurrentiel, il souhaite également se démarquer, en proposant aux visiteurs du site la possibilité de choisir la voiture de leur rêve à travers l'utilisation d'un configurateur riche. Nous détaillerons ce terme à la section 10.5.2. Le configurateur riche doit être accessible au sein du site sous forme d'applications Silverlight. Afin de créer un premier prototype, le client fournit une première ébauche de maquette sous forme de cinq croquis simples que nous avons numérisés, puis retouchés sous Photoshop. Nous avons ajouté un code couleur pour chaque page et menu afin de reconnaître facilement les pages du futur site. Ces croquis représentent l'ensemble des pages du site qu'il souhaite mettre en ligne (voir Figure 10.7).

La première page représente la page d'accueil du site donnant accès à toutes les autres pages. La deuxième liste les critères certifiant la qualité écologique des véhicules. La troisième décrit les prestations diverses et le service après-vente. La quatrième page met en avant l'innovation technologique et l'ergonomie d'utilisation des véhicules les plus récents, grâce à une galerie d'images et à un lecteur vidéo. La cinquième page contient le configurateur riche, celui-ci permettra au visiteur de trouver le véhicule idéal en fonction de critères. Nous allons utiliser SketchFlow pour lui proposer une maquette fonctionnelle du site global, puis nous mettrons l'accent sur le configurateur riche. De ce point de vue, SketchFlow peut vous permettre de prototyper n'importe quel type d'arborescence visuelle.

Figure 10.7

Les cinq pages de croquis retouchées sous Photoshop.



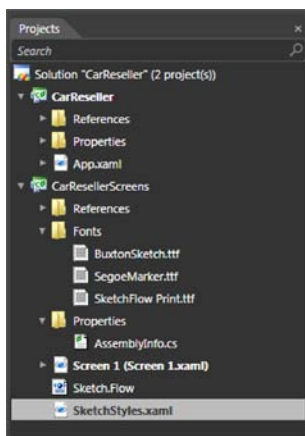
Nous allons maintenant créer une première solution SketchFlow.

10.2.2 Le projet SketchFlow

Ouvrez Expression Blend et créez un prototype nommé `CarReseller`. Vous remarquerez d'emblée l'apparition de nouveaux panneaux, dont nous verrons l'utilisation ultérieurement. Affichez le panneau Project s'il n'est pas visible, vous constatez que la solution est scindée en deux projets distincts (voir Figure 10.18).

Figure 10.18

Arborescence du projet SketchFlow CarReseller.



Le premier projet correspond au lecteur SketchFlow lui-même, il porte le nom original que vous avez défini (`CarReseller`). Le second contient le prototype, c'est le même nom suffixé du mot `Screens`. C'est dans ce dernier que vous travaillerez. Vous remarquez plusieurs différences avec les applications standard. Le second prototype possède tout d'abord un répertoire `Fonts` contenant trois polices utilisées par le style `Sketch` (croquis). Ce style est un élément important des projets SketchFlow, il est fourni par le fichier `SketchStyles.xaml` présent à la racine du projet. Ce type de fichier est assez nouveau pour nous. Il s'agit en fait d'un dictionnaire de ressources qui permet d'externaliser et de partager certains types de ressources utiles à un ou plusieurs projets.

INFO

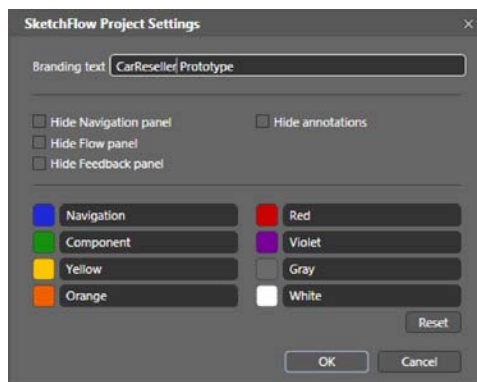
Il est facile de réutiliser le style Sketch au sein d'un projet standard. Il vous suffira d'ajouter le fichier `SketchStyles.xaml`, ainsi que les polices du répertoire `Fonts` à votre projet. Les styles définis dans le dictionnaire de ressources utilisent ces polices, c'est pourquoi il ne faut pas les oublier.

Comme nous l'avons déjà précisé, ce type de projet contient des comportements (Behaviors) spécifiques à SketchFlow. Attendez-vous donc à trouver des références à de nouvelles bibliothèques dynamiques dans le répertoire `References` du projet. Nous n'avons pas réellement besoin de décrire le rôle de chaque bibliothèque, mais celles-ci sont indispensables au bon fonctionnement du projet.

Le fichier `Sketch.Flow` est le dernier que nous évoquerons. Vous pouvez l'ouvrir sous l'application `NotePad`, par exemple. Il s'agit d'un fichier au format XML, qui décrit une partie du travail réalisé dans le prototype (écrans, transitions, etc.), ainsi que d'un ensemble de paramètres propres au projet, qu'il peut être utile de personnaliser (comme les couleurs d'écran par exemple). Vous pouvez modifier ce fichier de deux manières différentes, soit directement avec un éditeur de texte, soit en passant par le menu `Project`, puis en cliquant sur `SketchFlow Project Settings...` Une boîte de dialogue vous permettra de modifier quelques-unes des options présentes dans le fichier XML (voir Figure 10.9).

Figure 10.9

Options propres au projet SketchFlow en cours.



Vous allez modifier certains des paramètres afin de travailler plus confortablement. Comme nous l'avons précisé, le site est créé avec un code à quatre couleurs. Elles vont nous être utiles pour générer une carte de navigation avec ce code couleur. Si vous préférez modifier le fichier, vous pouvez remplacer les quatre dernières couleurs définies au sein de la balise `VisualTags` par celles exposées ci-dessous :

```
<VisualTags>
...
<VisualTag>
  <Name>OurServices</Name>
  <Color>FFFF9711</Color>
</VisualTag>
<VisualTag>
  <Name>MyCar</Name>
  <Color>FF48FEFF</Color>
</VisualTag>
<VisualTag>
```

```

    <Name>Eco2Way</Name>
    <Color>FF2EF872</Color>
  </VisualTag>
  <VisualTag>
    <Name>Innovation</Name>
    <Color>FFF93CA5</Color>
  </VisualTag>
</VisualTags>

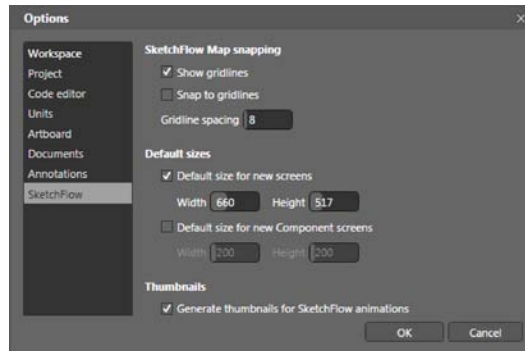
```

Sauvegardez le fichier, puis retournez dans Blend. Celui-ci a détecté la modification et vous propose de recharger le fichier Sketch.Flow. Dans la boîte de dialogue, cliquez sur OK ; si tout ce passe bien le fichier est rechargé proprement. Si les nœuds XML sont mal formatés, Blend vous proposera de recréer un nouveau fichier. Nous verrons le résultat de cette opération dans très peu de temps. Il reste encore un paramètre à régler avant de commencer notre prototype.

Chaque écran correspondra à un croquis fourni par le client, l'idéal serait que chaque nouvel écran que nous générerons possède, par défaut, des dimensions identiques à celles des croquis. Il existe plusieurs manières de procéder. Vous pouvez, par exemple, définir des dimensions d'écran par défaut pour l'ensemble des projets SketchFlow. Dans la barre du haut, choisissez Tools > Options... Dans la fenêtre qui s'affiche, sélectionnez l'onglet SketchFlow. Une série d'options propres à ce type de projets est proposée. Cochez l'option Default size for new screens, puis définissez une largeur de 660 pixels et une hauteur de 517 pixels. Ces dimensions correspondent aux croquis réalisés par le client et retouchés par nos soins. Tous les écrans que nous générerons posséderont par défaut ces dimensions (voir Figure 10.10).

Figure 10.10

Options de mise en pages propres à l'ensemble des projets SketchFlow.



Cette méthode présente un désavantage : si vous travaillez avec de nombreux prototypes aux dimensions différentes, ce réglage n'est pas vraiment pertinent et vous devrez le changer régulièrement. Vous pouvez également utiliser la carte de navigation représentée par le panneau SketchFlow Map. Cliquez sur Screen 1. Dans l'arbre visuel, sélectionnez le UserControl racine et affectez-lui une largeur (Width) de 660 pixels et une hauteur (Height) de 517 pixels. Faites ensuite un clic-droit sur Screen 1 dans la carte, puis cliquez sur l'option Set As default Navigation Screen Size. Cette option modifie les valeurs de l'une des balises XML contenues dans le fichier Sketch.Flow. Lorsque vous concevrez de nouveaux écrans, ceux-ci auront par défaut des dimensions correspondantes à l'écran d'origine. L'avantage de ce réglage est d'être lié aux propriétés du projet. Il est donc rechargé par défaut lorsque vous ouvrirez à nouveau le projet dans Blend.

10.2.3 La carte de navigation

Nous allons générer un premier prototype grâce au panneau SketchFlow Map. Cette fenêtre représente le cœur du travail de prototypage au sein de Blend (voir Figure 10.11).

Les prototypes sont constitués de différents écrans. Chacun d'eux représente une interface utilisateur. Ce panneau permet de gérer l'arborescence et l'enchaînement logique des interfaces utilisateur de manière globale. Celui-ci donne également accès à la gestion des transitions animées.

Figure 10.11

Le panneau SketchFlow Map affiche la carte de navigation du projet.

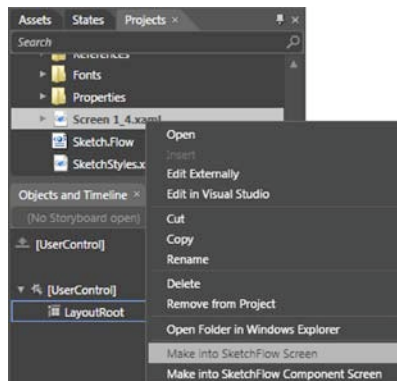


En bas du panneau se trouve une série d'outils sous forme d'icônes. Voici leur fonctionnalité de gauche à droite :

- Vous bénéficiez tout d'abord d'un zoom équivalent à celui présent dans le panneau de création et fonctionnant de manière identique.
- Les deux icônes représentant des flèches (↶ ↷) permettent d'annuler ou de refaire une action SketchFlow. Celles-ci sont particulièrement utiles car SketchFlow génère un couple de fichiers XAML et C# pour chaque nouvel écran créé. L'annulation classique ne fonctionnera donc pas toujours puisque SketchFlow manipule une arborescence de fichiers. Si vous avez créé un écran par erreur, utilisez ces actions pour revenir sur vos pas.
- Les troisième (➕) et quatrième icônes (➖) servent à ajouter un nouvel écran ou un nouveau composant d'écran. Nous reviendrons sur le principe des composants d'écran par la suite.
- Le cinquième pictogramme (✖) efface un écran sélectionné au sein de la carte SketchFlow, quel que soit son type (composant d'écran ou écran). Lorsque vous effacez un écran, le couple de fichiers qui lui est associé n'est pas réellement supprimé du projet. Si vous souhaitez réellement supprimer ces derniers, il vous faudra utiliser le panneau Projects. À l'opposé, vous pouvez définir un couple XAML/C# en écran SketchFlow (voir Figure 10.12).
- Les deux boutons suivants (🔍 🔍) vous permettent de zoomer, soit sur la totalité de la carte, soit sur les écrans en cours de sélection. Ces deux derniers sont vraiment très pratiques pour accéder aux écrans que vous souhaitez modifier. Vous pouvez également maintenir la barre d'espace et le bouton gauche de la souris appuyés pour vous déplacer dans la carte. La molette de la souris est également prise en compte pour le zoom.
- Vous pouvez diminuer l'opacité des liaisons de navigation ou de composants qui ne sont pas sélectionnées grâce aux deux dernières icônes : (🔍 🔍). Cela permet d'y voir un peu plus clair lorsque votre carte commencera à s'étoffer.

Figure 10.12

Ajouter un couple XAML / C# comme écran SketchFlow.



Tous ces outils vous permettent de créer une carte de navigation assez simplement. Blend propose toutefois d'autres options qui sont directement accessibles lors du survol d'un écran ou d'une liaison de la carte. Dans ce cas, un menu apparaît en dessous de l'écran survolé (voir Figure 10.13).

Figure 10.13

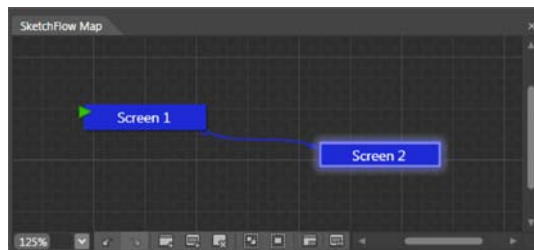
Menu déroulant affiché au survol d'un écran.



Il existe deux manières de créer des écrans. La première consiste à utiliser l'icône appropriée dans le menu bas de la fenêtre. En procédant ainsi, l'écran généré n'est relié à aucun autre par défaut. La deuxième façon de procéder consiste à employer les icônes du menu déroulé au survol. Surveillez l'écran nommé Screen1, maintenez le bouton gauche de la souris enfoncé sur la première icône, puis déplacez la souris pour générer un deuxième écran. Positionnez-le légèrement à l'extérieur. Relâchez le bouton gauche. Le nouvel écran est définitivement créé et celui-ci est relié à l'écran d'origine par une transition représentée par une courbe directionnelle (voir Figure 10.14).

Figure 10.14

Transition directionnelle de Screen1 vers Screen2.

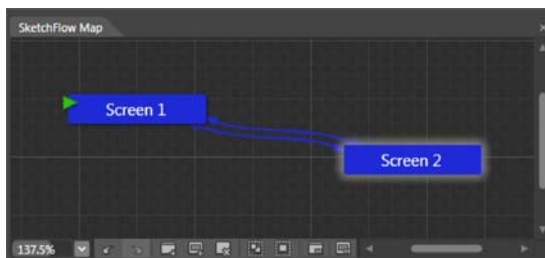


La courbe indique que l'utilisateur pourra naturellement passer de l'écran 1 (Screen1) à l'écran 2 (Screen2). La ligne représente également la transition animée qui aura lieu. Gardez toutefois à l'esprit que cette liaison ne représente pas la fonctionnalité en propre qui permettra de passer d'un écran à l'autre, mais simplement un lien de navigation logique et une transition. D'un seul coup d'œil sur une carte SketchFlow, vous devez être capable de traduire un scénario d'utilisation. Vous remarquez d'ailleurs que la courbe possède une direction, il est donc logique pour l'utilisateur de

passer de Screen1 à Screen2, mais pas forcément de Screen2 à Screen1. Une animation de transition sera jouée dans le premier cas, mais pas dans le second. Créez une seconde liaison au survol de Screen2 en cliquant sur la seconde icône et ciblez Screen1 (voir Figure 10.15).

Figure 10.15

*Transition
bidirectionnelle
entre Screen1
et Screen2.*



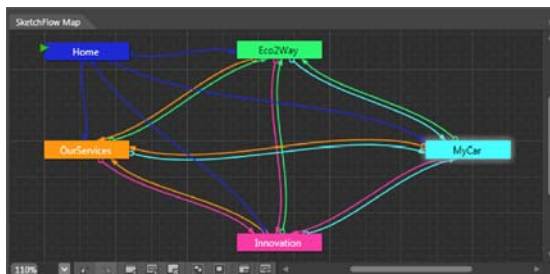
INFO

Screen1 possède une icône représentant une flèche verte, qui indique que Screen1 est l'écran de démarrage du prototype. Vous pouvez décider à tous moment de changer l'écran de démarrage *via* un clic-droit sur celui de votre choix au sein de la carte. Un menu contextuel vous permet de redéfinir cet écran comme écran initial.

Pour définir une transition animée personnalisée, vous pouvez cliquer-droit sur la courbe de transition et redéfinir l'option Transition Style. Par défaut, celle-ci est un fondu enchaîné assez efficace. Vous pouvez conserver le fondu actuel. Nous allons créer une première carte de navigation. Comme nous concevons un site Internet, nous partons du principe que l'utilisateur peut naviguer entre chaque page. Nous définirons donc une transition entre chaque écran, mise à part la page Home qui ne sert qu'à atteindre les autres. Ce n'est qu'une page d'accueil ne contenant aucune information importante. Elle pourrait contenir une vidéo ou une animation d'introduction plein écran. L'utilisateur, une fois sur le site, n'aura théoriquement pas besoin de revenir sur la page Home. Nous avons tout de même prévu un bouton à cet effet, dans le pied de page de notre site. Une transition ne serait pas forcément utile. Créez une carte de navigation (voir Figure 10.16).

Figure 10.16

*Ébauche de la carte
de navigation.*



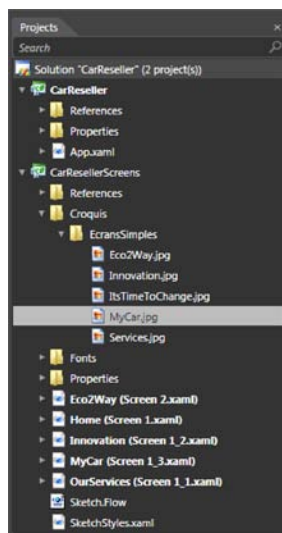
INFO

Afin de vous aider à mieux visualiser la carte, vous pouvez modifier la couleur de fond de chaque écran. Dans le menu déroulant au survol des écrans (la dernière icône de couleur) ou lors d'un clic-droit, vous trouverez l'option Change Visual Tag. Comme nous avons modifié la palette de couleur définie au sein du fichier Sketch.Flow, vous avez à disposition un code couleur correspondant à chaque page du site.

Décompressez le fichier au format zip téléchargé précédemment, à l'emplacement de votre choix sur votre disque dur. Dans le répertoire Assets décompressé, vous trouverez le répertoire FlatScreens contenant plusieurs images au format jpg. Au sein de Blend, créez un nouveau répertoire nommé Croquis dans le projet CarResellerScreens, puis un deuxième répertoire à l'intérieur nommé EcransSimples. Cliquez-droit sur le répertoire EcransSimples et choisissez l'option Add Existing Item... Sélectionnez toutes les images contenues dans le répertoire FlatScreens décompressé, puis cliquez sur OK. Vous venez d'importer toutes les images du répertoire d'un seul coup (voir Figure 10.17).

Figure 10.17

Importation des croquis au format jpg.



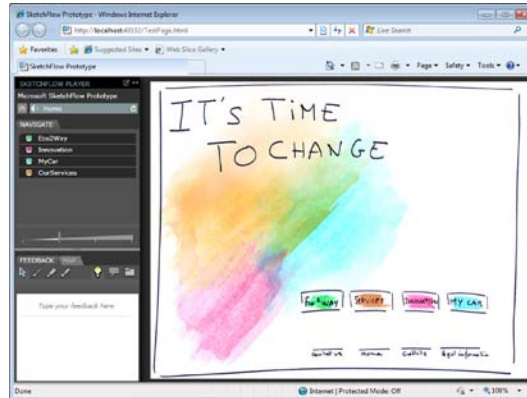
Le fichier nommé *ItsTimeToChange.jpg* correspond au visuel de la page Home. Les autres fichiers sont explicitement nommés. Dans la carte de navigation, double-cliquez sur la page *Eco2Way*. La fenêtre de design affiche la page, celle-ci est vide par défaut. Elle possède exactement la même structure que les pages d'applications Silverlight standard. Dans le panneau Projects, double-cliquez sur *Eco2Way.jpg*. Vous venez à l'instant de placer l'image dans l'écran *Eco2Way*. Procédez de même avec tous les autres écrans. Veillez aux dimensions du *UserControl* racine, celui-ci doit posséder des dimensions de 660 pixels de large par 517 de hauteur. Vous venez de créer un premier prototype simple, Sauvegardez tous les fichiers modifiés *via* le menu File ou le raccourci Ctrl+Maj+S. Compilez ensuite le projet pour voir le prototype dans le lecteur SketchFlow. Vous découvrirez ses fonctions dans la prochaine section. L'exercice corrigé est dans : *chap10/CarReseller_Simple.zip*.

10.3 Le lecteur SketchFlow

Le lecteur SketchFlow donne à vos collègues et à toute personne impliquée dans le projet la capacité de communiquer et de commenter le prototype. Lorsque vous avez compilé le projet, il a été embarqué dans le lecteur. Le navigateur affiche donc celui-ci ainsi que le prototype qu'il intègre en son sein. Le lecteur lance par défaut l'écran de démarrage que vous avez défini dans Blend (voir Figure 10.18).

Figure 10.18

Chargement de la page de démarrage au sein du lecteur SketchFlow.



Nous allons maintenant découvrir et utiliser les fonctionnalités du lecteur.

10.3.1 Navigation

La fenêtre de gauche nommée SketchFlow Player centralise toutes les fonctionnalités. Nous allons les décrire en partant du haut vers le bas. Elle donne tout d'abord accès à un mininavigateur : l'icône représentant une maison (🏠) permet à tout instant de revenir à l'écran de lancement. Les flèches retour et avant permettent de revenir sur le cheminement que vous avez suivi.

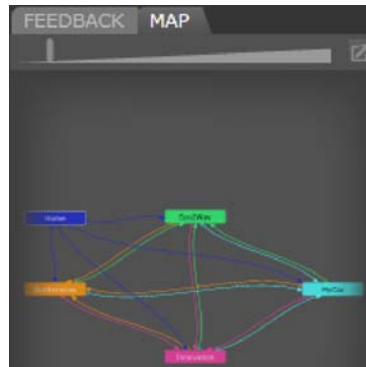
Vous remarquez une barre d'adresse indiquant l'écran dans lequel vous vous situez. Il n'est pas possible d'entrer manuellement une adresse. Rafraîchir la page actuelle s'effectue par un clic gauche sur la dernière icône (🔄). À chaque fois que vous naviguez dans un écran, l'onglet Navigate met à jour la liste des pages disponibles à partir de celle en cours. Cette liste est basée sur les transitions définies dans la carte sous Blend. Il est ainsi impossible d'utiliser cet onglet pour atteindre un écran qui ne serait pas lié par une transition. La direction de la transition est prise en compte pour son affichage.

La page Home possède une transition vers l'ensemble des autres écrans de l'application, ceux-ci sont donc tous présents dans la liste. Dans la liste, cliquez sur l'écran Eco2Way. La transition de fondu enchaîné est jouée, la liste est mise à jour, mais la page Home n'y figure pas. C'est tout à fait logique puisqu'aucune transition ne cible cet écran.

Pour finir, vous bénéficiez d'une réglette vous permettant de zoomer l'affichage du prototype en cours. Si vous avez un grand écran, cela devient vite indispensable car les croquis sont souvent sommaires et donc en faible résolution ; les agrandir est dans ce cas très pratique. Nous étudierons le panneau FEEDBACK à la section 10.3.2. L'onglet MAP autorise une navigation libre de toute transition (voir Figure 10.19).

Figure 10.19

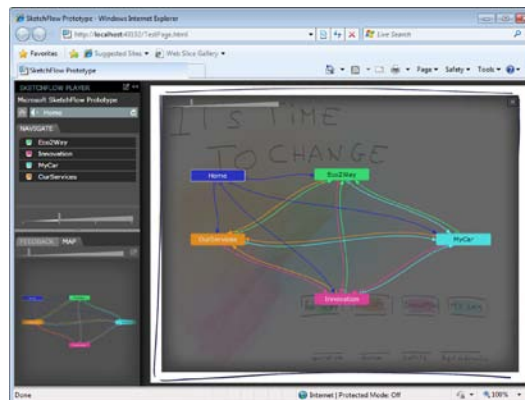
L'onglet MAP au sein du lecteur.



Lorsque vous double-cliquez sur l'un des écrans, il s'affiche à droite. Si une transition est définie entre l'écran initial et celui d'arrivée, elle est jouée. Dans le cas contraire, la page est simplement affichée de manière brute. Vous avez également la possibilité de zoomer dans la carte de navigation *via* la réglette située au-dessus. Vous aurez toutefois quelques difficultés à zoomer sur une zone lorsque les cartes de navigation seront complexes car l'onglet MAP n'est pas redimensionnable. Pour faciliter la navigation, la carte peut également être affichée en surimpression de l'application prototypée (voir Figure 10.20). Il suffit pour cela de cliquer sur l'icône d'agrandissement (🔍).

Figure 10.20

La carte de navigation en surimpression.



Nous venons de passer en revue les fonctionnalités de navigation. Vous remarquez qu'il n'est nul besoin pour l'instant de créer de réelles interactions utilisateur pour passer d'un écran à l'autre. Cela est très pratique et permet de se concentrer sur l'enchaînement des écrans. Nous aborderons les interactions utilisateur à la section 10.4.

10.3.2 Collaboration transverse avec SketchFlow

L'un des enjeux les plus importants de ces dernières années est certainement la communication et la collaboration intermétiers. C'est une problématique qui a toujours existé. Celle-ci passe aujourd'hui au premier plan du fait de la nécessité grandissante d'échanger et de partager les idées et contraintes de production entre créatifs et techniques. Si le XAML est l'un des moyens techniques permettant cette communication, SketchFlow en est bien l'héritier et l'un des plus puissants le-

viens œuvrant en ce sens. Les onglets Feedback présents à la fois dans le lecteur et dans Blend permettent à l'ensemble des acteurs de communiquer autour du prototype. Même si ces derniers ne fourmillent pas de milliers d'options, celles qui y sont présentes ajoutent une dimension supplémentaire et engendrent ainsi de nouvelles manières de produire.

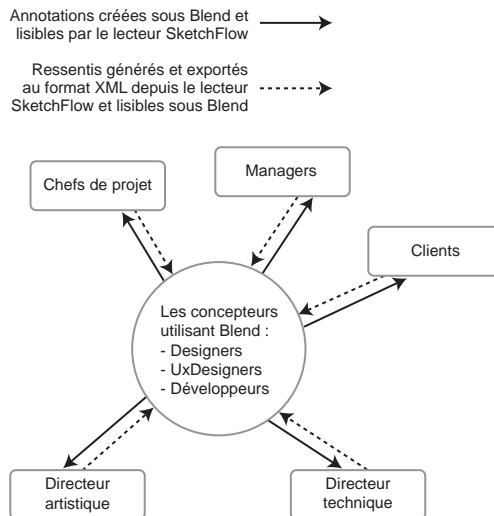
10.3.2.1 Annotation versus retour utilisateur

Dans tout projet, vous trouverez quatre types de feedback. Le premier vient toujours du client, puisque vous répondez avant tout à une problématique client et sans ce dernier, pas de projet. Ses critiques sont donc prépondérantes. Le deuxième est fourni par les chefs de projet, les directeurs technique ou artistique, les managers, ils ont une vision globale de la situation. L'utilisateur final a aussi son mot à dire. Il n'y a pas pire qu'une application qui n'est pas avant tout pensée pour l'utilisateur final et testée par ce type de public. Le diable est dans les détails, l'utilisateur final est souvent le critique le plus difficile sur un moyen terme. Pour finir, vous pouvez émettre des commentaires en tant que concepteur du prototype, que vous soyez développeur ou designer.

Les communications s'effectuent dans deux sens différents. D'une part, les ressentis utilisateur peuvent être adressés aux concepteurs, d'autre part les concepteurs peuvent faire remonter des informations et des remarques, ou encore exposer des problématiques imprévues (voir Figure 10.21).

Figure 10.21

Principe d'échanges de feedback utilisateur et d'annotation concepteur.



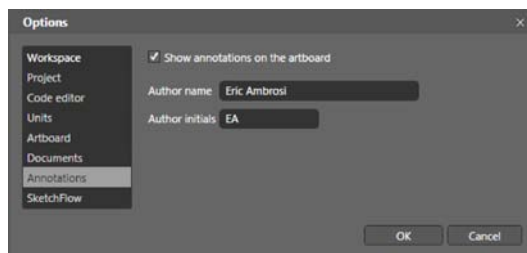
Les trois premiers types de feedback concernent les acteurs qui émettront des commentaires d'un point de vue utilisation. Ils navigueront dans le prototype *via* le lecteur SketchFlow et pourront partager leurs ressentis sous forme de fichiers XML. Ces fichiers sont importés et affichables directement au sein de Blend. À un autre niveau, les concepteurs du prototype peuvent créer des annotations au sein d'Expression Blend qui seront ensuite lisibles par le lecteur SketchFlow, lors de la navigation test.

Nous allons endosser le rôle du concepteur et créer des annotations. Au sein de Blend, sélectionnez l'écran MyCar. Vérifiez ensuite que l'affichage des annotations est correctement activé. L'icône de la bulle d'information (🗨️) en bas de la fenêtre de création vous permet d'activer ou de

désactiver l’affichage des annotations. Lorsque vous annotez un projet, vous utilisez un identifiant sous forme d’initiales. C’est exactement le même principe que sous Word ou Excel. Pour paramétrer l’identifiant, vous devez ouvrir le menu Tools (outils), sélectionner Options..., puis choisir l’onglet Annotations (voir Figure 10.22).

Figure 10.22

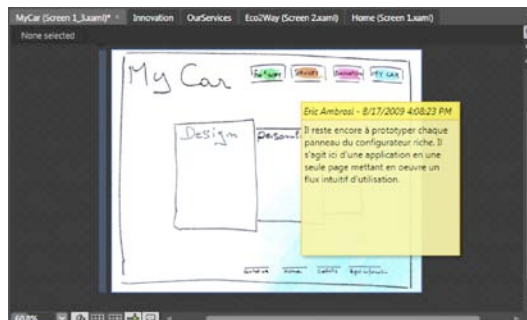
Paramétrage de l’onglet Annotations.



Pour créer un nouveau commentaire, utilisez le raccourci Ctrl+Maj+T. Vous pouvez également ouvrir le menu Tools, puis cliquer sur Create Annotation (voir Figure 10.23).

Figure 10.23

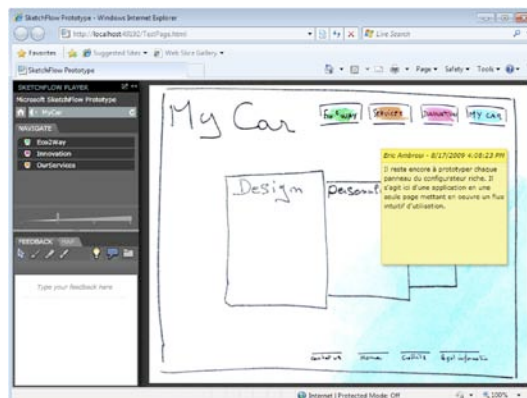
Une annotation dans l’écran MyCar.



Compilez la solution au sein du lecteur SketchFlow, sélectionnez la page MyCar, puis cliquez sur l’onglet FEEDBACK. Vous y trouverez une icône permettant l’affichage des annotations (). Lorsque vous la survolez, les annotations, s’il y en a, sont affichées temporairement. Cliquez-les pour activer leur affichage tout au long de la navigation (voir Figure 10.24).

Figure 10.24

Afficher les annotations.



Comme vous le constatez, les annotations sont assez simples à créer et à consulter. Laissez le navigateur ouvert, nous allons utiliser le lecteur SketchFlow afin de créer quelques notes du point de vue responsable de production, client ou utilisateur final. Il existe deux manières de générer un commentaire dans le lecteur. Soit vous créez une ou plusieurs notes associées pour chaque page, soit vous dessinez directement sur la page *via* les outils proposés par SketchFlow. Le panneau FEEDBACK contient à cet effet trois outils : un stylo et un surligneur dont vous pouvez régler l'épaisseur et la couleur ainsi qu'un correcteur. Pour ajouter une note de page, il suffit de cliquer sur le texte grisé *Type your feedback here* (voir Figure 10.25).

Il suffit maintenant d'exporter les croquis et commentaires sous forme d'un fichier .feedback au format XML. Cliquez sur l'icône représentant un répertoire. Deux options s'offrent à vous : vous pouvez supprimer tous vos commentaires par une réinitialisation ou les exporter. Choisissez l'option *Export Feedback...* Vous devez fournir votre nom ainsi que vos initiales. Nous nous mettrons dans la peau de notre chef des projets spéciaux, Nicolas. Il décide d'exporter ses commentaires pour nous les envoyer ensuite par e-mail, il crée donc un fichier nommé *Nicolas.feedback* (voir Figure 10.26).

Figure 10.25

Ajouter un feedback utilisateur.

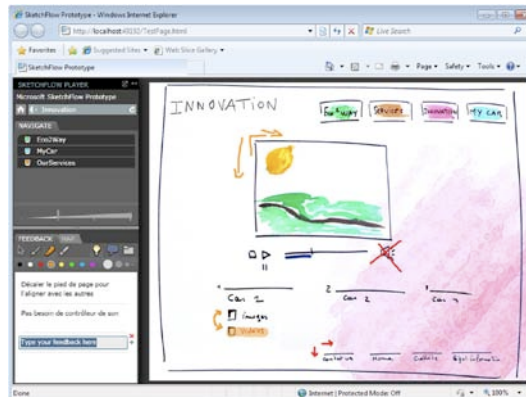


Figure 10.26

Un exemple de fichiers feedback.



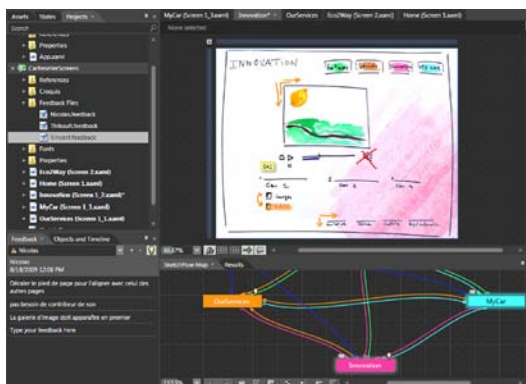
Le contenu de ce fichier est récupérable à tout instant. Pour cela, vous devez cliquer sur l'icône d'ajout depuis le panneau Feedback. Puis, sélectionnez le fichier .feedback. Vous pouvez importer plusieurs fichiers de ce type mais n'en visionner qu'un seul à la fois (voir Figure 10.27). Ces

fichiers sont automatiquement recopiés dans votre projet dans le répertoire Feedback Files. Si ceux-ci ont été générés alors que vous avez modifié le projet entre temps, Blend vous signale qu'ils peuvent être obsolètes. Dès cet instant, il vous appartiendra de déterminer la fiabilité de ces remarques. Lorsque des annotations ou des remontées utilisateur existent dans un écran, des icônes indiquent leur présence au-dessus de cet écran dans la carte de navigation (voir Figure 10.27). Vous pouvez activer ou désactiver l'affichage des commentaires utilisateur en cliquant sur l'icône en haut à droite du panneau Feedback (💡).

Vous connaissez maintenant les outils facilitant la collaboration et le travail de prototypage en équipe. Il reste toutefois quelques zones d'ombres concernant l'accès au prototype lui-même.

Figure 10.27

Visualiser les remontées utilisateur dans Expression Blend.



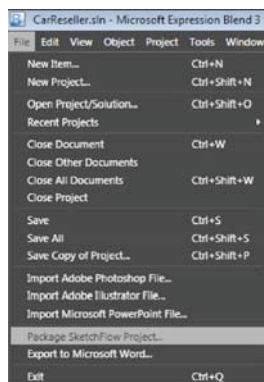
10.3.2.2 Partager un projet SketchFlow

Notre problématique à ce stade est assez simple. Nous pouvons créer des retours utilisateur parce que nous sommes détenteur du projet Blend. Nous pouvons compiler le projet à tout instant et avoir accès au lecteur ainsi qu'à la sauvegarde des commentaires. Toutefois, le partage des retours ne concerne pas réellement le concepteur, mais plutôt les directeurs de production, le client et l'utilisateur final. Il vous faut donc partager le projet SketchFlow sur un serveur web ou un espace disque accessible. Blend vous fournit une manière simple de gérer ce type de problématique et crée pour vous le site web complet contenant le prototype et le lecteur SketchFlow qui l'embarque. Il vous suffit simplement d'ouvrir le menu File, puis de cliquer sur l'option Package SketchFlow Project... (voir Figure 10.28).

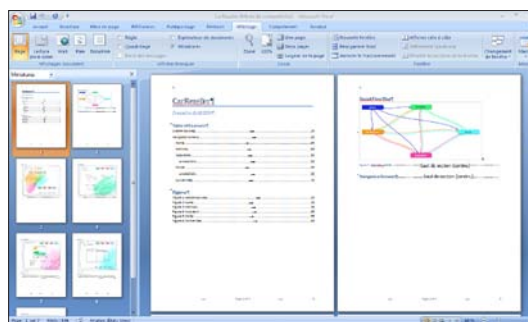
Choisissez le dossier qui recevra le projet compilé. Il vous suffira ensuite de télécharger ces fichiers sur le serveur web de votre choix *via* une connexion FTP ou SFTP. Vous n'avez rien d'autre à faire qu'attendre la réception des fichiers feedback par e-mail ou par tout autre moyen. Vous avez également la possibilité de créer un fichier au format Word contenant une description complète du prototype. Choisissez alors le menu Export to Microsoft Word... Vous obtiendrez un document comme montré à la Figure 10.29.

Figure 10.28

Mettre en forme un projet SketchFlow pour le partage.

**Figure 10.29**

Résultat d'une exportation au format Word.



Ce type de document est surtout avantageux lorsque les diverses parties se sont accordées sur un prototype. Vous pouvez le considérer comme un cahier des charges macroscopique que vous pouvez à tout instant consulter et modifier. Il pourra même vous être utile pour rédiger un cahier de recettes validant l'application livrable.

INFO

Un cahier de recettes est souvent rédigé lorsque les projets atteignent une certaine dimension en tout début de production. Il concentre toutes les informations propres à l'application livrable et à ses fonctionnalités, dans sa version finale. Ainsi, lors de la remise du projet au client, celui-ci peut recenser l'ensemble des fonctionnalités initialement prévues dans le cahier de recettes afin d'éviter tout oubli. Le cahier des charges est quant à lui dédié aux équipes techniques, il découle des impératifs et du livrable. Son contenu est utile tout au long de la production et guide les équipes techniques.

Dans les prochaines sections, nous allons grandement améliorer notre prototype et lui donner vie. Vous n'êtes pas obligé de connaître entièrement le logiciel Blend pour créer des prototypes. C'est là l'une des grandes forces de SketchFlow.

INFO

Il est possible de convertir un projet SketchFlow en projet de production et ainsi de le détacher du lecteur Silverlight. Les étapes sont différentes selon le langage et la plateforme utilisée, Silverlight ou WPF. Elles sont décrites dans la documentation accessible *via* l'interface d'Expression Blend.

10.4 Interactivité

Jusqu'à maintenant, l'utilisateur ne peut naviguer dans le prototype qu'à travers l'utilisation des fonctionnalités fournies par le lecteur SketchFlow. Cela est vraiment utile et évite de compliquer inutilement la conception du prototype dans la première phase de réflexion. Toutefois, lorsque vous aurez trouvé un consensus sur ses grandes lignes, il est possible de donner au client un aperçu de l'expérience utilisateur finale en lui appliquant une couche d'interactivité. Au sein de SketchFlow, l'interactivité utilisateur se traduit de deux manières différentes. Vous pouvez, comme dans toutes les autres applications, créer du code logique ou glisser des comportements interactifs sur les instances d'`UIElement`. Leur nombre est plus important dans ce type de projet et ils sont plus simples d'utilisation. Le prototype acquiert ainsi plus de profondeur et concrétise une partie de la conception. D'une manière complètement différente, vous pouvez simuler des interactions utilisateurs *via* le panneau SketchFlow Animation. Cette fonctionnalité complètement nouvelle peut être appréhendée de différentes manières. Nous étudierons son fonctionnement à la section 10.4.3.

10.4.1 Importer des fichiers PSD

Depuis sa version 3, Expression Blend rend possible l'importation de fichiers Photoshop (.psd). La répartition et la composition de visuels par le biais de calques confère aux fichiers .psd de nombreux avantages. Bien qu'il ne soit pas le seul dans ce cas, Photoshop est ainsi utilisé depuis des années dans le Web afin de maquetter de nombreux sites XHTML ou Flash. Désormais, les applications Silverlight peuvent être entièrement conçues graphiquement, puis être intégrées par les designers interactifs ou les développeurs dans un deuxième temps. Contrairement à ce principe, et à juste titre, Microsoft place la phase de croquis avant celle de la conception graphique pure et dure. L'utilisation de Photoshop de ce point de vue reste d'actualité. Il est très simple de scanner vos croquis, puis de les retoucher et de les organiser en calques dans un format .psd. Nous allons importer ce type de fichiers pour donner de la profondeur à notre application. Nous pourrions de cette manière séparer et gérer les éléments interactifs individuellement.

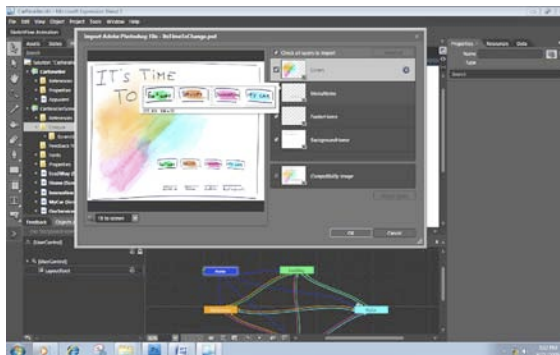
INFO

L'importation de fichiers Illustrator est également possible depuis la version 3 de Blend. Illustrator est avant tout utilisé pour concevoir le visuel finalisé, ainsi son emploi concerne moins le maquetage et le prototypage. Nous aborderons ce type d'importation au Chapitre 11.

Supprimez tous les composants Image au sein des écrans de la carte de navigation ainsi que le dossier EcransSimples. Dans le répertoire Assets décompressé, vous trouverez le répertoire PSD contenant les mêmes écrans, mais au format Photoshop et organisés en plusieurs calques. Dans la carte de navigation SketchFlow, double-cliquez sur la page Home. Dans le panneau Project, sélectionnez le répertoire Croquis. Ouvrez le menu File, puis l'option Import Adobe Photoshop File... Accédez au répertoire PSD et choisissez le fichier `ItsTimeToChange.psd`. Une fenêtre d'importation est affichée au premier plan, elle vous permet de gérer l'importation de ce type de fichiers (voir Figure 10.30).

Figure 10.30

Fenêtre d'importation
des fichiers Photoshop.



La structure du fichier est entièrement affichée. Il faut veiller à cocher l'option *Check all layers to import*. Lorsque le contenu d'un calque est caché dans le fichier .psd, il ne sera importé que si cette option ou la case à cocher adjacente sont cochées. Validez l'importation du fichier. Comme vous avez sélectionné l'écran *Home*, le contenu du fichier est directement intégré à la page. Si l'agence-ment et les marges sont sauvegardés, la transparence subira parfois quelques modifications chan-geant légèrement le visuel importé. Cela n'a que peu d'importance pour nous car nous travaillons sur un prototype. Lors de l'importation, chaque calque est transformé sous forme d'image au format png. Pour le vérifier, vous pouvez déplier le répertoire *ItsTimeToChange_Images* créé lors de l'importation. Vous y trouverez les images png du visuel final.

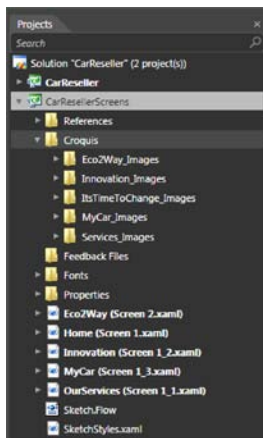
INFO

Pour obtenir un résultat visuel fidèle au document d'origine, il faut éviter toutes les spécificités propres au format psd. Rastérisez chaque calque, la plupart des effets ne sont pas conservés (seuls les projections d'ombre et le flou le sont). Il est également plus efficace de recréer les textes sous *Blend* ou de pixeliser ces derniers s'ils ne sont pas interactifs. Toutes les options de fusion propres aux calques Photoshop sont également à proscrire. Suivez cette logique et vous n'aurez pas de difficultés lors de l'importation.

Suivez les étapes décrites ci-dessus pour chaque écran du prototype. Veillez bien à sélectionner le dossier *Croquis* afin d'organiser proprement le projet (voir Figure 10.31).

Figure 10.31

Le répertoire généré
ItsTimeToChange_Images.

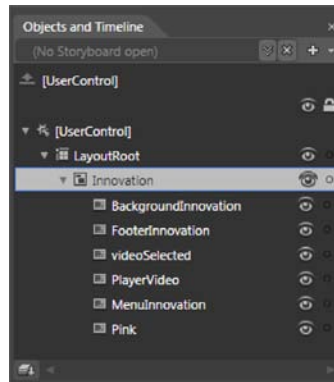


Dans l'arbre visuel de chaque écran, vous remarquez qu'un conteneur de type Canvas a été généré. Son nom correspond à celui de l'image importée. En son sein, plusieurs contrôles de type Image ont été créés et font référence à une image png. Ils représentent les calques d'origine et possèdent en conséquence un nom en correspondance avec chacun d'eux (voir Figure 10.32).

Comme vous le constatez, le nommage des calques et des objets est crucial. C'est un moyen simple et efficace pour faciliter la communication entre chaque pôle métier. Il n'y a qu'à lire pour comprendre l'utilité de chaque calque. Nous allons maintenant profiter du découpage des calques pour générer une interactivité utilisateur propre à chaque écran.

Figure 10.32

L'arbre visuel de l'écran Innovation.

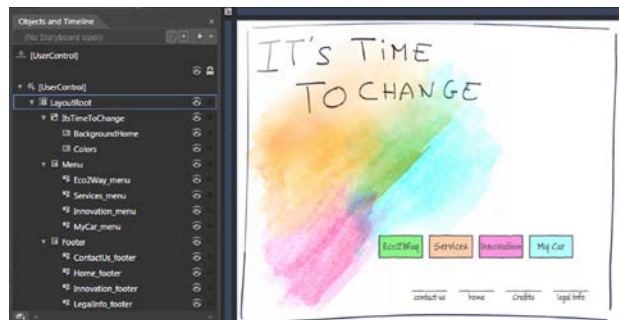


10.4.2 Navigation utilisateur

Afin de permettre à l'utilisateur de naviguer, il nous faut d'abord reconstruire le menu principal et le pied de page. Nous pouvons pour cela utiliser de simples instances de Button ayant le style Sketch et regroupées au sein d'un conteneur. Sélectionnez l'écran Home et supprimez l'instance d'Image correspondant au menu. Créez ensuite un conteneur StackPanel dans la grille principale et nommez-le Menu. Disposez-le de manière à l'aligner en bas à droite de la grille et définissez l'empilement de ses enfants en mode horizontal. Glissez quatre boutons à l'intérieur de manière à recréer le menu. Choisissez une couleur conforme au code couleur de chaque écran. Procédez de façon identique pour le pied de page, vous pouvez utiliser un dégradé sur le bord pour créer un effet correspond au croquis original (voir Figure 10.33).

Figure 10.33

Le menu et le pied de page recréés avec des instances de Button et le style Sketch.

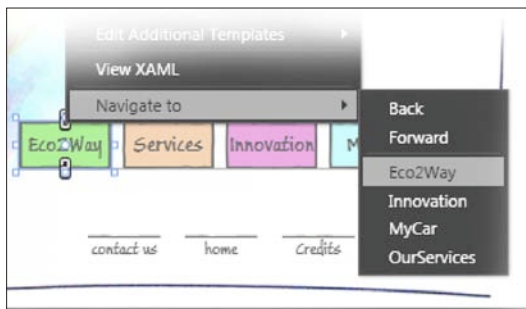


Nous allons maintenant ajouter un peu de logique au visuel. Rien de plus simple au sein d'un projet SketchFlow : faites un clic-droit sur le bouton Eco2Way_menu et sélectionnez l'option Navigate To. La liste des écrans accessibles apparaît, choisissez l'écran Eco2Way (voir Figure 10.34).

Répétez l'opération pour chaque bouton du menu. Ce que nous avons fait est non seulement traduit en XAML, mais consiste simplement à ajouter un comportement à chaque bouton sur lequel vous avez défini une navigation. Le comportement est donc accessible dans l'arbre visuel. Dans les projets SketchFlow, il n'est pas nécessaire de savoir ce fait pour utiliser le comportement. Une majorité d'interactions peut être réalisées par un simple clic-droit. Si vous êtes designer interactif, cela peut toutefois être utile dans les cas complexes d'interaction.

Figure 10.34

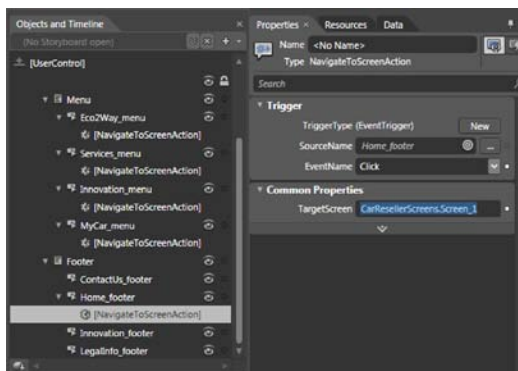
Définir une navigation vers l'écran Eco2Way.



Par défaut, l'accès aux écrans se déclenchera sur un clic de l'utilisateur, vous pourriez exécuter cette action lorsqu'un autre événement est diffusé. Une autre problématique se pose si vous souhaitez créer cette navigation pour le bouton Home_footer contenu dans le pied de page. Comme vous êtes sur la page Home, Blend ne vous propose pas de naviguer vers celle-ci. Si vous connaissez un peu les comportements, vous pouvez biaiser en choisissant un autre écran dans la liste afin de créer le comportement sur Home_footer. Vous n'avez plus qu'à modifier son paramètre TargetScreen (voir Figure 10.35).

Figure 10.35

Paramétrage du comportement affecté à Home_footer.



La page Home est maintenant presque aboutie. Il est très facile de dupliquer les deux StackPanel dans les autres écrans. Faites un copier-coller de ces derniers dans l'écran Eco2Way, remplacez le menu en haut de la page pour recouvrir l'image du menu déjà présente. Vous n'avez ensuite qu'à

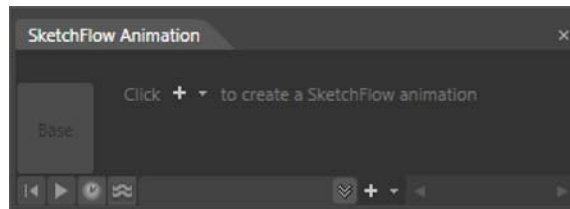
supprimer cette dernière ainsi que l'image du pied de page dans l'arbre visuel. Les comportements sont automatiquement copiés avec. Vous venez de créer une navigation utilisateur en moins de 5 minutes.

10.4.3 Simuler un flux d'utilisation

Nous allons maintenant simuler les actions de l'utilisateur *via* le panneau SketchFlow Animation. Ce dernier n'existe que dans ce type de projet, il permet de créer des animations temporisées et détectées par le lecteur SketchFlow lors de la navigation utilisateur. La personne testant le prototype pourra ainsi avoir un aperçu de l'interactivité tout en évitant un code fastidieux. Chaque écran du prototype peut posséder son propre jeu de simulations utilisateur. Vous allez simuler le clic de l'utilisateur sur un bouton afin de déplier une zone de texte cachée par défaut. Dans la carte de navigation (SketchFlow Map), double-cliquez sur l'écran OurServices. Vous constatez que le panneau SketchFlow Animation est vide (voir Figure 10.36).

Figure 10.36

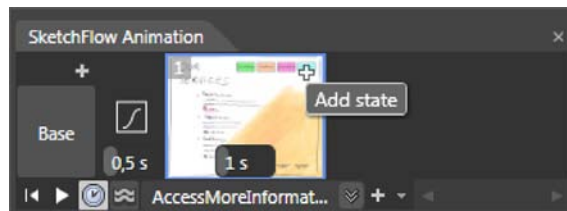
Panneau SketchFlow Animation vide.



Il contient un premier écran nommé Base, qui indique l'état par défaut de l'écran. Vous devez modifier quelques propriétés avant de créer la simulation utilisateur. Sélectionnez, dans le Canvas nommé Service, les composants Image MoreOver et ActivateMore. Passez les valeurs de leur propriété Opacity à 0. Ajouter une animation SketchFlow ou créer des animations standard sont des actions similaires. Cliquez sur l'icône représentant le signe plus, puis nommez l'animation AccessMoreInformation. Il suffit pour cela de cliquer sur le nom affiché par défaut et de le modifier. Sélectionnez le premier état nouvellement généré à droite de l'état de base. Il est représenté par une vignette affichant le visuel de l'écran. Vous passez en mode enregistrement d'état visuel. Au sein de SketchFlow, ces états sont appelés *Frame* ou image-clé (à ne pas confondre avec clé d'animation). Cliquez sur MoreOver, passez la valeur de la propriété Opacity à 100. Cliquez ensuite sur l'icône du signe plus situé en haut à droite de la vignette (voir Figure 10.37).

Figure 10.37

Création d'un nouvel état d'animation.

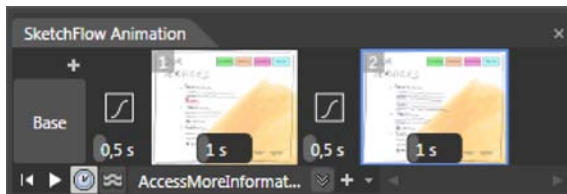


Vous créez ainsi un nouvel état à partir de l'état modifié. Cliquez sur ActivateMore, passez la valeur de la propriété Opacity à 100. Vous remarquez que lorsque vous survolez un écran, une valeur exprimée en secondes apparaît. Il s'agit du temps de pause durant lequel cet écran est affiché lors de la lecture de l'animation globale. Entre chaque écran, vous pouvez également spécifier une

durée ainsi qu'un type de transition. Vous pouvez activer définitivement l'affichage des tempos en cliquant sur l'icône de l'horloge.

Figure 10.38

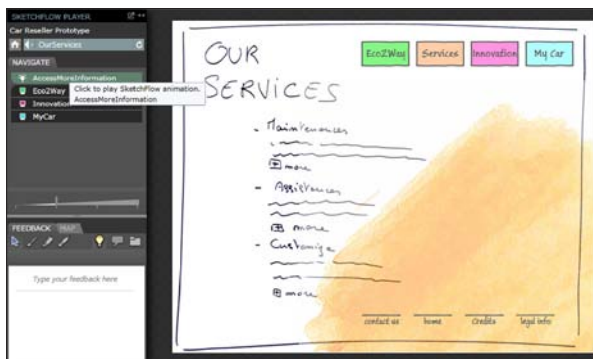
Les durées de pause affichées pour chaque écran.



Lorsque vous modifiez des propriétés dont les valeurs ne sont pas interpolables (donc différentes des valeurs de type Point, Double ou Color), vous pouvez activer le système d'agencement fluide qui assure une transition animée de ces propriétés (voir section 7.4). Testez l'animation SketchFlow directement au sein de Blend *via* le bouton de lecture. N'hésitez pas à revoir les tempos, si besoin, afin de simuler l'enchaînement des images-clés de manière réaliste. Compilez votre projet et naviguez jusqu'à l'écran OurServices pour accéder à l'animation du point de vue de l'utilisateur du prototype (voir Figure 10.39).

Figure 10.39

L'animation SketchFlow accessible.



Cliquez sur le lien pour jouer l'animation que vous venez de définir. Ce type d'animation est en fait basé sur le gestionnaire d'états visuels étudié au Chapitre 7. Nous allons maintenant aborder son fonctionnement au sein des projets SketchFlow.

10.4.4 États visuels

Chaque écran peut se comparer à une page de notre application. Les différentes pages peuvent donc posséder leurs propres états visuels. Au sein de la carte de navigation, double-cliquez sur l'écran nommé Innovation. Il permet à l'internaute de visualiser les tous derniers modèles de voiture disponibles sous forme de galerie d'images ou *via* un simple lecteur vidéo. La galerie ou la vidéo seront affichées à tour de rôle. Pour ce faire, vous allez créer deux états visuels au sein de l'écran Innovation *via* le panneau States. Créez un groupe d'états nommé DisplayStates, puis deux états nommés respectivement Galery et Video. Par défaut la vidéo est affichée. Sélectionnez l'état Galery, puis passez l'opacité des objets videoSelected et PlayerVideo à 0. Passez ensuite à false la valeur de la propriété IsHitTestVisible de l'objet videoSelected. Dans cet état, l'objet ne reçoit plus les interactions en provenance de la souris. Revenez dans l'état de base puis

faites un clic-droit sur l'objet `videoSelected`, sélectionnez l'option `Activate State` puis `Innovation / Galery` (voir Figure 10.40).

Vous venez de créer un comportement interactif sur cet objet. Lors du clic de la souris, l'utilisateur affichera l'état `Galery`. Vous pouvez créer un rectangle transparent, sous l'objet `videoSelected`, ayant pour objectif d'afficher l'état `Video`. Spécifiez une durée de transition pour le groupe d'états qui n'excède pas une seconde. Lorsque vous testez le prototype, vous avez la possibilité d'accéder aux états visuels de chaque écran (voir Figure 10.41).

Si vous êtes graphiste et que vous n'êtes pas un familier de Blend, l'accès simplifié aux états visuels est une vraie valeur ajoutée en termes de temps. L'utilisation de comportements est l'une des clés de cette réussite, nous les emploierons à nouveau de manières différentes dans le chapitre suivant. Le prototype interactif est dans le dossier : *chap10/CarReseller_Interactive.zip*.

Figure 10.40

Activer un état visuel via un clic-droit.

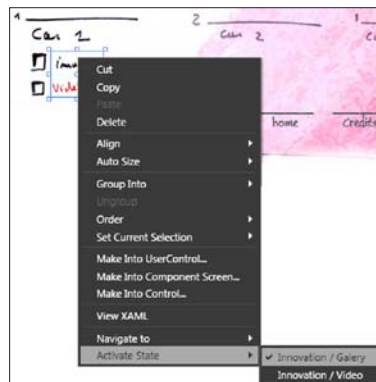
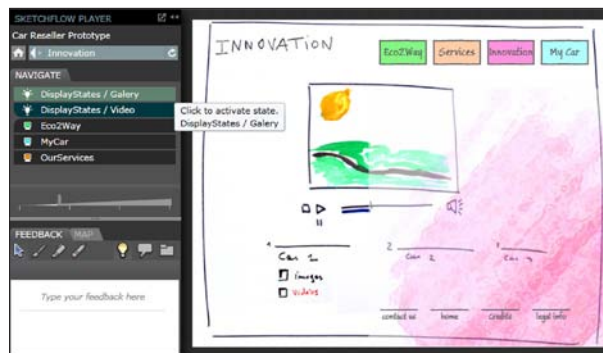


Figure 10.41

Accès aux états visuels via le lecteur SketchFlow.



10.5 Interface riche

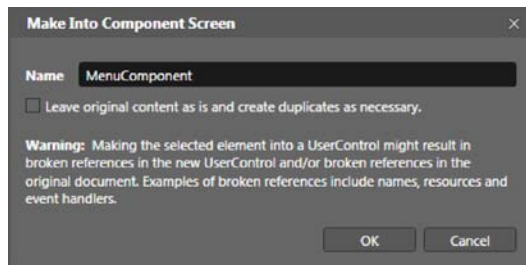
Jusqu'à présent l'arborescence de notre prototype est exclusivement constituée d'écrans simples, privilégiant ainsi une approche de conception page par page, très proche des sites XHTML classiques nombreux sur Internet. Cette approche possède certains défauts que nous allons identifier.

10.5.1 Écran *versus* composant

Comme vous le constatez, l'ensemble des pages du prototype partage deux éléments identiques, le menu principal et le pied de page. Le fait de conserver un design équivalent pour ces deux éléments facilite grandement la navigation et l'interprétation des pages pour l'internaute. D'un point de vue technique, cela pose toutefois problème car si vous souhaitez changer la disposition de ce menu, vous devrez le faire dans chaque page de votre projet. Ces deux éléments peuvent être considérés comme des instances de composants autonomes assurant leur propre logique. De cette manière, modifier le composant revient à changer toutes ses occurrences dans le projet. Au sein de SketchFlow, ce type de module est appelé composant d'écran. Il s'agit dans les faits d'un `UserControl` au même titre que notre application principale (voir Chapitre 12). Pour créer ce type d'objet, vous pouvez soit utiliser la carte de navigation, soit sélectionner un (ou plusieurs) contrôle dans l'un des écrans existant, pour le transformer par la suite en composant d'écran. Nous allons utiliser cette méthodologie dans un premier temps, puis nous utiliserons le panneau SketchFlow Map pour finaliser notre approche. Dans l'écran Home, faites un clic-droit sur le contrôle `Stack-Panel` nommé Menu, puis choisissez l'option `Make Into Component Screen...` Une boîte de dialogue apparaît vous demandant de nommer le composant d'écran (voir Figure 10.42).

Figure 10.42

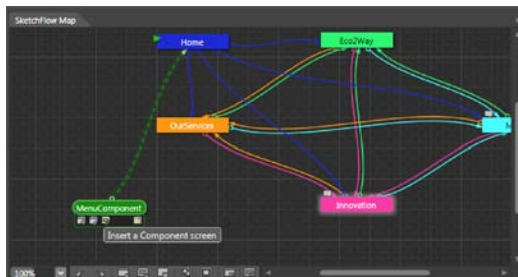
Fenêtre *Make Into Component Screen*.



Le nom `MenuComponent` proposé par défaut est éloquent, vous pouvez donc le conserver. Cliquez sur OK, la carte de navigation est automatiquement mise à jour. Vous remarquez qu'un nouveau type de connexion est généré. Celle-ci est en pointillés de couleur verte, apparence attribuée par défaut aux connexions de composants. Un couple de fichier XAML C# est également créé, Blend ouvre par défaut le fichier XAML correspondant à l'arbre visuel de notre menu. Il est possible d'instancier ce composant en le reliant (`MenuComponent`) à chaque écran. De cette manière, lorsque vous modifierez le composant source, chacune de ses instances sera mise à jour au sein des écrans du projet. Pour créer de nouvelles connexions à partir du composant, il suffit d'utiliser la dernière icône apparaissant lors du survol du composant au sein de la carte de navigation (voir Figure 10.43).

Figure 10.43

Connexion de composants.

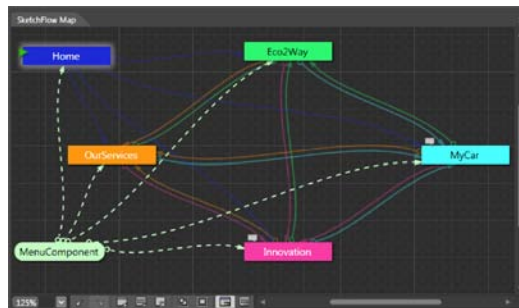


Il est nécessaire de repositionner correctement le composant pour chaque écran, non seulement dans l'espace, mais également dans l'arbre visuel afin de remplacer parfaitement l'espace occupé par les menus déjà présents. Il ne vous reste plus qu'à supprimer chaque ancien menu `StackPanel`. Vous pouvez procéder de manière identique pour le pied de page. Si vous souhaitez avoir une meilleure visibilité de l'un des deux types de connexions (composant ou écran), vous pouvez diminuer la luminosité de l'un ou de l'autre. Les composants d'écran possédant par nature de nombreuses connexions, vous pourriez également éclaircir leur couleur afin de les mettre en valeur. Un vert lumineux est idéal dans ce cas. Utilisez l'option `SketchFlow Project Settings...` (menu `Project`) pour modifier la couleur actuelle (voir Figure 10.44).

Les composants d'écran sont en fait de simples instances de `UserControl`. Si d'un point de vue technique `SketchFlow` n'est pas une révolution et repose sur de solides bases existantes, la facilité de conception qu'il apporte est réellement novatrice et pertinente. Le modèle technique `XAML / C#`, proposé dès le départ par `.Net 3`, est un terreau propice à ce genre d'avancées. Maintenant que les fondements sont posés, vous pouvez vous attendre à l'éclosion de ce type d'innovations de manière régulière dans l'avenir. Nous allons maintenant utiliser les composants d'écran d'une manière différente afin de créer un prototype simple d'interface riche.

Figure 10.44

Connexions de composants mise en valeur.



10.5.2 L'exemple du configurateur riche

La notion de configurateur riche est apparue dans le Web avec les technologies asynchrones comme `AJAX`. L'idée générale est de faciliter un processus à travers une interface enrichie. Cela peut aller de la réservation hôtelière à la configuration d'une cuisine comme le proposent certaines applications. Au sein de cette section, nous allons prototyper une interface permettant à un internaute de sélectionner et de configurer une voiture. L'objectif final est de faciliter la prise de rendez-vous pour un test de conduite.

10.5.2.1 Le flux d'utilisation

Comme nous l'avons précisé au début de cette section, la navigation page par page possède quelques défauts. Elle force notamment l'utilisateur à relire la totalité de chaque nouvelle page affichée. Cette contrainte peut évidemment être très utile si vous affichez un contenu radicalement différent des précédents. C'est toutefois rarement le cas et, dans 90 % des situations, le flux d'utilisation est complètement cassé par ce que l'on appelle l'*aveuglement d'informations* causé par le rafraîchissement.

Lorsque vous utilisez une application, vous entrez parfois dans le flux, un état idéal d'utilisation dans lequel tout vous paraît logique et évident. Vous enchaînez ainsi rapidement et simplement les actions et accédez facilement aux informations sans vous poser de questions existentielles. Dans ce cas, votre état de concentration est optimal. L'aveuglement d'informations peut vous forcer à quitter cet état. C'est un peu comme si un animateur décidait de faire bouger tous les personnages d'un dessin animé en même temps avec exactement la même amplitude pour chacun d'eux. Le traitement des informations par l'œil est tellement important qu'il n'est pas possible de tout analyser en même temps. On pourrait également percevoir cela comme une forme de censure ou de bruit visuel, tous deux engendrés par un trop plein d'informations impossibles à traiter. Dès lors, vous quittez le flux d'utilisation afin de conserver une distance suffisante et garder une vision globale de la situation. Cette situation est au final assez désagréable car elle vous force à prendre du recul et à réfléchir là où tout devrait être simple et compréhensible.

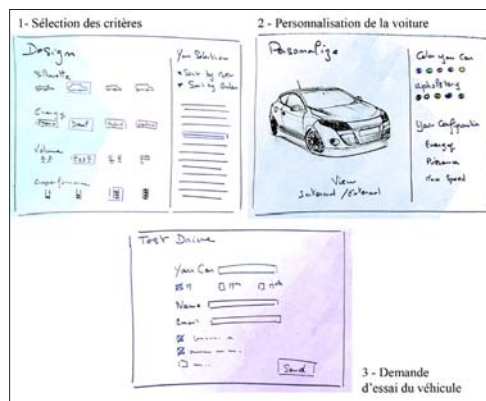
Le flux d'utilisation peut être déstabilisé par d'autres facteurs comme une mauvaise compréhension ou formalisation de l'interface ou des ralentissements qui obligent à prendre du recul du point de vue utilisation. Au final, vous pouvez très rapidement décourager l'internaute ou l'utilisateur de l'application si vous n'y prenez pas garde.

L'une des solutions permettant la mise en place de flux d'utilisation consiste à permettre une navigation au sein d'une seule et même page. Vous ne rafraîchissez ainsi qu'une portion de la page et attirez l'attention de l'utilisateur au bon endroit et au bon moment. Les composants d'écran permettent de réaliser ce type d'expérience assez simplement car vous ne rafraîchirez pas toute la page, mais juste une portion de celle-ci. L'utilisateur est beaucoup moins désorienté dans ce cas et peut même prévoir intuitivement un certain nombre de comportements de l'interface. Ce qui est prévisible le confortera dans son expérience et dans ses choix d'utilisation.

Nous allons maintenant illustrer ce principe à travers le prototypage d'un configurateur riche. Ce dernier est une petite application. Elle sera contenue dans l'écran MyCar et permettra à l'utilisateur de sélectionner un véhicule à partir d'une liste de critères. L'écran est constitué de trois panneaux représentant chacun une étape dans le processus d'utilisation (voir Figure 10.45).

Figure 10.45

Les trois panneaux du configurateur riche.



10.5.2.2 Créer les composants

Afin de se concentrer sur notre problématique, dézippez le projet : *chap10/CarReseller_Flux.zip*. Au sein du projet, l'écran MyCar contient une arborescence qui a été générée à partir de dessins

scannés et retouchés sous Photoshop et de composants ayant le style Sketch. Les écrans vus précédemment sont tout trois répartis au sein de grilles. Le premier permet de sélectionner les critères et affiche une liste mise à jour dynamiquement. Le deuxième écran donne un aperçu de la voiture et permet de personnaliser ses options. Le dernier écran est un formulaire renseigné par le client qui souhaite tester la voiture sélectionnée sur circuit ou sur route. Sélectionnez la grille Design, puis faites-en un composant d'écran. Dans le composant nouvellement créé, définissez des dimensions en mode automatique pour le composant de type `UserControl` racine, ainsi que pour la grille nommée Design. Pour finir, simplifiez l'imbrication en dégroupant la grille `LayoutRoot`. Vous pouvez réaliser cela *via* un clic-droit et sélectionner l'option `Ungroup`. Revenez dans l'écran `MyCar` et répétez l'opération pour les grilles `Perso` et `test`.

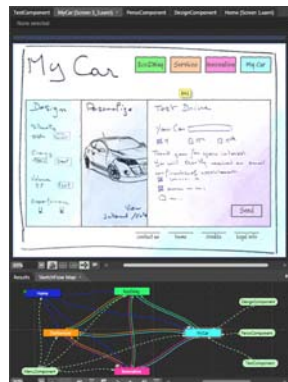
INFO

Dans l'écran `MyCar`, vous constatez la présence d'encadrés jaunes autour des composants nouvellement générés ainsi qu'un point d'exclamation en haut à gauche. Cela indique que vous devez compiler le projet pour voir le résultat visuel final de ces composants. Pour compiler sans tester le projet, utilisez le raccourci `Ctrl+Maj+B`. Lorsque la compilation est terminée, les bordures jaunes et le point d'exclamation disparaissent.

Vous obtenez trois composants d'écran liés à l'écran `MyCar` dans la carte de navigation, ainsi qu'un résultat visuel similaire à celui d'origine pour cet écran (voir Figure 10.46).

Figure 10.46

Les trois composants du configurateur riche et l'arbre visuel.



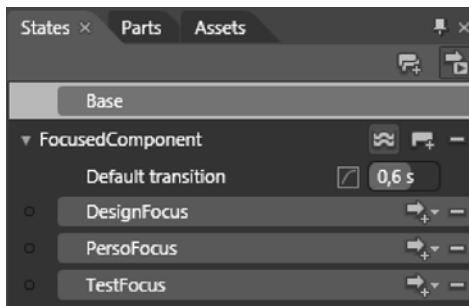
Comme vous le constatez, les deux premiers composants sont partagés en deux zones distinctes. Lorsque l'un ou l'autre de ces composants perdra le focus utilisateur, seule l'une des deux parties de ce composant sera visible, l'autre sera simplement rétractée avec une petite animation de fondu. Lorsque l'un des panneaux recevra le focus, il sera mis au premier plan avec un déplacement sur l'axe 3D en z. Ce comportement permet à l'utilisateur de ne pas être submergé d'informations inutiles.

Pour créer ce type de transition, il faut créer des états correspondant au focus utilisateur pour chaque composant, ainsi que pour l'écran `MyCar`. Dans l'écran `MyCar`, inversez l'ordre des composants de manière à ce que le panneau `DesignComponent` soit au premier plan et que le composant `TestComponent` soit au dernier. Créez ensuite un nouveau groupe d'états visuels nommé `FocusedComponent`, ainsi que trois états nommés en son sein, ayant pour noms respectifs, `DesignFocus`, `PersoFocus` et `TestFocus`. Définissez une transition globale pour le groupe d'états

de 6/10 de seconde environ avec une courbe de décélération de votre choix – Cubic Out semble une bonne option (voir Figure 10.47).

Figure 10.47

Les états visuels de l'écran MyCar.



Dans l'état DesignFocus, modifiez la profondeur sur l'axe 3D z global (GlobalOffsetZ) des composants PersoComponent et TestComponent avec des valeurs respectives de -100 et -200 pixels. Ceux-ci se trouvent à l'arrière-plan, automatiquement l'un derrière l'autre. Sélectionnez l'état PersoFocus, puis modifiez cette fois la position sur l'axe z global de DesignComponent et de TestComponent de -50 et de -200 pixels. Pour finir, dans l'état TestFocus, passez la position sur l'axe z global de DesignComponent à -200 et celle de PersoComponent à -100. Dans cet état, le panneau de TestComponent sera à l'avant-plan, puis viendra le panneau PersoComponent au second plan et au dernier plan DesignComponent.

INFO

Lorsque l'un des panneaux possède le focus, il ne subit aucune projection 3D. Cela est assez positif pour votre visuel final car lorsqu'un objet possède une projection, il est rendu sous forme de bitmap (voir Chapitre 9). Cela occasionne un lissage des pixels et peut engendrer des textes ou des vecteurs floutés. Les conséquences de ce rendu bitmap sont moins gênantes pour les panneaux situés à l'arrière-plan, car ils n'ont pas l'intérêt utilisateur. Vous pouvez également vous permettre de repositionner en x les panneaux afin de mettre celui qui a le focus utilisateur en valeur. Utilisez à cette fin les RenderTransform afin d'éviter de rasteriser les vecteurs en bitmap. Le mélange des deux types de transformations est donc non seulement possible mais conseillé pour les panneaux ayant le focus qui ne sont pas affectés de projections 3D.

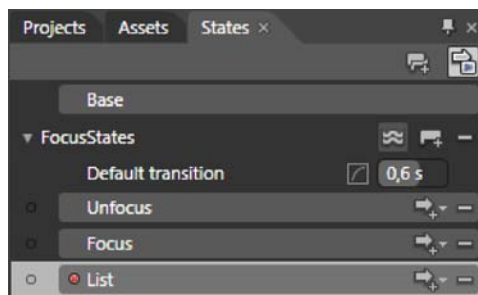
La problématique, à ce stade, est de toujours faire apparaître une portion des composants en arrière-plan de celui qui possède le focus. Allouer les marges de manière précise n'est pas forcément aisé à cet instant car vous devez anticiper le fait que les composants seront eux-mêmes dans des états visuels modifiant leur largeur. Vous pourrez également modifier l'agencement des composants à la suite d'une série de test de l'application et procéder ainsi de manière empirique.

10.5.2.3 Transitions et états visuels de composants

Nous avons réalisé la première partie de notre prototype. Il nous reste à créer les états visuels inhérents à chaque composant d'écran. Dans la carte de navigation, double-cliquez sur l'écran DesignComponent et créez un groupe d'états visuels nommé FocusStates ainsi que des états, la transition et une durée globale (voir Figure 10.48).

Figure 10.48

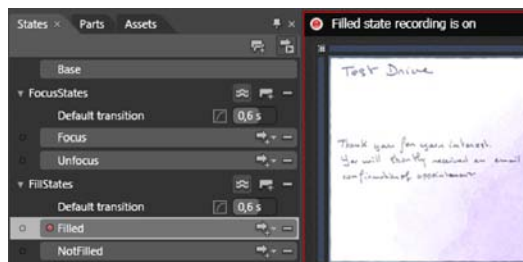
Les états visuels de DesignComponent.



Sélectionnez l'état `List`, puis passez la largeur (`Width`) ainsi que l'opacité de l'objet `Options-Design` à 0. Modifiez ensuite la largeur de l'objet `ColorDesign` à 150. Cliquez sur l'état `Unfocus` et définissez l'opacité et la largeur de l'objet `CurrentSelectionDesign` à 0. Passez ensuite la largeur de l'objet `ColorDesign` à 200. Dans la carte de navigation, double-cliquez sur le composant `PersoComponent`. Nous allons répéter les mêmes étapes. Créez deux états ayant les mêmes propriétés de transition, nommés respectivement `Focus` et `Unfocus`. Cliquez sur l'état `Unfocus`, puis sélectionnez l'objet nommé `PersoRight`. Passez la valeur de ses propriétés d'opacité et de largeur à 0. Définissez ensuite la largeur du composant `Image ColorPerso` à 220 pixels. Pour finir, nous allons modifier le composant `TestComponent`. Créez deux groupes d'états nommés `FocusStates` et `FillStates`. Le premier nous permettra d'afficher le visuel en fonction du focus utilisateur, le second gèrera le visuel selon que l'utilisateur aura rempli ou non le formulaire. Dans l'état de base, passez la propriété `Visibility` du composant `Message` à `Collapsed`. De cette manière, le texte de confirmation de l'envoi du formulaire n'est pas visible au chargement du configurateur. Créez les états `Focus` et `Unfocus` au sein du groupe d'état adéquat, puis les états `Filled` et `NotFilled` pour le groupe `FillStates` (voir Figure 10.49).

Figure 10.49

Les états visuels de TestComponent.



L'objectif est maintenant de ne laisser afficher que le titre et l'arrière-plan pour l'état `Unfocus`. Sélectionnez les objets `ContentTestDrive`, `Message` et `ColorTestDrive` pour modifier leur opacité à 0. Fixez également la largeur de `ContentTestDrive` et de `ColorTestDrive` à 170 pixels. Afin de gérer les états visuels concernant la saisie du formulaire, sélectionnez l'état `Filled`, puis modifiez la valeur de la propriété `Visibility` à `Collapsed` pour `ContentTestDrive` ainsi que pour le bouton. À l'opposé, changez la valeur de cette propriété à `Visible` pour le composant `Message`.

Nous venons d'éviter un conflit de propriété entre les états visuels du groupe `FillStates` et `FocusStates`. En effet, nous n'aurions pas pu utiliser la propriété `Opacity` dans les deux groupes d'états sans engendrer de bogues à l'exécution (voir Chapitre 7). Il ne nous reste plus qu'à créer le

code logique et les comportements interactifs nécessaires pour donner vie à l'interface. Le projet contenant tous les états visuels est accessible dans : *chap10/CarReseller_Etats.zip*.

10.5.2.4 Interactivité avancée

Vous trouverez un résumé de tous les états créés pour chaque écran ainsi que l'activation de ceux-ci en fonction de chaque étape d'utilisation au Tableau 10.1.

Ce type de récapitulatif est assez pratique pour concevoir des interfaces riches. C'est une sorte de feuille de route vous permettant de vous y retrouver. Par exemple, pour la première phase d'utilisation, l'écran MyCar affiche l'état DesignFocus, les composants DesignComponent, PersoComponent et TestComponent affichent respectivement les états Focus, Unfocus et Unfocus. Une fois le véhicule choisi dans la liste, l'écran MyCar active l'état PersoFocus, le panneau DesignComponent affiche l'état List, etc. Vous remarquez que pour la dernière étape, TestComponent est Focus, mais qu'il peut également être dans l'état Filled ou NotFilled. Celui-ci bénéficie en effet de deux groupes d'états visuels distincts et affiche donc deux états. Nous allons commencer par initialiser la première étape.

Tableau 10.1 : États visuels en fonction de l'étape

États visuels	Étape 1	Étape 2	Étape 3
Écran MyCar	DesignFocus	PersoFocus	TestFocus
Composant DesignComponent	Focus	List	Unfocus
Composant PersoComponent	Unfocus	Focus	Unfocus
Composant TestComponent	Unfocus	Unfocus	Focus / Filled / NotFilled

À cette fin, vous allez ajouter de la logique *via C#*. Vous pouvez vous référer au Tableau 10.1 pour vous faciliter ainsi la tâche. Ouvrez le fichier C# `Screen_1_3.cs` dans Blend ou Visual Studio et utilisez la classe statique `VisualStateManager`, comme montré ci-dessous :

```
public Screen_1_3()
{
    InitializeComponent();
    Loaded +=new System.Windows.RoutedEventHandler(Screen_1_3_Loaded);
}

private void Screen_1_3_Loaded(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToState(this,"DesignFocus",false);
    VisualStateManager.GoToState(designComponent,"Focus",false);
    VisualStateManager.GoToState(persoComponent,"Unfocus",false);
    VisualStateManager.GoToState(testComponent,"Unfocus",false);
}
```

Le troisième paramètre indique si une transtion animée est utilisée. Dans notre cas, la transition n'a pas besoin d'être jouée, ce paramètre est donc à `false`. Au sein du composant d'écran `DesignComponent`, cliquez-droit sur le composant `CurrentSelectionDesign`, puis dans le menu `Activate State`, sélectionnez l'état `List`. L'utilisateur passe automatiquement dans la seconde étape

d'utilisation lorsqu'il cliquera sur le composant. Si nous nous référons au Tableau 10.1, nous devrions également activer trois autres états : PersoFocus pour l'écran MyCar, Focus pour le composant PersoComponent et Unfocus pour le composant TestComponent. À ce stade, nous ne pouvons pas piloter ces états visuels de manière identique pour deux bonnes raisons. La première est que l'interface de Blend ne nous autorise pas à y accéder *via* la liste des états affichés dans le menu Activate State par le clic-droit. La seconde est assez ennuyeuse : même si nous y avons accès, l'état ainsi sélectionné remplacerait celui que nous avons déjà spécifié dans l'arbre visuel. Autrement dit, vous ne pouvez pas spécifier plus d'un comportement de même type sur un composant en utilisant le menu contextuel par clic-droit. En réalité, cela est possible, mais uniquement en XAML ou par glisser-déposer du comportement. Nous allons simplement ajouter ces comportements en modifiant le code XAML. Pour cela, passez en mode mixte afin que le code apparaisse, puis sélectionnez CurrentSelectionDesign. Voici la déclaration XAML du comportement :

```
<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
                          DesignComponent" TargetState="List" />
  </i:EventTrigger>
</i:Interaction.Triggers>
```

Ajoutez une balise EventTrigger (déclencheur d'événements) au sein de la liste des déclencheurs et modifiez le code comme ci-dessous :

```
<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
                          DesignComponent" TargetState="List" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.Screen_1_3"
                          TargetState="PersoFocus" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen=" CarResellerScreens.
                          PersoComponent" TargetState="Focus" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
                          TestComponent" TargetState="Unfocus" />
  </i:EventTrigger>
</i:Interaction.Triggers>
```

Testez le prototype dans le navigateur, lorsque vous cliquez sur la liste, trois transitions sont jouées simultanément. La première concerne le panneau design lui-même lorsqu'il passe dans l'état List. La deuxième est jouée au sein de l'écran MyCar et affiche l'état PersoFocus. La troisième affiche le composant PersoComponent en mode Focus. Il faudrait maintenant jouer la transition inverse lorsque l'utilisateur souhaite modifier ses critères de sélection. Cela consiste à réafficher l'état DesignFocus de l'écran MyCar lorsque l'utilisateur clique sur le titre du panneau TitleDesign dans DesignComponent. Comme le titre du panneau est situé en dessous de l'objet ColorDesign, le clic de l'utilisateur ne sera pas diffusé. Pour régler ce problème, il suffit de passer la propriété IsHitTestVisible de l'objet ColorDesign à false. De la même manière que précédemment, ajoutez les balises XAML comme montré ci-dessous :

```
<Image x:Name="TitleDesign" Height="44" HorizontalAlignment="Left"
       Margin="6,3,0,0" VerticalAlignment="Top" Width="99"
```

```

        Source="DesignPanel_Images/TitleDesign.png">
<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      DesignComponent" TargetState="Focus"/>
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      Screen_1_3" TargetState="DesignFocus"/>
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      PersoComponent" TargetState="Unfocus"/>
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      TestComponent" TargetState="Unfocus"/>
  </i:EventTrigger>
</i:Interaction.Triggers>
</Image>

```

Vous avez la possibilité de consulter le résumé des états pour visualiser les transitions. Testez le prototype ; vous constatez que cette fois, vous pouvez passer d'un état à l'autre à tout moment en cliquant alternativement sur la liste et le titre.

INFO

Notre application met en valeur une erreur de conception classique. Lorsque l'utilisateur a sélectionné sa voiture dans la liste, le composant PersoComponent est mis au premier plan et seule la liste des voitures sélectionnables est visible. Si cette voiture ne lui correspond pas et qu'il souhaite changer ses critères, aucun indice visuel ne lui indique que le titre lui permet de réafficher la totalité du panneau DesignComponent. Il n'aura pas forcément l'idée de le cliquer. Le mieux, dans ce cas, est d'afficher une icône sous forme de flèche de retour afin que l'internaute puisse aisément revenir sur ses pas.

Si besoin, n'hésitez pas à modifier les marges de chaque élément, pour obtenir un alignement des objets sans effets de bords. Une fois que l'utilisateur a personnalisé sa voiture (dans le deuxième panneau), il est temps pour lui de demander un rendez-vous pour la tester en condition réelle. Au sein du composant TestComponent, cliquez-droit sur le composant ColorTestDrive, puis au sein du menu Activate State, sélectionnez l'état Focus. Lorsque l'utilisateur cliquera sur ce panneau, celui-ci se déploiera et affichera le formulaire. Vous devez également gérer les quatre autres transitions que nous avons déjà évoquées auparavant. Vous pouvez une fois de plus modifier le code XAML :

```

<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.Screen_1_3"
      TargetState="TestFocus"/>
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      TestComponent" TargetState="Focus"/>
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      DesignComponent" TargetState="Unfocus"/>
  </i:EventTrigger>

```

```

</i:EventTrigger>
<i:EventTrigger EventName="MouseLeftButtonDown">
  <pb:ActivateStateAction TargetScreen="CarResellerScreens.
    PersoComponent" TargetState="Unfocus" />
</i:EventTrigger>
</i:Interaction.Triggers>

```

Dans un premier temps, il vous suffit de gérer cette navigation en ajoutant les comportements nécessaires sur le composant PersoComponent, directement au sein de l'écran MyCar :

```

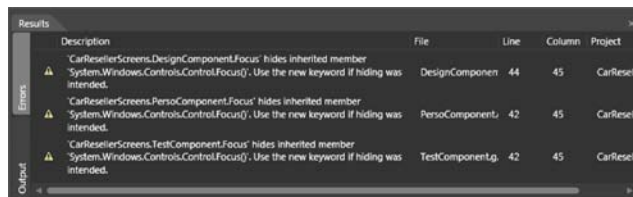
<i:Interaction.Triggers>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.Screen_1_3"
      TargetState="PersoFocus" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      TestComponent" TargetState="Unfocus" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      DesignComponent" TargetState="List" />
  </i:EventTrigger>
  <i:EventTrigger EventName="MouseLeftButtonDown">
    <pb:ActivateStateAction TargetScreen="CarResellerScreens.
      PersoComponent" TargetState="Focus" />
  </i:EventTrigger>
</i:Interaction.Triggers>

```

Dans la carte de navigation, double-cliquez sur le composant d'écran TestComponent. Vous devez encore activer l'état Filled affiché lorsque l'utilisateur soumet correctement le formulaire de rendez-vous. Cliquez-droit sur le bouton de soumission et choisissez l'état Filled, le projet est terminé. Vous avez peut-être remarqué plusieurs messages d'alerte dans l'interface de Blend tout au long de l'exercice (voir Figure 10.50).

Figure 10.50

Messages d'alerte.



Ces messages sont assez clairs, ils vous signalent que le mot Focus est déjà utilisé et hérité en interne par les contrôles utilisateur standard. Vous risquez de rendre cette propriété inaccessible, voir de l'écraser. Ce n'est pas grave dans la mesure où vous créez un prototype. Par contre, en mode de production, nommer les états avec des noms identiques à certains du framework Silverlight, peut engendrer des conflits et des bogues difficiles à prévoir. Pour éviter ce type de message, vous pouvez modifier le nom des états visuels liés au focus en les francisant. Le prototype finalisé est accessible dans : *chap10/CarReseller.zip*.

Au prochain chapitre, nous approfondirons nos connaissances des ressources graphiques utilisables dans les projets Silverlight. Nous aborderons également les bonnes pratiques de production concernant leur organisation et leur partage.

Partie III

Conception d'applications riches

11

Ressources graphiques

Depuis le début de ce livre, nous avons largement évoqué la capacité de Silverlight quant à la simplification de la communication entre chaque acteur. Depuis quelques années, cette collaboration intermétiers est une problématique centrale et incontournable. Le développement informatique a débuté dès les années quarante grâce à des personnages emblématiques comme Alan Turing. Dès lors, nous avons vu l'apparition de nombreux modèles de conception toujours plus performants. Ce secteur est aujourd'hui parvenu à une sorte de maturité et offre un large éventail de modèles de conception.

Toutefois, l'arrivée de technologies comme WPF ou Silverlight remet en cause un certain nombre d'habitudes acquises au profit de l'ergonomie et de l'expérience utilisateur. Comparativement, la technologie Winforms, tournée essentiellement vers la fonctionnalité, n'a que très peu besoin d'évoluer, elle produit du fonctionnel de manière très satisfaisante et ne prend pas vraiment en compte la conception d'interfaces graphiques avancées. Certaines entreprises ont simplement décidé de ne pas évoluer vers WPF pour cette raison. Cela peut se comprendre aisément bien que ce soit au détriment de l'utilisateur. Le design, l'ergonomie ou tout type de contrainte qui n'est pas directement liée à la fonctionnalité, met en danger sa conception même. Pourquoi ? Parce qu'il est bien plus facile de concevoir une application fonctionnelle en Winforms ou en Java, que d'obtenir la même qualité de conception tout en ajoutant des contraintes techniques et architecturales liées à l'expérience utilisateur ou au design.

Le but des plateformes WPF et Silverlight est précisément de répondre à cette problématique consistant à marier le design d'interfaces visuelles (sous tous ses aspects) et le développement informatique. Ces plateformes réussissent ce pari de manière élégante et efficace d'un point de vue architecture et productivité. Une partie de cette réussite provient de la notion de ressources. Une bonne gestion des ressources est l'une des clés du succès et facilite grandement l'adaptation des savoir-faire.

Si vous souhaitez simplifier votre manière de produire une application à travers la mise en place d'une collaboration intermétiers efficace, ce chapitre est fait pour vous. Dans un premier temps, vous apprendrez à catégoriser les ressources et découvrirez les principes de base inhérents à leur utilisation. Par la suite, vous utiliserez les ressources graphiques de manière concrète à travers l'intégration graphique d'un lecteur multimédia. Ainsi, vous aborderez l'utilisation des ressources de couleurs et de pinceaux, puis celles qui sont liées à l'utilisation de médias externes, comme les images ou les polices de caractères. Dans un second temps, vous concevrez des styles et des modèles personnalisés permettant de modifier l'affichage des contrôles Silverlight. Vous découvrirez ainsi le confort de conception apporté par les mécanismes de liaison et comment créer des ensembles de données fictives facilitant la personnalisation de contrôle de données.

11.1 Qu'est-ce qu'une ressource

Les ressources ont pour objectif d'être réutilisées, soit lors de la conception d'une application, soit durant leur exécution. Lorsque vous modifiez la valeur d'une ressource au sein d'Expression Blend ou de Visual Studio, toutes les occurrences d'objets utilisant cette ressource sont mises à jour. Ce principe est à la base du flux de production innovant et efficace proposé par Silverlight. C'est un héritage direct du workflow apporté par WPF, qui offre un modèle encore plus abouti.

11.1.1 Définition et types de ressources

Au sein de Silverlight, on distingue trois types de ressources : les ressources logiques, celles de type média et les ressources graphiques. Les ressources logiques font référence aux objets qui n'ont pas vocation à être directement affichés par le moteur vectoriel. Ainsi, vous pourriez définir une ressource logique de type `String` réutilisable en plusieurs endroits d'une même application. Les ressources de type média sont des fichiers externes que vous pouvez incorporer au sein des projets Silverlight. Les fichiers image (jpg ou png) font partie de cette catégorie. Ils ne sont pas décrits au sein du code XAML, mais directement référencés dans le projet *via* le fichier `csproj`.

Pour finir, les ressources graphiques sont codées en XAML ou en C#, elles sont utilisées par des objets destinés à être affichés. Les styles et les modèles de la classe `Button`, que nous avons créés au Chapitre 7, font partie de cette dernière catégorie. Le Tableau 11.1 fournit une liste non exhaustive des ressources classées par type.

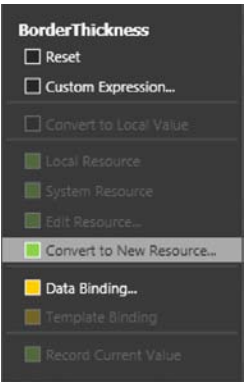
Tableau 11.1 : Divers types de ressources classées par genre

<i>Ressources logiques</i>	<i>Ressources médias</i>	<i>Ressources graphiques</i>
Double	Images : jpg / png	Couleurs
String	Vidéos : .wmv	Pinceaux
Boolean	Sons : .wma	Styles
Thickness,...	Polices de caractères : .ttf	Modèles

Les ressources graphiques définies par le code XAML sont très pratiques à utiliser et à maintenir ; nous allons le démontrer à travers un cas simple d'utilisation. Créez un nouveau projet nommé `RessourcesIntro`, placez ensuite plusieurs composants de type `Border` sur le conteneur `Layout-Root`. Définissez pour chacun d'eux une couleur d'arrière-plan mauve (par exemple `#FF937C9D`). Sélectionnez l'un d'eux au hasard puis, dans le panneau `Properties`, cliquez sur l'icône carrée à droite du champ `Opacity`. Sélectionnez l'option `Convert to New Resource...` (voir Figure 11.1).

Figure 11.1

Conversion d'une valeur en ressource.

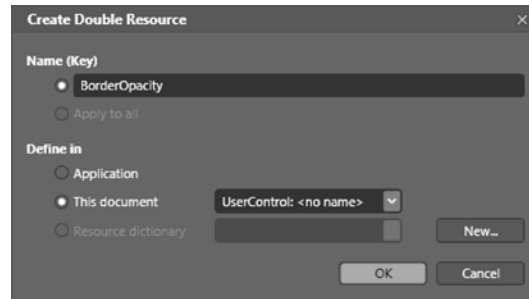


Une boîte de dialogue apparaît. Elle vous permet de configurer la clé de ressource ainsi que l'endroit où la stocker. Modifiez simplement le nom de la clé de ressource en `BorderOpacity`. Nommer les ressources permet, entre autres, de les identifier facilement par la suite. Veillez à nommer vos ressources le plus explicitement possible car un projet peut en contenir un nombre très important. Le but est de retrouver la ressource recherchée au plus vite et de permettre aux différents intervenants du projet d'identifier les ressources créées (voir Figure 11.2).

Ne changez pas les autres options, vous découvrirez leur intérêt tout au long de ce chapitre. Une fois le nom de la clé modifié, validez en cliquant sur `OK`. Comme vous le constatez, les ressources sont accessibles par des clés (`x:Key` en XAML). Ce comportement est tout à fait logique car elles sont stockées dans des listes particulières implémentant l'interface `IDictionary`. Ces listes sont constituées de couples clé/valeur et sont nommées dictionnaires. Accéder aux valeurs d'un dictionnaire est possible grâce aux clés qui leurs sont associées.

Figure 11.2

Fenêtre de création de ressource.



INFO

Ce principe s'applique parfaitement à WPF, mais diffère légèrement pour Silverlight. Les ressources Storyboard en sont un exemple. Au Chapitre 6, nous les avons évoquées à plusieurs reprises. Les animations sont des ressources qui ont la possibilité d'être directement accessibles *via* l'attribut `x:Name`. Ainsi, lorsque vous créez une animation au sein d'un projet Silverlight, celle-ci est bien contenue dans un dictionnaire de ressources propre au `UserControl` principal, mais elle est accessible par son nom d'exemplaire. Cela permet au développeur de les cibler directement *via* C#, comme il le ferait avec une instance d'objet standard.

Dans l'environnement WPF, gérer les animations se révèle légèrement plus complexe pour les débutants car ils doivent utiliser la clé afin de récupérer l'instance qu'ils transtypent en tant que `Storyboard`. Au final, la manière d'accéder aux animations dans Silverlight est plus pratique, mais se révèle limitée. En effet, il est impossible de déclencher un même `Storyboard` sur plusieurs objets à la fois, ce qui n'est pas le cas pour WPF.

Les dictionnaires de ressources sont des instances de la classe `Dictionary` spécifiques car les clés sont forcément de type `String`. Voici le code XAML de notre projet :

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:System="clr-namespace:System;assembly=mscorlib"
  x:Class="RessourcesIntro.MainPage"
  Width="640" Height="480">

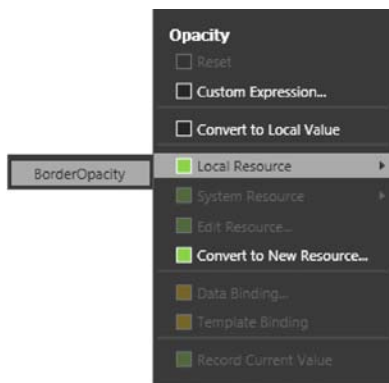
  <UserControl.Resources>
    <System:Double x:Key="BorderOpacity">1</System:Double>
  </UserControl.Resources>

  <Grid x:Name="LayoutRoot" Background="White">
    <Border Height="115" HorizontalAlignment="Left" Margin="49,36,0,0"
      VerticalAlignment="Top" Width="118" Opacity="{StaticResource
        BorderOpacity}" Background="#FF937C9D" BorderBrush="Black"
      BorderThickness="1"/>
    <Border Margin="223,203,294,180" Background="#FF937C9D"
      BorderBrush="Black" BorderThickness="1"/>
    <Border Height="138" HorizontalAlignment="Right" Margin="0,66,64,0"
      VerticalAlignment="Top" Width="134" Background="#FF937C9D"
      BorderBrush="Black" BorderThickness="1"/>
  </Grid>
</UserControl>
```

Comme vous le constatez, la valeur de l'opacité a été définie au sein d'une ressource de type double. La propriété `Opacity` du premier contrôle `Border` se contente de référencer la clé. Faites en sorte que chaque instance de la classe `Border` référence une valeur d'opacité pointant vers la ressource `BorderOpacity`. Pour cela, vous pouvez coder en XAML ou utiliser l'interface de Blend, ce dernier moyen reste le plus simple. Sélectionnez les deux autres objets `Border`, et dans le panneau `Properties`, cliquez sur l'icône carrée située à droite du champ `Opacity`. Sélectionnez ensuite l'option `Local Resource`, puis cliquez sur `BorderOpacity` (voir Figure 11.3).

Figure 11.3

Affectation d'une ressource à une propriété d'objet via Blend.



Désormais, toutes les instances de `Border` partagent la même opacité en faisant référence à la ressource `BorderOpacity`. Modifier leur opacité revient à changer la valeur de la ressource. À cette fin, vous pouvez utiliser le panneau `Resources`. Ouvrez-le et dépliez `UserControl1`. De cette manière, vous accédez à toutes les ressources définies pour `MainPage.xaml`. Une fois l'arborescence du `UserControl1` dépliée, la ressource apparaît en dessous. Changez sa valeur en passant le champ `BorderOpacity` à `0.4`. Vous constatez que l'opacité de chaque `Border` est mise à jour dynamiquement. Vous pouvez procéder de la même manière avec la propriété `BorderThickness` ; il est bien pratique d'homogénéiser l'épaisseur des bords des instances de `Border` au sein d'une application.

11.1.2 Les dictionnaires de ressources

Malgré les apparences du code XAML généré par Blend, toutes les ressources qui ne sont pas de type multimédia (image, vidéo,...) sont stockées au sein de dictionnaires de ressources. Nous expliquerons ce principe dans cette section.

11.1.2.1 Stockage et portée des ressources

Ces dictionnaires de ressources, déclarés par de simples balises XAML, peuvent être définis à plusieurs niveaux, ce qui détermine directement leur portée d'utilisation. Comme tous les objets de type `FrameworkElement` possèdent la propriété `Resources` typée `ResourceDictionary`, ils ont la capacité d'avoir leur propre dictionnaire de ressources. Voici les différents lieux de stockage possibles :

- Au sein de l'application elle-même, soit dans le fichier `App.xaml`. Dans ce cas, la ressource sera disponible dans toutes les pages du projet. Ce type de stockage peut apparaître limité. Différents projets, au sein d'une même solution, n'auront pas accès aux ressources que les uns ou les autres possèdent. Voici le code XAML généré dans Blend par défaut :


```
<Application.Resources>
  <CornerRadius x:Key="CornerRadius1">0</CornerRadius>
</Application.Resources>
```

ATTENTION

Ne confondez pas projet et solution. Lorsque vous créez un nouveau projet *via* le menu New Project..., vous générez en fait une solution qui contient un projet du même nom. Vous pouvez toutefois ajouter autant de projets à la solution que vous le souhaitez. Le fichier App.xaml est propre à chacun des projets de type Application Silverlight, contenus au sein d'une solution. Stocker les ressources dans le fichier App.xaml d'un projet limitera leur portée à ce projet, ce qui est souvent suffisant.

- À la racine d'une page de l'application, par exemple au sein du fichier MainPage.xaml. C'est l'option proposée par défaut, au sein de l'interface de Blend, lors de la création d'une ressource. Tous les objets de la page auront accès aux ressources de cette dernière. Les objets présents dans une autre page de l'application n'y auront pas droit sauf dans certains cas spécifiques. L'écriture XAML est équivalente à celle des ressources d'application, mais UserControl remplace Application :

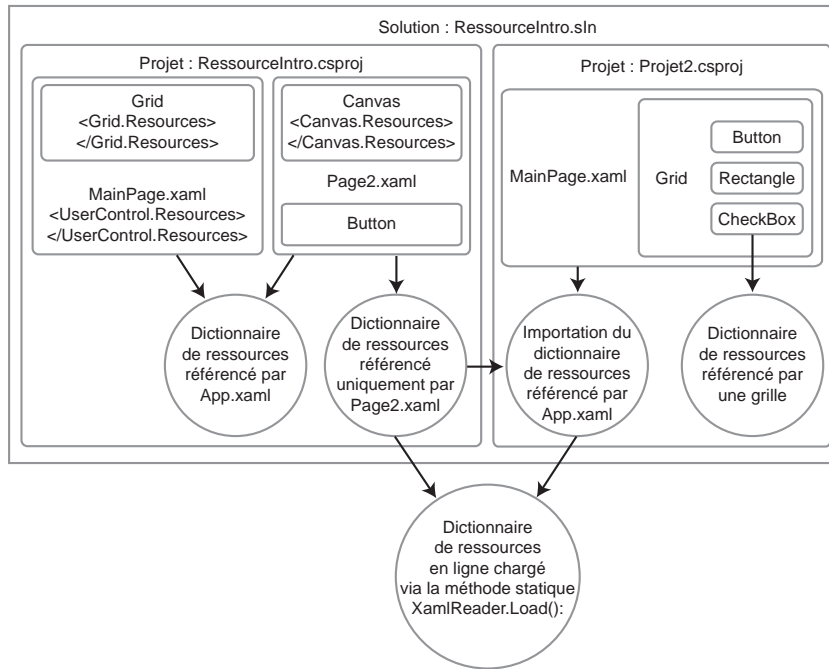
```
<UserControl.Resources>
  <System:Double x:Key="BorderOpacity">1</System:Double>
</UserControl.Resources>
```

- Directement dans la balise d'un contrôle. Un conteneur Grid posséderait ainsi une balise enfant <Grid.Resources> qui contiendrait les ressources uniquement disponibles à ses enfants. Par défaut, l'interface de Blend ne propose pas de paramétrer ce type de stockage. Cela n'est pas forcément pratique sauf cas exceptionnels.
- Dans un fichier externalisé et indépendant. Lorsque l'on parle de dictionnaires de ressources, on évoque généralement un dictionnaire de ressources externalisé. Au Chapitre 10, consacré à SketchFlow, les styles étaient centralisés dans un fichier unique nommé SketchStyles.xaml, celui-ci contenait des ressources au sein de la balise racine <ResourceDictionary>. Ce mode de stockage est sans doute le plus souple. Ses principaux avantages sont de faciliter la collaboration, le partage, la maintenance et l'accès aux ressources. Dès que vous concevez en ayant pour objectif la réutilisation et l'optimisation des flux de production, la création de dictionnaires de ressources externalisés se révèle idéale. Ils peuvent être référencés par tous les fichiers XAML qui ont alors accès à leur contenu. Ils sont par défaut référencés par App.xaml et bénéficient ainsi de la portée du projet. Ces dictionnaires ne souffrent donc pas des limitations qui existent pour les ressources directement contenues dans les fichiers App.xaml ou MainPage.xaml. Ils sont partageables entre plusieurs pages, projets et solutions. Il n'est toutefois pas toujours intéressant d'externaliser systématiquement les ressources. C'est notamment le cas lorsque vous souhaitez fournir un composant embarquant ses propres ressources. De plus, la gestion des accès peut parfois devenir délicate.

Vous pouvez consulter la Figure 11.4 illustrant différentes portées d'utilisation.

Figure 11.4

Portée d'utilisation des dictionnaires de ressources.



11.1.2.2 Déclaration XAML

Revenons sur le code XAML généré par Blend lorsque vous stockez les ressources dans une page de l'application (UserControl) :

```
<UserControl.Resources>
  <System:Double x:Key="BorderOpacity">1</System:Double>
</UserControl.Resources>
```

Cette écriture est en fait une version simplifiée de l'écriture d'un dictionnaire de ressources. Ci-dessous, vous pouvez lire un code XAML plus verbeux et précis, mais équivalent en terme de résultat :

```
<UserControl.Resources>
  <ResourceDictionary>
    <System:Double x:Key="BorderOpacity">0.4</System:Double>
  </ResourceDictionary>
</UserControl.Resources>
```

L'utilisation de la classe `ResourceDictionary` est souvent implicite car Blend adopte par défaut l'écriture simplifiée. Toutefois, lorsque vous référencerez des dictionnaires externes, vous obtiendrez une déclaration XAML complète ainsi qu'une nouvelle balise `<ResourceDictionary>`.

MergedDictionaries>. Celle-ci permet de référencer des dictionnaires supplémentaires *via* la propriété Source de la classe ResourceDictionary :

```
<UserControl.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="BobStyles.xaml">
      <ResourceDictionary Source="PatrickStyles.xaml">
    </ResourceDictionary.MergedDictionaries>
    <System:Double x:Key="BorderOpacity">0.4</System:Double>
  </ResourceDictionary>
</UserControl.Resources>
```

INFO

Il est possible de charger des dictionnaires de ressources de manière dynamique, cela peut-être très élégant et efficace dans le cas d'applications multiclients. Vous pourriez créer un ou plusieurs dictionnaires de ressources par client et les charger à la volée, selon le client connecté à l'application et sa charte graphique. Cela peut également rendre un designer plus autonome. Il pourrait modifier un dictionnaire de ressources sur son poste de travail puis remplacer celui présent sur le serveur web. De cette manière, le visuel de l'application serait mis à jour sans avoir besoin de la recompiler. Cela est réalisable grâce à la méthode statique Load de la classe XamlReader. Cette méthode est la version managée de la méthode JavaScript createFromXaml disponible depuis la version 1 de Silverlight.

11.1.2.3 Externaliser les ressources

Le stockage de ressources dans un fichier séparé est une méthodologie recherchée par les designers et les développeurs. Il suffit de modifier les ressources contenues dans le fichier, puis de recompiler l'application pour qu'elle soit mise à jour. Dans ces conditions chaque intervenant y gagne : vous instaurez une séparation des tâches tout en respectant le travail de toutes les parties impliquées. Le graphiste n'aura pas accès au code logique, et le développeur ne se préoccupera pas de savoir ce qui a changé graphiquement entre deux versions de l'application. Au final, il sera possible de mettre en place une réelle communication intermétiers de manière douce et progressive.

Comme nous l'avons vu (voir section 11.1.1), la création d'un dictionnaire de ressources peut se faire lorsque vous générez une nouvelle ressource (en cochant l'option adéquate). Il est également possible de produire un dictionnaire de ressources vide, puis de lui ajouter des ressources au fur et à mesure de la production. Le panneau Ressources d'Expression Blend est d'une grande aide pour ce genre de tâches. Ouvrez-le et cliquez sur l'icône située en haut à droite (📁). Une boîte de dialogue apparaît, vous demandant de saisir le nom du dictionnaire à créer (voir Figure 11.5).

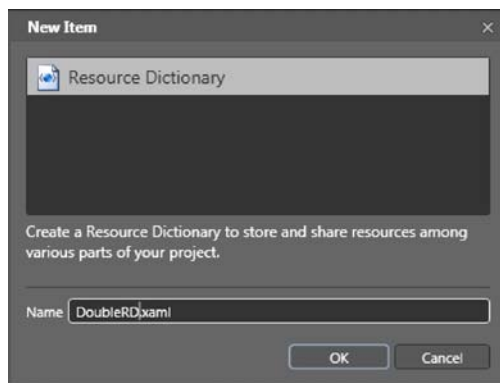
Comme les ressources stockées seront de type double, autant l'appeler DoubleRD.xaml.

INFO

Le nommage des dictionnaires est un point capital car il donne du sens et structure le projet. À brève échéance, un bon nommage facilite la compréhension du projet pour chacun des acteurs. Nous aborderons les bonnes pratiques de stockage et d'organisation des ressources tout au long des exercices de ce chapitre.

Figure 11.5

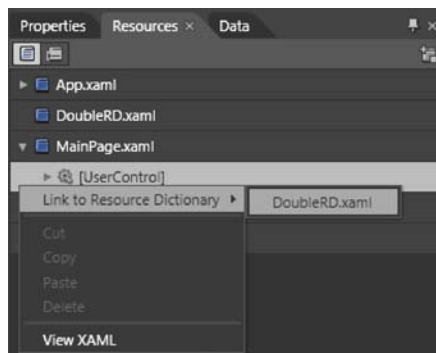
Création d'un nouveau dictionnaire de ressources.



Lorsque le dictionnaire est créé, le fichier `App.xaml` est modifié par défaut. En son sein, un lien logique, décrit en XAML, pointe désormais vers l'adresse du fichier `DoubleRD.xaml`. Il est possible de générer ou de supprimer ce type de lien *via* l'interface de Blend. Cela permet notamment de dédier des dictionnaires de ressources à certaines pages de l'application et pas à d'autres. Pour cela, chargez la page principale `MainPage.xaml` dans la fenêtre de création (si ce n'est pas déjà fait). Ensuite, au sein du panneau `Resources`, dépliez `App.xaml`, cliquez-droit sur le lien nommé `Link To : DoubleRD.xaml` et sélectionnez l'option `delete`. Une fenêtre apparaît vous indiquant qu'il est possible que certaines ressources ne soient plus correctement référencées. Ce n'est pas grave puisque notre dictionnaire est vide ; validez le choix. Vous détruisez de cette manière le lien pointant vers le dictionnaire externe sans supprimer le fichier lui-même. Nous allons maintenant ajouter un lien dans la page principale ciblant le dictionnaire. Cliquez-droit sur le `UserControl` principal, puis choisissez l'option `Link to Resource Dictionary` et le dictionnaire `DoubleRD.xaml` (voir Figure 11.6).

Figure 11.6

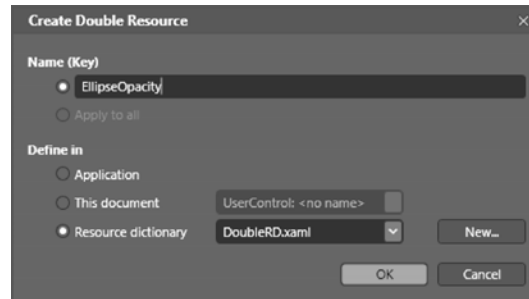
Créer un lien vers un dictionnaire de ressources.



Désormais, le dictionnaire est directement référencé par `MainPage.xaml` et ses ressources ne seront plus accessibles aux autres pages de l'application. Nous allons insérer une ressource dans ce dictionnaire en quelques clics de souris. Au sein de l'arbre visuel de `MainPage.xaml`, créez une instance d'`Ellipse`, puis cliquez sur l'icône carrée située à droite du champ `Opacity`. Choisissez l'option `Convert to New Resource`. Dans la boîte de dialogue affichée, entrez `EllipseOpacity` comme clé de ressource, sélectionnez ensuite l'option `Resource Dictionary`, vérifiez que le dictionnaire choisi dans la liste est bien `DoubleRD.xaml`, puis cliquez sur `OK` (voir Figure 11.7).

Figure 11.7

Créer un lien vers un dictionnaire de ressources.



Dans la section suivante, vous apprendrez à appliquer des ressources *via* l'interface de Blend et le langage XAML, mais également de manière dynamique en C#.

11.1.3 Appliquer des ressources

Maintenant que nous avons vu comment stocker les ressources, nous allons apprendre différentes façons de les affecter aux propriétés d'objets. La première manière de procéder consiste à cliquer sur l'icône carrée située à côté d'un champ, à sélectionner l'option Local Resource, puis de choisir la ressource dans la liste présentée. Dans tous les cas, Blend filtre les ressources accessibles en fonction du type de la propriété. Ainsi, il est impossible, *via* l'interface de Blend, d'appliquer une ressource de couleur à une propriété d'opacité car cette dernière n'accepte que les valeurs de type `double`. Dans Blend, la deuxième méthode consiste à ouvrir le panneau Resources, à déplier l'un des conteneurs de ressources, puis à glisser-déposer la ressource de votre choix sur un objet. Une liste apparaît directement au sein de la fenêtre de création. Il suffit de choisir l'une des propriétés proposées dans la liste pour lui affecter la ressource (voir Figure 11.8).

Figure 11.8

Affectation d'une ressource par glisser-déposer sur un Border.



Comme vous le constatez, Blend vous propose une liste de propriétés compatibles avec le type de la ressource déposée. La propriété `Tag` étant typée `Object`, celle-ci peut recevoir tout type de valeur. Le type `Thickness` est utilisable dans de nombreuses propriétés dont les marges intérieures et extérieures (respectivement `Padding` et `Margin`). Contrairement aux autres types de ressources, les modèles peuvent également être appliqués *via* un clic-droit sur l'objet. Nous utiliserons cette méthode à partir de la section 11.4.

INFO

Dans tous les cas et au sein d'Expression Blend, lorsqu'une ressource est appliquée à une propriété d'objet graphique (`UIElement`), la propriété est entourée d'un liseré vert visible dans le panneau Properties. Ce repère visuel est utile pour identifier les propriétés liées.

D'une toute autre manière, modifier le code déclaratif XAML est pratique lorsque vous avez besoin d'affecter rapidement une ressource à plusieurs objets à la fois. En XAML, vous définissez le plus souvent des valeurs en dur, par exemple `Opacity="0.4"`. Il est toutefois possible d'assigner des clés de ressources, des liaisons de données ou de modèles, ou encore la valeur null (via `{x:Null}`). Ces valeurs ne peuvent pas être directement écrites entre guillemets car elles sont en fait des expressions qui doivent être évaluées à la compilation ou à l'exécution (mis à part pour null qui est une valeur particulière). Leur notation se différencie par des accolades, vous obtiendrez par exemple :

```
<UserControl.Resources>
  <ResourceDictionary>
    <System:Double x:Key="BorderOpacity">0.4</System:Double>
  </ResourceDictionary>
</UserControl.Resources>
<Border x:Name="MonBorder" Opacity="{StaticResource BorderOpacity}">
```

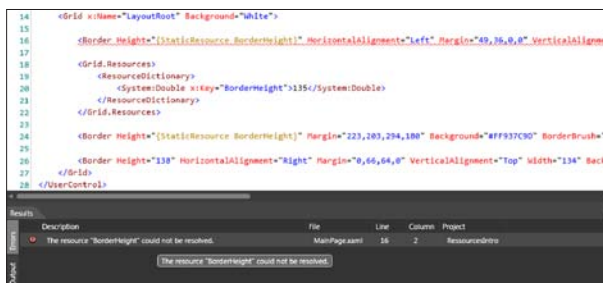
INFO

Le mot-clé `StaticResource` indique l'utilisation d'une ressource de type statique. Cela signifie que la ressource est définie à la compilation une bonne fois pour toutes. Le mot-clé `DynamicResource` n'existe pas encore du côté de Silverlight. Il est uniquement présent du côté WPF. Il permet à la propriété d'être mise à jour, durant l'exécution, chaque fois que la valeur de la ressource est modifiée de l'extérieur.

Il est important de noter que le nœud déclarant un dictionnaire de ressources doit toujours être le premier des nœuds enfants d'un document XAML. La lecture des nœuds XAML est toujours faite dans l'ordre des enfants. Si un objet référence une ressource statique qui n'est déclarée qu'après lui-même, l'interpréteur XAML lèvera une erreur (voir Figure 11.9).

Figure 11.9

Erreur levée lorsqu'une ressource est déclarée après un nœud enfant l'utilisant.



Théoriquement, ce type de problématique ne devrait pas survenir sous Expression Blend car les dictionnaires de ressources sont toujours déclarés en premier. Il peut toutefois être utile de connaître ce type de comportement.

Appliquer une ressource *via C#* est assez simple. Il vous faut tout d'abord récupérer la ressource en ciblant le dictionnaire qui la contient. L'écriture sera légèrement différente selon que le dictionnaire est déclaré dans l'application, une page ou une instance de `FrameworkElement` :

```
//Accès à la ressource dans le dictionnaire au sein de l'application
double MonBorderOpacity=(double)App.Current.Resources["BorderOpacity"];
```

```
//Dans la page principale de type MainPage
double MonBorderOpacity =(double)this.Resources["BorderOpacity"];

//Au sein de la grille principale LayoutRoot
double MonBorderOpacity =(double)LayoutRoot.Resources["BorderOpacity"];
```

Les valeurs stockées dans le dictionnaire sont toujours typées object afin de garantir un maximum de souplesse. Il vous faudra donc convertir le type de manière explicite pour récupérer une valeur utilisable :

```
foreach (UIElement uie in LayoutRoot.Children)
{
    uie.Opacity = MonBorderOpacity;
}
```

Les dictionnaires de ressources externalisés sont le plus souvent référencés par l'application *via* le fichier App.xaml. Dans ce cas, il faut les considérer comme faisant partie du dictionnaire contenu par App.Current.Resources. Vous pourriez toutefois créer une référence à un dictionnaire externalisé au sein de n'importe quelle page ou de n'importe quel composant. Ces dictionnaires sont simplement fusionnés avec ceux qui les référencent. Voici un exemple de code XAML décrivant ce type de cas de figure :

```
<UserControl
...
<UserControl.Resources>
    <ResourceDictionary>
        <System:Double x:Key="BorderOpacity">0.4</System:Double>
    </ResourceDictionary>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White">

    <Grid.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="ResourceDictionary1.xaml" />
            </ResourceDictionary.MergedDictionaries>
            <System:Double x:Key="BorderHeight">135</System:Double>
        </ResourceDictionary>
    </Grid.Resources>

    <Border Height="{StaticResource BorderHeight}"
        Opacity="{StaticResource BorderOpacity}" .../>
    <Border Height="{StaticResource BorderHeight}"
        Opacity="{StaticResource BorderOpacity}" .../>
    <Border Height="138" BorderThickness="{StaticResource
        BorderThickness}" Opacity="{StaticResource BorderOpacity}"
        CornerRadius="{StaticResource CornerRadius1}" .../>

</Grid>
</UserControl>
```

Dans la situation décrite ci-dessus, la référence d'un dictionnaire externe est déclarée dans le dictionnaire de la grille LayoutRoot. Ainsi, pour accéder à une ressource contenue dans ce dernier, il suffit de cibler la clé appropriée *via* la propriété Resources de cette grille :

```
Thickness monBorderThickness = (Thickness)LayoutRoot.
    Resources["BorderThickness"];
```

Nous avons abordé les principes majeurs inhérents à l'utilisation des ressources. Dans les prochaines sections, nous mettrons en pratique les connaissances acquises pour manipuler des ressources graphiques.

11.2 Les pinceaux

Les pinceaux représentent les ressources graphiques par excellence. Ce sont des instances d'objets héritant de la classe abstraite `Brush`. Ils sont à la base de tout visuel car les propriétés de remplissage comme `BackgroundColor`, `BorderBrush`, `Fill`, `Stroke` (`Shape`), propres aux objets graphiques, n'acceptent que ce type d'instances. Dans cette section, vous apprendrez à utiliser et à organiser vos pinceaux de manière efficace. Afin de travailler dans des conditions réelles, nous allons intégrer partiellement un lecteur multimédia. Les Figures 11.10 et 11.11 en donnent un aperçu.

Figure 11.10

L'écran d'accueil du lecteur multimédia.



Figure 11.11

La liste de lecture complète.



Le lecteur multimédia possède trois états visuels et l'utilisateur reste dans une seule page lorsqu'il utilise le lecteur. Le premier état représente un écran d'accueil composé d'une mire de bienvenue. Le deuxième est constitué d'une liste de lecture proposant différents types de fichiers multimédias lisibles par Silverlight. Lorsque l'un de ceux-ci est sélectionné, le troisième état visuel apparaît. Dans ce mode d'affichage, la liste des médias est repliée à gauche, ne laissant ainsi apparaître que

le titre et le type de chaque fichier. Dans l'espace libéré au centre et à droite, un lecteur multimédia jouera ou affichera la vidéo, le son ou l'image sélectionnés. Avant de continuer, il est nécessaire de télécharger les fichiers de travail ainsi que le projet `LecteurMultiMedia_Base` compressé au format zip. Les fichiers sont disponibles dans l'archive `Assets.zip` et le projet dans `LecteurMultiMedia_Base.zip` du dossier *chap11*. Ce dernier contient le visuel de base du lecteur multimédia, à partir duquel nous réaliserons les exercices qui suivent. Une fois les deux fichiers récupérés et décompressés sur votre disque dur, ouvrez le projet avec Expression Blend.

11.2.1 Couleurs et pinceaux

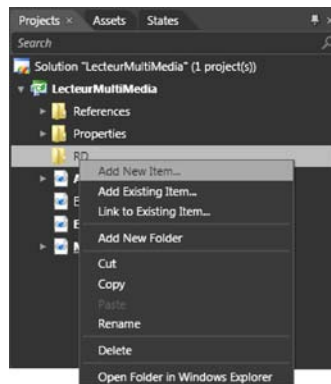
Les ressources de couleurs et les pinceaux de dégradés représentent l'une des bases fondamentales du flux de production Silverlight. Grâce à eux, il est facile de relouer tout un site en très peu de temps. Nous verrons comment créer et maintenir ces ressources.

11.2.1.1 Les ressources de couleurs

Lorsque vous concevez une application sous Expression Blend, l'une des premières étapes consiste à sauvegarder les couleurs de base définies par la charte graphique. Comme vous les utiliserez tout au long du projet, le mieux est de les centraliser au sein d'un dictionnaire de ressources externalisé. Dans le panneau Projects, créez un nouveau répertoire nommé RD et sélectionnez-le. Ce répertoire contiendra tous les dictionnaires de ressources utilisés dans l'application. Ajoutez-y un nouveau dictionnaire de ressources nommé Couleurs. Vous avez la possibilité d'utiliser le panneau Resources (référez-vous alors à la section 11.1). Une autre manière plus directe consiste à cliquer-droit sur le répertoire RD, puis à choisir l'option `Add New Item...` (voir Figure 11.12).

Figure 11.12

Menu affiché au clic-droit sur le répertoire RD.



Dans la boîte de dialogue apparaissant, choisissez `Resource Dictionary` et spécifiez `Couleurs.xaml` comme nom de fichier. Cliquez ensuite sur `OK` pour valider. Nous allons désormais créer chaque ressource de couleur dont nous avons besoin. Les couleurs nécessaires sont présentées au Tableau 11.2.

Tableau 11.2 : Liste des couleurs à sauvegarder comme ressource

<i>Couleurs</i>	<i>Noms de ressource x:Key</i>	<i>Valeurs hexadécimales</i>
Gris transparent	PanelBackground	#CD9E9E9E
Gris clair	ControlsBackground	#FFDCDBD8
Gris foncé	PlayerBackground	#FF444444
Rose foncé	VideoTheme	#FFC70963
Vert	LayoutModeTheme	#FF7AC909
Yellow	SoundTheme	#FFFFC800
Orange	TimeCodeTheme	#FFFF5900
Cyan	BitmapTheme	#FF08AFC8

Pour créer une ressource de couleur, il suffit de sélectionner un objet puis de cliquer sur l'une de ses propriétés de remplissage. Vous pouvez référencer la valeur de la propriété sous forme de ressource de couleur unie. Par exemple, vous pouvez sélectionner la grille principale LayoutRoot, puis sa propriété Background. Spécifiez ensuite la couleur souhaitée. Il suffit de cliquer sur l'icône représentant une double flèche située à gauche de la valeur hexadécimale (voir Figure 11.13).

Figure 11.13

Référencer une couleur unie sous forme de ressource de couleur.



Dans la boîte de dialogue Create Color Resource, définissez PlayerBackground comme clé de ressource, puis enregistrez la clé dans le dictionnaire Couleurs.xaml que nous avons créé précédemment. À cette fin, choisissez la dernière option de stockage. Vous venez de créer votre première ressource de couleur. Le code XAML généré est le suivant :

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Color x:Key="PlayerBackground">#FF444444</Color>
</ResourceDictionary>
```

L'écriture est relativement simple : une instance de la classe Color est stockée dans le dictionnaire. Elle accepte par défaut une valeur 32 bits (4 octets) décrivant quatre couches, alpha, rouge, vert et bleu, exprimées en hexadécimal. Par exemple, lorsqu'une couleur est complètement opaque,

elle débute par le couple FF qui indique la valeur hexadécimale maximale de la couche de transparence. Il est également possible de fournir une propriété statique de la classe Colors. Vous pourriez ainsi écrire :

```
<Color x:Key="PlayerBackground">Cyan</Color>
```

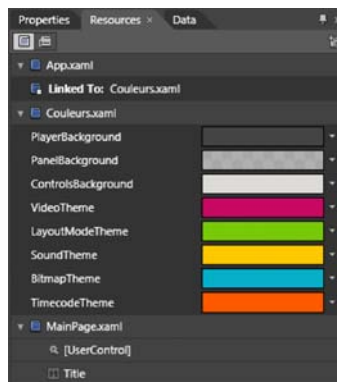
Finalement, il est encore plus rapide de dupliquer le nœud XAML déjà présent pour générer les ressources de couleurs listées au Tableau 11.2. Voici le code XAML généré une fois cette étape passée :

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Color x:Key="PlayerBackground">#FF444444</Color>
  <Color x:Key="PanelBackground">#CD9E9E</Color>
  <Color x:Key="ControlsBackground">#FFDCDBD8</Color>
  <Color x:Key="VideoTheme">#FFC70963</Color>
  <Color x:Key="LayoutModeTheme">#FF7AC909</Color>
  <Color x:Key="SoundTheme">#FFFFC800</Color>
  <Color x:Key="BitmapTheme">#FF08AFC8</Color>
  <Color x:Key="TimecodeTheme">#FFF5900</Color>
</ResourceDictionary>
```

Ouvrez maintenant le panneau Resources et déployez le dictionnaire Couleurs.xaml, vous y trouverez toutes les couleurs que vous avez définies sous forme visuelle (voir Figure 11.14).

Figure 11.14

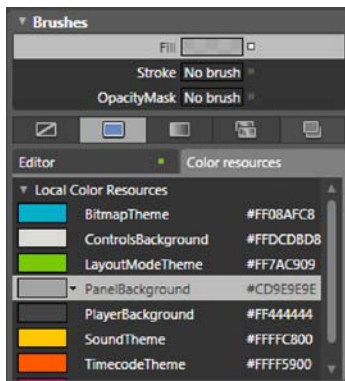
Accéder aux ressources de couleurs.



Maintenant que les ressources de couleurs sont créées, nous pouvons facilement les appliquer aux objets présents dans l'arbre visuel. Au sein de l'arbre visuel, dans la grille nommée Panneau-Liste, sélectionnez le tracé Panneau. Dans le panneau Brushes, sélectionnez sa couleur de remplissage, puis l'onglet Color Resources. Choisissez la ressource de couleur PanelBackground pour l'affecter en tant que remplissage (voir Figure 11.15).

Figure 11.15

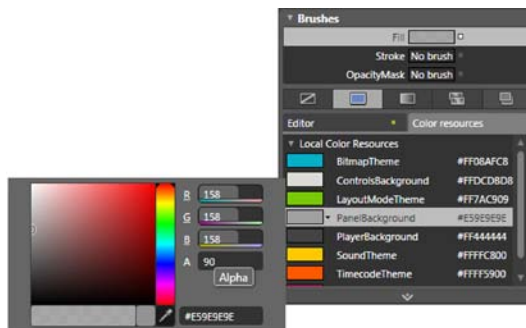
Appliquer une ressource de couleur via le panneau des propriétés.



Il est également possible de glisser-déposer une ressource de couleur directement sur un objet, puis de choisir la propriété sur laquelle vous souhaitez l'apposer. Comme nous l'avons évoqué au début de ce chapitre, l'avantage principal des ressources est de mettre à jour tous les objets graphiques qui y font référence lorsqu'elles sont modifiées. Vous pouvez mettre à jour une ressource de couleur de deux manières différentes : soit *via* le panneau Resources, soit directement dans l'onglet Color Resources du panneau Properties. Dans ces deux cas il suffira de cliquer sur la case de couleur pour modifier son remplissage. Comme vous l'avez peut-être remarqué, la couleur de remplissage a légèrement changé lorsque nous avons appliqué la ressource. Si vous souhaitez moins de transparence sur cette couleur, il suffit de modifier la ressource *via* le premier couple hexadécimal ciblant l'opacité (voir Figure 11.16).

Figure 11.16

Modifier une ressource de couleur.



Comme vous le constatez, modifier la ressource change les objets l'utilisant. Cachez les grilles nommées `PanneauListe` et `PseudoListe` afin de faire apparaître l'écran d'accueil. Affectez-lui la même ressource et faites de même avec tous les objets utilisant la même couleur au sein du projet.

11.2.1.2 Les pinceaux de couleurs

Les pinceaux de couleurs sont des ressources différentes en cela qu'elles contiennent également des options de remplissage. Elles héritent de la classe abstraite `Brush`. On en trouve trois catégories : les pinceaux de couleur unie de type `SolidColorBrush`, ceux de dégradé linéaire `LinearGradientBrush` et les pinceaux de dégradé radial `RadialGradientBrush`. Les premiers sont uti-

lisés quand vous définissez une couleur unie, par exemple #FFFF5900, ou lorsque vous affectez une ressource de couleur simple. Voici l'écriture XAML correspondante aux deux affectations :

```
<Path x:Name="FondAccueil" Stretch="Fill" StrokeThickness="1" ...>
  <Path.Fill>
    <SolidColorBrush Color="{StaticResource PanelBackground}" />
  </Path.Fill>
</Path>
<TextBlock x:Name="Title" ...>
  <TextBlock.Foreground>
    <SolidColorBrush Color="#FFFC800" />
  </TextBlock.Foreground>
</TextBlock>
```

Comme vous le constatez, les propriétés de remplissage acceptent des instances de type Brush. Le langage XAML accepte toutefois des raccourcis d'écriture impossibles du côté C#, ainsi on pourra écrire :

```
<Rectangle Height="35" Width="53" Fill="#FF000000" ... />
```

Alors qu'affecter une ressource ou une couleur en C# devra toujours passer par l'affectation d'une instance de Brush. Dans le cas d'une affectation de ressource de couleur cela donnera :

```
//Lorsqu'on récupère une ressource, on crée un pinceau de couleur unie
SolidColorBrush scb = new SolidColorBrush();

//on récupère une ressource qu'on transtype en tant qu'instance de Color
//puis on l'affecte à la propriété Color du pinceau
scb.Color = (Color)App.Current.Resources["PanelBackground"];

//Pour finir on applique le pinceau à la propriété Fill
//d'un exemplaire de Shape
Panneau.Fill = scb;
```

Nous pourrions tout aussi bien affecter le pinceau à la propriété Background d'une instance de Control. Pour affecter une couleur hexadécimale directement, écrivez :

```
//Lorsqu'on affecte une valeur brute, on crée un pinceau de couleur unie
SolidColorBrush scb = new SolidColorBrush();

//on utilise la méthode FromArgb de la classe Color
scb.Color = Color.FromArgb(0xFF,0xFF,0x59,0x00);

//On peut également utiliser une propriété statique de la classe Colors
scb.Color = Colors.Orange;

Panneau.Fill = scb;
```

Dès lors, il peut être intéressant pour le développeur que le graphiste crée des ressources de pin-
ceaux car elles sont faciles à affecter et conservent en mémoire des propriétés de remplissage
personnalisées *via* leurs propriétés Transform et RelativeTransform. L'intérêt d'un tel scénario
pour les instances de SolidColorBrush est moins grand car la couleur est unie. Toutefois, pour
les pin-
ceaux de dégradé, le potentiel d'optimisation est assez élevé. Afin de générer une ressource
pin-
ceau, quel que soit son type, il suffit de cliquer sur l'icône carrée située à droite d'un champ
de remplissage, puis de choisir l'option Convert to New Resource... Sélectionnez le TextBlock
nommé Title, en bas à gauche de l'interface. Affectez-lui un dégradé linéaire, par défaut du noir

au blanc. Nous allons créer une nouvelle ressource de type pinceau. Pour cela, le mieux est de la stocker, soit dans un dictionnaire prévu à cet effet, soit dans celui qui stocke déjà les couleurs.

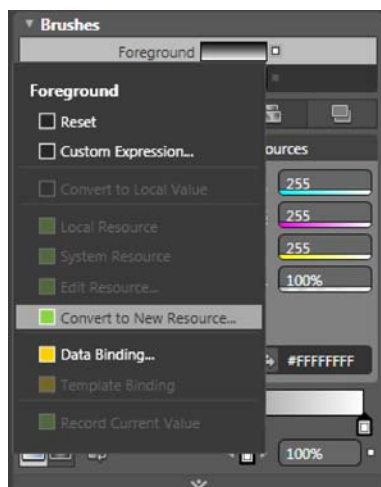
ATTENTION

Si vous créez deux dictionnaires de ressources et que les ressources de l'un font référence à celles contenues dans le second, il est alors nécessaire de créer une liaison. Ainsi, si vous décidez de créer un dictionnaire de pinceaux et que celui-ci utilise des ressources de couleurs unies définies dans un autre dictionnaire de ressources, il faudra les relier *via* le panneau Ressources.

Attention, avant toute action, sélectionnez le répertoire RD au sein du panneau Projects. Créez un nouveau dictionnaire nommé `BobifysBrushes.xaml` afin d'y ajouter le pinceau de dégradé linéaire. Vous pouvez maintenant convertir le dégradé en ressource pinceau (voir Figure 11.17).

Figure 11.17

Générer un pinceau de dégradé linéaire.



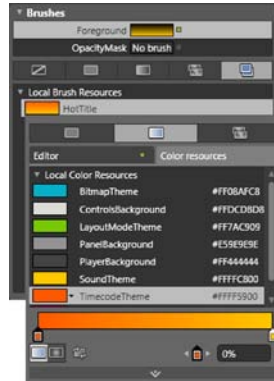
Dans la boîte de dialogue affichée, sélectionnez le dictionnaire de ressources `BobifysBrushes.xaml` comme espace de stockage, puis nommez la ressource `HotTitle`. Voici le code XAML généré lors de la création du pinceau :

```
<LinearGradientBrush x:Key="HotTitle" EndPoint="0.5,1" StartPoint="0.5,0">
  <GradientStop Color="Black" Offset="0"/>
  <GradientStop Color="White" Offset="1"/>
</LinearGradientBrush>
```

Comme vous le constatez, le pinceau est lui-même constitué de couleurs, vous pouvez facilement les remplacer par des ressources de couleurs. L'avantage de procéder de cette manière est de pouvoir mettre à jour le pinceau de dégradé *via* la modification des ressources de couleurs. À cette fin, cliquez sur la ressource pour accéder au sélecteur de couleur. Sélectionnez le premier point de couleur du dégradé et appliquez-lui la ressource "SoundTheme" *via* l'onglet Color resources. Répétez l'opération pour le second point de couleur du dégradé et appliquez-lui la ressource "TimecodeTheme" (voir Figure 11.18).

Figure 11.18

Modifier le pinceau de dégradé.



Le pinceau de dégradé est mis à jour au sein du dictionnaire de ressources, son code a légèrement changé car ses balises GradientStop référencent maintenant des ressources de couleurs :

```
<LinearGradientBrush x:Key="HotTitle" EndPoint="0.5,1" StartPoint="0.5,0">
  <GradientStop Color="{StaticResource TimecodeTheme}" Offset="0"/>
  <GradientStop Color="{StaticResource SoundTheme}" Offset="1"/>
</LinearGradientBrush>
```

À ce stade nous avons presque fini, si vous compilez vous risquez d'avoir des erreurs. C'est tout à fait logique, le dictionnaire BobifysBrushes.xaml utilise des ressources de couleurs appartenant à un autre dictionnaire. Celles-ci sont accessibles *via* l'interface de Blend car les deux dictionnaires sont référencés par App.xaml. Toutefois, il faut créer une liaison au sein du dictionnaire de pinceaux pointant vers celui contenant les couleurs afin qu'il puisse les utiliser. Dans cette optique, utilisez l'interface de Blend de la manière vue à la section 11.1.2.3. Le code XAML généré par Blend donne :

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Couleurs.xaml"/>
  </ResourceDictionary.MergedDictionaries>
  <LinearGradientBrush x:Key="HotTitle" EndPoint="0.5,1"
    StartPoint="0.5,0">
    <GradientStop Color="{StaticResource TimecodeTheme}" Offset="0"/>
    <GradientStop Color="{StaticResource SoundTheme}" Offset="1"/>
  </LinearGradientBrush>
</ResourceDictionary>
```

Vous remarquez qu'il n'y a pas besoin de spécifier l'accès au répertoire RD comme ce que nous avons dans le fichier App.xaml :

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="LecteurMultiMedia.App">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="RD/Couleurs.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

```

        <ResourceDictionary Source="RD/BobifysBrushes.xaml" />
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
</Application.Resources>
</Application>

```

C'est tout à fait logique car les dictionnaires de ressources sont dans le même répertoire. L'avantage d'utiliser l'interface de Blend est que les accès seront résolus automatiquement pour vous dans de nombreux cas. Recompilez le projet, cette fois l'application est correctement affichée dans le navigateur.

Revenons sur la traduction du code XAML exprimant le dégradé linéaire. Les propriétés `StartPoint` et `EndPoint` définissent le début et la fin du dégradé sous forme de coordonnées relatives `x` et `y`. Ainsi la valeur `0.5` signifie que leurs coordonnées sont situées à 50 % de la largeur totale de l'objet ; la valeur `1` de la propriété `EndPoint` indique que la fin du dégradé est située à 100 % de la hauteur totale de l'objet (donc sur l'axe `y`). Il est possible de modifier ces propriétés simplement en dépliant l'onglet situé sous la barre des dégradés. Vous pouvez également afficher les coordonnées absolues des points de départ et d'arrivée grâce aux champs qu'il propose (voir Figure 11.19).

Figure 11.19

Les options de remplissage des dégradés.



Pour apprendre le fonctionnement des dégradés, le mieux est encore de modifier les propriétés exposées dans l'onglet, sur des dégradés simples.

INFO


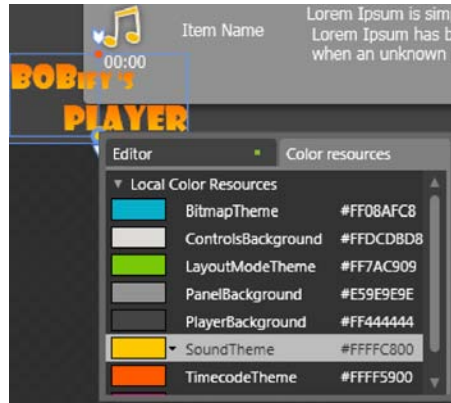
En tant que designer, il est possible que vous soyez tenté d'utiliser le manipulateur des dégradés dans la barre d'outil () afin de mettre à jour le pinceau (voir Figure 11.20). C'est une mauvaise idée : lorsque vous essayez de modifier le pinceau de dégradé ainsi, vous supprimez automatiquement son affectation au remplissage concerné. Ce comportement est logique puisque la configuration du dégradé est définie dans le pinceau lui-même. Blend considère que vous ne souhaitez plus utiliser le pinceau et génère un nouveau dégradé à partir du pinceau existant. Il est donc conseillé de finir complètement le dégradé avant de le transformer en pinceau. Vous pouvez également créer un nouveau dégradé et remplacer les balises au sein du pinceau par celles du nouveau dégradé. De cette manière, le pinceau est mis à jour plus facilement.


Figure 11.20

Essai de modification
du pinceau via l'outil
des dégradés.



Il pourrait être utile de sauvegarder l'un des dégradés gérant le reflet lumineux de certains éléments de l'interface. Dans la grille nommée *SliderTimeCode*, sélectionnez le deuxième enfant de type *Path*. Il possède un remplissage qui génère un léger effet de réflexion. Définissez-le en tant que ressource de dégradé de la même manière que précédemment, nommez-le *Reflection* et stockez-le dans le dictionnaire dédié aux pinceaux. Le code généré est légèrement différent car il s'agit cette fois d'un dégradé radial :

```
<RadialGradientBrush x:Key="Reflection" RadiusX="2.82514"
RadiusY="4.23125"
Center="0.504054,-2.13556" GradientOrigin="0.504054,-2.13556">
  <RadialGradientBrush.RelativeTransform>
    <TransformGroup/>
  </RadialGradientBrush.RelativeTransform>
  <GradientStop Color="#00FFFFFF" Offset="0.495" />
  <GradientStop Color="#7FFFFFFF" Offset="0.618605" />
  <GradientStop Color="#00FFFFFF" Offset="0.627" />
</RadialGradientBrush>
```

Comme vous le constatez, quatre propriétés sont utilisées pour le définir : *RadiusX*, *RadiusY*, *Center* et *GradientOrigin*. Vous notez qu'il est également possible d'affecter des transformations relatives aux dégradés grâce au nœud *RadialGradientBrush.RelativeTransform* (également valables pour les pinceaux d'images et de vidéos). Au sein de *Blend*, vous y avez accès *via* le panneau *Properties* ou la barre d'outils. Restez appuyé sur l'outil des dégradés situé dans la barre d'outils des dégradés pour faire apparaître le modificateur de transformations relatives des dégradés ().

11.2.2 Les pinceaux d'images et de vidéos

Les images ou les vidéos sont, la plupart du temps, chargées de manière dynamique pour créer un portfolio, un catalogue ou une webtv. L'utilisation de pinceaux d'images (ou de pinceaux vidéo) est spécifique à certaines problématiques de conception. Cela est particulièrement utile lorsqu'une même image doit être affichée plusieurs fois ou que l'utilisation d'un tracé vectoriel se révèle moins pratique dans le flux de production. Dans ce cas, l'image n'est qu'une seule fois en mémoire et son pinceau peut être affiché plusieurs fois sans affecter les performances de l'application.

Les étapes nécessaires à la création de pinceaux d'images ou de vidéos sont similaires. Assez étrangement, il est obligatoire d'utiliser un composant de type `Image` ou `MediaElement` pour fabriquer ce type de pinceau. Voici la procédure de création :

1. Vous devez commencer par importer une ressource média de type image ou vidéo compatible (png, jpg, wmv, etc.). Cette ressource sera par défaut compilée au sein du fichier xap dans la dll du même nom. C'est pourquoi il faut éviter qu'elle ne pèse trop lourd.
2. Ensuite il faut la double-cliquer au sein du panneau Projects afin de la positionner et de l'afficher *via* l'utilisation d'un composant adéquat. Ainsi, le composant `MediaElement` jouera une vidéo, le composant `Image` affichera quant à lui les bitmaps.
3. La dernière étape consiste à sélectionner le composant généré, puis à choisir depuis le menu `Tools > Make Brush Ressource > Make Image Brush Resource` ou `Make Video Brush Resource` selon le type de pinceau que vous souhaitez créer.

Via le panneau Projects, créez un nouveau répertoire nommé `bitmaps`. Cliquez-le droit, puis sélectionnez l'option `Add Existing Item...` Choisissez le répertoire `ImageBrushes`, dans le dossier Assets que vous avez téléchargé et décompressé précédemment. Sélectionnez les trois images au format png placées dans le répertoire et importez-les. Double-cliquez-les afin de créer trois composants `Image` dans le conteneur `LayoutRoot`. Sélectionnez le répertoire RD, puis le composant `Image` correspondant à l'icône de la note de musique, faites-en un pinceau d'image *via* le menu `Tools`. Choisissez de créer un nouveau dictionnaire de ressources dédié aux pinceaux d'images et de vidéos. Nommez le dictionnaire `VideosImagesBrushes.xaml` et le pinceau `IconMusic`. Procédez de même avec les deux autres instances d'`Image` et stockez les pinceaux dans le nouveau dictionnaire. Supprimez ensuite les composants `Image` car ils n'ont plus aucune utilité. Voici le code XAML généré :

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ImageBrush x:Key="IconBitmap" ImageSource="bitmaps/Bitmap.png"/>
    <ImageBrush x:Key="IconMusic" ImageSource="bitmaps/Music.png"/>
    <ImageBrush x:Key="IconVideo" ImageSource="bitmaps/video.png"/>
</ResourceDictionary>
```

Il est souvent nécessaire de régler certains détails pour que tout fonctionne correctement. Ouvrez le panneau Resources et déployez le dictionnaire `VideosImagesBrushes.xaml` afin de voir si les images apparaissent bien. En réalité, un bogue de l'interface de Blend peut être assez déroutant : notre dictionnaire est dans le répertoire RD alors que les images sont dans le répertoire `bitmaps`, le chemin d'accès décrit dans le XAML au-dessus est donc faux. Voici le code XAML à modifier pour que les pinceaux d'images soient correctement affichés :

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ImageBrush x:Key="IconBitmap" ImageSource="../bitmaps/Bitmap.png"/>
    <ImageBrush x:Key="IconMusic" ImageSource="../bitmaps/Music.png"/>
    <ImageBrush x:Key="IconVideo" ImageSource="../bitmaps/video.png"/>
</ResourceDictionary>
```

INFO

Comme vous le constatez, nous avons créé un nouveau dictionnaire dédié à ce type de ressources. Plusieurs contraintes techniques propres au flux de production nous orientent dans cette direction. La plus importante est liée au fait que les pinceaux d'images font référence à des objets physiques propres au projet. Si vous devez par la suite exporter les pinceaux d'images ou de vidéos dans un autre projet, il vous faudra également dupliquer le répertoire physique contenant les images ou les vidéos. Cette contrainte n'existe pas pour les dictionnaires de pinceaux de dégradés ou les ressources de couleurs qui sont autonomes par nature. Ces ressources sont en effet décrites entièrement en XAML, exporter le dictionnaire suffit. Dans ces conditions autant scinder ces deux types de pinceaux du point de vue de l'organisation. La deuxième raison est plus prosaïque : nous aurions pu stocker les pinceaux dans la page principale, mais autant organiser les ressources sainement dès le départ et faciliter la maintenance de l'application.

Afin d'appliquer les pinceaux, supprimez les enfants des composants Grid anonymes situés au sein des grilles respectivement nommées RadioMusic, RadioImage, RadioVideo. *Via* le mode d'édition XAML transformez la grille en Rectangle. Cette tâche est assez aisée : il vous suffit de supprimer les propriétés non supportées par le Rectangle. Voici le code XAML avant le changement envisagé :

```
<Grid x:Name="RadioVideo" Height="53" HorizontalAlignment="Left"
      Margin="166,1,0,0" VerticalAlignment="Top" Width="49">
  <TextBlock FontFamily="Tahoma" FontSize="12"
    HorizontalAlignment="Center"
    VerticalAlignment="Bottom" Margin="0,0,0,-2"
    RenderTransformOrigin="0.5,0.5"><Run Text="Video"
    Foreground="#FFFFFFFF" /></TextBlock>
  <Grid HorizontalAlignment="Center" VerticalAlignment="Center" Width="28"
    Height="32" Margin="0,0,0,4">
    <Path .../>
    <Path ...>
      <Path.Fill>
        <LinearGradientBrush StartPoint="0.497509,0.0614041"
          EndPoint="0.497509,0.929826">
          <GradientStop Color="#00FFFFFF" Offset="0" />
          <GradientStop Color="#BFFFFFFF" Offset="1" />
        </LinearGradientBrush>
      </Path.Fill>
    </Path>
  </Grid>
</Grid>
```

Voici le code XAML après modification :

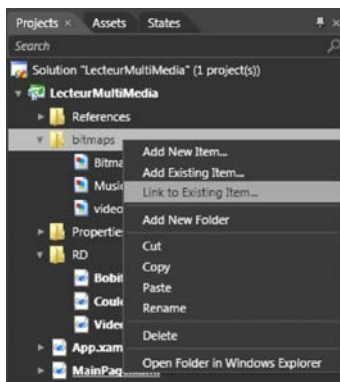
```
<Grid x:Name="RadioVideo" Height="53" HorizontalAlignment="Left"
      Margin="166,1,0,0" VerticalAlignment="Top" Width="49">
  <TextBlock FontFamily="Tahoma" FontSize="12"
    HorizontalAlignment="Center"
    VerticalAlignment="Bottom" Margin="0,0,0,-2"
    RenderTransformOrigin="0.5,0.5"><Run Text="Video"
    Foreground="#FFFFFFFF" /></TextBlock>
  <Rectangle HorizontalAlignment="Center" VerticalAlignment="Center"
    Width="48" Height="32" Margin="0,0,0,4"
    Fill="{StaticResource IconVideo}" />
</Grid>
```

Vous constatez que ce code est beaucoup plus sobre et surtout très efficace en terme de flux de production. Un graphiste n'aura qu'à mettre à jour les icônes *via* n'importe quel outil externe, tel

que Photoshop. Il n'aura pas besoin de connaître Expression Blend pour mettre à jour l'application, il lui suffira simplement de recompiler pour prendre en compte les icônes modifiées. L'une des pratiques dans ce domaine consiste à ne pas importer directement les ressources de type média dans le projet Silverlight, mais plutôt à créer un lien pointant sur elles. Vous pouvez accéder à cette option *via* un simple clic-droit au sein du panneau Projects en sélectionnant l'option Link To Existing Item... (voir Figure 11.21).

Figure 11.21

Ajouter une liaison pointant vers un fichier externe au projet.



Cette méthodologie est encore plus transparente pour le designer, mais donnera moins de souplesse car elle force à ne pas déplacer ou réorganiser les ressources médias liées trop souvent à un projet.

INFO

Il est également possible de créer un pinceau d'image en dur sans le définir comme ressource. Dans cette optique, il vous faudra utiliser le quatrième onglet, nommé Tile Brush au survol, situé au sein du panneau Brushes, puis choisir le média que vous souhaitez utiliser en tant que remplissage. Toutefois cette méthodologie est anecdotique et ne devrait pas être privilégiée.

Du côté C#, appliquer un pinceau d'image est très facile, car le développeur peut à tout moment cibler la ressource créée par le graphiste. Ainsi au lieu d'avoir un code verbeux du type :

```
//1 - on crée un objet Uri qui pointe vers notre fichier image à utiliser
Uri adresseImage = new Uri("video.png",UriKind.Relative);

//2 - on crée un objet de type BitmapImage en lui spécifiant
//l'adresse relative que nous venons de définir
BitmapImage bi = new BitmapImage( adresseImage );

//3 - On crée un pinceau d'image
ImageBrush ib = new ImageBrush();

//4 - on spécifie l'image source de ce pinceau
ib.ImageSource = bi;

//5 - on affecte la propriété Fill d'une instance de Shape
monRectangle.Fill = ib;
```

On obtient quelque chose de beaucoup plus simple du point de vue développement, mais également dans la répartition des tâches de chaque pôle métier. Le développeur n'a pas besoin de

connaître l'emplacement ou le nom de l'image à afficher. Il lui suffit de connaître le nom de la ressource. À cette fin, le mieux est encore de se baser sur une nomenclature définie dès le départ du projet en concertation avec l'ensemble des acteurs de la production :

```
//on récupère la ressource qu'on transtype en ImageBrush
ImageBrush ib = (ImageBrush)App.Current.Resources["IconVideo"];

//Dans ce cas, on applique le pinceau d'image à la propriété Fill
//d'une instance de Shape
monRectangle.Fill = ib;
```

À la section 11.3, nous découvrirons en quoi les pinceaux permettent en général d'éviter la prolifération excessive de styles et modèles de composant.

11.3 Les polices de caractères

Contrairement aux technologies habituelles du Web reposant sur la mise en forme XHTML, Silverlight offre de nombreux avantages en matière de conception graphique. L'un d'eux est de pouvoir afficher des polices de caractères personnalisées et de permettre des mises en forme assez avancées. L'affichage étant géré par un moteur vectoriel, certaines bonnes pratiques de conception doivent être connues pour bénéficier d'un affichage propre et performant.

11.3.1 Afficher et mettre en forme du texte

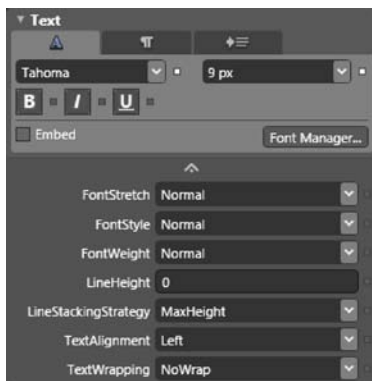
Au sein de Silverlight, deux mécanismes de mise en forme de texte cohabitent. Celui qui est le plus couramment utilisé repose sur les composants `TextBlock`, `TextBox` et sa sous-classe `PasswordBox`. Tous ces contrôles sont directement hérités de `FrameworkElement`. Ils n'ont donc pas de propriété `Template` permettant de modifier leur modèle. Autrement dit, leur affichage est uniquement déterminé par le texte qu'ils contiennent. Le composant `TextBlock` a pour unique but d'afficher du texte non interactif. Il est très simple d'utilisation et possède des options de mise en forme assez similaires à ce que vous trouverez dans n'importe quel éditeur de texte. Les contrôles `TextBox` et `PasswordBox` sont des champs de saisie utilisateur n'ayant pas pour objectif de proposer d'options de mises en forme avancées. Le deuxième mécanisme de mise en forme consiste à utiliser la classe `Glyphs` qui fournit une API bas niveau de formatage de texte. Celle-ci est réellement utile lorsque vous avez besoin d'un contrôle total et dynamique sur le texte affiché dans Silverlight. Voici un exemple XAML simple de l'écriture d'une instance de `Glyphs` :

```
<Glyphs Height="100" Width="100" Fill="Red"
  FontRenderingEmSize="12" FontUri="Fonts/tahoma.ttf"
  UnicodeString="Hey" OriginX="10" OriginY="20" Indices=",85;,64;,67;6"/>
```

Nous n'allons pas nous étendre sur l'API bas niveau car cela sortirait du cadre de ce livre. Nous apprendrons donc à utiliser le composant `TextBlock` qui répond à 80 % des cas d'utilisation. Grâce à ce dernier, nous pouvons appliquer une mise en forme générale *via* ses propriétés, qui sont accessibles dans la section `Text` du panneau `Properties` (voir Figure 11.22).

Figure 11.22

Les propriétés de mise en forme du composant TextBlock.



Voici une liste non exhaustive des propriétés que vous pouvez configurer :

- Le choix de la police de caractères (Arial, Verdana, Calibri, etc.). En pratique, lorsque vous choisissez une police différente de celle proposée par défaut, il vous faudra l'embarquer dans 90 % des cas. Nous abordons plus en détail l'intégration de polices à la section 11.3.2.
- La taille de police exprimée en pixels ou en points selon la configuration de l'interface de Blend. Pour choisir l'une de ces deux unités, ouvrez le menu Tools > Options... et dans l'onglet Units, choisissez Points ou Pixels. Notez que dans tous les cas, le XAML est formaté en tenant compte de valeurs exprimées en pixels.
- Afficher le texte en gras, italique ou souligné est réalisé grâce aux propriétés respectives FontStyle, FontWeight et TextDecoration. Pour avoir un vrai gras, il faudra utiliser une police correspondante, dans le cas contraire il est simulé. C'est exactement le même principe avec l'italique.
- Il est également possible de modifier l'espacement entre les caractères (l'interlettrage) en utilisant la propriété FontStretch. La police n'est toutefois pas modifiée dynamiquement. Lorsque vous l'utilisez, cette propriété fait référence à une police correspondant à l'espacement spécifié (Condensed par exemple). Lorsque cette police n'est pas disponible, modifier la valeur de cette propriété ne change rien et ne renvoie aucune erreur ou alerte.
- Vous pouvez aligner le texte à droite, à gauche ou au centre, via la propriété TextAlignment qui accepte une valeur de l'énumération du même nom. Malheureusement, il n'est pas possible de justifier le texte car la valeur TextAlignment.Justify n'existe que dans WPF.
- Le retour à la ligne automatique est gérée grâce à la propriété TextWrapping et à l'énumération correspondante contenant les valeurs Wrap et NoWrap. Définir cette propriété à NoWrap empêchera tout retour à la ligne automatique. Cela est visible lorsque la largeur du champ texte est définie en pixels et non en mode automatique. *A contrario*, si la propriété TextWrapping est affectée de la valeur Wrap, que vous fixez la largeur du TextBlock en dur et que la hauteur est définie en mode Auto, alors le texte contenu ira automatiquement à la ligne. La hauteur du champ s'adaptera dynamiquement à la quantité de texte à afficher.
- Pour finir, il est possible de spécifier une hauteur de ligne via la propriété LineHeight.

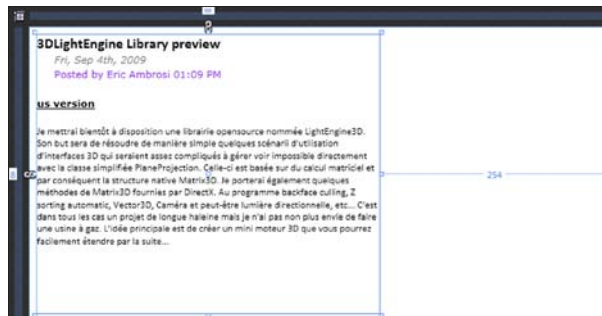
Toutes ces capacités sont assez utiles, mais elles demeurent insuffisantes en l'état car elles sont propres à TextBlock. *A priori*, cela nous oblige à avoir autant de TextBlock que de mises en

forme. Fort heureusement, les `TextBlock` ont la capacité de posséder des éléments enfants mis en forme et qui sont des instances de la classe `Inline`. Ainsi, `TextBlock` possède la propriété `Inlines` qui n'est rien d'autre qu'une collection d'instances de type `Inline`. La classe `Inline` est abstraite, elle sert de base à deux classes concrètes en héritant : `Run` et `LineBreak`. Les instances de type `Run` possèdent presque toutes les capacités d'un `TextBlock`. Grâce à ces dernières, vous pouvez appliquer des mises en forme différentes au sein d'un même champ `TextBlock`. Ces balises sont générées dans Blend lorsque vous double-cliquez dans un `TextBlock`, que vous sélectionnez une portion spécifique du texte contenu, puis que vous définissez les options de mise en forme. Il est également possible de le faire *via* le panneau des propriétés en cliquant sur la propriété `Inlines` et en ajoutant chaque portion de texte manuellement. Les instances de `LineBreak` permettent, quant à elles, de faire des retours à la ligne. Voici le code XAML d'un exemple de texte mis en forme (voir Figure 11.23) *via* l'utilisation de ces balises dans Blend :

```
<TextBlock Margin="70,87,192,86" TextWrapping="Wrap">
  <Run FontFamily="Calibri" FontSize="16" FontWeight="Bold"
    Text="3DLightEngine Library preview"/>
  <LineBreak/><Run FontStyle="Italic" Foreground="Gray" Text="    Fri,
    Sep 4th, 2009"/>
  <LineBreak/><Run Foreground="BlueViolet" Text="    Posted by Eric
    Ambrosi 01:09 PM"/>
  <LineBreak/><LineBreak/><Run FontWeight="Bold"
    TextDecorations="Underline" Text="us version"/>
  <LineBreak/><LineBreak/><Run FontFamily="Calibri" Text="Je mettrai
    bientôt à disposition une librairie ..."/>
</TextBlock>
```

Figure 11.23

Un TextBlock contenant plusieurs mises en forme.



Cette écriture est simplifiée, vous pourriez tout aussi bien affecter la propriété `Inlines` en XAML de cette manière :

```
<TextBlock Margin="70,87,192,86" TextWrapping="Wrap">
  </TextBlock.Inlines>
  <Run FontFamily="Calibri" FontSize="16" FontWeight="Bold"
    Text="3DLightEngine Library preview"/>
  <LineBreak/><Run FontStyle="Italic" Foreground="Gray" Text="Fri,
    Sep 4th, 2009"/>
  <LineBreak/><Run Foreground="BlueViolet" Text="    Posted by Eric
    Ambrosi 01:09 PM"/>
  <LineBreak/><LineBreak/><Run FontWeight="Bold"
    TextDecorations="Underline" Text="us version"/>
  <LineBreak/><LineBreak/><Run FontFamily="Calibri" Text="Je mettrai
    bientôt à disposition une librairie ..."/>
</TextBlock>
```



```
</TextBlock.Inlines>
</TextBlock>
```

INFO

Comme vous le constatez, deux propriétés assez utiles ne sont pas fournies par la classe `Run`. La première, `LineHeight`, concerne la hauteur des lignes, la seconde nommée `TextAlignment`, gère l'alignement horizontal du texte. Ainsi le texte contenu dans un `TextBlock` aura en commun ces deux propriétés que le texte soit, ou non, décrit dans des balises `Run`. Il n'y a pas vraiment de solution élégante à cette problématique. L'idéal pour des mises en forme plus abouties est d'employer la classe `Glyphs`, mais elle est moins facile à utiliser au quotidien car d'assez bas niveau.

Obtenir une mise en forme équivalente en C# est relativement simple. Il suffit de créer des instances de type `Inline` puis de les ajouter à la collection `Inlines` qui est une propriété de `TextBlock` :

```
Run r1 = new Run();
r1.FontFamily = new FontFamily("Calibri");
r1.FontSize = 13;
r1.FontWeight = FontWeights.Bold;
r1.Text = "Ce texte est formaté via une balise Run héritant de la classe
          abstraite Inline.";

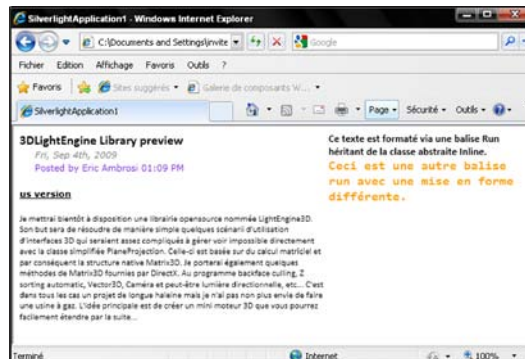
Run r2 = new Run();
r2.FontFamily = new FontFamily("Consolas");
r2.Foreground = new SolidColorBrush(Colors.Orange);
r2.FontSize = 16;
r2.FontWeight = FontWeights.Bold;
r2.Text = "Ceci est une autre balise run avec une mise en forme
          différente.";

//on ajoute la première balise Run
MiseEnFormeDynamique.Inlines.Add(r1);
//on va à la ligne en ajoutant une instance de LineBreak
MiseEnFormeDynamique.Inlines.Add(new LineBreak());
//on ajoute une seconde balise Run
MiseEnFormeDynamique.Inlines.Add(r2);
```

Vous trouverez dans le dossier *chap11* la solution contenant ces exemples : *MiseEnForme.zip*. Le résultat de cet exercice est visible à la Figure 11.24.

Figure 11.24

Résultat visuel de l'exercice de mise en forme.



11.3.2 Polices personnalisées

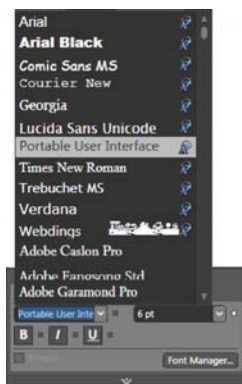
Les polices de caractères occupent une position centrale dans la charte graphique et définissent en partie l'identité visuelle d'une entreprise. Il est donc important de comprendre comment intégrer ces dernières de manière efficace.

11.3.2.1 Les polices par défaut

Le lecteur Silverlight contient nativement un certain nombre de polices de caractères. Pour les identifier, référez-vous à la liste des polices accessibles par défaut au sein de tout composant TextBlock. Lorsque vous les utilisez, il n'est pas nécessaire de les embarquer puisqu'elles le sont déjà. Les polices intégrées par défaut possèdent l'icône Silverlight à droite de leur nom (voir Figure 11.25).

Figure 11.25

Les polices embarquées par défaut affichant le logo Silverlight.



Parmi celles qui sont intégrées, on distingue Portable User Interface. Celle-ci n'en est pas vraiment une à proprement parler. Dans les faits, elle est constituée d'un agglomérat de plusieurs polices de caractères. Pour les cultures occidentales, elle affiche l'équivalent de la police Lucida Sans Unicode ou Lucida Grande. Elle permet en outre de supporter l'affichage de texte pour d'autres cultures à travers le monde. Ainsi sur un système d'exploitation chinois ou russe, elle apparaîtra de manière différente. Toutes les autres polices intégrées par défaut engendreront un rendu identique quelle que soit la culture du système d'exploitation client. Il est peu fréquent qu'une entreprise, une association ou une marque utilise l'une de ces polices car elles sont trop standard et passe-partout. Les créatifs choisissent souvent des polices moins connues permettant ainsi à l'utilisateur d'identifier plus facilement une marque ou un produit.

INFO

Produire souvent du contenu visuel pour différentes marques, produits ou entreprises, a pour conséquence d'installer de nombreuses polices sur le système d'exploitation du concepteur ou des créatifs. Certaines des polices installées ne seront utiles que 15 jours ou moins. Dans cette optique, le mieux est toujours d'utiliser des gestionnaires de polices permettant de les activer ou de les désactiver à volonté. De cette manière, vous éviterez de polluer votre poste et vous pourrez même organiser ces dernières de manière plus élégante que ne le fait le répertoire Fonts. Le logiciel SuitCase de la société Extensis est l'exemple type d'un gestionnaire de polices. Vous le trouverez à l'adresse : <http://www.extensis.com/fr/products/suitcasefusion2/>.

11.3.2.2 Embarquer les polices

Vous avez sans doute remarqué les multiples messages d'alerte dans le panneau Results. Ils indiquent que vous utilisez des polices qui ne sont pas embarquées par défaut dans le lecteur Silverlight (voir Figure 11.26).

Figure 11.26

Messages d'alerte indiquant que des polices utilisées ne sont pas embarquées.



Si vous codez directement en XAML dans Expression Blend, la propriété `FontFamily` est soulignée afin de vous alerter. Lorsque vous compilez, les polices ne sont pas affichées au sein de l'application dans le navigateur. Plusieurs choix s'offrent à vous pour résoudre cette problématique :

- Vous pouvez utiliser l'une des polices intégrées par défaut. D'un point de vue expérience utilisateur, ce n'est pas un très bon choix. Il offre toutefois l'avantage de ne pas augmenter le poids final du fichier compilé.
- Il est également possible de vectoriser un composant `TextBlock`. Cette solution est viable pour un nombre limité de champs texte utilisant une police spécifique. Cela n'est valable que si ces derniers n'ont pas besoin d'être mis à jour dynamiquement ou fréquemment dans Blend. De cette manière, vous en faites des éléments graphiques servant de décors. Cette approche est désavantageuse en termes de maintenance puisque vous ne pourrez plus changer leur contenu. Il faut toutefois la prendre en considération car cette méthode vous permet également de profiter des avantages propres aux tracés vectoriels en matière de design.
- La troisième façon consiste à embarquer manuellement la police. Cette opération peut être réalisée sous Visual Studio ou Blend, mais, le plus souvent, c'est le designer interactif qui s'en occupe puisqu'il joue le rôle d'intégrateur. De cette manière, la police sera compilée au sein du fichier xap et accessible par les composants qui y feront référence.
- Une autre manière de procéder consiste à télécharger dynamiquement une police de caractères présente sur le serveur web. Nous étudierons cette voie à la section 11.5.
- La dernière solution, présente depuis Silverlight 3, consiste à utiliser des polices système. Il existe deux principes différents dans ce cas. Le premier consiste à faire référence à la police dans le XAML ou en C#, en affectant directement la propriété `FontFamily`. Dans ce cas, le nombre de polices système accessibles est limité : Calibri, Consolas, Constantia, etc. (voir la documentation pour une liste complète). La deuxième possibilité est de charger dynamiquement n'importe quelle police du système d'exploitation. Toutefois, cette méthode n'est possible qu'en JavaScript et non *via* le code managé C# ou VB. De plus, il est nécessaire d'utiliser des mécanismes de mise en forme de texte de bas niveau reposant sur la classe `Glyphs`. Cette solution peut être avantageuse mais nécessite beaucoup plus d'efforts.

Dans tous les cas, si la police ciblée n'est pas sur le système client, elle sera remplacée par la police Portable User Interface dans le premier cas. Dans le second cas, le texte ne sera pas affiché. Ces solutions ne sont donc viables que dans des environnements maîtrisés, dans un intranet d'entreprise par exemple.

Pour notre part, nous allons embarquer les polices au sein du fichier xap *via* Expression Blend, puis nous aborderons les options offertes par ce dernier. Dans le projet actuel, nous devons intégrer deux polices. La première, Showcard Gothic, est utilisée pour le logotype du lecteur multimédia. La seconde est Tahoma, qui concerne 95 % du texte affiché dans l'application.

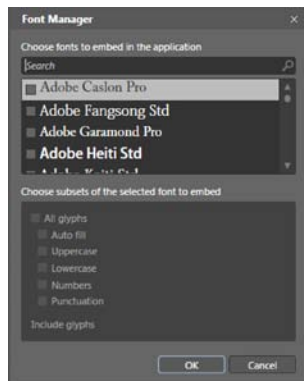
INFO

La police Tahoma fait partie de la liste des polices que Silverlight peut charger dynamiquement lorsqu'elles sont installées sur le poste client. Pour cette dernière, nous pourrions donc décider de nous reposer sur ce mécanisme et éviter de l'embarquer. Dans le cas où elle ne serait pas présente sur le système client, la police "Portable User Interface" la remplacerait. Ce choix de conception est à votre discrétion. Tout dépendra de vos objectifs et contraintes de production, mais également du type d'environnement et d'utilisateur que vous ciblez. L'impact peut être désastreux si la typographie initialement prévue est très différente de la police `PortableUserInterface` : taille, empâtement, interlettrage, les caractéristiques typographiques sont autant de paramètres qui diffèrent d'une police à l'autre. Le visuel global de votre application peut en être très fortement affecté.

Dans le menu `Tools`, choisissez le menu `Font Manager...` Une boîte de dialogue s'ouvre, c'est en fait un gestionnaire de polices propre à la solution Silverlight (voir Figure 11.27).

Figure 11.27

Le gestionnaire de polices.



À travers ce gestionnaire, vous avez la capacité d'embarquer une ou plusieurs polices ainsi que de lister les caractères à intégrer pour chacune d'entre elles. Nous allons aborder plusieurs points de vue différents, l'objectif final étant d'optimiser au maximum le poids du fichier compilé tout en gardant le visuel intact.

L'une des manières d'aborder cette problématique est illustrée par l'intégration de la police Showcard Gothic. Trouvez cette dernière en tapant les premières lettres de son nom dans le champ de recherche. Cliquez sur la case à cocher à côté de la police. Vous avez le choix entre embarquer la police dans sa totalité ou seulement une partie des caractères dont vous avez besoin. Comme les champs `Text` l'utilisant sont des éléments purement graphiques, il n'est pas nécessaire de tout embarquer. Décochez l'option `All glyphs`. Seule l'option de remplissage automatique `Auto fill` reste cochée. Cette propriété est très utile car elle permet de n'embarquer que les caractères dont la présence est détectée à la compilation. La différence entre les deux options est flagrante. Le poids du fichier xap est inférieur de 40 Ko dans le second cas, cela n'est pas négligeable sur Internet. Le remplissage automatique est une bonne option de ce point de vue. Vous ne pourrez toutefois pas

créer de textes affectés de cette police, à l'exécution, car vous prendriez le risque de ne pas pouvoir afficher certains caractères non embarqués à la compilation.

INFO

Lorsque vous avez embarqué la police, vous avez sans doute remarqué une très légère pause de fonctionnement. Celle-ci est due au fait que Blend a ajouté la police directement au sein du projet dans un répertoire Fonts spécialement créé à cet effet. Ce principe est très avantageux, il vous permet de partager le projet sans risque d'erreurs à la compilation. La personne à qui vous donnerez le projet n'aura pas besoin d'installer la police sur le système puisque celle-ci fait partie du projet.

Ouvrez à nouveau le gestionnaire de polices si ce n'est pas fait. Cherchez la police Tahoma, vous devez maintenant vous poser une question. Devez-vous tout embarquer ou seulement certains caractères ? Pour répondre à cette question, vous devez répondre à plusieurs autres :

- Quelle est la zone internationale ciblée par votre application ? Par exemple, celle-ci est-elle faite pour le continent nord américain, l'Europe, les pays nordistes ou l'ensemble de ces contrées ? Si vous connaissez la réponse, vous savez quels caractères doivent être embarqués en cas de saisie utilisateur dans l'application. Vous pourrez ainsi prévoir les caractères auquel l'internaute aura accès pour remplir un formulaire par exemple.
- S'il n'y a pas de saisie utilisateur prévue, dans quelles langue(s) ou culture(s) les champs textes dynamiques sont-ils susceptibles d'être mis à jour ?

Nous allons essayer de fournir la réponse la mieux adaptée. Laissez coché All glyphs, puis compilez le projet. Dans le répertoire Debug, vous pouvez constater que le poids du fichier est d'environ 780 Ko, nous avons environ 400 Ko de police embarquée en trop. Cela est normal car la police Tahoma est très complète et supporte de nombreux caractères différents, intégrer tous ses caractères est une erreur d'optimisation. Nous n'avons besoin que d'une fraction de ceux-ci. Partant du principe que notre application ne ciblera que les cultures occidentales, autant intégrer uniquement ceux que nous sommes susceptibles d'afficher. Voici une liste de caractères que je garde sous la main et que j'incrémente au besoin au fur et à mesure des projets que je réalise :

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZàâäçèéêëîïôûüÿÀÁÂÃ
 ÈÉÊËÌÍÎÏÜÜ'.,;:!"«»Ð&~#{ } () - _ ` \ / @ [] ° = + - * % μ | 0 1 2 3 4 5 6 7 8 9 © ® ¨ ¯ ± × ÷ æ ø ß ... £ \$ ¥

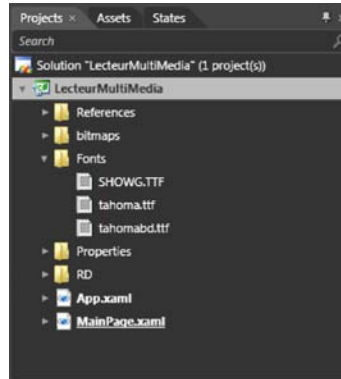
Attention à bien ajouter l'espace normal car c'est un caractère à part entière. Vous pouvez vous inspirer de cette liste ou créer la vôtre. Dans le gestionnaire de polices, décocher l'option All Glyphs, laissez l'option Auto fill sélectionnée, mais ajoutez la liste des caractères ci-dessus dans le champ en bas nommé Include glyphs. Cliquez sur OK. Une fois encore la police a été ajoutée au répertoire Fonts (voir Figure 11.28).

Recompilez l'application. Le fichier xap fait désormais 380 Ko, ce qui est beaucoup mieux. Le gain est d'environ 400 Ko ce qui est vraiment appréciable en terme de téléchargement. L'utilisateur passera moins de temps à attendre l'affichage de votre application. Vous remarquez qu'aucune alerte n'est désormais visible dans le panneau Results.

Vous pouvez télécharger le projet *LecteurMultiMedia_BF.zip* contenant les différents pinceaux ainsi que l'intégration des polices.

Figure 11.28

Les polices importées dans le projet facilitent son partage.



11.4 Styles et modèles de composants

Les styles et les modèles sont des ressources avancées et représentent le fer de lance du principe de réutilisation mis en exergue par Silverlight et WPF. Ils ont pour objectif d'améliorer le flux de production en facilitant la réutilisation du jeu de composants fourni sur les plateformes Silverlight et WPF. La création de composants visuels personnalisés *via* l'héritage, bien que possible, n'est plus autant nécessaire qu'avec la bibliothèque Winforms grâce aux notions de styles et de modèles de contrôles introduits avec .Net 3. Dans les prochaines sections, nous concevrons des composants personnalisés à travers les exemples du `Slider` et de la `ListBox`. Ce faisant, nous apprendrons les bonnes pratiques et approfondirons nos connaissances des méthodologies de production. Comme bon nombre d'environnements de développement, la grande majorité des interactions et des fonctionnalités est apportée par le jeu de composants proposé par défaut ou par des bibliothèques comme Silverlight Toolkit. Chaque composant possède un rôle propre et délimité, ce qui lui permet d'être souvent réutilisé au sein de plusieurs applications. Afin d'apporter un maximum de souplesse de conception, la fonctionnalité et l'interactivité supportées par un contrôle (`Slider` par exemple) ne sont pas couplés fortement au design ou au style visuel de ce dernier. Réutiliser des composants existants sera toujours plus productif que les créer par vous-même. Grâce aux styles et modèles, vous pourrez conserver les fonctionnalités des contrôles tout en modifiant complètement leur aspect visuel.

11.4.1 Les styles

Un style est un groupe de propriétés définies que l'on peut affecter à n'importe quel objet de type `FrameworkElement`. Ainsi, un objet de type `Shape`, `Panel` ou `Control` a la capacité de posséder un style. Toutes ces classes possèdent la propriété `Style` héritée de `FrameworkElement`, cette propriété recevra une instance de la classe `Style`. Voici l'écriture XAML d'un style ciblant la classe `Rectangle` :

```
<Style x:Key="RectangleStyle" TargetType="Rectangle">
  <Setter Property="Width" Value="150" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Fill" Value="Red" />
</Style>
```

Comme vous le remarquez, un style nécessite plusieurs attributs pour fonctionner. Le premier est le nom de la clé de ressource, le second définit le type de l'objet qui pourra recevoir le style. La balise `Style` doit elle-même contenir des balises enfants de type `Setter` qui permettront de cibler une propriété et d'y associer une valeur. Voici l'affectation en XAML du style défini plus haut :

```
<Grid x:Name="LayoutRoot" Background="White">
  <Rectangle Style="{StaticResource RectangleStyle}" />
</Grid>
```

Lorsque vous appliquez un style à un objet, les valeurs par défaut des propriétés de ce dernier sont automatiquement écrasées par celles contenues dans le style. Les attributs d'objet définis en brut dans la déclaration XAML ou C# de l'objet seront toutefois prioritaires à celles définies par le style. Vous pourriez par exemple décider qu'un style propre aux composants `Rectangle` affecte leur largeur (`Width`) de 200 pixels. Toutefois à la compilation, si un `Rectangle` affecté du style possède déjà sa propriété `Width` définie à 300, alors celle-ci outrepassera celle du style :

```
<Grid x:Name="LayoutRoot" Background="White">
  <!-- la propriété Width définie dans le rectangle écrase celle
        récupérée par le style -->
  <Rectangle Style="{StaticResource RectangleStyle}" Width="300"... />
  <!-- la propriété Width définie dans le style définit la largeur par
        défaut du Rectangle -->
  <Rectangle Style="{StaticResource RectangleStyle}" />
</Grid>
```

Créer un style en C# est une tâche simple, mais apporte en général moins d'intérêt. Créer un style dans Expression Blend et le stocker au sein d'un dictionnaire de ressources a plus de sens car ce dernier reste accessible au designer qui peut le modifier directement de manière sensible *via* l'interface de Blend. Voici toutefois un exemple de style généré et appliqué en C# :

```
//on crée un nouveau style
Style s = new Style( typeof(Ellipse) );

//on crée une instance de Setter
//Le premier paramètre indique la propriété ciblée de l'objet
//Le second paramètre définit la valeur de la propriété ciblée
Setter setter = new Setter(Ellipse.FillProperty, new SolidColorBrush
    (Colors.Blue));

//on ajoute un élément de type Setter à la collection Setters
s.Setters.Add(setter);

//il n'est pas obligatoire d'ajouter le style au dictionnaire
//de ressources via la ligne ci-dessous
//Resources.Add("EllipseStyle", s);

//on affecte le style généré à la propriété Style
monEllipse.Style = s;
//monEllipse.Style = Resources["EllipseStyle"] as Style;
```

Depuis Silverlight 3, il est possible de baser des styles à partir d'autres déjà existant. À cette fin vous pouvez utiliser la propriété `BasedOn`. Elle vous permet de cibler le style hérité. Voici un exemple simple illustrant cette méthode :

```
<UserControl.Resources>
  <Style x:Key="RectangleStyle" TargetType="Rectangle">
```

```

        <Setter Property="Width" Value="150" />
        <Setter Property="Height" Value="100" />
        <Setter Property="Fill" Value="Red" />
    </Style>
    <Style x:Key="RoundSquareStyle" BasedOn="{StaticResource RectangleStyle}"
          TargetType="Rectangle">
        <Setter Property="Width" Value="100" />
        <Setter Property="RadiusX" Value="10" />
        <Setter Property="RadiusY" Value="10" />
    </Style>
</UserControl.Resources>

```

Cela est réalisable uniquement côté XAML. Ce type d'écriture permet d'éviter de recopier un code fastidieux ou de dupliquer des styles entiers lorsque vous souhaitez les décliner. Il est possible d'ajouter ou d'écraser des balises Setter.

11.4.2 Affectation dynamique

Dans l'idéal, le style est créé dans l'interface de Blend, mais il peut être affecté dynamiquement, *via* C# ou VB, à plusieurs instances de FrameworkElement à la fois. De cette manière, vous profitez du meilleur des deux mondes, le graphisme et le ressenti utilisateur apportés par les créatifs, couplés à la fonctionnalité et l'automatisation des tâches fournies par le développeur. Nous allons le démontrer en créant un style *via* l'interface de Blend du point de vue d'un designer interactif, puis nous affecterons dynamiquement ce style à des objets de la liste d'affichage. Dans un projet de votre choix, créez un StackPanel centré horizontalement et aligné vers le haut. Instanciez dans ce conteneur cinq exemplaires de Button et affectez leur propriété Content des valeurs respectives : Accueil, Innovation, Technologie, Services, Portfolio. Sélectionnez le premier et dans le menu Object, sélectionnez Edit Style > Create Empty... Dans la fenêtre qui apparaît, nommez le style et stockez-le dans un dictionnaire de ressources externe. L'interface de Blend vous place en mode d'édition de style. Passez en mode de création mixte afin de voir le code XAML généré par vos actions sur le panneau Propriétés. L'idée est de personnaliser entièrement les propriétés du bouton, voici un exemple de style généré :

```

<Style x:Key="ButtonStyle1" TargetType="Button">
    <Setter Property="Background">
        <Setter.Value>
            <SolidColorBrush Color="#FF550024" />
        </Setter.Value>
    </Setter>
    <Setter Property="BorderThickness" Value="0,0,2,2" />
    <Setter Property="BorderBrush" Value="#FF743A3A" />
    <Setter Property="Foreground" Value="#FF8E4C4C" />
    <Setter Property="FontFamily" Value="Courier New" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Cursor" Value="Hand" />
</Style>

```

Du côté développeur, il suffit de lister tous les objets contenus dans le StackPanel, puis d'affecter le style dynamiquement à chacun d'eux mis à part au premier qui le possède déjà. Au sein de Blend, sélectionnez la grille LayoutRoot, puis dans le panneau des événements, dans le champ MouseLeftButtonUp, entrez ApplyStyle comme nom de méthode. Selon les paramètres de l'onglet Projects du menu Options... cette action ouvre le fichier MainPage.xaml.cs dans Blend ou

directement dans Visual Studio. Voici la méthode qui permet d'affecter le style créé par le designer dans Blend, à chaque bouton du StackPanel de manière dynamique :

```
private void ApplyStyle(object sender, MouseButtonEventArgs e)
{
    foreach (UIElement uie in MenuHaut.Children)
    {
        Button b = uie as Button;
        if (b != null && b.Content != "Accueil")
        {
            b.Style = App.Current.Resources["ButtonStyle1"] as Style;
        }
    }
}
```

Testez le projet. Lorsque vous relâchez le bouton gauche de la souris n'importe où sur la grille LayoutRoot, le style est dynamiquement affecté à chaque instance de bouton présent dans le StackPanel. Concrètement, chacun des acteurs de la production joue son rôle sans influencer ou interférer dans le travail des autres. Le projet *StyleAndTemplate_Base.zip* est disponible dans le dossier *chap11* des exemples du livre.

11.4.3 Organiser et nommer les ressources

Le code XAML ci-dessus est montré à titre indicatif, le designer n'a pas forcément besoin de savoir ce qui a été généré par la personnalisation des propriétés du bouton. Toutefois, comme nous l'avons précisé, pour que le développeur et le designer interactif puissent collaborer, il est au moins nécessaire de s'accorder sur une nomenclature des clés de ressource. Revenons un peu en arrière afin de mieux comprendre comment nommer et organiser nos dictionnaires de ressources, nous pourrions alors envisager une nomenclature des ressources. La Figure 11.4 décrit les capacités d'organisation des dictionnaires de ressources d'un point de vue technique. Ce point de vue est important mais n'aborde pas les bonnes pratiques de découpage des ressources. Plusieurs facteurs importants se révèlent décisifs lorsque vous souhaitez organiser vos ressources de manière optimisée :

- Vous pouvez tout d'abord gérer vos dictionnaires de ressources en les nommant selon les thèmes visuels qu'ils traitent. Par exemple, *SketchStyles.xaml* contenu dans les projets *SketchFlow*, est un dictionnaire de ressources dédié au style visuel croquis. Ainsi, nous pourrions créer un autre dictionnaire de ressources nommé *GlossyStyles.xaml* ou *Seventies-Styles.xaml*.
- Il est également possible de nommer les dictionnaires et de centraliser les ressources en fonction des fonctionnalités qu'ils couvrent. Par exemple, si votre application possède un lecteur multimédia, vous pouvez fédérer les ressources dans un dictionnaire dédié au lecteur.
- Si vous travaillez pour des clients différents et que vous souhaitez industrialiser le développement, il est peut-être utile d'avoir des dictionnaires dédiés à chacun d'eux (*Banque1Brush.xaml* et *Banque2Brush.xaml*). Ce modèle d'organisation est assez intéressant car il vous permet de simplifier en factorisant l'affectation de dictionnaires.
- Les contraintes techniques propres aux ressources influent grandement sur les possibilités d'organisation. Par exemple, un *Storyboard* est une ressource mais il cible une instance de *UIElement* au sein d'un arbre visuel. Il ne sera donc pas possible de l'externaliser dans un

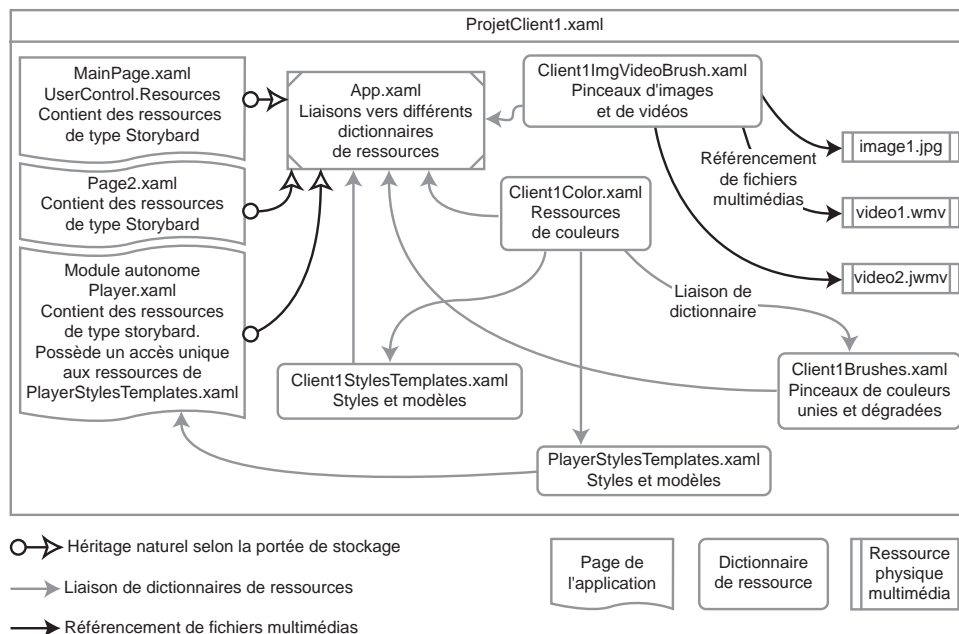
dictionnaire puisque la référence de l'instance animée n'existe pas dans le dictionnaire. Les pinceaux d'images (ImageBrush) suivent un peu la même logique dans le sens où les partager entre différents projets passera forcément par l'importation des images qu'ils ciblent. On peut donc se demander s'il est légitime de centraliser les pinceaux d'images (et de vidéos) dans des dictionnaires externes. La réponse est oui dans la mesure où organiser les ressources dans des dictionnaires facilite la maintenance, la lecture du XAML ainsi que l'évolutivité des applications. Toutefois, comme leur logique de partage est contraignante, il faudra éviter de les insérer dans des dictionnaires de ressources contenant des pinceaux dont le visuel est uniquement dépendant du code déclaratif XAML (et non d'une quelconque ressource de type média).

- Dans le même ordre d'idée, le type des ressources que vous organisez est à prendre en compte. Vous pouvez décider de scinder les différents types de ressources dans autant de dictionnaires externes. Les pinceaux dans un dictionnaire, les ressources de couleurs dans un autre, les styles et les modèles encore dans un autre. L'avantage de procéder ainsi est de conserver une organisation propre et facile à maintenir. Changer les couleurs propres à la charte graphique reviendra à modifier les ressources de couleurs contenues dans le dictionnaire adéquat. Comme le code XAML est relativement simple pour ce type de ressource, il est facile pour un graphiste ou un designer de changer les valeurs des couleurs sans altérer l'architecture ou le travail de conception déjà réalisé. Si vous mélangez les styles, les pinceaux et les couleurs, cela complexifiera leur accès et diminuera leur visibilité au sein du projet.

La Figure 11.29 met en avant une organisation structurée tenant compte de certaines de ces pistes de réflexion.

Figure 11.29

Organisation des dictionnaires de ressources par client, type de ressource et fonctionnalité.



Les concepts décrits ci-dessus influencent non seulement l'organisation des ressources, mais nous apportent également des indices concernant la conception d'une nomenclature. Par exemple, si vous deviez créer un style ayant pour thème visuel un effet vitré conçu pour les boutons de soumission, vous pourriez le nommer `SubmitGlassyButtonStyle`. De cette manière, vous saurez en lisant ce nom que cette ressource est un style avec un rendu "verre" pour les boutons de soumission de formulaire. Le développeur voit son travail facilité et n'a pas forcément besoin de s'entretenir avec le designer s'il doit l'appliquer dynamiquement.

11.4.4 Les modèles

Les modèles de composants sont souvent apparentés à la notion de style. Ils sont représentés par la classe `ControlTemplate` dont les instances peuvent être affectées à la propriété `Template` de tout objet héritant de `Control`. Ils permettent de remplacer l'arbre visuel et logique défini au sein des objets héritant de la classe `Control`. Par exemple, l'arbre visuel et logique d'une barre de défilement (`ScrollBar`) est entièrement modifiable si vous affectez à sa propriété `Template` une instance de `ControlTemplate`. À première vue, il peut vous sembler que les modèles sont une version plus puissante des styles que nous venons d'évoquer puisque la forme même des composants peut être réinventée. En réalité, les instances de `Style` et de `ControlTemplate` n'agissent pas aux mêmes niveaux. Leurs objectifs sont complémentaires, un modèle est toujours affecté à la propriété `Template` d'un `Control`. Par ailleurs, comme les styles sont des groupes de propriétés définies, un style peut contenir une balise `Setter` dont le rôle sera d'appliquer une instance de `ControlTemplate` à la propriété `Template` d'un `Control`. Si créer un modèle est relativement simple du côté XAML, il est peu pertinent de procéder de cette manière. Que vous soyez développeur ou non, et même si cela est réalisable avec du temps et du café, concevoir un modèle en codant directement du XAML est contre-productif lorsque vous pouvez l'éviter. Cela s'avère fastidieux et peu rentable, notamment si vous souhaitez coder à la main les tracés vectoriels (même si vous utilisez une extension comme `ReSharper`). D'autre part, le modèle influence fortement le visuel final d'un composant (encore plus que les propriétés définies au sein d'un style) et contient souvent des transitions animées internes au contrôle.

Ainsi, coder les modèles en XAML est souvent un non-sens, il est très difficile de générer du visuel de manière sensible tant le code XAML est abstrait. Les créatifs procèdent souvent par tâtonnement pour arriver au visuel final d'une interface, pour cela il faut nécessairement utiliser une interface telle qu'en proposent `Illustrator`, `Expression Design` et `Expression Blend`. Il va sans dire que créer un modèle du début à la fin en C# est encore plus aberrant lorsqu'on souhaite un design abouti.

Voici l'exemple d'un style et d'un modèle de bouton, tous deux créés à la main en XAML et affectés à une instance de bouton :

```
<Style x:Key="ButtonStyle2" TargetType="Button">
  <Setter Property="Background">
    <Setter.Value>
      <SolidColorBrush Color="#FF550024" />
    </Setter.Value>
  </Setter>
  <Setter Property="BorderThickness" Value="0,0,2,2" />
  <Setter Property="BorderBrush" Value="#FF743A3A" />
  <Setter Property="Foreground" Value="#FFFFFF" />
  <Setter Property="FontFamily" Value="Courier New" />
</Style>
```

```

        <Setter Property="FontSize" Value="18"/>
        <Setter Property="Cursor" Value="Hand"/>
    </Style>
    <ControlTemplate x:Key="RoundButton" TargetType="Button">
        <Grid >
            <Ellipse Fill="BlueViolet" Width="100" Height="Auto" />

            <ContentPresenter VerticalAlignment="Center"
                            HorizontalAlignment="Center" />
        </Grid>
    </ControlTemplate>
    ...
    <Button
        Content="Accueil"
        Margin="0,0,10,0"$
        Style="{StaticResource ButtonStyle2}"
        Template="{StaticResource RoundButton}" />

```

L'exemple ci-dessus est assez brut. Comme vous le constatez, les balises `Template` et `Style` n'entretiennent aucun lien particulier. Cette structure ne correspond pas au code généré sous Blend. Ce dernier possède une manière bien à lui d'articuler les styles et les modèles. Dans la grande majorité des cas, lorsque vous créez un modèle de composant dans Blend, celui-ci l'encapsule automatiquement au sein d'un style. C'est pour cette raison que, dans Expression Blend, la fenêtre de création de modèles a pour titre "Create Style Resource". Même si ce comportement peut déconcerter au premier abord, il est efficace et permet d'appliquer le modèle indirectement à travers l'affectation du style. Voici le même exemple revu et corrigé afin de mettre en valeur ce concept :

```

<Style x:Key="ButtonStyle2" TargetType="Button">
    <Setter Property="Background">
        <Setter.Value>
            <SolidColorBrush Color="#FF550024" />
        </Setter.Value>
    </Setter>
    <Setter Property="BorderThickness" Value="0,0,2,2" />
    <Setter Property="BorderBrush" Value="#FF743A3A" />
    <Setter Property="Foreground" Value="FFFFFFFF" />
    <Setter Property="FontFamily" Value="Courier New" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Cursor" Value="Hand" />
    <Setter Property="Template" >
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <Grid>
                    <Rectangle Fill="BlueViolet" RadiusX="8" RadiusY="8" />
                    <ContentPresenter VerticalAlignment="Center"
                                    HorizontalAlignment="Center" />
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>

```

Le XAML généré peut paraître verbeux, mais il est au final assez simple et rapide à assimiler. Il est possible d'améliorer ce concept en référençant une ressource modèle dans plusieurs styles différents :

```

<ControlTemplate x:Key="RoundButton" TargetType="Button">
    <Grid >
        <Ellipse Fill="BlueViolet" Width="100" Height="Auto" />

```

```

        <ContentPresenter VerticalAlignment="Center"
                        HorizontalAlignment="Center" />
    </Grid>
</ControlTemplate>

<Style x:Key="ButtonStyle1" TargetType="Button">
    <Setter Property="Background">
        <Setter.Value>
            <SolidColorBrush Color="#FF550024" />
        </Setter.Value>
    </Setter>
    <Setter Property="BorderThickness" Value="0,0,2,2" />
    <Setter Property="Foreground" Value="#FFFFFFFF" />
    <Setter Property="FontFamily" Value="Courier New" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Template" Value="{StaticResource RoundButton}" />
</Style>

<Style x:Key="ButtonStyle2" TargetType="Button">
    <Setter Property="Background">
        <Setter.Value>
            <SolidColorBrush Color="#FF240055" />
        </Setter.Value>
    </Setter>
    <Setter Property="BorderThickness" Value="0,0,4,4" />
    <Setter Property="Foreground" Value="#FFFFFFFF" />
    <Setter Property="FontFamily" Value="Tahoma" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="Template" Value="{StaticResource RoundButton}" />
</Style>

```

ATTENTION

De manière générale, il faut privilégier les styles aux modèles. L'application n'en sera que plus facile à maintenir et évolutive. Tout ce que vous définissez au sein d'un modèle est propre à l'ensemble des instances qui possèdent le modèle. La seule exception à cette règle est lorsque vous recourrez à la liaison de modèles (TemplateBinding). Au contraire, toutes les propriétés définies au sein d'un style restent par la suite modifiables et personnalisables.

Le projet *StyleAndTemplate_ModeleSimple.zip* est disponible dans le dossier *chap11* des exemples du livre.

Nous allons examiner deux cas concrets reposants sur le design d'un Slider et d'une ListBox. Ces derniers concentrent, à eux seuls, une grande majorité des problématiques communes à la personnalisation de tous types de contrôles.

11.5 Le modèle *Slider*

La personnalisation d'un Slider est très différente de celle d'un bouton (voir Chapitre 7). Un bouton ne possède par défaut aucun objet logique au sein de son arbre visuel. Autrement dit, rien dans le modèle d'un bouton n'est crucial en terme de fonctionnalités. Le seul objet logique, contenu par défaut dans tous les boutons, est le ContentPresenter. Il fournit au bouton la capacité d'afficher du texte ou tout autre objet comme unique enfant. Il est toutefois facultatif car le bouton diffuse l'événement Click et réagit aux interactions utilisateur (MouseEnter, MouseLeave, etc.) sans lever d'erreurs lorsque le ContentPresenter est absent. Ce n'est pas le cas du Slider : ce

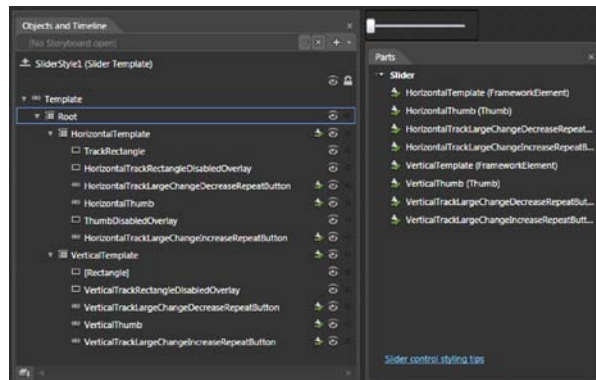
dernier contient plusieurs éléments logiques nécessaires à son fonctionnement. Dans cette section, vous apprendrez les méthodologies et les bonnes pratiques de la personnalisation de composant.

11.5.1 Principes et architecture

Lorsque vous créez un modèle, deux manières de procéder s'offrent à vous. Soit vous le concevez à partir d'un objet d'affichage sélectionné dans l'arbre visuel de l'application, soit vous modifiez le modèle d'un composant fourni par défaut au sein de la bibliothèque Silverlight. Au Chapitre 7, nous avons utilisé la première méthode. C'était la meilleure option car la classe `ButtonBase`, et celles qui en découlent comme `Button`, `ToggleButton` ou `RadioButton`, ne contiennent aucun objet logique. *A contrario*, comme l'architecture du composant `Slider` est plus complexe, il est préférable dans un premier temps de partir du modèle fourni en standard. Pour cela, utilisez le projet `StyleAndTemplate_ModelSimple.sln` ou créez un nouveau projet de test. Placez un exemplaire de `Slider` n'importe où dans la grille principale. Cliquez-droit dessus, sélectionnez `Edit Template > Edit a copy...` Dans la fenêtre de création de styles, vous pouvez cliquer sur OK. Dans un premier temps, notre objectif est de comprendre l'arbre visuel d'un `Slider`, il n'est donc pas nécessaire d'organiser le modèle dans un dictionnaire de ressources. Le modèle du `Slider` est affiché juste après cette étape (voir Figure 11.30).

Figure 11.30

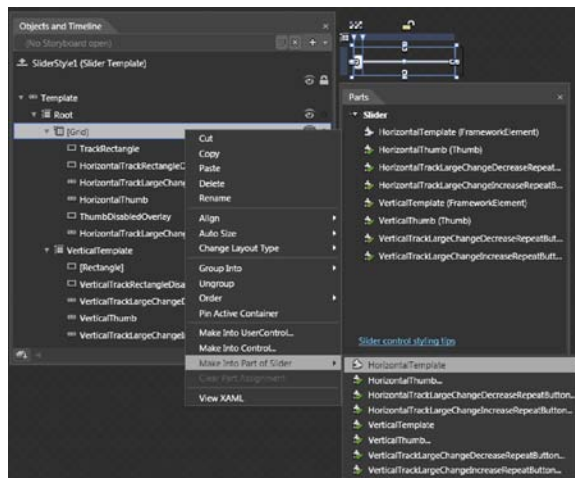
L'arbre visuel d'un `Slider`.



Vous remarquez qu'un `Slider` contient deux grilles correspondant respectivement à un mode d'affichage horizontal et vertical. Pour identifier facilement les éléments logiques d'un contrôle, Blend affiche, à droite de ceux-ci, une icône représentant un morceau de puzzle surmonté d'une coche verte. La liste des parties nécessaires au fonctionnement d'un contrôle est gérée par le panneau `Parts` situé en haut à gauche de l'interface. Lorsqu'un de ces éléments logiques n'est pas présent dans l'arbre visuel, ce panneau affiche le morceau de puzzle sans coche verte. De cette manière, il est très facile de savoir si toutes les parties logiques du composant sont correctement définies dans l'arbre visuel. De fait, l'identification automatique des éléments logiques repose sur le nommage correct de ces derniers. Pour vous en convaincre, il suffit de supprimer le nom de la grille `HorizontalTemplate`, le panneau `Parts` vous indique dès lors que celui-ci manque en affichant l'icône puzzle sans coche verte. Vous n'avez pas besoin de connaître par cœur le nom des parties logiques pour chaque composant. Si vous souhaitez redéfinir la grille en tant que partie logique du `Slider`, il vous suffit de cliquer-droit sur celle-ci, de sélectionner l'option `Make Into Part of Slider`, puis de cliquer sur `HorizontalTemplate` (voir Figure 11.31).

Figure 11.31

L'arbre visuel d'un Slider.



Dès lors, la grille est à nouveau nommée `HorizontalTemplate`. Nous allons maintenant essayer de comprendre comment le modèle `Slider` horizontal est structuré. Tout d'abord, lorsque la grille est sélectionnée, on remarque trois colonnes dont les deux premières sont en mode de redimensionnement automatique :

```
<Grid x:Name="HorizontalTemplate" Background="{TemplateBinding
                                     Background}">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
```

Les deux colonnes s'adaptent aux dimensions de leur contenu. La troisième colonne est en mode relatif et occupera par conséquent le reste de l'espace disponible. Dans les première et dernière colonne se trouvent deux boutons de répétition (de type `RepeatButton`) transparents mais cliquables. Ils représentent des zones interactives qui permettent de déplacer le curseur de manière indirecte. Lorsque l'utilisateur clique sur l'un d'eux, la largeur du premier `RepeatButton`, `HorizontalTrackLargeChangeRepeatButton`, augmente ou diminue. Cela a pour conséquence de déplacer le curseur dont la largeur est fixée en dur et de réduire ou d'augmenter la place restante allouée à la troisième colonne. Le curseur, dans la colonne du milieu est de type `Thumb`. Ce composant diffuse trois événements correspondant à ceux d'une interaction de type glisser-déposer : `DragStarted`, `DragDelta` et `DragCompleted`. L'objet événementiel, de type `DragDeltaEventArgs`, récupéré par l'écouteur de l'événement de `DragDelta`, contient deux propriétés `HorizontalChange` et `VerticalChange`.

Lorsque vous personnalisez un contrôle, vous n'avez pas accès au code logique, nous pouvons toutefois prédire que ces valeurs servent à modifier la largeur du premier `RepeatButton` dans les limites des dimensions du `Slider`. La modification de la largeur décale automatiquement les deux autres colonnes. Nous pouvons maintenant créer un premier `Slider` en partant d'une base simple sans trop de difficultés. Vous devrez souvent choisir entre créer des styles personnalisés à partir de ceux fournis, ou créer entièrement l'arbre visuel de ces composants. Cette dernière méthode, bien

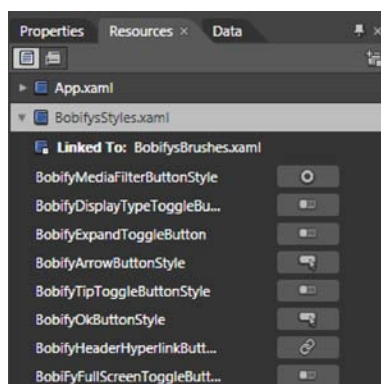
qu'un peu plus éprouvante au début, est une excellente manière d'apprendre l'agencement ainsi que l'imbrication de contrôles.

11.5.2 Un *Slider* personnalisé

Nous allons utiliser la seconde méthode évoquée plus haut en nous basant sur une arborescence simple située dans `MainPage.xaml`. Nous produirons donc un modèle de `Slider` à partir de celle-ci. Il est nécessaire de décompresser la solution *LecteurMultiMedia_BaseModele.zip* du dossier *chap11*. Vous trouverez dans l'arbre visuel plusieurs boutons (`Button`, `ToggleButton`, `HyperlinkButton`, `RadioButton`) dont les styles et modèles sont contenus dans le dictionnaire de ressources nommé `BobifyStyles.xaml`. Ouvrez le panneau Ressources afin de prendre connaissance des différents styles et modèles présents qu'il propose (voir Figure 11.32).

Figure 11.32

Le dictionnaire de ressources dédié aux styles et aux modèles.



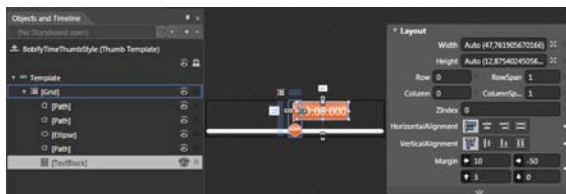
Ce dictionnaire contient une liaison ciblant un dictionnaire dédié aux pinceaux ; il sera sans doute nécessaire d'en ajouter une autre pointant vers le dictionnaire contenant les ressources de couleurs. Tout ce qui est fait dans ce projet est une synthèse des précédents chapitres, nous n'y reviendrons donc pas. Le `Slider` aura pour but de filtrer les vidéos et les sons en fonction d'une durée maximum, le composant `ListBox` affichera la liste des médias en remplacement de la grille fictive `PseudoListe` (cette dernière n'est qu'une maquette).

Sélectionnez la grille nommée `SliderTimeCode`. Elle nous servira de base pour créer un `Slider` personnalisé. Définissez-lui trois colonnes, dont les deux premières doivent être en mode de redimensionnement automatique et la dernière en mode relatif. Positionnez ensuite deux instances de `RepeatButton` dans la première et la dernière colonne avec une opacité fixée à 0. Le premier `RepeatButton` doit posséder une largeur exprimée en pixels égale à 0, le second doit être en mode de redimensionnement automatique pour s'adapter dynamiquement à la troisième colonne. Faites en sorte que le futur composant `Thumb`, qui correspond à la grille contenant le curseur et l'étiquette, soit dans la colonne du milieu. Transformez cette grille en contrôle de type `Thumb` via le menu `Tools`, puis `Make Control...`. Nommez le style généré `BobifyTimeThumbStyle`. Une fois dans le mode d'édition du modèle, définissez une largeur de 12,5 pixels pour la grille principale de ce composant. Sélectionnez ensuite les deux premiers tracés ainsi que le `TextBlock` et appliquez-leur une marge négative à droite de -50 pixels. Cette opération est importante car elle réduit l'espace utilisé par le `Thumb` dans notre futur `Slider`, tout en conservant l'affichage de l'étiquette.

Si nous ne procédions pas ainsi, le curseur ne pourrait pas se déplacer sur la totalité du Slider car il serait bloqué par la largeur de l'étiquette (voir Figure 11.33).

Figure 11.33

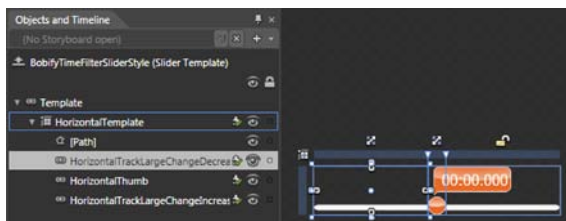
Réglage des marges pour les tracés du composant Thumb.



Une fois cette opération réalisée, revenez au niveau de l'application. Vérifiez bien que les colonnes en largeur automatique ne possèdent pas une valeur minimale en pixels. À cette fin, il suffit de cliquer à côté de l'icône de redimensionnement Auto, pour les sélectionner et afficher leurs propriétés. Oublier cette étape contraindrait les déplacements du curseur. Sélectionnez la grille contenant les éléments du Slider, faites-en un nouveau style *via* le menu Make Control... Nommez-le BobifyTimeFilterSliderStyle. Une fois dans le modèle, vérifiez que la largeur de la grille est en mode Auto. Cliquez-droit dessus et définissez-la en tant que partie logique HorizontalTemplate, celle-ci est dès lors renommée. Le premier enfant de la grille correspond à un tracé et ne fait pas partie des enfants logiques obligatoires. Il est toutefois nécessaire de vérifier que sa largeur est en mode d'étirement automatique (afin qu'elle s'adapte à la grille dynamiquement). Ce tracé doit être étalé sur les trois colonnes, veillez à ce que sa propriété ColumnSpan soit égale à 3. Sélectionnez ensuite chacun des composants restants dans l'arbre visuel et donnez-leur un rôle logique adéquat (voir Figure 11.34).

Figure 11.34

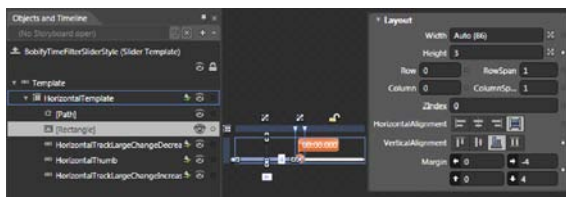
Association des objets de l'arbre visuel en tant que partie logique du contrôle Slider.



Si tout s'est correctement déroulé, le Slider fonctionne parfaitement. Revenez au niveau de l'application, définissez une valeur minimale de 0 et une maximale de 600. Cela représentera le temps minimal et maximal exprimé en secondes. Lorsque l'utilisateur déplacera le curseur, il filtrera les vidéos ou les sons dont le temps sera supérieur à la valeur choisie. L'étiquette a pour but d'afficher cette durée mise à jour lors de déplacements du curseur. Vous pouvez encore améliorer le rendu en ajoutant une barre de couleur (Rectangle) superposée au tracé dans la première colonne (voir Figure 11.35).

Figure 11.35

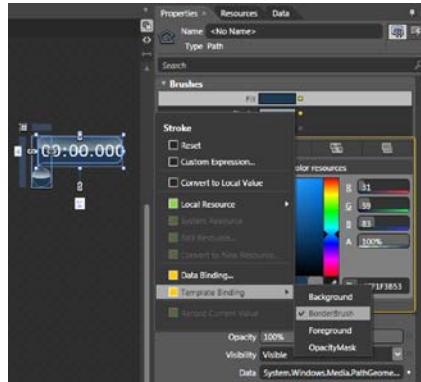
Arbre visuel finalisé du modèle Slider.



Il est également possible d'animer les différents éléments durant le survol du `Slider` ou du `Thumb`. À cette fin, le mieux est de créer des liaisons de modèles entre les propriétés de remplissage des tracés présents dans l'arbre visuel du `Thumb` et les propriétés `Background` et `BorderBrush` de celui-ci (voir Figure 11.36).

Figure 11.36

Liaison de modèles du Thumb.



De cette manière, il est possible d'animer les couleurs du `Thumb` *via* l'état `MouseOver` du `Slider`. Le `Slider` est maintenant presque terminé, il nous reste à trouver un moyen efficace de lier la valeur affichée par le `TextBlock` contenu dans le `Thumb`, et la valeur du `Slider`.

Le projet est dans l'archive *LecteurMultiMedia_SliderModele.zip* du dossier *chap11*.

11.6 Les liaisons

Au sein de Silverlight, deux familles de liaisons cohabitent : les liaisons de modèles et les liaisons de données. La première correspond à la notation `XAML TemplateBinding`. Elle ne fait pas référence à une classe proprement dite, mais à une expression spécifique, uniquement disponible au sein d'un modèle de composant. Elle n'est utile que dans le cadre de la personnalisation de composants et a été conçue en grande partie à l'intention des designers. La seconde, un sur-ensemble plus vaste, dotée des mêmes capacités que la première, offre toutefois beaucoup plus de capacités. Celle-ci est exprimée par la classe concrète `Binding`. Vous pouvez considérer que la notation `TemplateBinding` est une utilisation spécifique de la classe `Binding` facilitant et simplifiant l'écriture `XAML`.

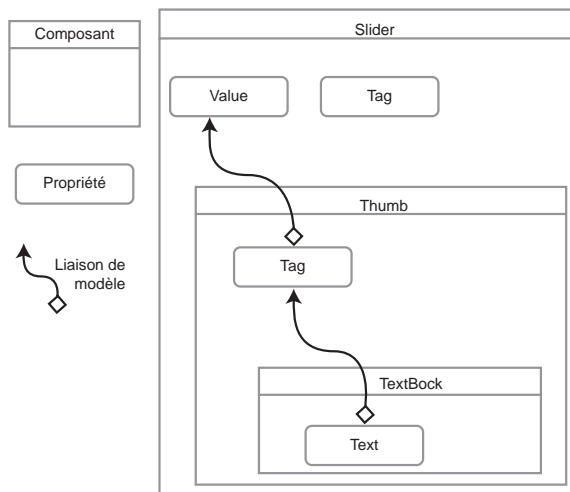
11.6.1 Créer une liaison

Reprenez le projet lecteur multimédia là où vous l'aviez laissé, puis accédez au modèle `Thumb` contenu dans le `Slider`. Nous allons créer une suite de liaisons de modèles partant de la propriété `Text` du `TextBlock` contenu dans le `Thumb` jusqu'à la propriété `Value` du `Slider`. Lorsque cette dernière sera modifiée, le composant `TextBlock` affichera la durée au format `mm:ss.ms`. Le problème consiste à savoir sur quelle propriété du `Thumb` nous pouvons créer la liaison. La propriété `Tag` est idéale dans ce cas de figure puisqu'étant typée `Object`, elle est capable de recevoir n'importe quel type de valeur. Il est ensuite possible de lier la propriété `Tag` du `Thumb` à la propriété `Value` du `Slider` (voir Figure 11.37).

Dans la Figure 11.37, il faut comprendre que la propriété `Text` du `TextBlock` contenu dans le `Thumb` est liée (`TemplateBinding`) à l'attribut `Tag` de celui-ci ; puisque la propriété `Tag` du `Thumb` est liée à l'attribut `Value` du `Slider`. Autrement dit, lorsque la propriété `Value` du `Slider` sera mise à jour par l'utilisateur, elle affectera la propriété `Tag` du `Thumb` qui impactera à son tour la valeur `Text` du `TextBlock` (contenu dans le modèle du `Thumb`). Créez les trois liaisons de modèle décrites ci-dessus.

Figure 11.37

Schéma de la liaison de modèles en cascade.



INFO

Vous constatez que le texte est affiché dans une autre couleur correspondant à la propriété `Foreground` du `Thumb`. Ce comportement est assez logique, le contrôle `Thumb` ne contient par défaut aucun `TextBlock` ou `ContentPresenter` dans son modèle, il possède cependant la propriété `Foreground` héritée de la classe `Control`. Celle-ci correspond à une couleur de texte et ne cible, initialement, aucun contrôle. Toutefois, dès qu'une liaison de modèles est mise en place pour la propriété `Text`, le champ `TextBlock` est détecté à l'intérieur du modèle et sa propriété `Foreground` est liée à l'attribut `Foreground` du `Thumb`. Vous n'avez pas besoin de définir cette liaison.

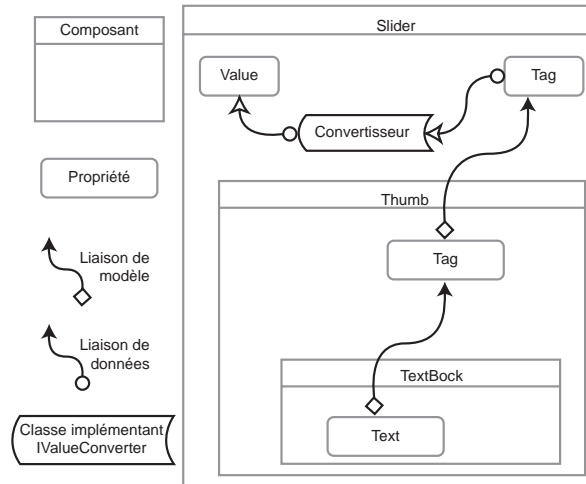
Compilez et déplacez le curseur du contrôle `Thumb`, il affiche une valeur de type `Double` appelant en interne sa méthode `ToString` afin que le champ `TextBlock` puisse l'afficher. Cela met en valeur la nécessité de convertir les secondes au format `mm:ss.ms`. À cette fin, nous devons créer une liaison de données ainsi qu'une classe de conversion. Cette classe doit implémenter l'interface `IValueConverter` et définir les méthodes qui y sont décrites. Ouvrez le projet dans Visual Studio – cet environnement est bien plus adapté au développement C# qu'Expression Blend. Pour réaliser cette opération, nous avons deux choix :

- Le premier consiste à utiliser notre classe de conversion à l'intérieur de la définition du modèle donc dans notre dictionnaire de ressources. C'est le choix le plus logique et correct en terme d'architecture. Cela devrait fonctionner sans problème. Toutefois, il arrive parfois que l'interface de Blend lève une erreur pour des raisons d'accès et d'initialisation. Ce type d'erreurs est appelé `Design Time Errors`. Celles-ci sont souvent dues à l'éclatement des ressources logiques et graphiques dans des dictionnaires de ressources ainsi qu'à une interprétation limitée du code logique par Blend. Nous aborderons ce principe à la fin de cette section.

- Le second choix possède l'avantage d'afficher correctement la donnée convertie dans Expression Blend sans que ce dernier ne lève d'erreurs. L'objectif est d'utiliser la liaison de modèles standard vers une propriété du contrôle, par exemple la propriété Tag dans le cas de notre Slider. Il est ensuite nécessaire de lier la propriété Tag à une autre du même contrôle, Value dans notre exemple, en définissant une liaison de données (Binding). De cette manière, nous définissons une conversion au niveau de l'application, ce qui limite grandement les risques d'erreur d'interprétation du code logique sous Blend (voir Figure 11.38).

Figure 11.38

Schéma de la liaison de modèles en cascade avec conversion des valeurs.

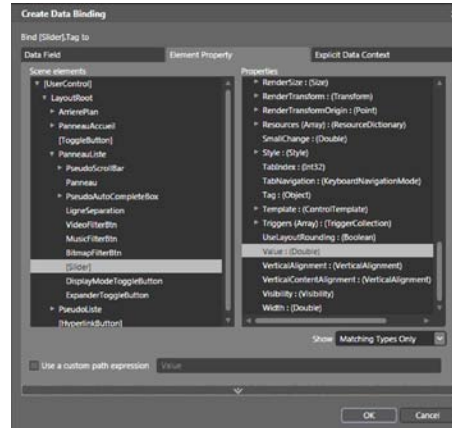


Nous allons aborder cette solution en premier. Comme vous le constatez à la Figure 11.37, il est possible de lier des propriétés du même objet ou d'objets différents situés dans le même arbre visuel. Générez une liaison de modèles pointant vers la propriété Tag du Slider en lieu et place de la propriété Value initialement ciblée. Cliquez ensuite sur l'icône carrée située à droite de la propriété Tag du Slider et sélectionnez le menu Data Binding... pour définir une liaison de données. Une boîte de dialogue apparaît, elle vous permet de créer une liaison d'UIElement à UIElement tant que les propriétés liées acceptent des types équivalents. Cliquez sur le second onglet nommé Element Property. La partie gauche de la fenêtre liste les objets présents dans l'arbre visuel de l'application. Conservez le Slider sélectionné. La partie droite montre les propriétés du Slider, choisissez Value (voir Figure 11.39).

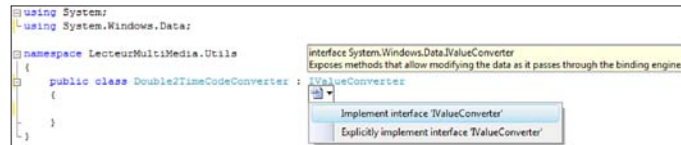
Lorsque vous compilez, vous obtenez le même comportement qu'auparavant tout en ayant ajouté une liaison. Nous allons maintenant coder notre classe de conversion. Celle-ci doit implémenter l'interface IValueConverter. Dans Visual Studio, créez un répertoire nommé Utils si celui-ci n'est pas présent. Cliquez-droit sur ce répertoire, puis ajoutez une classe nommée Double2TimeCodeConverter.cs. Supprimez tous les espaces de nom générés par défaut mis à part System, puis ajoutez System.Windows.Data. Générez ensuite le code implémentant les méthodes décrites par l'interface (voir Figure 11.40).

Figure 11.39

*Liaison de modèles
et de donnée avec
conversion des valeurs.*

**Figure 11.40**

*Implémentation des
méthodes de l'interface
IValueConverter.*



Une fois cette étape passée, vous obtenez le code ci-dessous :

```
using System;
using System.Windows.Data;

namespace LecteurMultiMedia.Utils
{
    public class Double2TimeCodeConverter : IValueConverter
    {
        #region IValueConverter Members

        public object Convert(object value, Type targetType, object
            parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }

        public object ConvertBack(object value, Type targetType, object
            parameter, System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }

        #endregion
    }
}
```

Les deux méthodes ont exactement le même objectif : convertir un type ou une valeur en un autre type ou valeur. La première, nommée `Convert`, correspond au sens défini par défaut lorsque vous créez une liaison de données. La seconde, `ConvertBack`, permet de gérer une liaison à double sens qu'il faut préciser en XAML. Si nous définissions un champ de saisie `TextBox` au lieu d'un champ texte classique de type `TextBlock`, l'utilisateur sera alors en mesure de saisir une valeur dans le `TextBox`. La propriété `Text` de ce dernier peut alors être convertie dynamiquement en `Double` et

affectée à la propriété Value si la méthode est correctement implémentée. Dans cet exemple, il n'est pas très pertinent d'avoir ce type d'interaction, nous n'allons donc pas nous attarder sur la méthode ConvertBack. Modifiez la méthode Convert comme ci-dessous :

```
public object Convert(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
{
    TimeSpan ts = new TimeSpan();
    if (value is double) ts = TimeSpan.FromSeconds((double)value);

    return ts.Minutes.ToString("00")
        + ":" + ts.Seconds.ToString("00")
        + "." + ts.Milliseconds.ToString("000");
}
```

Maintenant que nous avons créé le code adéquat, il faut référencer la classe de conversion en tant que ressource logique. Le mieux serait de la stocker dans un dictionnaire de ressources dédié. Créez-en un *via* Blend dans le répertoire RD. Le code ci-dessous décrit la manière dont vous pouvez déclarer des ressources logiques et référencer l'espace de noms Utils :

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:Utils="clr-namespace:LecteurMultiMedia.Utils"
>

    <Utils:Double2TimeCodeConverter x:Key="D2TCConverter" />

</ResourceDictionary>
```

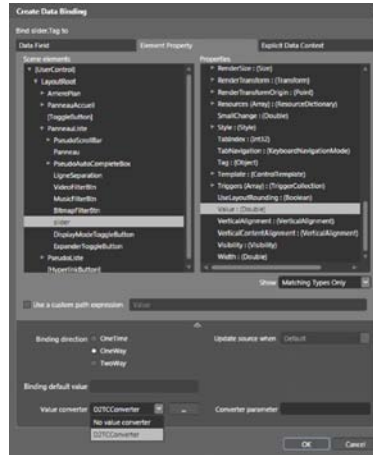
INFO

Comme ce dictionnaire est référencé par App.xaml, la classe de conversion est utilisable n'importe où dans l'application. Vous devrez toutefois prêter attention au fait qu'une seule instance de celle-ci sera utilisée systématiquement. Il faudra donc faire attention à ne pas employer de champs ou de propriétés de manière inconséquente. Dans l'exemple ci-dessus, la variable de type TimeSpan est locale à la méthode, de ce fait, celle-ci possède une référence unique à chaque appel de la méthode Convert. Il était important de ne pas utiliser de propriété ou de champ privé car nous pourrions avoir plusieurs instances de Slider différents utilisant la même opération de conversion.

Nous allons maintenant modifier la liaison de données afin d'utiliser le convertisseur. Cliquez à nouveau sur l'icône carrée située à droite de la propriété Tag du Slider, puis choisissez le menu Data Binding... Dépliez la zone des options située en bas de la fenêtre affichée, choisissez D2TC-Converter comme classe de conversion (voir Figure 11.41).

Figure 11.41

Sélection de la ressource
D2TCCConverter.



Le code XAML correspondant à la liaison de données a évolué :

```
<Slider ... Tag="{Binding Value,
                        Converter={StaticResource D2TCCConverter},
                        ElementName=slider, Mode=OneWay
                        }"
... />
```

Vous constatez que, mis à part le membre indiquant d'éventuels paramètres de conversion, le XAML reflète fidèlement les options de liaison décrites dans la fenêtre. Nous pourrions encore améliorer la conversion en passant un paramètre sous forme de chaîne de caractères par exemple :

```
<Slider ... Tag="{Binding Value,
                        Converter={StaticResource D2TCCConverter},
                        ElementName=slider,
                        ConverterParameter=00:00.000,
                        Mode=OneWay
                        }"
... />
```

Du côté C#, il nous faudrait modifier la classe de cette manière :

```
public object Convert(object value, Type targetType, object parameter,
                    System.Globalization.CultureInfo culture)
{
    TimeSpan ts = new TimeSpan();
    if (value is double)
        ts = TimeSpan.FromMilliseconds((double)value*1000);

    string p = parameter as string;
    if (p == "00:00.000")
    {
        return ts.Minutes.ToString("00")
            + ":" + ts.Seconds.ToString("00")
            + "." + ts.Milliseconds.ToString("000");
    }

    return ts.Minutes.ToString("00")
        + ":" + ts.Seconds.ToString("00");
}
```

Si aucun paramètre n'est précisé, alors on affiche les minutes et secondes, dans le cas contraire et si la chaîne de caractères fournie correspond à "00:00.000", alors on montre également les millisecondes.

Revenons maintenant sur la première solution consistant à utiliser la classe de conversion dans le modèle du Slider (voir Figure 11.42).

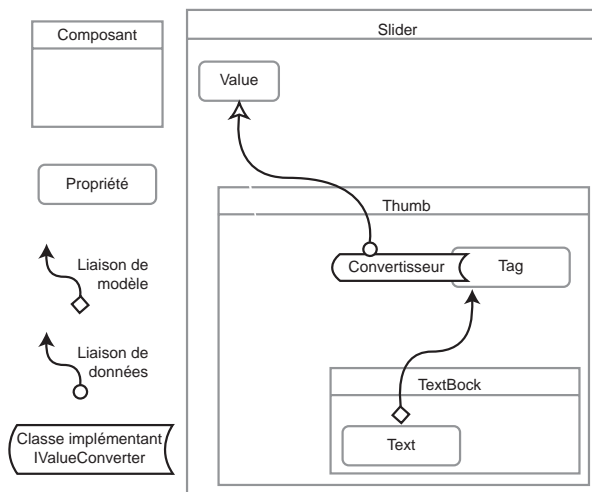
Comme nous l'avons dit au début de cette section, la liaison de modèles n'est qu'une liaison de données spécifique. Il est possible de créer une liaison de données en précisant une source relative pointant vers la classe du modèle personnalisé. De cette manière, nous reproduisons le comportement d'une liaison de modèles :

```
Tag="
{ Binding
  RelativeSource={RelativeSource TemplatedParent},
  Path=Value,
  Converter={StaticResource D2TCConverter },
  Mode=OneWay
}"
```

Cette liaison, bien que fonctionnelle à l'exécution, n'est toutefois pas correctement interprétée par Expression Blend. Le panneau Results affiche un message indiquant un code XAML invalide. Ce type d'erreurs est appelé *Design Time Errors*. Ces erreurs peuvent être handicapantes pour les designers. Dans certains cas, le composant est mal affiché dans Blend, dans d'autres la totalité de l'application ne sera pas affichée en mode création. Ceci est essentiellement dû au fait que le style est défini au sein d'un dictionnaire de ressources.

Figure 11.42

Schéma avec utilisation de la classe Double2TimeCodeConverter dans le style.



INFO

Créer une liaison de données en C# est assez simple, cela peut être intéressant lorsque vous développez votre propre framework. Admettons deux instances de `Rectangle`. Lorsque l'on met à jour la largeur du premier rectangle, cela modifie dynamiquement la largeur du second. Le code ci-dessous montre comment faire :

```
Random rnd = new Random();
public MainPage()
{
    InitializeComponent();

    //on crée une nouvelle instance de Binding
    Binding b = new Binding();

    //on définit le type de liaison
    b.Mode = BindingMode.OneWay;

    //on précise le nom du DependencyObject source
    b.ElementName = rectangleSource.Name;

    //on déclare le chemin d'accès à la propriété source
    b.Path = new PropertyPath("Width");

    //on affecte la liaison à la propriété Width du rectangle cible
    rectangleCible.SetBinding(Rectangle.WidthProperty, b);

    MouseLeftButtonDown +=
        new MouseButtonEventHandler(MainPage_MouseLeftButtonDown);
}

void MainPage_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    //lorsque l'on clique sur LayoutRoot
    //on met à jour la propriété source,
    //ce qui modifie la propriété cible
    rectangleSource.Width = (double)rnd.Next(100, 400);
}
```

Le projet *LecteurMultiMedia_SliderLiaisonModele.zip* est disponible dans le dossier *chap11* des exemples.

11.6.2 Les données fictives

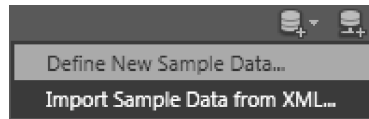
Nous allons maintenant simuler l’affichage de données externes au sein d’Expression Blend. Nous utiliserons à cette fin une instance de `ListBox` que nous personnalisons à la section 11.7. Lorsque vous modifiez le style d’un contrôle, il est nécessaire d’avoir le retour visuel des styles et des modèles en cours de modification. Bien évidemment, les contrôles standard, tels que la `ScrollBar`, sont assez faciles à tester puisque ce sont des éléments visuels dont les enfants ne sont pas générés dynamiquement à l’exécution. Ce n’est pas le cas des éléments d’une liste qui ne sont théoriquement visibles que lorsque les données sont chargées et affectées à la `ListBox`. Les ingénieurs de Microsoft ont pensé à cette éventualité et ont créé à cette fin la notion de données fictives. L’idée est simple : les designers ont la capacité de créer des collections de données typées aux valeurs aléatoires. Une fois créées, ils ont la possibilité de les affecter à une `ListBox` ou à n’importe quel autre contrôle dont le rôle est d’exposer des données.

Procéder ainsi permet aux designers de personnaliser les composants de données de manière autonome, sans solliciter le développeur. Lorsque les données réelles sont prêtes, il suffit de modifier une ligne de code pour les affecter en lieu et place des données fictives. Ouvrez le projet *Lecteur-*

"Multimedia" dans sa dernière version. Nous allons commencer par créer un jeu de données fictives *via* le panneau Data. Cliquez sur la première icône située en haut à gauche (🗄️) de ce panneau, puis sélectionnez le menu Define New Sample Data... (voir Figure 11.43).

Figure 11.43

Définir une collection de données fictives.



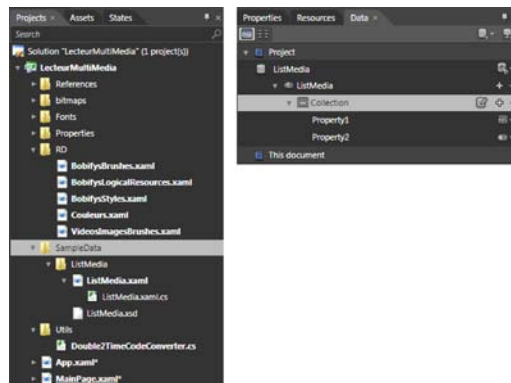
Dans la fenêtre de création qui s'affiche, choisissez de définir cette source pour l'ensemble du projet et nommez-la `ListMedia`. Vous pouvez décider d'afficher cette source de donnée à l'exécution ou uniquement dans l'interface de Blend. Laissez cette option inchangée et cliquez sur OK. Blend a automatiquement généré un répertoire dédié au stockage de ces données, celles-ci sont également affichées dans le panneau Data (voir Figure 11.44).

Afin que les données fictives soient accessibles au sein de l'ensemble du projet, Blend a modifié le fichier `App.xaml` en ajoutant une ressource de type `SampleDataSource` :

```
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="RD/BobifysLogicalResources.xaml" />
    <ResourceDictionary Source="RD/Couleurs.xaml" />
    <ResourceDictionary Source="RD/BobifysBrushes.xaml" />
    <ResourceDictionary Source="RD/VideosImagesBrushes.xaml" />
    <ResourceDictionary Source="RD/BobifysStyles.xaml" />
  </ResourceDictionary.MergedDictionaries>
  <SampleData:ListMedia x:Key="ListMedia" d:IsDataSource="True" />
</ResourceDictionary>
```

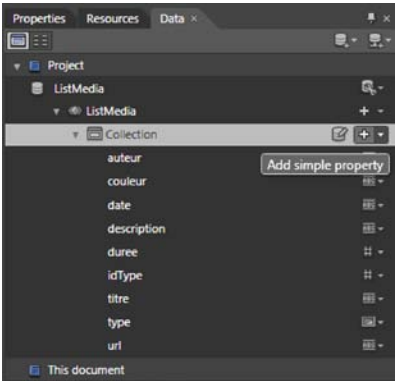
Figure 11.44

Le répertoire généré et l'affichage du modèle de données dans le panneau Data.



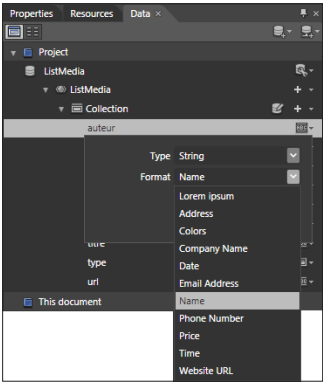
La propriété `IsDataSource` n'est là que pour préciser si le panneau Data doit afficher ou non cette source de donnée. Au sein du panneau Data, deux propriétés sont créées par défaut. Nous allons les personnaliser et en ajouter d'autres afin de simuler une réelle source de données. Cliquez sur l'icône plus, située à droite de "Collection" afin d'ajouter autant de propriétés qu'il est nécessaire (voir Figure 11.45).

Figure 11.45
Les propriétés définissant un modèle de collection.



À l'exception des propriétés *duree*, *idType* (Number), *type* (Image), la majorité sera des chaînes de caractères String. L'équipe des ingénieurs Expression a poussé la conception relativement loin puisqu'il est possible de configurer le format de donnée représenté par un champ. Ainsi le champ *auteur* contiendra des noms de personnes fictives (voir Figure 11.46).

Figure 11.46
Spécifier un format de données.



Vous pouvez vous référer au Tableau 11.3 pour configurer les champs que vous avez créés.

Tableau 11.3 : Liste des couleurs à sauvegarder comme ressource

<i>Nom des champ</i>	<i>Type de valeur</i>	<i>Format et spécificités de la valeur</i>
auteur	String	Name
couleur	String	Color
		bitmap : #FF08AFC8
		vidéo : #FFC70963
		son : #FFFFC800
date	String	Date

Tableau 11.3 : Liste des couleurs à sauvegarder comme ressource (suite)

Nom des champ	Type de valeur	Format et spécificités de la valeur
description	String	Lorem Ipsum
		Nombre de mots 40
		Longueur max 12
duree	Number	Nombre à 3 chiffres (inutile pour les médias)
idType	Number	Nombre entre 1 et 3
		bitmap : 1
		vidéo : 2
		son : 3
titre	String	Lorem Ipsum
		Nombre de mots max 4
		Longueur max 12
type	Image	Préciser le chemin d'accès vers le répertoire <i>bitmaps</i> du projet. Les images choisies aléatoirement par Blend déterminent le type de média.
url	String	Website
		Url


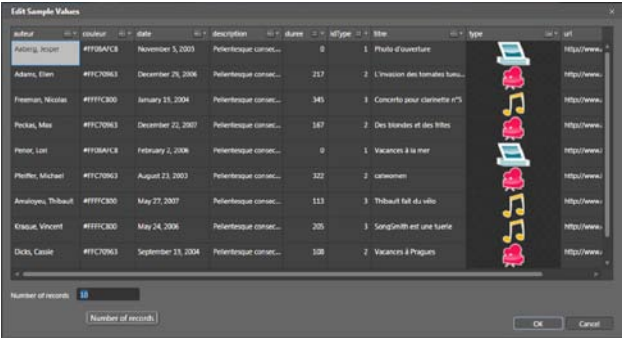
La configuration des données fictives n’est pas encore terminée, vous pouvez modifier à la main les valeurs que vous souhaitez en cliquant sur l’icône de modification de données () située à droite de Collection. Dans le tableau qui s’affiche, appuyez-vous sur les images bitmap du champ Type afin de définir les champs idType et Color dont les valeurs doivent être en accord avec le type de média. La durée exprimée en secondes est également inutile pour les médias de type image, autant passer sa valeur à 0 dans ce cas (voir Figure 11.47).

Figure 11.47
Modifier directement
les données fictives.



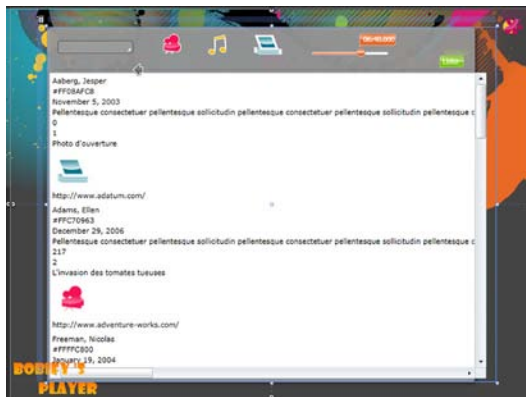
Vous pouvez également modifier les titres si vous souhaitez obtenir un contenu réaliste. Une fois la configuration terminée, cliquez sur OK. La collection est prête à être affectée à une instance de `ListBox`. Si vous êtes designer, développer en C# peut vous sembler compliqué ou ne faisant pas partie de votre périmètre de travail. Ce n'est pas un problème puisque Blend vous permet de réaliser cette opération sans une ligne de code logique. Vérifiez que les grilles nommées `PseudoListe` et `PseudoScrollBar` sont cachées en cliquant sur l'icône de l'œil située à droite des grilles dans l'arbre visuel. Ensuite, depuis le panneau Data, glissez la collection dans la grille nommée `PanneauListe`. Blend génère automatiquement un composant `ListBox` dont la propriété `ItemsSource` est liée à notre collection de données fictives. Définissez ses marges afin qu'il occupe l'espace laissé vacant par la grille cachée `PseudoListe` :

```
<Slider x:Name="slider" ... />
<ToggleButton x:Name="DisplayModeToggleButton" ... Cursor="Hand"/>
<ListBox Margin="2,69,14,4" ItemTemplate="{StaticResource ItemTemplate1}"
    ItemsSource="{Binding Collection}"/>
<ToggleButton x:Name="ExpanderToggleButton" ... />
```

Nous avons parcouru la moitié du chemin. Si tout s'est correctement déroulé, vous devriez obtenir un visuel correspondant à la Figure 11.48

Figure 11.48

Modifier les données fictives directement.



Le projet *LecteurMultiMedia_SampleData.zip* est dans le dossier *chap11* des exemples de ce livre.

11.6.3 Le contexte de données

Dans l'exercice précédent, nous avons défini, sans nous en rendre compte, un contexte de données. Cette notion est importante en terme d'architecture, nous allons donc l'étudier dans cette section. Créez un projet temporaire de test, puis déposez côte à côte sur `LayoutRoot` une instance de conteneur `Grid` et un exemplaire de `StackPanel`. La grille contiendra une liste ; le `StackPanel` affichera le détail de l'élément sélectionné dans la liste. Comme nous l'avons vu, lorsque vous déposez une collection en provenance du panneau Data, vous générez par défaut une instance de `ListBox` exposant la totalité des champs décrits par la collection pour chaque élément de la liste. Il est toutefois possible de ne glisser-déposer qu'un ou plusieurs champs au choix *via* l'utilisation des touches `Maj` ou `Ctrl`. Dans ce cas, vous instanciez une liste n'affichant que le ou les champs qui ont été déposés sur le conteneur. Créez une collection de données fictives, puis choisissez l'une

ou l'autre de ces méthodes pour déposer un exemplaire de `ListBox` dans la grille. Voici le code XAML généré par Blend :

```
<Grid x:Name="LayoutRoot" Background="White" DataContext="{Binding
    Source={StaticResource ListMedia}}" >
  <Grid HorizontalAlignment="Left" Margin="54,25,0,206" Width="205">
    <ListBox x:Name="listBox" Margin="0,0,5,0"
      ItemTemplate="{StaticResource ItemTemplate1}"
      ItemsSource="{Binding Collection}" SelectedIndex="-1"/>
  </Grid>
</StackPanel ... />
```

Comme nous l'avons dit plus haut, une collection de données fictives est stockée comme ressource lorsqu'elle est générée dans Expression Blend. Cette dernière n'est pas directement affectée à l'instance de `ListBox`. En réalité, Blend a affecté la ressource `ListMedia` à la propriété `DataContext` de la grille principale. La grille principale est alors considérée comme le contexte de données principal. À ce titre, le contenu de la ressource `ListMedia` est accessible à tous les enfants de la grille `LayoutRoot`. Si nous abordons maintenant le composant `ListBox` instancié, nous remarquons que la propriété `ItemsSource` est liée au premier enfant de la ressource nommé `Collection`. Vous pourriez toutefois modifier le nom de la collection. Cela signifie que vous pouvez en créer plusieurs au sein d'une même ressource de données fictives. Cette organisation est assez pratique dans certains scénari d'applications multiclients, par exemple. Lors de l'affectation de `ItemsSource`, il n'est nul besoin de préciser une liaison complexe car `Collection` est directement accessible par le contexte de données en cours.


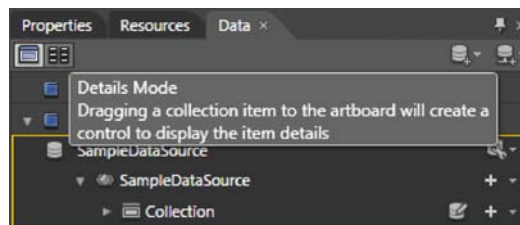
Si vous êtes perdus à un moment ou à un autre, il vous suffit de consulter le panneau `Data` pour assimiler le code XAML. L'arborescence décrite dans le panneau montre explicitement que `Collection` est un enfant de la ressource `listMedia`. Pour finir, la propriété `ItemTemplate` accepte une valeur de type `DataTemplate` automatiquement générée par Expression Blend et formalisant les données de manière visuelle. Nous verrons ultérieurement comment modifier ou manipuler une instance de `DataTemplate`. Si vous appuyez sur `Alt` en même temps que vous glissez une collection, ce n'est pas une `ListBox` qui est générée. Dans ce cas, vous créez une liaison de données sur le conteneur ciblé. De sorte que lorsque vous glisserez des champs de la collection dans un conteneur en maintenant la touche `Alt` enfoncée, les composants générés par cette action posséderont une liaison relative au contexte de donnée de ce conteneur. Cela est particulièrement utile lorsque vous souhaitez afficher le détail d'un élément sélectionné depuis un composant `ListBox`. Vous pouvez également utiliser l'icône de détail () située en haut à gauche du panneau `Data` pour arriver à ce résultat (voir Figure 11.49).

Figure 11.49

Les modes liste et détail.

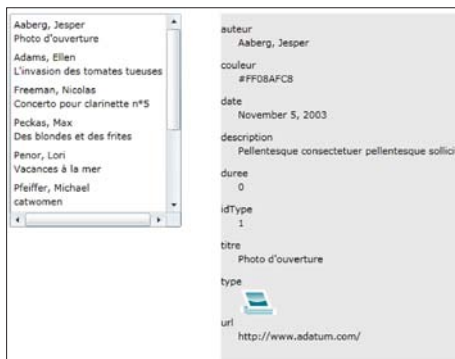


Maintenez la touche `Alt` appuyée et déposez la collection du panneau `Resources` sur le `StackPanel`, celui-ci est dorénavant affecté en tant que contexte de donnée. Sélectionnez tous les champs de la collection *via* la touche `Maj` ou `Ctrl`, puis glissez-les sur le `StackPanel` en maintenant la touche

Alt enfoncée. Les intitulés sous forme de `TextBlock`, ainsi que les contrôles exposants la valeur de chaque champ, sont instanciés à l'intérieur du conteneur. Vous pouvez ajuster la mise en forme de ces derniers au sein du `StackPanel` (voir Figure 11.50).

Figure 11.50

La liste et le détail des champs mis en forme.



Expression Blend a également anticipé le fonctionnement de l'application et a généré le code XAML dans ce sens. À l'exécution, le contexte de données du `StackPanel` est dépendant de l'élément sélectionné dans la liste. Le code XAML qui suit décrit ce fonctionnement :

```
<StackPanel ... DataContext="{Binding SelectedItem, ElementName=listBox}"
    d:DataContext="{Binding Collection[0]}" ...>
    <TextBlock ... Text="auteur" />
    <TextBlock Text="{Binding auteur}" ... />
    <TextBlock ... Text="couleur" />
    <TextBlock Text="{Binding couleur}" ... />
    <TextBlock ... Text="date" />
    <TextBlock Text="{Binding date}" ... />
    <TextBlock ... Text="description" />
    <TextBlock Text="{Binding description}" ... />
    <TextBlock ... Text="duree" />
    <TextBlock Text="{Binding duree}" ... />
    <TextBlock ... Text="idType" />
    <TextBlock Text="{Binding idType}" ... />
    <TextBlock ... Text="titre" />
    <TextBlock Text="{Binding titre}" ... />
    <TextBlock ... Text="type" />
    <Image Source="{Binding type}" ... />
    <TextBlock ... Text="url" />
    <TextBlock Text="{Binding url}" ... />
</StackPanel>
```

Dans ce cas, la propriété `DataContext` ne pointe pas vers une ressource, mais directement vers l'élément en cours de sélection dans la liste. Lorsqu'une propriété est préfixée de `d` : cela signifie qu'elle possède une affectation qui ne sera valable que durant la phase de conception sous Blend et non lors de l'exécution de l'application. Ainsi, dans le code XAML ci-dessous, il faut comprendre que lorsque vous êtes en cours de conception, le contexte de donnée du `StackPanel` est le premier élément de la collection :

```
d:DataContext="{Binding Collection[0]}"
```

Cela vous permet de tester en temps réel l'affichage des valeurs de la collection dans la `ListBox`. Au contraire, durant l'exécution de l'application, le contexte de donnée évoluera en fonction de l'élément sélectionné dans la liste :

```
DataContext="{Binding SelectedItem, ElementName=listBox}"
```

INFO

A priori, il serait possible d'obtenir le même comportement dans l'un ou l'autre des cas en modifiant l'aperçu dans Blend et en spécifiant exactement la même valeur que le contexte de données à l'exécution :

```
d:DataContext="{Binding SelectedItem, ElementName=listBox}"
```

Toutefois, Blend supporte très mal l'accès temps réel à l'élément sélectionné d'une liste, la compilation échoue si vous modifiez cette ligne. Laissez donc le code XAML tel qu'il est généré par Blend afin d'éviter les complications inutiles.

Compilez et testez l'application. Vous venez de créer de la logique applicative sans une seule ligne de code C#, cela en utilisant les mécanismes de liaison de données Silverlight ainsi que les outils fournis par Expression Blend.

11.6.4 Paramètres optionnels

Pour l'instant, l'écriture des liaisons (déclarées par le mot-clé `Binding`) est relativement simple, il existe toutefois un certain nombre de paramètres optionnels dont voici une liste non exhaustive :

- `Source` définit une source de données de type `StaticResource` dans la majorité des cas. Cette propriété est utilisée pour cibler des objets contenus au sein d'un dictionnaire de ressources.
- `RelativeSource` permet de définir un objet source relatif au contexte actuel. Cette propriété accepte deux valeurs, `Self` et `TemplatedParent`. `Self` cible l'objet définissant la liaison en cours comme source de données. `TemplatedParent` permet de cibler une propriété du modèle en cours de personnalisation et permet ainsi de reproduire le principe de liaison de modèle.
- `ElementName` permet de créer des liaisons entre différentes instances d'objets au sein d'un même arbre visuel. La valeur attendue est de type `x:Name`. L'objet ciblé est considéré comme source.
- `Path` définit le chemin d'accès vers la propriété d'un objet source.
- `Converter` définit une classe de conversion personnalisée.
- `ConverterParameter` est un attribut optionnel permettant d'envoyer un paramètre aux méthodes de conversion `Convert` et `ConvertBack`.
- `Mode` permet de gérer le type de liaison établie, elle prend trois types de valeur, `OneTime`, `OneWay` et `TwoWay`. La valeur `OneTime` permet d'établir une liaison de données prise en compte une seule fois au chargement du composant. `OneWay` et `TwoWay` synchroniseront les données durant l'exécution de manière unidirectionnelle ou bidirectionnelle. Pour finir, `OneWay` est le mode utilisé par défaut lorsqu'aucune valeur n'est définie pour `Mode`.

- `ValidatesOnExceptions` accepte une valeur booléenne, très utile dans le cadre de liaisons bidirectionnelles. Nous aborderons son utilisation au Chapitre 12.

Vous remarquez que `Source`, `RelativeSource` et `ElementName` ont exactement le même rôle consistant à définir un objet source. Il n'est donc pas possible de déclarer plusieurs de ces attributs en même temps. Lorsque vous créez un jeu de données fictives, les liaisons sont en majorité à sens unique, il est toutefois possible de générer des liaisons à deux sens.

INFO

Par défaut, les liaisons bidirectionnelles ne sont générées que lorsque la collection de données fictives contient des valeurs booléennes. Pour ce type de champ, le composant généré est une case à cocher cliquable. Il faut donc modifier la donnée lorsque le composant `CheckBox` est coché ou décoché par l'utilisateur. La mise à jour des données est dans ce cas bidirectionnelle.

Pour tester le mode deux voies, il suffit de remplacer l'un des contrôles `TextBlock` par `TextBox` et préciser l'utilisation de ce mode en XAML :

```
<TextBlock ... Text="titre" />
<TextBox Text="{Binding Path=titre, Mode=TwoWay}" ... />
```

Compilez et modifiez à la main la valeur du champ texte de saisie correspondant au titre. Ensuite, cliquez sur un autre composant pour qu'il perde le focus, la liste affiche alors la valeur mise à jour. Attention toutefois, mettre à jour la liste revient à modifier la collection.

INFO

De la même manière que pour le `Slider` personnalisé précédemment, il est possible d'utiliser une classe de conversion. Cela est même souvent conseillé puisque dans de nombreux cas, les données présentées à l'utilisateur dans les vues ne sont pas stockées ou formatées de la même manière en base de données et dans l'interface utilisateur. Par exemple, en base, une date sera souvent stockée sous forme d'un nombre de seconde ou de millisecondes, qui représente le temps écoulé depuis le 1er janvier 1970 à minuit. Ainsi, le temps exprimé en seconde dans la base sera convertit en date intelligible afin d'être affichée dans l'interface. À l'opposé, lorsque l'utilisateur saisira une date intelligible dans l'application, celle-ci sera convertie sous forme d'un nombre de secondes écoulées, lors de sa mise à jour dans la base SQL.

11.7 Le modèle *ListBox*

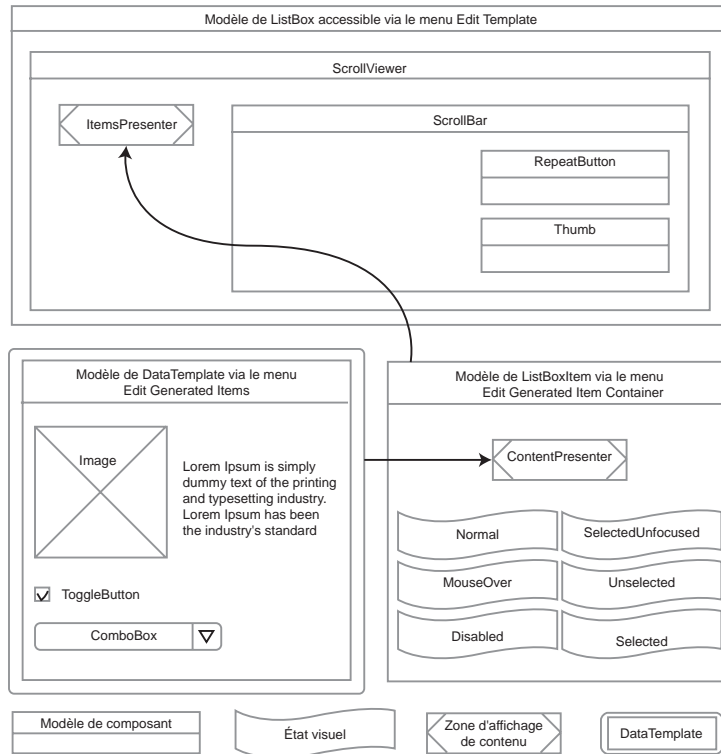
Nous allons maintenant personnaliser la `ListBox` que nous avons créé précédemment afin d'obtenir une liste de lecture dont le visuel correspond à la grille nommée `PseudoListe`. Cette grille simule l'affichage d'une liste de médias que nous diffuserons au sein du lecteur.

11.7.1 Structure et principes

Le contrôle `ListBox` figure parmi les plus complexes à personnaliser pour deux raisons. En premier lieu, il est constitué de nombreux composants de natures différentes qui sont situés à plusieurs niveaux d'imbrication. Il est nécessaire de les localiser et de les personnaliser les uns après les autres pour arriver au résultat final (voir Figure 11.51).

Figure 11.51

Structure et composition d'un contrôle ListBox.



En second lieu, il est parfois obligatoire d'effectuer quelques manipulations délicates lorsque vous souhaitez arriver à vos fins. Au final, l'architecture d'une *ListBox*, sans être réellement compliquée, est éclatée et permet une personnalisation très poussée de ce composant.

Cette architecture est articulée autour de trois familles de composants décrites sur le schéma de la Figure 11.42. La première catégorie concerne les contrôles courants de type *ScrollView*, *ScrollBar*, *RepeatButton* ou *Thumb*. Nous avons abordé leur personnalisation dans les grandes lignes et de ce point de vue, la seule difficulté consiste à les localiser dans l'arbre visuel. La deuxième famille est du même genre, sauf que c'est la première fois dans ce livre que nous y sommes confrontés de cette manière. Dans le cas d'une *ListBox*, elle est représentée par la classe *ListBoxItem*, pour les instances de listes déroulantes (de type *ComboBox*), il s'agira de la classe *ComboBoxItem*. Ces deux classes héritent en droite ligne de la classe abstraite *ContentControl*. Cela signifie qu'elles ont pour but premier d'afficher un ou plusieurs objets, tout en fournissant des bases logiques et graphiques minimum. C'est ce que vous constatez à la Figure 11.42 : un *ListBoxItem* contient un jeu d'états visuels permettant de personnaliser en partie l'affichage graphique et l'interactivité de chaque élément d'une liste. À l'instar d'un simple bouton, elles possèdent par défaut un enfant de type *ContentPresenter* qui définit la zone d'affichage des données. Dans la grande majorité des situations et au même titre que les boutons, un *ListBoxItem* affichera une simple chaîne de caractères. Cela est transparent car chaque élément d'une collection appellera par défaut sa méthode *ToString* lorsque rien n'est précisé. Ces composants ont également

la capacité de contenir un modèle de données personnalisé (*DataTemplate*) pour les cas plus élaborés.

Cette capacité est héritée de *ContentControl* via la propriété *ContentTemplate*, qui accepte les instances de *DataTemplate*. C'est en réalité une affectation en cascade générée via une liaison de modèles établie entre la propriété *ContentTemplate* du *ListBoxItem* et la propriété *ContentTemplate* de l'objet *ContentPresenter* contenu dans le modèle de ce dernier. Le *DataTemplate* est le troisième type d'objet utilisé pour personnaliser l'affichage d'une *ListBox*. Celui-ci n'est pas un contrôle à proprement dit car il n'hérite pas la classe *Control*. Son but consiste à représenter de manière visuelle des données simples ou complexes, et centralise à cette fin tous les contrôles nécessaires. Si vous avez besoin d'afficher un bitmap pour chaque élément d'une *ListBox*, c'est dans le *DataTemplate* que le composant *Image* sera placé par défaut pour afficher cette image. Il est en fait possible d'y imbriquer n'importe quel type de composant visuel. Les *ListBox* sont souvent utilisées pour présenter des objets métier. Chaque élément peut donc au besoin afficher des valeurs booléennes, des chaînes de caractères, des nombres, des sous-listes, des vidéos et de nombreux autres types d'objets adaptés à l'affichage de contenu. Le composant *DataTemplate* a pour rôle de centraliser tous ces objets.

INFO

Ce principe est à la fois pertinent car généré par défaut dans Blend, mais peut également apparaître limité dans certains cas où le graphisme est élaboré. Au final, l'idée consistant à séparer au maximum la pure donnée et sa représentation visuelle est toujours privilégiée. Il faut considérer ce principe comme un idéal à atteindre sans pour autant en faire un dogme. Le graphisme, l'expérience utilisateur et l'ergonomie sont également là, avec leur cahier des charges et leurs propres contraintes qui sont aujourd'hui des notions déterminantes en termes d'adoption. Ces contraintes remettent parfois en cause des architectures bien conçues mais laissant peu de place à l'innovation. Bien heureusement, Silverlight a pour particularité d'offrir suffisamment de souplesse de conception pour contenter tous les appétits.

Nous allons maintenant aborder ces problématiques à travers une manipulation concrète de la *ListBox*.

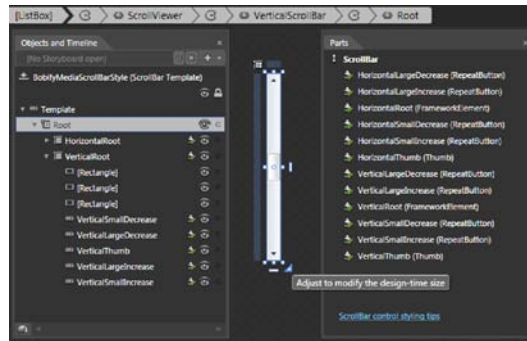
11.7.2 Modifier l'apparence d'une liste d'éléments

Ouvrez le projet *LecteurMultimedia* de l'archive *LecteurMultiMedia_SampleData.zip* du dossier *chap11* des exemples. Sélectionnez la *ListBox* affichant les données fictives puis supprimez les remplissages de son arrière-plan et de sa bordure. Ensuite, créez un nouveau modèle de *ListBox* à partir de celui existant. Nommez le nouveau style *BobifyMediaListBoxStyle*, puis stockez-le dans le dictionnaire de ressources dédié aux styles *BobifyStyles.xaml*. Dans l'arbre visuel du modèle, vous remarquez que seule l'enveloppe de la liste sera concernée par le modèle que nous sommes en train de personnaliser. Ainsi, les éléments actuellement contenus par celle-ci ne peuvent pas être personnalisés à ce niveau bien qu'ils soient tout de même accessibles par ailleurs. Ce n'est pas très grave car dans un premier temps, vous allez vous concentrer sur les éléments interactifs gérant le défilement de la liste. Le contrôle *ScrollViewer* contient une instance de *ScrollBar* vertical qu'il va falloir modifier pour arriver à vos fins. Le fonctionnement d'une barre de défilement (*ScrollBar*) est très semblable à celui d'un *Slider* mis à part que son modèle contient deux instances de *RepeatButton* correspondant aux flèches du bas et du haut. Créez

autant de styles personnalisés que nécessaire afin d'accéder au modèle de la barre de défilement. Étirez les dimensions fictives propres au mode design, de la grille nommée Root, afin de pouvoir personnaliser le composant ScrollBar plus confortablement (voir Figure 11.52).

Figure 11.52

Visuel et logique d'un contrôle ScrollBar.



Identifiez tous les composants logiques qu'il est nécessaire de personnaliser afin de reproduire le visuel de la grille nommée `PseudoScrollBar`. Elle est située sur la page principale dans le conteneur `PanneauListe`. Vous pouvez également faire des copier-coller de tracés si nécessaire pour les `RepeatButton` ainsi que pour le `Thumb`. Attention au fait que l'arrière-plan de `PseudoScrollBar` est un tracé (`Path`) qu'il vaudra mieux remplacer par un composant de type `Border` dans le composant final. Utiliser ce dernier sera beaucoup plus pertinent car il est plus adapté au redimensionnement qu'un tracé. L'ombre portée à gauche dans le visuel original, n'est pas simple à réaliser, mais il est tout de même possible, soit de la remplacer, soit de la simuler en créant une épaisseur de la bordure à gauche du `Border`.

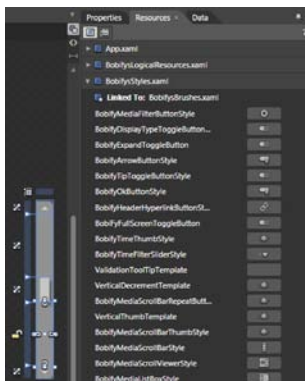
Même si ce travail a des aspects fastidieux, cela n'est pas plus difficile à faire que ce que nous avons déjà réalisé avec le `Slider`. Dans tous les cas, vous n'avez pas besoin de concevoir plus de deux modèles personnalisés car les objets `VerticalSmallDecrease` et `VerticalSmallIncrease` peuvent accepter le même style. Il vous suffit d'appliquer une symétrie verticale sur l'un des deux boutons pour obtenir des directions opposées. Au final, il est nécessaire de créer cinq styles différents qui sont dans l'ordre :

- `BobifyMediaListBoxStyle`, du contrôle `ListBox` lui-même ;
- `BobifyMediaScrollViewerStyle` associé au contrôle `ScrollViewer` contenu dans le composant `ListBox` ;
- `BobifyMediaScrollBarStyle` comme style la barre de défilement ;
- `BobifyMediaScrollBarRepeatButtonStyle` correspondant aux flèches basses et hautes ;
- `BobifyMediaScrollBarThumbStyle` pour le curseur la barre de défilement.

Une fois que vous avez modifié le visuel de chaque contrôle, l'idéal est de créer des animations entre les états visuels `Normal`, `MouseOver` et `Pressed`. La Figure 11.53 montre le visuel ainsi qu'une liste des styles finalisés sans toutefois afficher les transitions d'interactions utilisateur.

Figure 11.53

Contrôle ScrollBar modifié
et styles générés.



Le projet, *LecteurMultiMedia_ScrollBar.zip*, est dans le dossier *chap11* des exemples du livre. Dans la prochaine section, nous allons nous consacrer à la personnalisation des éléments de la liste à afficher.

11.7.3 DataTemplate versus ListBoxItem

Nous allons maintenant aborder la partie plus délicate concernant les éléments affichés par liste. Comme nous l'avons dit plus haut, ces derniers sont de type `ListBoxItem` et héritent de la classe `ContentControl`. À ce titre, ils possèdent un `ContentPresenter` au sein de leur modèle définissant la zone d'affichage du contenu. Lorsque vous affectez une instance de `List<Object>` à la propriété `ItemsSource` d'une `ListBox`, par défaut, chacun des objets invoque sa méthode `ToString()`. Le retour de cette méthode est affecté à la propriété `Content` dont l'affichage est géré par l'objet `ContentPresenter` de la `ListBox`. Ce n'est pas ce qui se passe dans notre cas. Par défaut, Blend a généré un `DataTemplate` stocké comme ressource afin de pouvoir afficher les champs de chaque enregistrement. Ce `DataTemplate` est affecté à la propriété `ContentTemplate` de chaque `ListBoxItem`. Il lui indique comment représenter visuellement l'enregistrement (également appelé *value object*) qui est affecté à la propriété `Content` de chacun des éléments de la liste. Pour accéder au `DataTemplate`, cliquez-droit sur l'instance de `ListBox`, sélectionnez le menu `Edit Additional Templates > Edit Generated Items` (voir Figure 11.54).

Figure 11.54

Accéder au modèle de
données (DataTemplate).



Vous accédez à l'arbre visuel du modèle de donnée, vous y trouverez tous les composants nécessaires à l'affichage des données. Passez en mode d'édition XAML, voici le code correspondant :

```
<DataTemplate x:Key="ItemTemplate">
    <StackPanel>
        <TextBlock Text="{Binding auteur}"/>
        <TextBlock Text="{Binding couleur}"/>
        <TextBlock Text="{Binding date}"/>
        <TextBlock Text="{Binding description}"/>
        <TextBlock Text="{Binding duree}"/>
    </StackPanel>
</DataTemplate>
```

```

        <TextBlock Text="{Binding idType}"/>
        <TextBlock Text="{Binding titre}"/>
        <Image Source="{Binding type}"
            HorizontalAlignment="Left" Height="64" Width="64"/>
        <TextBlock Text="{Binding url}"/>
    </StackPanel>
</DataTemplate>

```

Incidentement, la ressource `DataTemplate` influence beaucoup le rendu final de la liste, elle n'est toutefois pas contenue directement dans le modèle du `ListBoxItem`. Cela engendre des limitations en terme de design. Par exemple, seul le contrôle `ListBoxItem` est capable de réagir aux interactions utilisateurs telles que le survol de la souris. Le mieux serait de copier-coller le `StackPanel` et son contenu dans l'arbre visuel du `ListBoxItem` afin de récupérer toutes les liaisons nécessaires simplement.

Copiez le code XAML décrivant la balise `StackPanel` et revenez au niveau de l'application. Cliquez-droit sur la liste, choisissez le menu `Edit Additional Templates > Edit Generated Item Container > Edit a Copy...` Dans la fenêtre qui s'ouvre, nommez le style `BobifyMediaListBoxItemStyle`, puis stockez-le dans le dictionnaire adéquat. Collez le code XAML au bon endroit comme montré ci-dessous :

```

<Style x:Key="BobifyMediaListBoxItemStyle" TargetType="ListBoxItem">
    <Setter Property="Padding" Value="3"/>
    ...
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="ListBoxItem">
                <Grid Background="{TemplateBinding Background}">
                    <VisualStateManager.VisualStateGroups> ...
                    <Rectangle x:Name="fillColor" Fill="#FFBADDE9" .../>
                    <Rectangle x:Name="fillColor2" Fill="#FFBADDE9" .../>
                    <ContentPresenter x:Name="contentPresenter" .../>
                    <Rectangle x:Name="FocusVisualElement" .../>
                    <StackPanel>
                        <TextBlock Text="{Binding auteur}"/>
                        <TextBlock Text="{Binding couleur}"/>
                        <TextBlock Text="{Binding date}"/>
                        <TextBlock Text="{Binding description}"/>
                        <TextBlock Text="{Binding duree}"/>
                        <TextBlock Text="{Binding idType}"/>
                        <TextBlock Text="{Binding titre}"/>
                        <Image Source="{Binding type}" .../>
                        <TextBlock Text="{Binding url}"/>
                    </StackPanel>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>

```

Vous n'avez plus besoin du composant `ContentPresenter` dans l'arbre visuel, vous pouvez le supprimer. L'objectif est maintenant d'agencer les composants récupérés afin d'obtenir un visuel équivalent à celui exposé par la grille `PseudoListe`. Grâce à cette manipulation, nous pourrions obtenir une expérience utilisateur ainsi qu'une ergonomie plus poussées. La première étape consiste à remplacer l'affectation des remplissages des rectangles `fillColor`, `fillColor2` et `FocusVisualElement` par une liaison pointant vers la couleur :

```

<Rectangle x:Name="fillColor" Fill="{Binding couleur}" RadiusX="1"
           RadiusY="1" IsHitTestVisible="False" Opacity="0"/>

<Rectangle x:Name="fillColor2" Fill="{Binding couleur}" RadiusX="1"
           RadiusY="1" IsHitTestVisible="False" Opacity="0"/>

<Rectangle x:Name="FocusVisualElement" Stroke="{Binding couleur}"
           StrokeThickness="2" RadiusX="1" RadiusY="1"
           Visibility="Collapsed"/>

```

Compilez et testez le projet. Vous remarquez que les couleurs de survol et de sélection correspondent désormais au type de média. Vous pouvez modifier les valeurs d'opacité des couleurs au sein des états visuels adéquats, afin d'obtenir un peu plus de punch. Pour plus de fluidité, créez des transitions d'environ 0,4 seconde avec une courbe de décélération de votre choix. Nommez ensuite les champs texte afin de pouvoir les identifier plus facilement dans l'arbre visuel. Voici la liste des tâches à réaliser pour arriver à une version plus aboutie :

- Dégrouppez le `StackPanel` et créez à la place trois colonnes et deux lignes dans la grille. Ceci permet de mettre en page les composants de manière simplifiée, tout en prenant en compte d'éventuels redimensionnements de la liste.
- La première et la seconde colonne possèdent respectivement des largeurs de 64 et de 100 pixels. La dernière colonne est en mode relatif et occupe l'espace restant à disposition.
- Placez l'image et le champ exposant la durée, dans la première colonne.
- Le composant `TextBlock` qui est lié au titre doit être dans la deuxième colonne.
- La description, l'auteur et la date sont à positionner dans la dernière colonne.
- Vous pouvez supprimer les champs texte exposant l'URL, la couleur et la propriété `idType`.
- Définissez les marges et les options d'alignement adéquates.
- La couleur des textes affichés doit par défaut être blanche, puis virer vers le noir lorsque l'un des éléments de la liste est sélectionné, qu'il ait le focus utilisateur ou non.
- Choisissez la police de caractères Tahoma que nous avons embarquée précédemment. Spécifiez des tailles de polices différentes en fonction de l'importance de chaque champ.
- Créez une instance de `Border` s'adaptant à la totalité de la grille, puis définissez une bordure basse de 0,4 pixel d'épaisseur. Cette ligne sépare les éléments les uns des autres et améliore ainsi la lisibilité.

Une fois ces quelques étapes abouties, vous obtenez un visuel très proche de celui proposé par la maquette de départ. Il nous reste à trouver un moyen élégant d'afficher la durée au format adéquat. Pour cela il nous faut encore une fois utiliser notre classe de conversion. Il suffit juste de définir la valeur du paramètre `Convert` :

```

<TextBlock x:Name="duree"
  Text="{Binding duree,
    Converter={StaticResource D2TCCConverter},
    Mode=OneWay
  }"
  HorizontalAlignment="Center" VerticalAlignment="Bottom" Foreground="White"
  FontFamily="/LecteurMultiMedia;Component/Fonts/Fonts.zip#Tahoma"
  FontSize="10.667"
  Grid.Row="1" Margin="0,0,0,8"/>

```

Nous aurons tout de même à modifier la classe de conversion car dans le cas d'une image, il n'y a pas besoin d'afficher de durée, il nous suffit juste de tester la valeur et de renvoyer une chaîne de caractères vide si la durée est égale à zéro :

```
public object Convert(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
{
    double v ;

    try
    {
        v = (double)value;
    }
    catch (Exception e)
    {
        return String.Empty;
    }

    if (v == 0) return String.Empty;

    TimeSpan ts = new TimeSpan();
    if (value is double) ts = TimeSpan.FromMilliseconds(v*1000);

    string p = parameter as string;
    if (p == "00:00.000")
    {
        return ts.Minutes.ToString("00")
            + ":" + ts.Seconds.ToString("00")
            + "." + ts.Milliseconds.ToString("000");
    }

    return ts.Minutes.ToString("00")
        + ":" + ts.Seconds.ToString("00");
}
```

Compilez et testez le projet, nous avons presque atteint nos objectifs (voir Figure 11.55).

Figure 11.55

Le composant ListBox personnalisé.



Il nous reste encore à gérer le redimensionnement de la liste. Diminuez sa largeur pour voir comment l'agencement de ses éléments réagit. Comme vous le constatez plusieurs points méritent notre attention. Tout d'abord, la grille ne doit pas excéder une hauteur trop importante. Dans l'idéal, les instances de `ListBoxItem` devront avoir les mêmes dimensions, une valeur de 100

pixels pour la propriété Height est idéale. Le bloc de texte contenant la description ne doit pas excéder 46 pixels de hauteur. Voici le code XAML définitif concernant la personnalisation du modèle de ListBoxItem :

```
<Border Grid.RowSpan="2" BorderThickness="0,0,0,0.4" Grid.ColumnSpan="3"
        Opacity="0.4" Margin="4,0">
    <Border.BorderBrush>
        <SolidColorBrush Color="{StaticResource PlayerBackground}" />
    </Border.BorderBrush>
</Border>

<Rectangle x:Name="fillColor" Fill="{Binding couleur}" RadiusX="1"
        RadiusY="1" IsHitTestVisible="False" Opacity="0" Grid.
        ColumnSpan="3" Grid.RowSpan="2" />

<Rectangle x:Name="fillColor2" Fill="{Binding couleur}" RadiusX="1"
        RadiusY="1" IsHitTestVisible="False" Opacity="0" Grid.
        ColumnSpan="3" Grid.RowSpan="2" />

<Rectangle x:Name="FocusVisualElement" Stroke="{Binding couleur}"
        StrokeThickness="2" RadiusX="1" RadiusY="1"
        Visibility="Collapsed" Grid.ColumnSpan="3" Grid.RowSpan="2" />

<TextBlock x:Name="auteur" Text="{Binding auteur}"
        HorizontalAlignment="Left" Margin="4,4,0,0"
        VerticalAlignment="Top" Grid.Column="2" Grid.Row="1"
        Foreground="White" FontFamily="/LecteurMultiMedia;Component/
        Fonts/Fonts.zip#Tahoma" FontSize="10.667" />

<TextBlock x:Name="date" Text="{Binding date}" HorizontalAlignment="Right"
        Margin="0,4,4,0" VerticalAlignment="Top" Grid.Column="2" Grid.
        Row="1" Foreground="White" FontFamily="/
        LecteurMultiMedia;Component/Fonts/Fonts.zip#Tahoma"
        FontSize="10.667" />

<TextBlock x:Name="description" Text="{Binding description}"
        Margin="4,8,4,0" VerticalAlignment="Top" Grid.
        Column="2" TextWrapping="Wrap" Foreground="White" FontFamily="/
        LecteurMultiMedia;Component/Fonts/Fonts.zip#Tahoma"
        FontSize="10.667" Height="46" />

<TextBlock x:Name="duree" Text="{Binding duree, Converter={StaticResource
        D2TCConverter}, Mode=OneWay}" HorizontalAlignment="Center"
        VerticalAlignment="Top" Foreground="White" FontFamily="/
        LecteurMultiMedia;Component/Fonts/Fonts.zip#Tahoma"
        FontSize="10.667" Grid.Row="1" Margin="0,4,0,0" />

<TextBlock x:Name="titre" Text="{Binding titre}" Margin="4,0"
        VerticalAlignment="Center" Grid.Column="1" TextWrapping="Wrap"
        d:LayoutOverrides="HorizontalAlignment" Foreground="White"
        FontFamily="/LecteurMultiMedia;Component/Fonts/Fonts.zip#Tahoma"
        FontSize="12" Grid.RowSpan="2" Height="40" />

<Image x:Name="icon" Source="{Binding type}"
        HorizontalAlignment="Center" Height="32"
        VerticalAlignment="Center" Width="48" />
```

La personnalisation de la liste est terminée, vous pourriez encore l'améliorer ou bien vous entraîner avec le composant AutoCompleteBox. Ce dernier est un mélange entre une liste et un champ texte de saisie ; il constitue un petit défi. Vous possédez déjà le style de la ScrollBar contenue

dans la liste sur lequel vous pouvez vous baser. Vous pouvez également supprimer les objets dont le nom est préfixé par *Pseudo*, puis créer les états visuels de l'application permettant d'étendre ou de replier la liste. Le bouton nommé *ExpanderToggleButton* a été créé à cet effet. En mode replié, seuls les titres, les icônes ainsi que la durée restent visibles. Vous pouvez télécharger le projet finalisé *LecteurMultiMedia_DesignFinal.zip* dans le dossier *chap11* des exemples du livre.

Au Chapitre 12, nous aborderons la conception de composants. Nous apprendrons ainsi à centraliser et à formaliser les fonctionnalités au sein de composants dédiés que vous pourrez partager et réutiliser dans vos projets.

Composants personnalisés

La conception de composants réutilisables a toujours été au centre de nombreux débats. Créer un composant *from scratch* pose un certain nombre de questions. Par exemple, est-ce que le composant n'existe pas déjà tout fait sur le Web ? Ou encore, est-il productif de concevoir un contrôle si celui-ci est spécifique à votre problématique actuelle ? Certains indices peuvent vous éclairer, avant de vous décider à concevoir un tel composant. Il vous faut d'abord envisager tous les cas de réutilisation possible. En effet, qui dit conception de composant dit réutilisation générique, cette notion vous fera automatiquement sortir du cadre de production fixé par le projet en cours. Autrement dit, pour que le composant soit réutilisable, il faut évaluer un certain nombre de cas d'utilisation qui ne sont pas forcément prévus dans le cahier des charges de votre projet. Il est bien sûr possible d'implémenter de nouvelles fonctionnalités au fur et à mesure de vos besoins et des projets. Toutefois, concevoir un composant pose obligatoirement une réflexion plus vaste que celle imposée par les nécessités du moment. Ce qui est fait n'est plus à faire, lorsque vous vous lancez dans ce type de développement, vous faites en quelque sorte un pari sur l'avenir, si le composant n'est pas réutilisé, c'est que ce pari aura échoué. Afin d'optimiser le temps, il vous faut clairement définir le cadre d'utilisation ainsi qu'une certaine granularité dans les niveaux de services fournis par vos composants. Une fois lancé, il est difficile de savoir où s'arrêter. De ce point de vue, le projet en cours joue le rôle d'un garde-fou. Dans ce chapitre, vous apprendrez à identifier vos besoins et à faire les bons choix de conception. Nous aborderons deux manières différentes de concevoir des contrôles.

12.1 Contrôle utilisateur

Créer un composant de toute pièce revient toujours à hériter d'une classe. Celle-ci doit implémenter la logique ainsi que les mécanismes de base que vous utiliserez. Dans cette section, nous listerons les différents types d'héritage possibles, puis nous construirons notre premier composant utilisateur.

12.1.1 Un héritage pertinent

Plus vous dériverez vos composants de classes spécialisées, plus vous embarquerez de code à l'initialisation de ce dernier. Le tout est de connaître les capacités supportées par chaque famille de composants Silverlight afin de créer le minimum de logique tout en répondant au cahier des charges. Cela permet d'optimiser le poids et de minimiser la logique embarquée en ne laissant accès qu'au strict nécessaire. De cette manière, vous ne polluez pas l'utilisateur du composant par d'inutiles propriétés. On peut considérer plusieurs grandes familles de composants, mais seules certaines d'entre elles sont réellement exploitables dès lors que vous souhaitez créer vos propres composants :

- La première famille concerne les contrôles dont les modèles sont personnalisables. Ils héritent de la classe abstraite `Control`.
- Un sous-ensemble de la classe `Control` correspond aux contrôles utilisateur (de type `UserControl`) que nous allons aborder dans cette section.
- Les composants non personnalisables d'un point de vue modèle, comme `TextBlock`, qui héritent de `FrameworkElement` et ne possèdent donc pas la propriété `Template`. Plus vous utiliserez une classe générique, plus votre composant sera léger et pertinent. Hériter de `FrameworkElement` s'avère pertinent dans certains cas particuliers.
- La famille des conteneurs à plusieurs enfants, tels que `Grid` ou `StackPanel` joue un rôle important. Il est toutefois plus rare d'avoir besoin d'un nouveau panneau. Ils héritent de la classe abstraite `Panel` et utilisent en interne les mécanismes propres au système d'agencement Silverlight. Nous ne concevons pas de conteneur personnalisé dans ce chapitre, le blog <http://www.tweened.org> contient un article décrivant un tutoriel pas à pas à ce propos.
- Les conteneurs spécialisés à enfant unique tels que `Popup`, `Border`. Ils héritent directement de `FrameworkElement` et sont des classes fermées à la modification. La classe `ContentControl` qui hérite de `Control` en fait partie car elle possède l'attribut `ContentPropertyAttribute` qui fournit cette capacité. La classe `ViewBox` hérite de `ContentControl`.
- Les classes de présentation telles que `ContentPresenter` et `ItemsPresenter` dont l'objectif est de formaliser des données ou certains types d'objets de manière visuelle.
- Les formes simples comme `Ellipse` ou `Rectangle` qui dérivent de `Shape`. En apparence, il est possible d'hériter de `Shape`, toutefois contrairement à WPF, cette classe ne possède pas la méthode qui permet de définir une nouvelle géométrie. Il n'est donc pas possible de créer de formes personnalisées par ce biais.

Cette classification est arbitraire et n'est valable que d'un point de vue fonctionnel et pratique. Pour notre part, nous aborderons la création de composants héritant de `UserControl` et de `Control`.

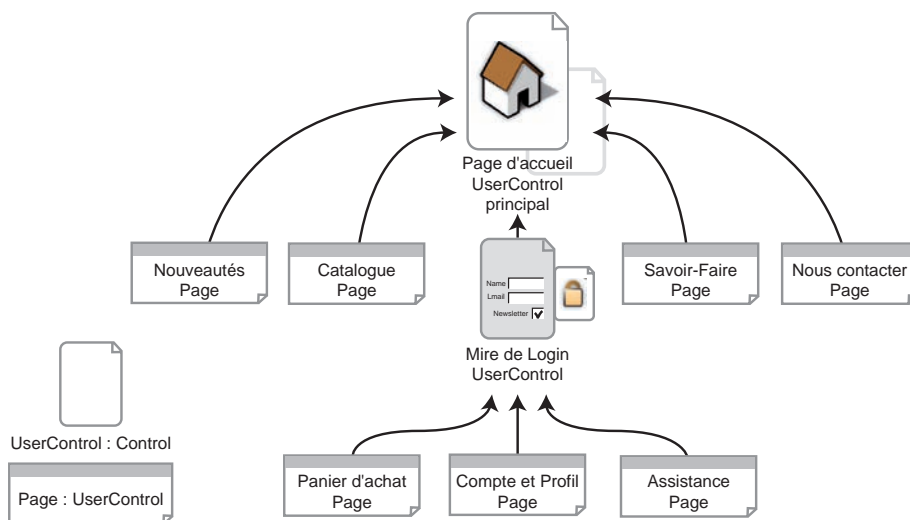
12.1.2 Navigation page par page

Lorsque vous concevez un composant, il convient d'identifier correctement ses objectifs. Apporte-t-il une nouvelle ergonomie, ou traite-t-il une fonctionnalité macro telle qu'un processus utilisateur. Dans le premier cas, les contrôles représentent une ergonomie, un type d'interaction ou une fonctionnalité spécifiques. Un `RadioButton`, une liste déroulante (`ComboBox`), une grille d'alignement (`Grid`) ou un champ de complétion automatique (`AutoCompleteBox`) sont autant d'exemples faisant partie de cette catégorie. Nous verrons leur conception à la section 12.3.

Dans le second cas, le contrôle est très souvent composé de plusieurs autres contrôles et peut être considéré comme un module applicatif à part entière. Un lecteur vidéo, une fenêtre de connexion ou un catalogue de produits sont des cas représentatifs de cette catégorie. Ils font référence à des nécessités applicatives et contiennent en interne un flux d'utilisation métier. On les considère comme des modules autonomes et ils sont, à ce titre, généralement conçus sous forme d'instances de type `UserControl`. La classe principale `MainPage` étant elle-même issue de type `UserControl`, il est naturel et sain qu'un `UserControl` en contienne plusieurs autres ou soit lui-même chargé ou instancié dynamiquement (voir Figure 12.1).

Figure 12.1

Arborescence constituée de diverses instances de type `UserControl` et `Page`.



La classe `Page`, dont vous pouvez voir plusieurs instances sur la Figure 12.1, hérite de `UserControl` et implémente des capacités supplémentaires en matière de navigation. À l'instar de `UserControl`, la classe `Page` est formalisée à l'aide de deux fichiers contenant respectivement le langage déclaratif XAML et le code logique C#. Ouvrez le projet de l'archive *WebNavigation_Base.zip* du *chap12* des exemples de ce livre. Nous allons partir de ce projet pour créer un site web multipages avec gestion de l'historique.

Supprimez la grille grise située au centre de `LayoutRoot`, en faisant attention au code logique qui pilote le composant `TextBlock`. Remplacez la grille par un composant `Frame` que vous trouverez dans le panneau `Assets` et nommez-le `frame`. Il doit posséder les mêmes marges et modes de redimensionnement que la grille supprimée. L'espace de noms `navigation` est automatiquement généré lors de l'instanciation de `Frame`. Ce composant a pour but de gérer l'affichage des différentes pages, leur mise en cache ainsi que l'historique de navigation. Celui-ci est directement pilotable *via* votre navigateur web favori, l'utilisation d'URL avec ancres (#) ou les flèches avant et arrière. Cliquez-droit sur le projet, puis sélectionnez le menu `Add New Item...` Dans la boîte de dialogue affichée, choisissez `Page` et nommez-la `Nouveautes.xaml` (voir Figure 12.2).

Répétez l'opération pour générer des pages correspondant aux menus affichés en haut de `LayoutRoot` sans oublier la page `Accueil.xaml` qui sera chargée par défaut. Pour une meilleure organi-

sation du projet, vous pouvez créer un répertoire nommé `navPages`, puis ajouter chaque page au projet par un clic-droit sur celui-ci (voir Figure 12.3).

Figure 12.2

Création des pages de navigation.

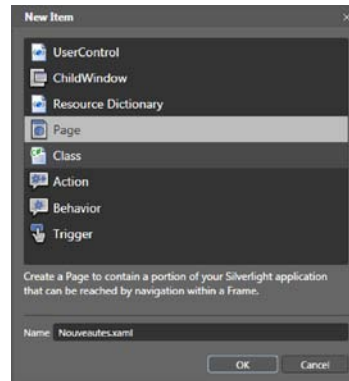
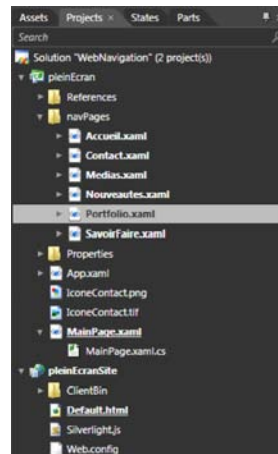


Figure 12.3

Le projet avec les pages générées.



Mis à part l'espace de noms et le type, le code XAML de chaque page générée est similaire à celui d'un `UserControl` standard. Une grille principale transparente nommée `LayoutRoot` est générée par défaut et les dimensions de l'instance de `Page` sont en mode `Auto`. Celles-ci s'adaptent en fait par défaut à celles de l'objet `Frame` de la page principale car il est responsable de leur affichage :

```
<navigation:Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=
                        System.Windows.Controls.Navigation"
    x:Class="pleinEcran.Nouveautes"
    Title="Nouveautes Page"
    d:DesignWidth="640" d:DesignHeight="480" >
    <Grid x:Name="LayoutRoot" />
</navigation:Page>
```

Vous pouvez créer du contenu pour ces pages. Afin de se concentrer sur le principe de navigation, définissez une couleur d'arrière plan ainsi qu'un titre central pour chacune d'entre elles. Une fois cette tâche effectuée, revenez dans la page principale, sélectionnez le composant Frame, puis affectez la chaîne de caractères `/navPages/Accueil.xaml` à sa propriété Source :

```
<navigation:Frame x:Name="navFrame" Margin="30,60,30,30"
    Background="#00742323" Source="/navPages/Accueil.xaml"/>
```

Compilez l'application pour tester le chargement de la page `Accueil.xaml`. Il s'agit maintenant de mettre à jour cette propriété lors du clic sur chaque menu. Sélectionnez le premier bouton du `WrapPanel`, puis affectez la propriété Tag de la chaîne de caractères `/navPages/Nouveautes.xaml`. Répétez cette opération pour l'ensemble des boutons. Nommez le `WrapPanel` `MenuHaut`, puis ouvrez la solution dans Visual Studio. Nous allons parcourir les enfants du `WrapPanel` et affecter un écouteur identique pour chacun d'eux. Nous allons récupérer le diffuseur de l'événement `Click` dynamiquement afin de récupérer la valeur de sa propriété Tag :

```
public MainPage()
{
    InitializeComponent();

    foreach (Button b in MenuHaut.Children)
    {
        b.Click += new RoutedEventHandler(showPage);
    }
}

void showPage(object sender, RoutedEventArgs e)
{
    Button b = sender as Button;
    if (b != null)
    {
        frame.Navigate(new Uri(b.Tag.ToString(), UriKind.Relative));
        //frame.Source = new Uri(b.Tag.ToString(), UriKind.Relative);
    }
}
```

Vous pouvez soit utiliser la méthode `Navigate` soit la propriété `Source` pour arriver à vos fins. Testez l'application dans le navigateur, le composant `Frame` affiche chaque page l'une après l'autre après qu'un clic ait été diffusé par chacun des menus. `Frame` permet également de gérer la navigation *via* l'utilisation directe de la barre d'adresse du navigateur. La propriété `UsesJournalParent` est responsable de ce comportement. Par défaut, elle est en mode automatique, si l'instance de `Frame` n'est pas affichée dans une autre `Frame`, elle utilise le journal de navigation du navigateur web. Vous obtenez des URL du type :

```
http://localhost:49160/Default.html#/navPages/Nouveautes.xaml
```

INFO

La navigation, dans ce type d'application, est également réalisable *via* des instances de `HyperlinkButton`. Il suffit de renseigner la propriété `Target` avec le nom du composant `Frame` puis de préciser l'URL de la page chargée dans la propriété `Source` de l'`HyperlinkButton`.

Les instances de Page associées au composant Frame offrent des fonctionnalités de navigation puissantes. Il est ainsi possible de générer des URL plus standard et lisibles de manière dynamique en mappant les adresses XAML de type `#/navPages/Nouveautes.xaml`. Il est également possible de détecter le chargement ou le déchargement d'une page. Cela permet d'ajouter des transitions ou de la logique métier comme le chargement de données ou la sauvegarde d'actions effectuées dans la page. Vous trouverez le projet finalisé dans le dossier *chap12* des exemples : *WebNavigation.zip*.

Nous allons maintenant créer un UserControl d'un point de vue composant.

12.2 Boîte de connexion

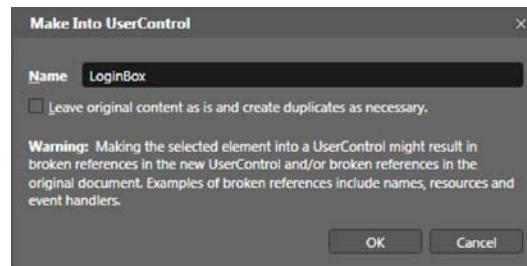
Dans la grande majorité des cas, vous concevrez un UserControl lorsque vous aurez besoin de centraliser la logique et l'interface utilisateur dédiées à une fonctionnalité spécifique de votre application. L'exemple le plus emblématique est la mire de connexion. Téléchargez le projet *LecteurMultimedia_BaseUC* présent dans le *chap12* des exemples de ce livre.

12.2.1 Conception visuelle

Sélectionnez la grille nommée *PanneauAccueil*, puis utilisez le raccourci F8 ou le menu Tools et l'option Make Into UserControl... Une fenêtre apparaît vous demandant de renseigner le nom du nouveau UserControl. Renseignez *LoginBox* dans le champ de saisie correspondant et laissez l'autre option décochée. Celle-ci vous permet de laisser l'arborescence intacte si vous la cochez. Dans notre cas, nous souhaitons remplacer la grille *PanneauAccueil* par notre UserControl (voir Figure 12.4).

Figure 12.4

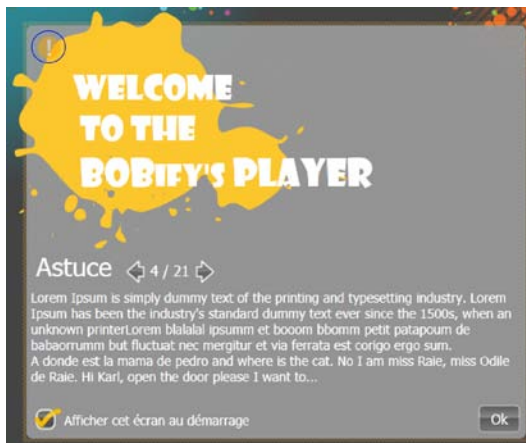
La fenêtre de création d'un UserControl.



Blend génère automatiquement un UserControl à la racine du projet, qui est constitué d'un fichier XAML et d'un document C#. Le code logique contient une classe partielle *LoginBox* faisant directement référence au code déclaratif XAML contenu dans *LoginBox.xaml*. Les principes d'architecture sont strictement identiques à ceux déjà évoqués dans *MainPage.xaml*. Il est donc possible de créer des états visuels pour *LoginBox*, de la logique, des données fictives et tout autre type de ressources. La seule différence est que *LoginBox* est affiché au sein d'un autre UserControl. À ce titre, il ne peut être visualisé correctement dans ce dernier sans être préalablement compilé. Dans le cas contraire, si vous ouvrez à nouveau *MainPage.xaml*, une icône en forme de point d'exclamation est affichée en haut à gauche et un cadre jaune/orange entoure l'instance du UserControl *LoginBox* (voir Figure 12.5).

Figure 12.5

Prévisualisation du rendu du UserControl avant sa compilation.

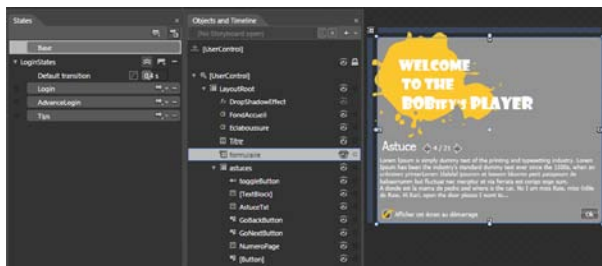


Compilez afin de visualiser le rendu final : le point d'exclamation et la bordure jaune disparaissent. L'arbre visuel de l'application principale a été allégé puisque seule l'instance du `UserControl` géré est visible, son arborescence est dorénavant située dans le fichier `LoginBox.xaml`.

Vous allez concevoir plusieurs états visuels permettant à un éventuel utilisateur de se connecter avant de voir les astuces s'afficher. Il vous faut un état visuel affichant la mire de login par défaut, un état de connexion avec une option avancée ainsi que l'état affichant une astuce une fois que la connexion est acceptée. Créez un groupe nommé `LoginStates` dans le panneau `States`, puis ajoutez les états `Login`, `AdvanceLogin` et `Tips` (voir Figure 12.6). Vous pourriez également créer un état correspondant au refus de connexion côté serveur, vous allez toutefois procéder de manière différente pour gérer les erreurs de connexion. Il vous faut également déplacer le fond gris, l'éclaboussure jaune et le titre dans la grille `LayoutRoot`. De cette façon, nous allons centraliser les éléments relatifs aux astuces et ceux du futur formulaire de connexion dans des conteneurs différents, cela facilitera la gestion des transitions (voir Figure 12.6).

Figure 12.6

États visuels du composant `LoginBox`.

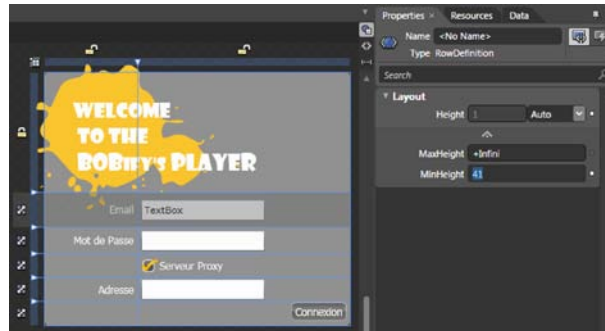


Renommez et réorganisez selon vos besoins l'arbre visuel de l'application, vous pouvez également vous fier à la Figure 12.6. Cachez la grille nommée `astuces` via l'icône de l'œil dans l'arbre visuel et logique. De cette manière vous allez pouvoir travailler sur le formulaire sans vous encombrer d'éléments inutiles. Dans la grille formulaire, créez cinq lignes en redimensionnement automatique en cliquant sur le liseré bleu présent à gauche de celle-ci, puis créez une colonne en redimensionnement relatif (icône du cadenas ouvert) de 30 % à gauche et de 60 % à droite. Cela va nous permettre de positionner les éléments du formulaire simplement. Mis à part pour la première

ligne qui doit posséder environ 180 pixels de hauteur en dur, vous devez vérifier que les lignes en redimensionnement automatique possèdent 0 comme valeur minimum allouée (voir Figure 12.7).

Figure 12.7

Conception du formulaire.



Arrangez-vous pour que la hauteur (Height) de tous les éléments du formulaire soient en mode de redimensionnement automatique, mis à part le TextBlock "Adresse" ainsi que le champ de saisie renseignant l'adresse du serveur proxy. Garder des valeurs en dur permet de les animer plus simplement lorsque vous aurez besoin de gérer la transition permettant d'afficher ou non le mode de connexion avancé. Sélectionnez le UserControl racine principal puis, dans le code XAML, supprimez la propriété `d:DesignHeight` afin d'avoir un aperçu de l'adaptation de sa hauteur, en fonction du contenu affiché. Vérifiez également que le fond et la grille astuces possèdent une hauteur en mode Auto (voir Figure 12.8).

Figure 12.8

Mise en forme des conteneurs et arbre visuel et logique de la fenêtre de connexion.



Sélectionnez chaque état visuel et modifiez les états respectifs de chaque élément en générant des animations traditionnelles. Cela permet de désactiver les interactions utilisateur sur les objets cachés en passant la valeur de leur propriété `Visibility` à `Collapsed` en toute fin d'animation. Vous avez de nombreuses modifications à faire, mais la bonne pratique consiste à définir l'état de base comme celui affiché par défaut. Dans notre cas, il s'agit de faire apparaître la mire de login sans le paramétrage éventuel d'un proxy. Il sera ensuite plus aisé de configurer les trois états visuels. N'oubliez pas également que certaines modifications sont à prévoir dans le fichier `MainPage.xaml`. Vous pouvez télécharger le projet avec les états visuels et l'intégration du contrôle utilisateur. Il se trouve dans l'archive *LecteurMultimedia_LoginBox.zip* du dossier *chap12*. Ce dernier pourra être utile pour la deuxième partie de cet exercice.

12.2.2 Validation par liaison de données

Depuis Silverlight 3, les objets de l'arbre visuel peuvent capter les exceptions levées lors de la mise à jour de sources de données. Le cas le plus flagrant concerne les formulaires. Il est souvent pratique de définir un objet métier comme contexte de données du formulaire, les champs de saisie définissent des liaisons à deux voies permettant la mise à jour de l'objet métier lors de la saisie utilisateur. La classe métier C# possède elle-même des accesseurs qui lèvent des erreurs lorsque les données renseignées ne correspondent pas au format attendu. Ces erreurs sont captées par les instances de FrameworkElement de la liste d'affichage, qui affichent en retour une alerte visuelle indiquant l'erreur rencontrée. Cela est assez simple à réaliser, reprenez le projet Lecteur-Multimedia dans sa dernière version.

Dans Visual Studio, créez un répertoire `classes`, cliquez-droit dessus et ajoutez une classe nommée `UserCredentials`. Celle-ci possède une suite de champs privés et de propriétés associées qui contiennent des accesseurs gérant la validation des données fournies. L'une de ces données permet de valider le formatage d'un e-mail et lève une exception en cas d'erreur de format détectée :

```
...
using System.Text.RegularExpressions;

namespace LecteurMultiMedia.classes
{
    public class UserCredentials
    {
        private string email;

        private string password;

        private bool useProxy = false;

        private string proxyAddress = String.Empty;

        public string Email
        {
            get
            {
                return this.email;
            }
            set
            {
                Regex myRegex = new Regex(@"^([a-zA-Z0-9_\-\.\.])@(\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\] | (([a-zA-Z0-9_\-\.\.]+)\.))([a-zA-Z]{2,4}|[0-9]{1,3})(\?)?$", RegexOptions.IgnoreCase);
                if (!myRegex.IsMatch(value))
                {
                    throw new Exception
                        ("L'adresse email fournie n'est pas valide.");
                }
                else
                {
                    this.email = value;
                }
            }
        }

        public string Password
        {

```

```

        get
        {
            return this.password;
        }
        set
        {
            if (String.IsNullOrEmpty(value) || value.Length < 4)
            {
                throw new Exception("Le mot de passe possède au moins 4
                                     caractères.");
            }
            else
            {
                this.password = value;
            }
        }
    }

    public bool UseProxy
    {
        get
        {
            return this.useProxy;
        }
        set
        {
            this.useProxy = value;
        }
    }

    public string ProxyAdress
    {
        get
        {
            return this.proxyAdress;
        }
        set
        {
            if (this.useProxy == true)
            {
                this.proxyAdress = value;
            }
            else
            {
                this.proxyAdress = String.Empty;
            }
        }
    }
}
}
}

```

Chaque propriété peut lever une exception personnalisée lors de l'affectation d'une nouvelle valeur, cela est réalisé assez simplement *via* l'instruction `throw new Exception()`. Dans le code ci-dessus, vous pouvez améliorer le contrôle d'affectation de l'adresse proxy à l'aide d'une autre expression régulière. Une fois la classe de données créée, il suffit d'en affecter une instance comme contexte de données de la grille formulaire. Celle-ci est située au sein du contrôle utilisateur `LoginBox`. Le mieux est de faire de l'objet de données un membre de la classe `LoginBox`, de cette manière, il reste disponible tout au long de l'application :

```

public partial class LoginBox : UserControl
{
    private UserCredentials currentUserCredentials =
        new UserCredentials();

    public LoginBox()
    {
        InitializeComponent();

        formulaire.DataContext = currentUserCredentials;
    }
}

```

Du côté d'Expression Blend, nous n'avons plus qu'à définir une liaison de données à deux voies. Lorsque l'utilisateur renseignera les informations de connexion demandées, il mettra à jour l'objet de donnée et sera notifié des éventuelles erreurs de saisie levées par les accesseurs de la classe `UserCredentials`. Le code XAML est assez simple :

```

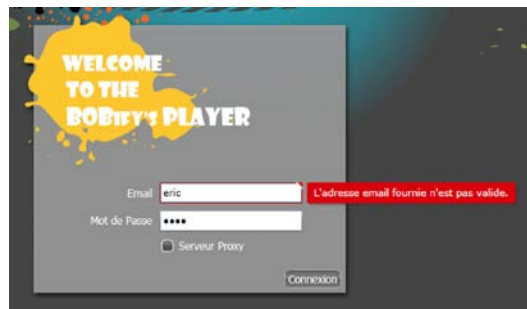
<TextBox ... Text="{Binding Email, Mode=TwoWay, ValidatesOnExceptions=True}" ...>
<PasswordBox ... Password="{Binding Password, Mode=TwoWay,
    ValidatesOnExceptions=True}" />
<ToggleButton Content="Serveur Proxy " ... IsChecked="{Binding UseProxy,
    Mode=TwoWay}" ... />
<TextBox x:Name="UrlProxyTxt" Text="{Binding ProxyAdress, Mode=TwoWay,
    ValidatesOnExceptions=True}" .../>

```

Il est important de remarquer que les données saisies sont affectées à l'objet source lorsque l'élément du formulaire de saisie perd le focus utilisateur. Ainsi les exceptions ne sont éventuellement levées qu'à cet instant. Compilez l'application et entrez des valeurs erronées. Afin que le champ de saisie de l'e-mail perde le focus, sélectionnez celui correspondant au mot de passe. La valeur de sa propriété `Text` est ainsi mise à jour et une erreur est levée. Le champ de saisie de l'e-mail est entouré d'un cadre rouge et la chaîne de caractères définie dans l'exception est affichée dans une étiquette qui apparaît sur le côté (voir Figure 12.9).

Figure 12.9

Retour visuel des levées d'exceptions.



Le visuel affiché en cas d'erreur de saisie est entièrement paramétrable, et cela à différents niveaux. Dans un premier temps, vous pouvez configurer l'aspect visuel du cadre rouge apparaissant autour du composant `TextBox` lorsqu'une exception est levée. À cette fin, vous n'avez qu'à modifier le modèle du composant et à accéder au groupe d'états nommés `ValidationStates`. Dans un second temps, il est possible de modifier l'aspect de l'étiquette qui apparaît sur le côté du champ de saisie. Lorsque vous créez un modèle personnalisé de `TextBox`, Blend génère automatiquement un modèle nommé `ValidationToolTipTemplate`. Il est accessible *via* le panneau `Resources` et

contient deux états nommés Open et Closed. Vous pouvez paramétrer la forme et la couleur de l'étiquette à cet endroit. En dernier ressort, la gestion visuelle et logique des exceptions de saisie peut-être contrôlée *via* la diffusion de l'événement `BindingValidationError` par le moteur de liaison. L'événement est en général diffusé par le contexte de donnée, il faut toutefois qu'au moins un de ses enfants définisse à true la valeur de la propriété `NotifyOnValidationError` pour que l'événement soit écouté. Voici le code logique correspondant à l'écoute de l'événement ainsi que la personnalisation du visuel résultant de cet événement :

```
public partial class LoginBox : UserControl
{
    private UserCredentials currentUserCredentials = new UserCredentials();

    public LoginBox()
    {
        InitializeComponent();

        formulaire.DataContext = currentUserCredentials;

        formulaire.BindingValidationError += new EventHandler<
            <ValidationErrorEventArgs>(formulaire_BindingValidationError);
    }

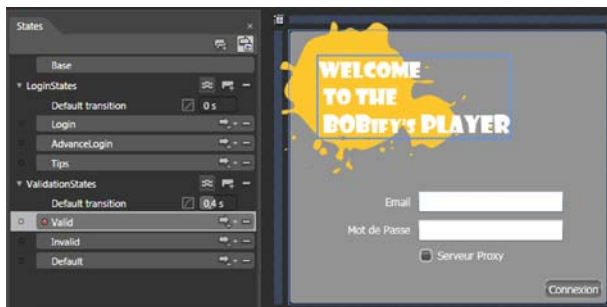
    void formulaire_BindingValidationError(object sender,
        ValidationErrorEventArgs e)
    {
        if (e.Action == ValidationErrorEventAction.Added)
        {
            FondAccueil.BorderBrush = new SolidColorBrush(Colors.Red);
        }
        else if (e.Action == ValidationErrorEventAction.Removed)
        {
            FondAccueil.BorderBrush = new SolidColorBrush(Colors.White);
        }
    }
}
```

Du côté XAML, il suffit d'ajouter l'attribut de liaison `NotifyOnValidationError` permettant la diffusion de l'événement :

```
<TextBox ... Text="{Binding Email,
    Mode=TwoWay,
    ValidatesOnExceptions=True,
    NotifyOnValidationError=true}" .../>
```

Cette fois-ci, non seulement une étiquette rouge apparaît lors d'une erreur, mais la bordure de la fenêtre de login change également de couleur. Pour bien faire, le mieux est de créer des états de validation propres au contrôle `LoginBox` puis d'afficher les uns ou les autres selon qu'il y ait ou non diffusion d'événements indiquant une erreur. Vous laisserez ainsi au designer le soin de paramétrer le visuel générique de la fenêtre de connexion en cas d'erreur de saisie (voir Figure 12.10).

Figure 12.10
Les états visuels
de validation.



Tant qu'une erreur de saisie est détectée, le bouton de connexion est désactivé. Vous évitez ainsi de polluer le serveur. Autrement dit, si le format de l'e-mail renseigné est faux de prime abord, autant ne pas soumettre ce dernier à l'approbation du serveur. Du côté C#, il faut légèrement modifier le code afin d'utiliser les états de validation créés auparavant. Dans l'état *Invalid*, la propriété *IsEnabled* du bouton de connexion est passée à *false* :

```
public partial class LoginBox : UserControl
{
    private UserCredentials currentUserCredentials = new UserCredentials();

    public LoginBox()
    {
        InitializeComponent();

        VisualStateManager.GoToState(this, "Default", false);

        formulaire.DataContext = currentUserCredentials;

        formulaire.BindingValidationError += new EventHandler<
            <ValidationErrorEventArgs>(formulaire_BindingValidationError);
    }

    void formulaire_BindingValidationError(object sender,
        ValidationErrorEventArgs e)
    {
        Debug.WriteLine("diffuseur" + (sender as FrameworkElement).Name);
        if (e.Action == ValidationErrorEventAction.Added)
        {
            VisualStateManager.GoToState(this, "Invalid", true);
        }
        else if (e.Action == ValidationErrorEventAction.Removed)
        {
            VisualStateManager.GoToState(this, "Valid", true);
        }
    }
}
```

Du côté XAML, l'idéal est de définir la propriété *NotifyOnValidationError* à *true* pour le contrôle *PasswordBox*. L'événement est diffusé lorsque la saisie du mot de passe ne correspond pas au format attendu. Dans notre cas, le mot de passe fait au moins quatre caractères. Le code C# ci-dessus permet également de vérifier que les deux conditions, login et mot de passe, sont réunies avant de donner accès à l'état visuel *Valid* permettant la soumission du formulaire. Le projet est dans le dossier *chap12* des exemples du livre : *LecteurMultimedia_Validation.zip*.

12.2.3 Rafrâichissement manuel de liaison de données

Nous avons encore quelques problématiques d'expérience utilisateur. Pour l'instant, les liaisons de données sont rafraîchies de manière automatique. Les composants `TextBox` et `PasswordBox` doivent perdre le focus pour rafraîchir leur valeur et permettre ainsi à la liaison de données de lever ou non une erreur. Le mieux serait d'afficher les erreurs de saisie lorsque l'utilisateur clique sur le bouton de connexion. Pour cela, il nous faut configurer des liaisons de données en mode `Explicit`. Ce mode permet de rafraîchir les valeurs manuellement. Nommez les champs de saisie respectivement `emailTxt` et `passwordTxt`. Ensuite, laissez le bouton de connexion actif (`IsEnabled=true`) quel que soit l'état visuel `Valid`, `Invalid` ou `Default`. Il pourra ainsi être cliquable et rafraîchir les liaisons. Toujours côté XAML, définissez l'attribut `UpdateSourceTrigger` à `Explicit`. Voici le code XAML modifié :

```
<TextBox x:Name="emailTxt"
    Text="{Binding Email,
    Mode=TwoWay,
    NotifyOnValidationError=true,
    ValidatesOnExceptions=True,
    UpdateSourceTrigger=Explicit}"
/>
<PasswordBox x:Name="passwordTxt" ...
    Password="{Binding Password,
    Mode=TwoWay,
    ValidatesOnExceptions=True,
    NotifyOnValidationError=true,
    UpdateSourceTrigger=Explicit}"
/>
```

Le rafraîchissement de la liaison est désormais en mode manuel. Afin de mettre les valeurs, vous devez récupérer la liaison côté C#, puis appelez la méthode `UpdateSource`. Nous allons le faire lorsque le bouton de connexion diffuse l'événement `Click`. Nommez ce dernier `connectionButton`, puis définissez l'écoute de l'événement `Click` dans le constructeur. Pour finir, récupérez les instances de `BindingExpression` au sein de l'écouteur, et appelez leur méthode `UpdateSource` :

```
public LoginBox()
{
    InitializeComponent();

    VisualStateManager.GoToState(this, "Default", false);

    formulaire.DataContext = currentUserCredentials;

    formulaire.BindingValidationError += new EventHandler<
        <ValidationErrorEventArgs>(formulaire_BindingValidationError);

    connectionButton.Click += new RoutedEventHandler
        (connectionButton_Click);
}

void connectionButton_Click(object sender, RoutedEventArgs e)
{
    BindingExpression be1 = emailTxt.GetBindingExpression
        (TextBox.TextProperty);
    be1.UpdateSource();
}
```

```

        BindingExpression be2 = passwordTxt.GetBindingExpression
            (PasswordBox.PasswordProperty);
        be2.UpdateSource();
    }

```

Cette fois, les liaisons sont rafraîchies lors du clic sur le bouton de connexion et lèvent une erreur si besoin. Compilez dans Visual Studio l'application sans le mode Debug *via* le raccourci Ctrl+F5. Le compilateur n'arrêtera pas l'exécution et vous bénéficierez de l'expérience utilisateur final. L'archive du projet est dans le *chap12* des exemples : *LecteurMultimedia_LiaisonManuelle.zip*.

12.2.4 Notification de changement de propriété

Pour le moment, lorsque vous cliquez sur le bouton aucun mécanisme ne permet de tester réellement si les valeurs saisies par l'utilisateur sont correctes. Pour cela, il suffit d'ajouter quelques membres à notre classe `UserCredentials`. Il nous faut créer deux champs privés booléens ainsi qu'une troisième propriété publique booléenne. Ces membres représentent respectivement : la validité de l'e-mail (`isEmailFilled`), celle du mot de passe (`isPasswordFilled`) et pour finir, la combinaison des deux précédentes (`IsCredentialsFilled`). Cette dernière valide la totalité du formulaire. Voici le code C# de la classe `UserCredentials` mise à jour :

```

public class UserCredentials
{
    private string email;

    private string password;

    private bool useProxy = false;

    private string proxyAdress = String.Empty;

    private bool isEmailFilled = false;

    private bool isPasswordFilled = false;

    private bool isCredentialFilled = false;
    public bool IsCredentialsFilled
    {
        get { return isCredentialFilled; }
        set { isCredentialFilled = value; }
    }

    public string Email
    {
        get
        {
            return this.email;
        }
        set
        {
            Regex myRegex = new Regex(@"^([a-zA-Z0-9_\-\.\.])@((\[[0-9]{1,3}\.
                [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\.
                |([a-zA-Z0-9\-\.]
                ([a-zA-Z]{2,4}|[0-9]{1,3})
                (\.?)$",
                RegexOptions.IgnoreCase);

            isEmailFilled = myRegex.IsMatch(value);
            IsCredentialsFilled = (isEmailFilled && isPasswordFilled);
        }
    }
}

```



```
        if (!isEmailFilled)
        {
            throw new Exception
                ("L'adresse email fournie n'est pas valide.");
        }
        else
        {
            this.email = value;
        }
    }
}

public string Password
{
    get { return this.password; }
    set
    {
        isPasswordFilled = (value.Length > 3);
        IsCredentialsFilled = (isEmailFilled && isPasswordFilled);
        if (!isPasswordFilled)
        {
            throw new Exception
                ("Le mot de passe contient au moins 4 caractères.");
        }
        else
        {
            this.password = value;
        }
    }
}

public bool UseProxy
{
    get
    {
        return this.useProxy;
    }
    set
    {
        this.useProxy = value;
    }
}

public string ProxyAddress
{
    get
    {
        return this.proxyAddress;
    }
    set
    {
        if (this.useProxy == true)
        {
            this.proxyAddress = value;
        }
        else
        {
            this.proxyAddress = String.Empty;
        }
    }
}
}
```

Du côté de la classe `LoginBox`, il nous suffit de tester la valeur de la propriété `IsCredentialsFilled`. Dans un premier temps, ce test peut être réalisé lorsque l'utilisateur clique sur le bouton de connexion et après rafraîchissement des liaisons de données :

```
void connectionButton_Click(object sender, RoutedEventArgs e)
{
    BindingExpression be1 = emailTxt.GetBindingExpression
        (TextBox.TextProperty);
    be1.UpdateSource();
    BindingExpression be2 = passwordTxt.GetBindingExpression
        (PasswordBox.PasswordProperty);
    be2.UpdateSource();

    if (currentUserCredentials.IsCredentialsFilled)
    {
        //On fait quelque chose
        //si les identifiants sont correctement saisis
    }
}
```

Concrètement, le rafraîchissement des liaisons déclenche la mise à jour des propriétés liées `Password` et `Email` de l'instance `currentUserCredentials`. Celles-ci réaffectent la propriété `IsCredentialsFilled` au sein de leur accesseur `Set`. Ce mécanisme est intéressant, mais nous pouvons encore le perfectionner.

Par exemple, il est possible de lier la propriété `IsEnabled` du bouton de connexion à la propriété `IsCredentialsFilled` de notre classe `UserCredentials`. L'utilisateur ne sera pas tenté de cliquer sur le bouton s'il est désactivé lorsque le format des identifiants est incorrect. Le bouton de connexion n'a plus besoin de rafraîchir les liaisons des champs de saisie. De plus, il ne doit plus gérer les erreurs d'identifiant lors de l'écoute de l'événement `Click` car ceux-ci seront testés en amont :

```
void connectionButton_Click(object sender, RoutedEventArgs e)
{
    //Si le clic est possible, c'est que les identifiants sont
    //correctement renseignés. Il n'est donc plus nécessaire de les tester
}
```

Il est nécessaire de modifier `UserCredentials`. Jusqu'à maintenant, nous avons évoqué et utilisé la liaison de données à de nombreuses reprises sans réellement nous pencher sur les mécanismes internes fournissant cette fonctionnalité. Les propriétés d'objets n'héritant pas de `DependencyObject` ne possèdent pas cette faculté par défaut, il vous faudra l'implémenter.

Deux méthodologies sont possibles. La première consiste à utiliser des propriétés de dépendance (`DependencyProperty`) qui contiennent par défaut le mécanisme de liaison. La seconde est d'implémenter l'interface `INotifyPropertyChanged`. Dans le premier cas, seules les classes héritant de `DependencyObject` peuvent contenir ce type de propriété. Ce n'est pas une solution adéquate car elle concerne avant tout les objets graphiques. Nous opterons pour la seconde méthode car implémenter l'interface est plus simple, et donc plus logique, dans le cas d'objets métiers ou de classes non graphiques. Cette interface est avant tout constituée d'un événement de type `PropertyChangedEventHandler`. Voici son implémentation dans la classe `UserCredentials` :

```
public class UserCredentials : INotifyPropertyChanged
{
```

```
#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged ;

#endregion
```

Plusieurs propriétés de la classe `UserCredentials` peuvent désormais utiliser le moteur de liaisons de manière transparente. Pour cela, il suffit de diffuser l'événement lorsque la propriété est modifiée. La bonne pratique consiste à ajouter une méthode gérant la diffusion de l'événement :

```
public void NotifyPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Il nous faut maintenant diffuser l'événement dans l'accesseur `set` de la propriété `IsCredentialsFilled` *via* l'appel de la méthode `NotifyPropertyChanged` :

```
public bool IsCredentialsFilled
{
    get { return isCredentialFilled; }
    set
    {
        isCredentialFilled = value;
        NotifyPropertyChanged("IsCredentialsFilled");
    }
}
```

La propriété `IsCredentialsFilled` peut désormais être liée à n'importe quelle autre *via* l'utilisation d'une liaison standard. Dans notre cas, nous pouvons lier la propriété `IsEnabled` de notre bouton de connexion à `IsCredentialsFilled`. De cette manière, notre bouton sera actif uniquement lorsque les identifiants seront correctement renseignés. Dans le constructeur de la classe du contrôle utilisateur `LoginBox`, nous ajoutons une liaison reliant ces deux propriétés :

```
public LoginBox()
{
    ...
    Binding b = new Binding();
    b.Path = new PropertyPath("IsCredentialsFilled");
    b.Source = currentUserCredentials;
    b.Mode = BindingMode.OneWay;
    connectionButton.SetBinding(Button.IsEnabledProperty, b);
}
```

Ce n'est pas suffisant. Nous avons paramétré les liaisons des champs de saisie en mode explicite (*via* l'attribut `UpdateSourceTrigger=Explicit`). Nous pourrions revenir sur nos pas et les définir en mode automatique. Toutefois cela ne résoudrait pas notre problématique car elles ne seraient rafraîchies que lorsque les champs perdraient le focus. Contrairement à WPF, la propriété `UpdateSourceTrigger` n'accepte pas la valeur `PropertyChanged`. Nous devons donc rafraîchir ces liaisons manuellement en écoutant deux événements correspondants à la modification temps réel du mot de passe et de l'identifiant :

```
public LoginBox()
{
```

```
//...

passwordTxt.PasswordChanged += new RoutedEventHandler
    (passwordTxt_PasswordChanged);

emailTxt.TextChanged += new TextChangedEventHandler
    (emailTxt_TextChanged);
}

void emailTxt_TextChanged(object sender, TextChangedEventArgs e)
{
    BindingExpression be = emailTxt.GetBindingExpression
        (TextBox.TextProperty);
    be.UpdateSource();
}

void passwordTxt_PasswordChanged(object sender, RoutedEventArgs e)
{
    BindingExpression be = passwordTxt.GetBindingExpression
        (PasswordBox.PasswordProperty);
    be.UpdateSource();
}
```

Vous pouvez tester l'application. Le mieux est de la compiler dans Visual Studio, sans le débogueur *via* le raccourci Ctrl+F5. De cette manière, vous n'aurez pas d'arrêt lors de l'exécution dû aux exceptions levées par les liaisons.

Nous avons abordé divers mécanismes permettant d'indiquer à l'utilisateur une erreur de saisie ou lui évitant de cliquer inutilement sur le bouton de connexion. Ce projet est un cas d'école et permet d'aborder diverses techniques. Toutefois en termes d'ergonomie, choisissez l'une de ces méthodologies, mais pas toutes en même temps : vous risqueriez de perturber ou de noyer votre utilisateur avec trop d'informations visuelles à traiter. L'idéal est de réserver l'état *Invalid* lorsque les identifiants sont inconnus en base de données et qu'ils ont le bon format. Vous pourriez ajouter un message indiquant une erreur d'identifiants dans cet état. Vous pouvez commenter ou supprimer le code gérant la souscription à l'événement *BindingValidationError*.

INFO

Nous pourrions penser que cette gestion des erreurs est assez compliquée puisqu'il est possible d'utiliser les événements directement et de tester les valeurs en leur sein. Vous pourriez ainsi afficher un visuel spécifique lors des erreurs de formulaire. Bien que cette solution soit possible, vous vous priveriez non seulement de la gestion des erreurs levées par les liaisons de données mais également de leur gestion visuelle associée nativement au sein des modèles de composants. Ces mécanismes permettent au designer de s'intégrer dans le flux de production sans avoir à élaborer de stratégies avec le développeur.

Le projet est dans l'archive *LecteurMultimedia_LiaisonNotification.zip* du *chap12* des exemples du livre.

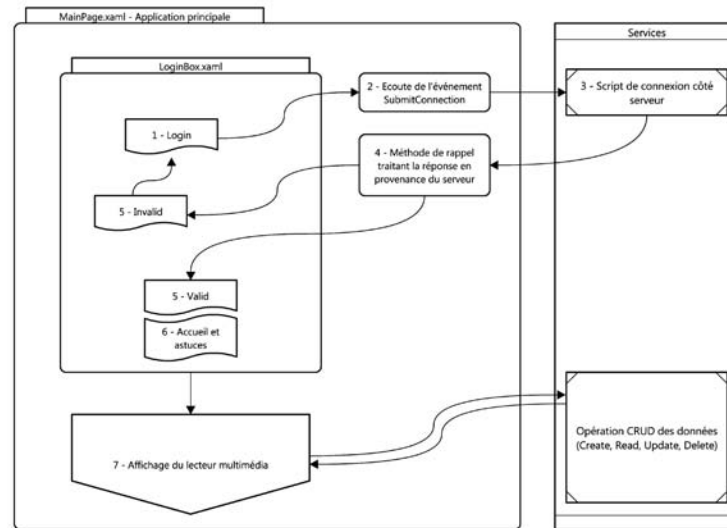
12.2.5 Événements personnalisés

Nous avons encore un peu de travail à accomplir pour finaliser notre contrôle utilisateur. Nous devons maintenant concevoir le processus de connexion à l'application et piloter la lecture des transitions de manière pertinente. Lorsque l'utilisateur saisit ses identifiants et demande une connexion,

la requête est traitée côté serveur qui valide ou invalide la connexion au service. Comme nous n'avons pas encore abordé l'échange de données client-serveur, nous allons simuler une réponse du serveur. Si la connexion est valide, l'écran d'accueil contenant les astuces devient disponible, dans le cas contraire, nous affichons les états Invalid et Login (voir Figure 12.11).

Figure 12.11

Processus de connexion.



Comme vous le constatez, il serait bienvenu que notre UserControl `LoginBox` diffuse un événement de soumission lors de la demande de connexion. De cette manière, une méthode de l'application principale jouera le rôle d'écouteur et sera déclenchée dès la diffusion de ce dernier. Cette méthode a pour objectif d'appeler un service distant sur le serveur dont la réponse est attendue par l'application (via l'utilisation d'une méthode de rappel). À réception de celle-ci, l'application principale pilotera les états visuels du contrôle utilisateur afin d'afficher l'écran d'accueil, ou de rediriger l'utilisateur vers l'état de connexion Invalid dans le cas où les identifiants fournis sont incorrects.

Prenez le projet *LecteurMultimedia_LiaisonNotification.zip* du *chap12* des exemples du livre. La première étape consiste à créer un événement personnalisé. Vous pouvez y parvenir de différentes manières. La première méthode n'est pas très standard : elle consiste à créer une délégation décrivant le contrat de diffusion de l'événement. Il suffit ensuite de créer un événement personnalisé membre de la classe `LoginBox` du type correspondant à celui défini par la délégation :

```
public delegate void LeDelegate (object param1, object nParam);

public class LoginBox : UserControl
{
    public event LeDelegate UnEvenement;

    //une méthode déclenchée commençant par On (Lorsque)
    private void OnQuelqueChoseArrive ()
    {
        if (UnEvenement != null)
        {
```

```

        //on déclenche l'événement si un écouteur y a souscrit
        UnEvenement( UneInstanceDObjet, UneAutreInstanceDObjet ) ;
    }
}
...
}

```

Cette méthode ne respecte pas vraiment les bonnes pratiques. Par convention la signature de l'événement prend toujours deux arguments. Le diffuseur de l'événement est le premier paramètre. Dans le cas présent, il s'agira de `this` correspondant à l'instance de la `LoginBox` diffusant l'événement. Dans le but de faciliter la maintenance, nous partons du principe que l'événement pourrait être diffusé par différents types d'objets. Celui-ci est typé `object` afin de garantir ce principe. En second paramètre, on passe l'objet événementiel souvent hérité de la classe `EventArgs`. Il permet d'envoyer des informations pertinentes concernant l'événement lui-même. C'est exactement ce qui arrive lorsque vous écoutez les événements `MouseLeftButtonChanged` d'un bouton, ou `ValueChanged` diffusé entre autres par les instances de `Slider`. Pour respecter ces conventions, il n'est pas nécessaire de créer une délégation spécifique. Il suffit de créer un événement générique de type `EventHandler` :

```

public class LoginBox : UserControl
{
    public event
        EventHandler<SubmittedConnectionEventArgs> SubmittedConnection;
    ...
}

```

Un type générique est une classe dont le nom est toujours suivi de balises entourant le type supporté par l'instance. Lorsque vous procédez de cette manière, il n'y a pas de contrat de délégation à définir, le diffuseur est l'instance en cours et l'objet événementiel correspond à la classe renseignée comme type géré (entre balise inférieur et supérieur). Pour des raisons de simplicité et de performance, l'objet événementiel hérite de la classe `EventArgs`. L'héritage permet d'ajouter des méthodes, des propriétés ou tout type de données utiles lors du déclenchement de la méthode d'écoute. Dans Visual Studio, ajoutez une classe dans le répertoire `classes` et nommez-la `SubmittedConnectionEventArgs`. Voici le code logique décrivant l'objet événementiel :

```

...
using LecteurMultiMedia.classes;

namespace LecteurMultiMedia.classes
{
    public class SubmittedConnectionEventArgs : EventArgs
    {
        public UserCredentials Credentials { get; set; }
    }
}

```

Le mieux est encore de déclarer directement une instance de type `UserCredentials` en tant que membre de la classe `SubmittedConnectionEventArgs`. De cette manière, nous récupérerons directement toutes les informations renseignées depuis le formulaire *via* la liaison à deux voies. Nous stockons le mot de passe, le login, l'éventuelle URL du proxy saisie par l'utilisateur. Il devient trivial de les récupérer du côté de l'application au sein de `MainPage.xaml.cs`, puis de les envoyer

à un service distant de connexion disponible côté serveur. Voici le code final de `LoginBox`. Nous déclenchons l'essai de connexion lorsque l'utilisateur relâchera le bouton de connexion. Comme vu précédemment, celui-ci est actif et cliquable dès que le formulaire est correctement renseigné :

```
void connectionButton_Click(object sender, RoutedEventArgs e)
{
    if (SubmittedConnection != null)
    {
        SubmittedConnectionEventArgs sce = new SubmittedConnectionEventArgs();
        sce.Credentials = currentUserCredentials;
        SubmittedConnection(this, sce);
    }
}
```

Il nous suffit d'écouter l'événement, puis de récupérer les identifiants de connexion afin de décider d'une éventuelle action à entreprendre :

```
public partial class MainPage : UserControl
{
    UserCredentials aknowledgeUserCredentials = new UserCredentials();

    public MainPage()
    {
        InitializeComponent();

        loginBox.SubmittedConnection += new EventHandler
            <SubmittedConnectionEventArgs>(loginBox_SubmittedConnection);
    }

    void loginBox_SubmittedConnection(object sender,
        SubmittedConnectionEventArgs e)
    {
        string email = e.Credentials.Email;
        string pass = e.Credentials.Password;
        string proxy = e.Credentials.ProxyAdress;

        if (email == "eric@tweened.org" && pass == "eric")
        {
            aknowledgeUserCredentials = e.Credentials;
            VisualStateManager.GoToState(loginBox, "Tips", true);
            //supprime le cadre rouge en cas d'identifiants erronés
            //lors d'une précédente tentative
            VisualStateManager.GoToState(loginBox, "Valid", true);
        }
        else
        {
            VisualStateManager.GoToState(loginBox, "Invalid", true);
        }
    }
}
```

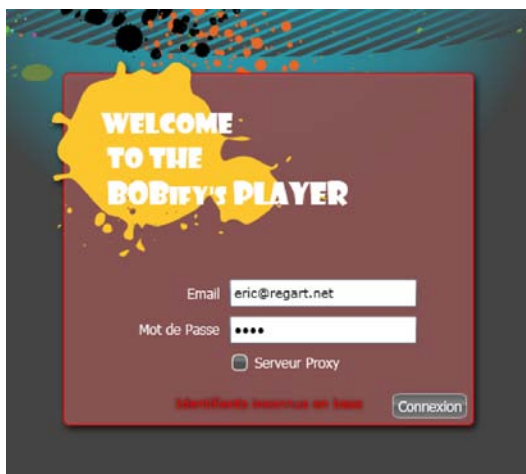
ATTENTION

Nous simulons un appel serveur, il est évident que toute gestion des identifiants en dur dans le code C# présente un risque de sécurité important puisqu'un fichier compilé peut-être décompilé par des outils disponibles pour le grand public. L'exemple suivant n'est en aucun cas à suivre car les identifiants doivent être stockés en base de données distante et gérés dynamiquement.

Lorsque l'événement SubmittedConnection est diffusé par l'instance loginBox, nous vérifions, dans la méthode d'écoute, si le couple d'identifiants e-mail et mot de passe correspond à un utilisateur existant en base. Si oui, nous affichons l'état Tips de la mire de connexion. Dans le cas contraire, nous affichons l'état Invalid décrivant une erreur d'identifiants (voir Figure 12.12).

Figure 12.12

Une erreur d'identifiants affiche l'état Invalid.



Nous avons pratiquement terminé la création de notre contrôle utilisateur. Il nous reste à gérer deux comportements : d'une part, la transition vers l'état AdvanceLogin permettant de renseigner un éventuel serveur proxy, d'autre part la disparition du composant LoginBox dans l'application principale afin de laisser apparaître la liste des médias. La première étape est assez simple à réaliser *via* des comportements, ou du code logique, associés au ToggleButton correspondant. Dans ce cas, nommez le ToggleButton en optionalProxyToggleButton. Le code logique créé est assez simple :

```
public LoginBox ()
{
    ...
    optionalProxyToggleButton.Checked += new RoutedEventHandler
        (optionalProxyToggleButton_Checked);

    optionalProxyToggleButton.Unchecked += new RoutedEventHandler
        (optionalProxyToggleButton_Unchecked);
}

void optionalProxyToggleButton_Unchecked(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToState(this, "Login", true); ;
}

void optionalProxyToggleButton_Checked(object sender, RoutedEventArgs e)
```



```

{
    VisualStateManager.GoToState(this, "AdvanceLogin", true);
}

```

Concernant la disparition de la fenêtre de connexion, l'idéal est que celle-ci diffuse un événement lorsque le bouton OK de la fenêtre des astuces est cliqué :

```

public event EventHandler<RoutedEventArgs> RemoveRequest;

...

public LoginBox ()
{
    ...

    removeLoginBoxButton.Click += new RoutedEventArgs
        (removeLoginBoxButton_Click);
}

void removeLoginBoxButton_Click(object sender, RoutedEventArgs e)
{
    if (RemoveRequest!=null)
    {
        RemoveRequest(this,e);
    }
}

```

De cette manière, l'application gère la suppression de l'instance de manière optimale.

Nous pourrions décider que ce comportement appartient à la mire de login mais cela poserait certaines problématiques ; l'une d'elles étant que la fenêtre de connexion peut-être contenue par divers types de contrôles qu'il faudra tester pour la supprimer de la liste d'affichage. Les propriétés Child, Content et Children peuvent chacune contenir au moins une instance de UIElement. Notre contrôle utilisateur ne peut pas savoir à l'avance laquelle de ces propriétés contient sa référence, il doit donc tester chaque type de conteneur. De ce point de vue, diffuser un événement et déléguer la gestion de sa suppression à l'application est à la fois plus simple et plus souple en termes de conception. Le code complet de notre application est succinct :

```

public partial class MainPage : UserControl
{
    UserCredentials aknowledgeUserCredentials = new UserCredentials();

    public MainPage()
    {
        InitializeComponent();

        loginBox.SubmittedConnection += new EventHandler
            <SubmittedConnectionEventArgs>(loginBox_SubmittedConnection);

        loginBox.RemoveRequest += new EventHandler<RoutedEventArgs>
            (loginBox_RemoveRequest);
    }

    void loginBox_RemoveRequest(object sender, RoutedEventArgs e)
    {
        loginBox.SubmittedConnection -= loginBox_SubmittedConnection;
        loginBox.RemoveRequest -= loginBox_RemoveRequest;
        LayoutRoot.Children.Remove(loginBox);
    }
}

```

```
        loginBox = null;
        VisualStateManager.GoToState(this, "Liste", true);
    }

    void loginBox_SubmittedConnection(object sender,
                                     SubmittedConnectionEventArgs e)
    {
        string email = e.Credentials.Email;

        string pass = e.Credentials.Password;

        string proxy = e.Credentials.ProxyAddress;

        if (email == "eric@tweened.org" && pass == "eric")
        {
            acknowledgeUserCredentials = e.Credentials;
            VisualStateManager.GoToState(loginBox, "Tips", true);
            VisualStateManager.GoToState(loginBox, "Valid", true);
        }
        else
        {
            VisualStateManager.GoToState(loginBox, "Invalid", true);
        }
    }
}
```

Vous constatez que nous détruisons par principe tous les écouteurs propres aux événements diffusés par la fenêtre de connexion, puis nous supprimons l'instance de `LoginBox` de la liste d'affichage et passons sa référence à `null`. De cette manière, le passage du ramasse-miettes (*garbage collector*) est facilité. Il libère aisément les ressources utilisées par l'instance de `LoginBox`.

Les contrôles utilisateur possèdent plusieurs qualités importantes en termes de conception. Ils sont faciles à concevoir, à utiliser en production, à partager et à faire évoluer au cours du temps. Il est, par exemple, assez simple de modifier la fenêtre de connexion que nous avons conçue. Remplacer le champ de saisie de l'e-mail ou du proxy par d'autres ne prend pas plus de 5 minutes. Ces contrôles possèdent toutefois quelques désavantages mis en valeur lors de la conception de contrôles personnalisables. Le projet finalisé est dans l'archive *LecteurMultimedia_Evenement.zip* du dossier *chap12* des exemples de ce livre.

12.3 Contrôles personnalisables

Dans l'exemple précédent, nous avons démontré la souplesse de conception apportée par les contrôles utilisateur. Toutefois, leurs codes logique et déclaratif sont ouverts à la modification sans garde-fou. Le code logique étant accessible et fortement couplé au code déclaratif, une mauvaise manipulation du designer ou de l'intégrateur peut très facilement conduire à des dysfonctionnements. De plus, les contrôles utilisateur ne possèdent pas réellement de style et de modèle proprement dit, alors que les contrôles personnalisables apportent beaucoup de souplesse en la matière.

Les contrôles personnalisables répondent à ces différentes problématiques de manière élégante mais demandent beaucoup plus de réflexion. Lorsque vous créez des contrôles de ce type, vous permettez au designer de concevoir des styles et des modèles pour ces contrôles. Le code logique est inaccessible et seul l'arbre visuel et logique peut être modifié. Cette architecture est un gage de stabilité et de robustesse car le fond et la forme sont complètement séparés. Toutefois, les

contrôles personnalisables s'inscrivent généralement dans des problématiques d'utilisation plus vastes que la production en cours ne le suggère. Ils sont faits pour être réutilisés, mais il est difficile de savoir ce qui peut ou va être réutilisé dans vos futurs projets. Attention donc à ne pas perdre de vue le projet lui-même en vous lançant dans une conception trop éparpillée. Préparez un cahier des charges du composant et ne le dépassez qu'en cas d'extrême nécessité.

Dans cette section, nous allons concevoir deux composants qui permettront de sélectionner une couleur dans un nuancier en mode HSL – *hue, saturation, lightness* (teinte, saturation, luminosité) – et d'affecter cette dernière à d'autres contrôles.

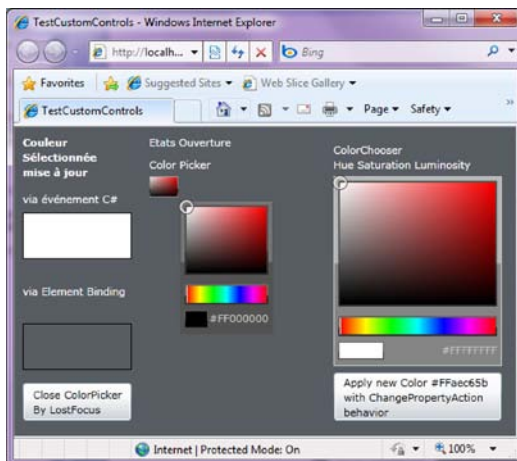
12.3.1 *ColorChooser* et *ColorPicker*

Les contrôles *ColorChooser* et *ColorPicker* ont des rôles très différents et n'existent ni dans la bibliothèque Silverlight fournie par défaut, ni dans le Silverlight Toolkit. On peut facilement imaginer leur utilité lorsque l'utilisateur doit paramétrer son interface graphique ou tout type de contenu visuel. Comme ils n'existent pas, vous pourriez être amené à en concevoir par nécessité.

Les objectifs de ces composants sont complémentaires. *ColorChooser* permet à l'utilisateur de choisir une couleur au sein d'un nuancier, il est possible de récupérer la valeur hexadécimale de cette couleur ou une instance de *SolidColorBrush*. La couleur sélectionnée peut ensuite être affectée à l'une des propriétés de remplissage de tout autre contrôle durant l'exécution. *ColorPicker* permet quant à lui de faire apparaître le *ColorChooser*, qui est plus discret et occupe moins de place dans l'interface. Le *ColorPicker* récupère un certain nombre de propriétés du *ColorChooser*. Nous allons concevoir le *ColorChooser* en premier. Il possède la propriété publique *SelectedColor* ainsi que l'événement *ColorChanged*. Ce dernier est diffusé lorsque la couleur est modifiée au sein du nuancier. *ColorPicker* expose exactement les mêmes membres et en ajoute quelques-uns. Il contient notamment la propriété *IsOpen* qui permet d'afficher ou non le nuancier. Cette propriété est associée à deux événements *FillBoxOpened* et *FillBoxClosed* qui sont diffusés lorsque le *ColorChooser* apparaît ou disparaît. De plus, *ColorPicker* possède également la capacité d'associer un style personnalisé à *ColorChooser* qu'il instancie à la compilation. Cette opération est réalisée grâce à sa propriété publique *ColorChooserStyle*. La Figure 12.13 expose nos composants finalisés mis en situation.

Figure 12.13

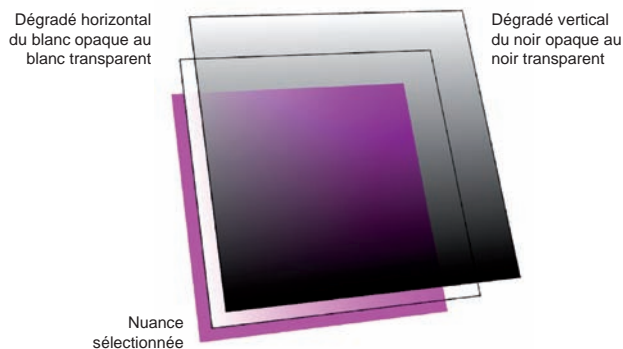
Les contrôles *ColorPicker* et *ColorChooser* finalisés.



Concernant la création du dégradé principal, il peut être généré par code *via* l'écriture directe de pixels au sein d'une instance de type `WritableBitmap`. Cette technique n'est toutefois pas très optimisée en termes de performance car il faut réécrire, à l'aide d'un algorithme, une nouvelle image bitmap lorsque la nuance est modifiée par l'utilisateur ou lorsque le contrôle est redimensionné. Une surface de dégradé de 200 pixels par 200 signifie que 40 000 pixels doivent ainsi être redéfinis à chaque fois. Nous prenons donc le parti de créer deux dégradés en dur superposés à un aplat de couleur symbolisant la nuance sélectionnée (voir Figure 12.14).

Figure 12.14

Principe de conception du dégradé central.



Cette technique est très utilisée sur le Web depuis de nombreuses années. Toutefois nous n'utiliserons pas de bitmap de taille fixe, mais des instances de pinceaux de dégradé (`LinearGradientBrush`) affichées par le moteur vectoriel Silverlight. De cette manière, notre composant sera redimensionnable.

12.3.2 Créer des contrôles

Créez un nouveau projet de type Silverlight Application + Website et nommez-le `TestCustomControls`. Le mieux est d'ouvrir la solution dans Visual Studio si ce n'est pas déjà le cas. Ajoutez un nouveau projet Silverlight de type Class Library à la solution en cours et nommez-le `TweenedControls`.

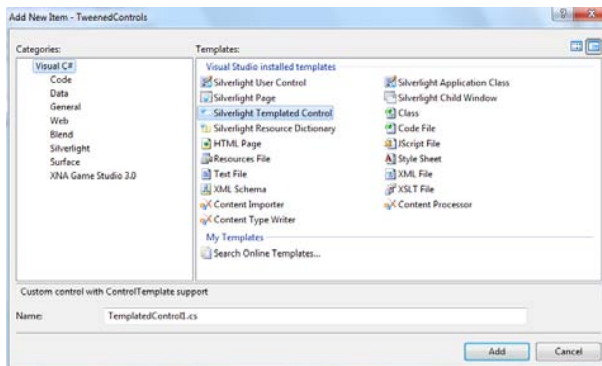
INFO

Si vous codez avec Expression Blend, ajoutez un projet de type Silverlight Control Library. Il n'est pas conseillé de concevoir le contrôle uniquement dans Expression Blend. La complétion et l'IntelSense ne sont pas aussi performants que sous Visual Studio. Expression Blend est beaucoup plus efficace pour la conception visuelle de votre composant. L'idéal est de travailler avec ces deux logiciels tout au long du processus de conception.

Supprimez le ou les fichiers générés par défaut à la racine du nouveau projet. Cliquez-droit sur le projet et ajoutez un nouvel élément de type "Silverlight Templated Control" (voir Figure 12.15).

Figure 12.15

La fenêtre de création du composant personnalisable.



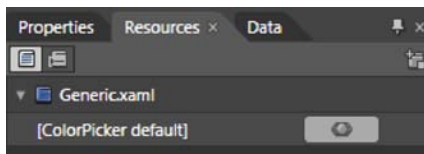
Nommez ce dernier `ColorPicker`, puis validez. Compilez directement l'application pour qu'Expression Blend puisse rafraîchir son interface et prendre en compte le composant généré. Visual Studio a généré un fichier de code logique C# nommé `ColorPicker.cs` ainsi qu'un fichier nommé `generic.xaml` contenant du code déclaratif XAML. Celui-ci contient le style et le modèle que le composant `ColorPicker` utilise par défaut. C'est un dictionnaire de ressources spécifique associé aux contrôles personnalisables conçus dans notre projet. Il est toujours placé dans le répertoire `Themes` des projets. Du côté C#, la classe `ColorPicker` hérite de `Control`, ce qui en fait un contrôle personnalisable. Elle fait référence au style appliqué par défaut dans son constructeur via la propriété `DefaultStyleKey` :

```
namespace TweenedControls
{
    public class ColorPicker : Control
    {
        public ColorPicker()
        {
            this.DefaultStyleKey = typeof(ColorPicker);
        }
    }
}
```

Retournez sous Blend et ouvrez le fichier `Generic.xaml` présent dans le répertoire `Themes`. Le panneau `Resources` liste les styles contenus dans le dictionnaire. Cliquez sur le style (voir Figure 12.16).

Figure 12.16

Accès au style par défaut du `ColorPicker` contenu dans `Generic.xaml`.



Dans l'arbre visuel, cliquez-droit sur le style afin de modifier le modèle. Blend a instancié un `Border`, remplacez-le par une grille nommée `Root`. Passez en mode de design mixte : cela vous permet d'afficher le XAML :

```
<Style TargetType="local:ColorPicker">
    <Setter Property="Template">
        <Setter.Value>
```

```

        <ControlTemplate TargetType="local:ColorPicker">
            <Grid x:Name="LayoutRoot" />
        </ControlTemplate>
    </Setter.Value>
</Setter>
</Style>

```

Le contenu de ce contrôle peut être apparenté à celui d'un simple bouton. Il ne possède en effet aucun élément logique, une seule instance de `Rectangle` suffit à son fonctionnement. Il est toutefois nécessaire que l'instance de `Rectangle` – ou la grille principale – possède un remplissage afin de définir une surface cliquable. Vous pouvez bien sûr concevoir un arbre logique plus élaboré. Répétez les étapes précédentes afin de générer un contrôle de type `ColorChooser`. Si le style n'est pas créé dans le fichier `Generic.xaml`, n'hésitez pas à ajouter la balise `Style` comme ci-dessous :

```

<Style TargetType="local:ColorChooser">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="local:ColorChooser">
                <Border x:Name="LayoutRoot">
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>

```

Laissez le composant racine en tant que `Border` et nommez-le `LayoutRoot`. Son objectif est de contenir les éléments de l'arbre visuel tout en étant un élément de design. Le dictionnaire de ressources `Generic.xaml` contient à présent les styles du `ColorPicker` et du `ColorChooser`. Nous allons maintenant aborder leur conception à proprement parler.

12.3.3 Le contrat de modèle

Un contrat de modèle est un ensemble d'attributs propre au contrôle. Il fournit des indices sur ses capacités ainsi qu'un certain nombre de contraintes à son fonctionnement. Un contrat de modèle est très utile aux graphistes qui souhaitent personnaliser un contrôle. Concrètement ces attributs définissent les états visuels ainsi que les parties logiques du contrôle qui sont liées à son fonctionnement.

12.3.3.1 Gérer les états visuels

`ColorPicker` contient des attributs d'états visuels ainsi qu'un autre rarement utilisé nommé `StyleTypedProperty`. `ColorPicker` permet d'appliquer un style au contrôle `ColorChooser` qu'il contient directement *via* l'interface de `Blend`. Le contrat de modèle est toujours à placer directement au-dessus de la classe définissant le contrôle personnalisable. Le voici pour `ColorPicker` :

```

namespace TweenedControls
{
    #region Contrat du modèle
    [TemplateVisualState(Name = "Normal", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "MouseOver", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "Pressed", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "Disabled", GroupName = "CommonStates")]

    [TemplateVisualState(Name = "Focused", GroupName = "FocusStates")]

```

```

[TemplateVisualState(Name = "Unfocused", GroupName = "FocusStates")]

[TemplateVisualState(Name = "Opened", GroupName="ColorChooserStates")]
[TemplateVisualState(Name = "Closed", GroupName="ColorChooserStates")]

[StyleTypedProperty(Property = "ColorChooserStyle", StyleTargetType =
                                     typeof(ColorChooser))]
#endregion
public class ColorPicker : Control
{

```

Nous reviendrons sur l'utilisation du dernier attribut ultérieurement. Compilez le projet. Ensuite, dans le modèle du `ColorPicker`, ouvrez le panneau `States`. Il affiche désormais tous les états visuels que le contrôle gère par défaut. Le designer prend connaissance de ces derniers par une simple lecture du panneau. Arrangez-vous pour définir des noms explicites pour chaque état.

INFO

Vous devrez parfois fermer le fichier `Generic.xaml` puis ouvrir à nouveau le modèle pour que les attributs soient correctement pris en compte par l'interface d'Expression Blend. Cela est particulièrement vrai pour les attributs définissant des parties logiques de contrôles. N'hésitez pas à recharger entièrement le dictionnaire de ressources et à recompiler l'application sous Blend si nécessaire *via* le raccourci `Ctrl+B`.

L'idée est de naviguer entre les différents états visuels en ajoutant un peu de code logique à nos composants. Plutôt que d'écouter directement les événements, il est préférable de surcharger leurs méthodes afin de les gérer en amont. L'événement `IsEnabled` est le seul qui ne peut être surchargé, il faut donc l'écouter pour accéder aux états correspondants :

```

public ColorPicker()
{
    this.DefaultStyleKey = typeof(ColorPicker);

    this.IsEnabledChanged += new DependencyPropertyChangedEventHandler
        (this.ColorPicker_IsEnabledChanged);
}

#region handler and override

protected override void OnLostFocus(RoutedEventArgs e)
{
    base.OnLostFocus(e);
    VisualStateManager.GoToState(this, "Unfocused", true);
}

protected override void OnGotFocus(RoutedEventArgs e)
{
    base.OnGotFocus(e);
    VisualStateManager.GoToState(this, "Focused", true);
}

protected override void OnMouseLeave(MouseEventArgs e)
{
    base.OnMouseLeave(e);
    VisualStateManager.GoToState(this, "Normal", true);
}

```

```

protected override void OnMouseEnter(MouseEventArgs e)
{
    base.OnMouseEnter(e);
    VisualStateManager.GoToState(this, "MouseOver", true);
}

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
{
    base.OnMouseLeftButtonDown(e);
    VisualStateManager.GoToState(this, "Pressed", true);
}

private void ColorPicker_IsEnabledChanged(object sender,
                                           DependencyPropertyChangedEventArgs e)
{
    {
        if (this.IsEnabled)
        {
            VisualStateManager.GoToState(this, "Normal", true);
        }
        else
        {
            VisualStateManager.GoToState(this, "Disabled", true);
        }
    }
}

```

Le code logique ci-dessus est assez simple et sera similaire pour les deux composants. L'une des différences concerne les états `Opened` et `Closed` liés tous deux à la propriété `IsOpen` du `ColorPicker`. Nous aborderons la conception de cette propriété ultérieurement.

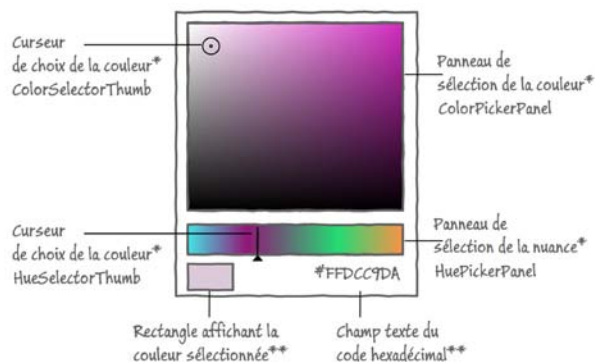
12.3.3.2 Associer des parties logiques

Ouvrez maintenant le fichier de code logique `ColorChooser.cs`. A *contrario* du `ColorPicker`, `ColorChooser` contient des parties logiques. Afin de les identifier facilement, le mieux est toujours de créer un croquis de votre futur contrôle (voir Figure 12.17).

Figure 12.17

Les parties logiques du contrôle `ColorChooser`.

Croquis du ColorChooser



* parties logiques du contrôle obligatoires à son fonctionnement

** parties du contrôle facultatives

Pour que `ColorChooser` fonctionne correctement, il faut qu'au moins quatre composants en fassent partie. Les deux premiers, `ColorPickerPanel` et `HuePickerPanel`, sont des panneaux (`Panel`) qui créent dynamiquement les dégradés. Les deux autres, `ColorSelectorThumb` et `HueSelectorThumb`, sont des instances de la classe `Thumb` qui ont pour avantage de faciliter les opérations de glisser-déposer. L'utilisateur pourra les déplacer à volonté afin de sélectionner la nuance ainsi que la couleur de son choix. Voici le contrat de modèle de `ColorChooser` :

```
//Espace de noms nécessaire à l'utilisation de la classe Thumb
using System.Windows.Controls.Primitives;

namespace TweenedControls
{
    #region Contrat du modèle ColorChooser
    [TemplateVisualState(Name = "Normal", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "MouseOver", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "Pressed", GroupName = "CommonStates")]
    [TemplateVisualState(Name = "Disabled", GroupName = "CommonStates")]

    [TemplateVisualState(Name = "Focused", GroupName = "FocusStates")]
    [TemplateVisualState(Name = "Unfocused", GroupName = "FocusStates")]

    [TemplatePart(Name = "ColorPickerPanel", Type = typeof(Panel))]
    [TemplatePart(Name = "HuePickerPanel", Type = typeof(Panel))]

    [TemplatePart(Name = "ColorSelectorThumb", Type = typeof(Thumb))]
    [TemplatePart(Name = "HueSelectorThumb", Type = typeof(Thumb))]
    #endregion

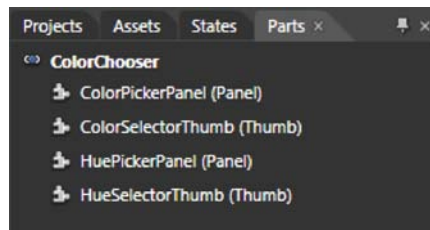
    public class ColorChooser : Control
    {
        ...
    }
}
```

L'idéal est de typer les parties de contrôle de la manière la plus large possible. Ce n'est pas réellement possible dans le cas présent puisque ce sont des fonctionnalités propres aux classes `Thumb` et `Panel` qui nous importent.

Compilez le projet dans l'interface d'Expression Blend. Le panneau `Parts` est mis à jour et affiche les parties logiques du contrôle qui sont nécessaires. Si ce n'est pas le cas, n'hésitez pas à fermer le fichier `Generic.xaml` et à le rouvrir. Les parties ne sont pas encore associées à d'éventuelles instances d'objet au sein de l'arbre visuel, une coche verte n'est donc pas visible à côté de chacune d'elles (voir Figure 12.18).

Figure 12.18

Le panneau `Parts` affichant les parties logiques du contrôle.



Pour associer une partie logique de contrôle à un élément présent dans l'arbre visuel, il est nécessaire de surcharger la méthode `OnApplyTemplate` héritée de la classe `Control`. Au sein de cette méthode, nous affecterons un élément de l'arbre à un membre de notre classe *via* l'appel de la

méthode `GetTemplateChild`. Il nous faut tout d'abord générer deux propriétés privées dont le but sera de gérer la création des dégradés grâce à leur accesseur `Set` :

```
//Bitmap du dégradé de nuances
private WriteableBitmap hueBitmap;

private Panel huePickerPanel = null;

private Panel HuePickerPanel
{
    get
    {
        return this.huePickerPanel;
    }
    set
    {
        this.huePickerPanel = value;
        if (this.huePickerPanel != null)
        {
            this.CreateHueGradientShape();
        }
    }
}

//Bitmap du dégradé des HSL (Hue-Saturation-Luminosity)
private WriteableBitmap colorBitmap;

private Panel colorPickerPanel = null;

private Panel ColorPickerPanel
{
    get
    {
        return this.colorPickerPanel;
    }
    set
    {
        this.colorPickerPanel = value;
        if (this.colorPickerPanel != null)
        {
            this.CreateLuminositySaturationGradients();
        }
    }
}
```

Chaque propriété est associée à une instance de `WriteableBitmap` qui a pour but de photographier le dégradé afin d'en conserver une image bitmap. Grâce à ce mécanisme, nous pouvons récupérer la couleur d'un des pixels selon la position des curseurs de sélection. Nous allons maintenant associer ces propriétés *via* la méthode `GetTemplateChild`. Voici le code C# correspondant :

```
public override void OnApplyTemplate()
{
    //on appelle la méthode de la super-classe Control dans un premier temps
    base.OnApplyTemplate();

    this.HuePickerPanel = GetTemplateChild("HuePickerPanel") as Panel;
    this.ColorPickerPanel = GetTemplateChild("ColorPickerPanel") as Panel;
}
```

Lorsque les propriétés privées sont affectées, elles appellent une méthode générant les dégradés. Voici le contenu de cette méthode nommée `CreateHueGradientShape` :

```
private void CreateHueGradientShape()
{
    Shape hueGradientShape = new Rectangle();

    LinearGradientBrush lgb = new LinearGradientBrush();
    lgb.StartPoint = new Point(0, 0.5);
    lgb.EndPoint = new Point(1, 0.5);
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0xFF, 0x00, 0x00), Offset = 0 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0xFF, 0xFF, 0x00), Offset = 0.1666666 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0x00, 0x00, 0xFF), Offset = 0.3333333 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0x00, 0xFF, 0xFF), Offset = 0.5 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0x00, 0x00, 0xFF), Offset = 0.6666666 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0xFF, 0x00, 0xFF), Offset = .83333333 });
    lgb.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0xFF, 0xFF, 0x00), Offset = 1 });
    hueGradientShape.Fill = lgb;
    huePickerPanel.Children.Add(hueGradientShape);
    Canvas.SetZIndex(hueGradientShape, -1);
}

// currentHueFillShape représente le rectangle situé sous les dégradés
private Shape currentHueFillShape;

private void CreateLuminositySaturationGradients()
{
    currentHueFillShape = new Rectangle();
    currentHueFillShape.Fill = new SolidColorBrush(Colors.Red);
    colorPickerPanel.Children.Add(currentHueFillShape);
    Canvas.SetZIndex(currentHueFillShape, -1);

    //dégradé transparent blanc
    Rectangle wr = new Rectangle();
    LinearGradientBrush wlg = new LinearGradientBrush();
    wlg.StartPoint = new Point(0, 0.5);
    wlg.EndPoint = new Point(1, 0.5);
    wlg.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0xFF, 0xFF, 0xFF), Offset = 0 });
    wlg.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0x00, 0xFF, 0xFF, 0xFF), Offset = .99 });
    wr.Fill = wlg;
    colorPickerPanel.Children.Add(wr);
    Canvas.SetZIndex(wr, -1);

    //dégradé transparent noir
    Rectangle br = new Rectangle();
    LinearGradientBrush blg = new LinearGradientBrush();
    blg.StartPoint = new Point(0.5, 1);
    blg.EndPoint = new Point(0.5, 0);
    blg.GradientStops.Add(new GradientStop() { Color = Color.
        FromArgb(0xFF, 0x00, 0x00, 0x00), Offset = 0 });
}
```

```

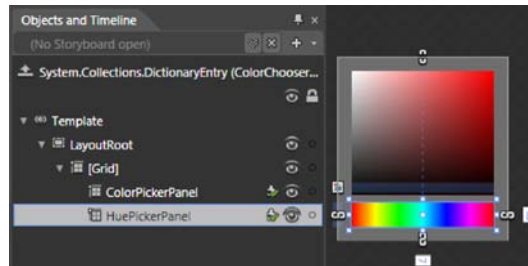
        blgb.GradientStops.Add(new GradientStop() { Color = Color.
            FromArgb(0xFF, 0x00, 0x00, 0x00), Offset = 0.01 });
        blgb.GradientStops.Add(new GradientStop() { Color = Color.
            FromArgb(0x00, 0x00, 0x00, 0x00), Offset = 1 });
        br.Fill = blgb;
        colorPickerPanel.Children.Add(br);
        Canvas.SetZIndex(br, -1);
    }

```

Compilez le projet dans Visual Studio, puis dans Blend afin de le recharger complètement. Dans le modèle du ColorChooser, créez une grille en contenant deux autres dans le Border racine. Agrandissez la zone de prévisualisation si nécessaire, placez les deux grilles l'une en dessous de l'autre et définissez-les en tant que partie de contrôle. Au moment où cette opération est réalisée, la méthode OnApplyTemplate est appelée, les grilles sont associées comme propriétés privées de la classe ColorChooser et les dégradés sont générés (voir Figure 12.19).

Figure 12.19

Résultat de
l'association de
parties logiques
sous Blend.



Comme les dégradés sont créés dynamiquement, ils n'apparaissent pas dans l'arbre visuel et logique. Nous devons recommencer le même travail avec les deux curseurs de sélection. Voici le code logique générant les propriétés privées associées :

```

private Thumb colorSelectorThumb=null;

private Thumb ColorSelectorThumb
{
    get { return this.colorSelectorThumb; }
    set
    {
        if (this.colorSelectorThumb != null)
        {
            this.colorSelectorThumb.DragDelta -=
                this.ColorSelectorThumb_DragDelta;
        }

        this.colorSelectorThumb = value;
        if (this.colorSelectorThumb != null)
        {
            this.colorSelectorThumb.DragDelta += new
                DragDeltaEventHandler(this.ColorSelectorThumb_DragDelta);
        }
    }
}

private int oriX = 0;
private int oriY = 0;
private void ColorSelectorThumb_DragDelta(object sender, DragDeltaEventArgs e)
{
    if (this.colorBitmap == null)

```

```

    {
        this.colorBitmap = new WriteableBitmap(this.colorPickerPanel, null);
        this.colorBitmap.Pixels[0] = (255 << 24) | (255 << 16) | (255 << 8) | 255;
    }

    this.oriX += (int)e.HorizontalChange;
    this.oriX = Math.Max(0, this.oriX);
    this.oriX = Math.Min(this.colorBitmap.PixelWidth - 1, this.oriX);

    this.oriY += (int)e.VerticalChange;
    this.oriY = Math.Max(0, this.oriY);
    this.oriY = Math.Min(this.colorBitmap.PixelHeight - 1, this.oriY);

    this.colorSelectorThumb.Margin =
        new Thickness(this.oriX - 8, this.oriY - 8, 0, 0);

    this.RefreshColorPreview(false);
}

private Thumb hueSelectorThumb = null;

private Thumb HueSelectorThumb
{
    get { return this.hueSelectorThumb; }
    set
    {
        {
            if (this.hueSelectorThumb != null)
            {
                this.hueSelectorThumb.DragDelta -=
                    this.hueSelectorThumb_DragDelta;
            }

            this.hueSelectorThumb = value;
            if (this.hueSelectorThumb != null)
            {
                this.hueSelectorThumb.DragDelta += new DragDeltaEventHandler(this.
                    hueSelectorThumb_DragDelta);
            }
        }
    }
}

private int oriHueX = 0;

private void hueSelectorThumb_DragDelta(object sender, DragDeltaEventArgs e)
{
    if (this.hueBitmap == null)
    {
        this.hueBitmap = new WriteableBitmap(this.huePickerPanel, null);
        this.hueBitmap.Pixels[0] =
            (255 << 24) | (255 << 16) | (0 << 8) | 0;
    }
    this.oriHueX += (int)e.HorizontalChange;
    this.oriHueX = Math.Max(0, this.oriHueX);
    this.oriHueX = Math.Min(this.hueBitmap.PixelWidth - 1, this.oriHueX);

    this.hueSelectorThumb.Margin =
        new Thickness(this.oriHueX - 10, 0, 0, 0);

    int pixelIndice = this.oriHueX;

    string hexaColor = Convert.ToString(this.hueBitmap.
        Pixels[pixelIndice], 16).ToUpper();

```

```

byte r = Convert.ToByte(hexaColor.Substring(2, 2), 16);
byte g = Convert.ToByte(hexaColor.Substring(4, 2), 16);
byte b = Convert.ToByte(hexaColor.Substring(6, 2), 16);

SolidColorBrush scb = new SolidColorBrush(Color.FromArgb(0xFF, r, g, b));

if (this.currentHueFillShape != null)
{
    this.currentHueFillShape.Fill = scb;
}
this.RefreshColorPreview(true);
}

```

Dans le code ci-dessus, plusieurs éléments sont à prendre en considération. D'un point de vue global, nous ajoutons deux propriétés privées, `ColorSelectorThumb` et `HueSelectorThumb`. Leur but est de garantir le fonctionnement des curseurs de sélection. Au sein de leur attribut `Set`, nous vérifions si l'instance du curseur n'est pas nulle. Si c'est le cas nous supprimons l'écoute de l'événement `DragDelta`. Puis nous écoutons cet événement diffusé par le nouvel élément défini comme partie logique. De cette manière, vous évitez de consommer des ressources systèmes inutiles. Le curseur de choix de la nuance subit un déplacement horizontal alors que celui du choix de la couleur peut être déplacé sur les axes `x` et `y`. Dans le cas où l'instance de chaque curseur est nulle, le `ColorChooser` ne fonctionnera pas. C'est tout à fait normal puisque ces propriétés font référence à des parties logiques du contrôle. Si elles sont nulles, c'est que le designer interactif ne les a pas assignées dans `Blend`.

ATTENTION

Vous remarquez que le déplacement des curseurs de sélection est réalisé *via* l'affectation de la propriété `Margin`. Cette solution n'est pas idéale car elle signifie que les conteneurs parents de chacun des instances de `Thumb` doivent être de type `Grid` ou posséder cette propriété. `Margin` n'est pas accessible au sein d'un `Canvas` par exemple. L'idéal serait d'utiliser une transformation relative. La bibliothèque `ProxyRenderTransform` que je mets à disposition sur le portail `Code Plex` permet de faciliter cette approche. Vous pouvez la télécharger à l'adresse : <http://proxyrd.codeplex.com/wikipage?title=Home&ProjectName=proxyrd>. Importez la bibliothèque en tant que `dll`, puis faites référence à l'espace de noms au sein de votre projet. Cette bibliothèque ajoute des méthodes d'extension aux instances de `UIElement`. La ligne de code suivante :

```
this.hueSelectorThumb.Margin = new Thickness(this.oriHueX - 10, 0, 0, 0);
```

et celles qui sont similaires peuvent être remplacées par le type d'expression suivante :

```
this.hueSelectorThumb.SetX(this.oriHueX);
```

Dans les deux méthodes gérant les déplacements, nous créons des images `bitmap` des dégradés, ce sont ces dernières qui vont être utilisées pour récupérer la valeur des pixels :

```
this.hueBitmap = new WriteableBitmap(this.huePickerPanel, null);
```

Le deuxième paramètre du constructeur, de type `Transform`, permet de prendre en compte une éventuelle transformation appliquée à l'objet. Le fait de passer la valeur `null` enregistre une image `bitmap` ne tenant pas compte des transformations appliquées à l'objet. Vous pouvez également

créer un agrandissement en passant une matrice de transformation ou une instance de `Render-Transform`.

INFO

Lorsque nous récupérons l'image bitmap de l'instance `HuePickerPanel`, nous remplaçons la valeur du premier pixel par du rouge. Cela permet d'obtenir un nuancier cohérent :

```
this.hueBitmap.Pixels[0] = (255 << 24) | (255 << 16) | (0 << 8) | 0;
```

Nous procédons de même avec la partie logique `ColorPickerPanel`, mais en affectant un blanc pur à l'image récupérée. Nous n'entrerons pas dans le détail des explications concernant les opérations sur les bits, il faut juste savoir que les expressions entre parenthèses symbolisent les quatre octets d'un pixel, à savoir l'alpha (transparence), le rouge, le vert et le bleu :

```
this.colorBitmap.Pixels[0] = (255 << 24) | (255 << 16) | (255 << 8) | 255
```

Cela peut paraître quelque peu brouillon, mais nous évite bien des complications. Les dégradés sont générés sous forme vectorielle *via* des instances de `LinearGradientBrush`, de plus ils sont interprétés par le moteur vectoriel `Silverlight`. Vous n'avez donc que peu de chances d'obtenir des couleurs primaires pures aux extrémités. Modifier les pixels manuellement nous évite d'utiliser des dégradés sous forme d'images bitmap embarquées.

Lorsque nous déplaçons le curseur des nuances, nous régénérons constamment l'image du dégradé de couleur. Limitez à tout prix la création dynamique de bitmap *via* la classe `WritableBitmap`, celle-ci est très puissante, mais consomme des ressources. La méthode `RefreshColorPreview`, exposée ci-dessous, permet de rafraîchir la couleur en cours de sélection :

```
private void RefreshColorPreview(bool mustGenerate)
{
    if (mustGenerate)
    {
        this.colorBitmap =
            new WritableBitmap(this.colorPickerPanel, null);

        //on crée un pixel blanc en haut à gauche
        //afin d'avoir un blanc pur
        this.colorBitmap.Pixels[0] =
            (255 << 24) | (255 << 16) | (255 << 8) | 255;
    }

    int pixelIndice = this.oriY * this.colorBitmap.PixelWidth + this.oriX;

    string hexaColor = Convert.ToString(this.colorBitmap.
        Pixels[pixelIndice], 16).ToUpper();

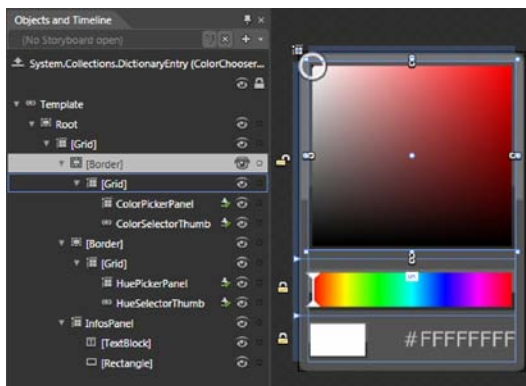
    this.SelectedColor = Color.FromArgb(
        0xFF
        , Convert.ToByte(hexaColor.Substring(2, 2), 16)
        , Convert.ToByte(hexaColor.Substring(4, 2), 16)
        , Convert.ToByte(hexaColor.Substring(6, 2), 16)
    );
}
```

Si besoin, la méthode `RefreshColorPreview` réactualise l'image bitmap de l'objet `ColorPickerPanel`. Elle accepte à cette fin un paramètre booléen. Si vous testez le code suivant, plusieurs erreurs sont levées dès la compilation. C'est tout à fait logique car la propriété `SelectedColor` de notre composant `ColorChooser` n'est pas encore déclarée dans notre classe. Commentez `SelectedColor` dans le code afin d'éviter toute erreur. L'arbre visuel et logique de notre composant est visible à la Figure 12.20.

Le projet est disponible dans l'archive *TestCustomControl_ControlParts.zip* du *chap12* des exemples.

Figure 12.20

Arbre visuel et logique du composant finalisé.



12.3.4 Les propriétés de dépendance

À l'instar de son grand frère WPF, Silverlight propose un système de propriétés facilitant l'implémentation de fonctionnalités avancées. Concrètement, ce système ajoute de nouvelles capacités aux propriétés CLR existantes *via* la création de propriétés de dépendance typées `DependencyProperty`. Celles-ci permettent le support des liaisons de données au même titre que l'implémentation de l'interface `INotifyPropertyChanged` que nous avons abordée à la section 12.2.

Ce n'est toutefois pas le seul cas où vous utiliserez les propriétés de dépendance. À partir du moment où vous souhaitez qu'une propriété supporte le système d'animation, de style, de notification de changement ou qu'elle possède une valeur par défaut, il devient préférable d'utiliser une propriété de dépendance. Seules les classes héritant de `DependencyObject` possèdent des propriétés de dépendance. Cela ne suffit toutefois pas à implémenter des liaisons de données. Le concept de liaison de données est avant tout conçu pour mettre en relation le modèle et la vue. De ce point de vue, l'implémentation réelle des mécanismes de liaison *via* la résolution de valeur de membres et de contextes de données est réalisée par la classe `FrameworkElement`. Celle-ci est en effet le socle sur lequel repose la création d'interfaces visuelles et contient de ce fait les fondations des API de liaison. Le code ci-dessous expose la déclaration d'une propriété de dépendance :

```
public int MyProperty
{
    get { return (int)GetValue(MyPropertyProperty); }
    set { SetValue(MyPropertyProperty, value); }
}

public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register(
```



```
"MyProperty", // le nom de la propriété CLR enregistrée
typeof(int), // le type de la propriété
typeof(ownerclass), //La classe qui possède la propriété
null); //définition d'une valeur par défaut et/ou méthode de rappel
```

Pour coder rapidement ce type de propriété sous Visual Studio, il vous suffit d'écrire `propdp` puis d'appuyer deux fois la touche Tabulation. Comme vous le constatez, la propriété publique utilise en interne les méthodes `SetValue` et `GetValue` qui sont fournies par l'API du système de propriétés Silverlight. Pour des raisons liées à l'initialisation de l'application, les propriétés de dépendance sont statiques. En pratique, la méthode statique `Register` de la classe `DependencyProperty` renvoie une propriété de dépendance. Pour cela, la méthode `Register` accepte comme paramètre, le nom de la propriété CLR publique sous forme de chaîne de caractères, suivi de son type et du type de l'objet qui l'expose.

Dans la grande majorité des cas, la classe qui possède la propriété est celle qui contient la propriété publique. Un dernier paramètre permet de spécifier une valeur par défaut à l'initialisation ainsi qu'une méthode statique de rappel déclenchée lorsque la valeur de la propriété est modifiée. Cela est particulièrement utile lorsque vous souhaitez diffuser des événements ou définir une rétroaction. Ce dernier paramètre n'est toutefois pas obligatoire, il vous suffit de passer `null` pour l'éluider. Si vous pouvez écrire ces quelques lignes, c'est que la classe dans laquelle vous codez hérite de `DependencyObject`. Vous pourriez être tenté de créer un objet métier comme ci-dessous :

```
namespace SilverlightApplication1
{
    public class Customer : DependencyObject
    {
        public bool IsConnected
        {
            get { return (bool)GetValue(IsConnectedProperty); }
            set { SetValue(IsConnectedProperty, value); }
        }

        private static readonly DependencyProperty IsConnectedProperty =
            DependencyProperty.Register("IsConnected", typeof(bool),
                typeof(Customer), new PropertyMetadata(false, new
                    PropertyChangedCallback( OnChangedStatus ) ));

        public static void OnChangedStatus
            (DependencyObject dp, DependencyPropertyChangedEventArgs e)
        {
            //rétroaction
        }
    }
}
```

Comme vous le constatez, le dernier paramètre permet de définir une valeur par défaut, dans notre cas à `false`, ainsi qu'une méthode statique privée de rappel (*callback*) déclenchée lorsque la propriété change de valeur. Il n'est pas forcément recommandé d'utiliser cette méthodologie. Comme nous l'avons dit précédemment, hériter de `DependencyObject` n'apporte pas la totalité des fonctionnalités. Seule la notification du changement de propriétés est supportée. Il vaut donc mieux implémenter l'interface `INotifyPropertyChanged` pour les objets métiers ou pour ceux qui ne font pas partie de la liste d'affichage. Ceci est une meilleure option car l'héritage multiple n'est pas supporté par C#, alors que l'implémentation multiple d'interfaces est possible ainsi que largement recommandée. Une fois que vous aurez hérité de `DependencyObject`, vous ne pourrez

pas hériter d'une autre classe si besoin et vous devrez peut-être revoir votre conception dans le pire des cas.

Voyons maintenant ce que cela donne pour notre ColorChooser. Il possède une propriété de dépendance SelectedColor de type Color :

```
[Category("Common Properties")]
[Description("La couleur courante choisie par l'utilisateur.")]
[EditorBrowsable(EditorBrowsableState.Advanced)]
public Color SelectedColor
{
    get { return (Color)GetValue(SelectedColorProperty); }
    set { SetValue(SelectedColorProperty, value); }
}

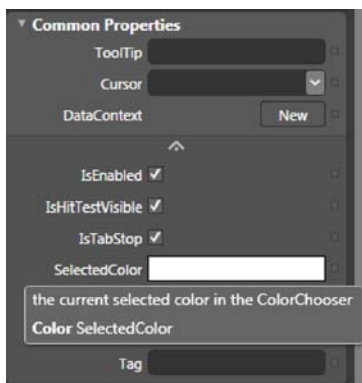
public static readonly DependencyProperty SelectedColorProperty =
    DependencyProperty.Register(
        "SelectedColor",
        typeof(Color),
        typeof(ColorChooser),
        new PropertyMetadata(Colors.White, new PropertyChangedCallback
            (OnSelectedColorChanged))
    );
```

Il est possible de définir des métadonnées à deux niveaux. Vous pouvez en tout premier lieu les placer directement au-dessus de la propriété entre crochets. Ces métadonnées permettent, entre autres, l'apport d'informations complémentaires lisibles dans l'interface d'Expression Blend.

L'attribut Category nous permet de définir la catégorie au sein du panneau des propriétés de Blend, dans lequel la propriété sera accessible : Description offre la possibilité d'afficher un résumé de l'intérêt de cette propriété au survol de la propriété dans Blend. L'attribut EditorBrowsable permet, quant à lui, de ne pas afficher la propriété dans le panneau, de la rendre visible ou alors de la placer dans une sous-partie d'option avancée accessible *via* la double flèche (voir Figure 12.21).

Figure 12.21

La propriété SelectedColor accessible dans les options avancées de la rubrique Common Properties.



La deuxième possibilité consiste à définir les métadonnées dans le dernier paramètre de la méthode Register comme nous l'avons évoqué plus haut. Nous y définissons une couleur blanche comme valeur par défaut car le curseur de sélection est placé en haut à gauche du sélecteur. De

plus, la méthode de rappel `OnSelectedColorChanged` est exécutée chaque fois que la couleur est modifiée. Cette dernière diffuse alors l'événement `ColorChanged` :

```
private static void OnSelectedColorChanged
    (DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    ColorChooser cc = d as ColorChooser;

    if (cc.ColorChanged != null)
    {
        ColorChangedEventArgs cce = new ColorChangedEventArgs();
        cce.NewValue = (Color)e.NewValue;
        cce.OldValue = (Color)e.OldValue;
        cc.ColorChanged(cc, cce);
    }
}
```

L'objet événementiel de type `ColorChangedEventArgs` hérite de la classe `EventArgs` et contient deux propriétés supplémentaires, `NewValue` et `OldValue`, vous permettant de récupérer respectivement la nouvelle couleur et l'ancienne couleur sélectionnée. Nous partons du principe que ces propriétés ne doivent pas être accessibles en écriture à l'extérieur de l'espace de noms `Tweened-Controls` :

```
public class ColorChangedEventArgs : EventArgs
{
    public Color NewValue
    {
        get;
        internal set;
    }

    public Color OldValue
    {
        get;
        internal set;
    }
}
```

Du côté de la classe `ColorPicker`, la propriété est définie de manière similaire. Toutefois l'implémentation est légèrement différente car la classe `ColorPicker` possède une instance de `ColorChooser` et écoute l'événement `ColorChanged` de cette instance afin de modifier sa propre propriété `SelectedColor` :

```
public ColorPicker()
{
    ...

    this.colorChooser.ColorChanged += new
        EventHandler<ColorChangedEventArgs>(this.colorChooser_ColorChanged);
}

private void colorChooser_ColorChanged
    (object sender, ColorChangedEventArgs e)
{
    this.SelectedColor = e.NewValue;
}
```

Le principe mis en œuvre est une délégation au sens propre du terme. Il reste encore une problématique à régler : il faut donner au concepteur la capacité de définir un style sur le ColorChooser contenu par le ColorPicker. Il est nécessaire de créer une propriété de dépendance à cette fin :

```
[Category("Common Properties")]
[Description("The current style applied to the ColorPicker.")]
[EditorBrowsable(EditorBrowsableState.Advanced)]
public Style ColorChooserStyle
{
    get { return (Style)GetValue(ColorChooserStyleProperty); }
    set { SetValue(ColorChooserStyleProperty, value); }
}

public static readonly DependencyProperty ColorChooserStyleProperty =
    DependencyProperty.Register(
        "ColorChooserStyle",
        typeof(Style),
        typeof(ColorPicker),
        new PropertyMetadata(OnColorChooserStyleChanged));

private static void OnColorChooserStyleChanged(DependencyObject d,
        DependencyPropertyChangedEventArgs e)
{
    Style s = e.NewValue as Style;

    ColorPicker cp = d as ColorPicker;

    if (cp != null)
    {
        Style os = e.OldValue as Style;

        cp.colorChooser.Style = s;
    }
}
```

Cette solution est tout à fait cohérente, mais nous n'avons pas fini. À ce stade, un développeur peut facilement affecter un style au ColorChooser. Un designer le pourra également en passant par le panneau des propriétés, bien que cela ne soit pas habituel pour lui. Là encore il vous faut vous arranger pour que le designer puisse, *via* Expression Blend, appliquer un style de son choix en passant par les menus habituels ou par un clic-droit. Il vous faut pour cela ajouter une métadonnée au-dessus de la classe ColorPicker, et recompiler, voire reconstruire le projet sous Blend. L'attribut StyleTypedProperty responsable de cette intégration est exposé ci-dessous :

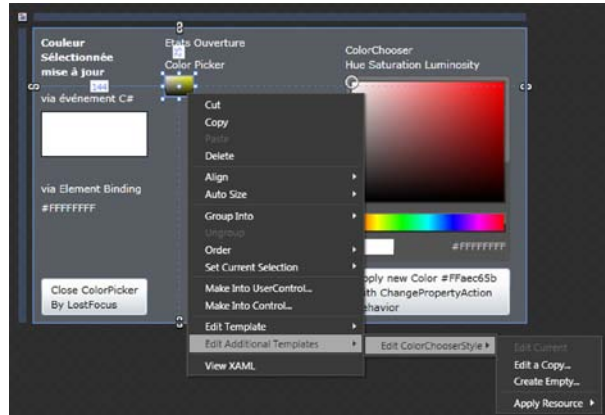
```
[StyleTypedProperty(Property = "ColorChooserStyle",
        StyleTargetType = typeof(ColorChooser))]

public class ColorPicker : Control
{
```

Concrètement, vous pouvez désormais accéder au style de manière directe et standard dans Blend par le menu Object de la barre du haut ou *via* un simple clic-droit (voir Figure 12.22).

Figure 12.22

Accès au style du ColorChooser contenu dans le ColorPicker via le clic-droit.



Vous avez encore un peu de travail pour finaliser les deux composants par vous-même. Vous pouvez toutefois télécharger la version finalisée, *TestCustomControl*, dans le *chap12* des exemples.

Au prochain chapitre, nous aborderons la lecture de médias à travers l'utilisation de différents contrôles utilisateur spécialement créés pour l'occasion. Nous étudierons ensuite le chargement de données distantes au sein d'un projet basé sur le modèle de conception MVVM.

Médias et données

Parmi tous les principes apportés par le développement web, le plus représentatif est sans aucun doute le comportement asynchrone lié au chargement de contenus distants. Celui-ci est toujours présent et brille soit par son absence de prise en charge, soit par un support étendu qui améliore la qualité et l'intérêt des applications. Il est au cœur de nombreuses problématiques de production qui se sont complexifiées avec l'évolution rapide des besoins. L'un d'eux concerne l'échange de données entre, d'une part, les applications clientes, et d'autre part, les applications ou services distants hébergés côté serveur. Les technologies dites asynchrones comme AJAX, Flash ou dernièrement Silverlight y apportent des réponses plus ou moins efficaces et élégantes. La technologie AJAX (pour *Asynchronous JavaScript and XML*) est en fait résumée par un seul objet asynchrone, il ne peut donc pas se positionner au même niveau qu'une plateforme comme Silverlight qui possède de nombreux objets et méthodes asynchrones. Ce chapitre est consacré à l'apprentissage des mécanismes propres au chargement et à l'envoi de données. Nous aborderons tout d'abord le chargement de médias externes de type bitmap et vidéo, puis nous étudierons les contraintes liées à leur diffusion. Par la suite, nous apprendrons à concevoir une application orientée données en nous basant sur le modèle de conception MVVM. Puis, nous apprendrons à charger et à consommer des flux de données au format JSON et XML *via* différentes technologies dont LINQ. Pour finir, nous aborderons les principes de base inhérents à la sécurité et à l'échange d'informations interdomaines.

13.1 Chargement de médias

Dans cette section, vous allez apprendre à télécharger des images. Nous créerons à cette occasion une classe qui centralisera la logique de chargement et diffusera des événements personnalisés, ainsi qu'une vue gérant l'affichage de l'image. Dans un second temps, nous allons reprendre le lecteur vidéo créé au Chapitre 7 pour diffuser de la vidéo. Nous en profiterons pour en faire un contrôle réutilisable.

13.1.1 Chargement dynamique d'images

Deux formats d'image, JPG et PNG, sont actuellement supportés par Silverlight. En règle générale, la compression JPG est efficace lorsque vous n'avez pas besoin de couche d'opacité ainsi que pour des résolutions supérieures à 128 pixels de largeur par 128 pixels de hauteur. Le format PNG possède la capacité d'être défini par une palette de couleurs indexées 8 bits (256 couleurs) ou 24 bits (32 millions de couleurs) ainsi qu'un mode "couleurs véritables non indexées". Lorsque le fichier PNG est encodé en 8 bits, son poids est relativement léger pour des images de faible résolution, tout en conservant une qualité optimale. Ce format permet en outre d'obtenir une intégration simplifiée de l'image au sein de l'arbre visuel et logique d'une application *via* sa couche de transparence. De ce point de vue, son utilisation est pertinente lorsque vous souhaitez afficher ou concevoir des icônes. Quel que soit le format utilisé, charger dynamiquement des images se révèle payant à plus d'un titre. D'une part, cela allège le poids de votre application car les images ne sont pas stockées dans le fichier compilé mais directement sur le serveur. D'autre part, cela dynamise vos sites et vos applications, les rendant ainsi plus attractifs. Pour finir, cela facilite leur maintenance et leur évolution car une partie de la maquette graphique peut être gérée indépendamment de l'utilisation de Blend.

13.1.1.1 La classe `BitmapImage`

Nous avons vu à plusieurs reprises que le composant `Image` possède la capacité d'afficher des images bitmap. Dans ce cas, les images peuvent au choix être compilées au sein du fichier xap ou être accessibles directement sur un serveur web. Dans ce dernier cas, les fichiers seront dans le même répertoire que le xap ou dans un autre. À noter que dans le cas d'un référencement relatif, le chemin de l'image est calculé à partir de l'emplacement du xap. Si vous ajoutez des images *via* le menu `Add Existing Item`, celles-ci seront par défaut compilées dans le fichier xap. Il est possible de modifier ce comportement au sein de Visual Studio en changeant les options. Dans tous les cas, il est possible de définir la propriété `Source` directement en XAML à partir du moment où les images sont embarquées ou placées dans le répertoire adéquat :

```
<Image Source="monImage.jpg" />
```

Du côté C#, cela est légèrement moins évident lorsque l'on n'utilise pas la classe `BitmapImage`. Il nous faudra convertir la chaîne de caractères en instance de type `ImageSource` comme montré ci-dessous :

```
public partial class MainPage : UserControl
{
    string urlImage = "http://www.tweened.org/wp-content/files/livre/
                        images/lapin-cretin.jpg";

    public MainPage()
    {
        InitializeComponent();

        ImageSourceConverter isc = new ImageSourceConverter();

        ImageSource imgSource =
            isc.ConvertFromString(urlImage) as ImageSource;
        uneInstanceImage.Source =imgSource;
    }
}
```

Il est possible d'écouter les événements `ImageOpened` et `ImageFailed` respectivement diffusés lorsque l'image est correctement chargée ou, au contraire, quand son chargement échoue :

```
public partial class MainPage : UserControl
{
    string urlImage = " http://www.tweened.org/wp-content/files/livre/
                        images/lapin-cretin.jpg ";

    public MainPage()
    {
        // Required to initialize variables
        InitializeComponent();

        ImageSourceConverter isc = new ImageSourceConverter();

        ImageSource imgSource = isc.ConvertFromString(urlImage) as
            ImageSource;

        uneInstanceImage.ImageFailed += new EventHandler
            <ExceptionRoutedEventArgs>(uneInstanceImage_ImageFailed);

        uneInstanceImage.ImageOpened += new EventHandler
            <RoutedEventArgs>(uneInstanceImage_ImageOpened);

        uneInstanceImage.Source =imgSource;
    }

    void uneInstanceImage_ImageOpened(object sender, RoutedEventArgs e)
    {
        TextBlock t = new TextBlock();

        t.Text = "L'image " + urlImage + " a été chargée.";

        LayoutRoot.Children.Add(t);
    }

    void uneInstanceImage_ImageFailed(object sender,
        ExceptionRoutedEventArgs e)
    {
        throw new NotImplementedException("Le format et/ou le chemin d'accès
            de l'image " + urlImage + " sont incorrects.");
    }
}
```

ATTENTION

Le code ci-dessus ne fonctionne qu'à une seule condition : il vous faut exécuter l'application en environnement `http` lorsque le chemin d'accès est une URL absolue commençant par `http://localhost:49188/TestPage.html`. Autrement dit, sous `Blend` vous n'aurez aucun problème puisque ce dernier lance systématiquement l'application *via* une instance du serveur de développement. Sous `Visual Studio`, si vous n'êtes pas dans un projet `Silverlight 3 + Website` ou utilisant une page `ASP.NET`, le chemin d'accès au site sera du type `file:///C:/Users/votre-nom/Desktop/test/test/Bin/Debug/TestPage.html`. Dans ce cas, l'événement `ImageFailed` sera diffusé indiquant l'échec du chargement. Cela est lié à des contraintes de sécurité. Une dernière restriction au chargement d'images concerne leur emplacement. Il n'est pas possible d'utiliser un chemin d'accès relatif du type `../images/uneImage.jpg`, et cela également pour des raisons de sécurité. Ce n'est pas un problème puisqu'il est possible

de récupérer l'URL absolue du site puis de concaténer le bon chemin d'accès aux images afin de passer une URL de type absolue, comme montré ci-dessous :

```
string SitePath = "";

public MainPage()
{
    InitializeComponent();

    if (App.Current.Host.Source != null)
    {
        SitePath = new Uri(App.Current.Host.Source, "../").ToString()
    }
    ...
}
```

Notre conception précédente est trop directe et n'est pas idéale pour différentes raisons. D'une part, nous ne pouvons pas surveiller l'état de téléchargement de l'image distante. D'autre part, nous n'avons pas accès simplement aux dimensions natives de l'image à la réception de celle-ci. La classe `BitmapImage` résout ces deux problématiques assez simplement en donnant accès à des propriétés et capacités supplémentaires (voir Tableau 13.1).

Tableau 13.1 : Description de la classe *BitmapImage*.

<i>Méthode asynchrone</i>	<i>Propriétés</i>	<i>Événements</i>
<code>SetSource(Stream)</code>	<code>UriSource</code>	<code>ImageFailed</code>
Correspond à un flux de données binaires à utiliser conjointement avec une instance de type <code>WebClient</code> .	Attend une valeur de type <code>Uri</code>	Diffusé lorsque l'image n'est pas trouvée ou lorsque le format attendu ne correspond pas.
Non documenté	<code>PixelWidth</code>	<code>ImageOpened</code>
	Largeur originelle en pixels de l'image bitmap, de type <code>int</code> .	Événement diffusé lorsque l'image est entièrement téléchargée.
Non documenté	<code>PixelHeight</code>	<code>DownloadProgress</code>
	Hauteur originelle en pixels de l'image bitmap, de type <code>int</code> .	Diffusé durant la progression du téléchargement.
Non documenté	<code>CreateOptions</code>	Non documenté
	Options de création de l'image.	

Comme vous le constatez, tous les ingrédients sont présents pour assurer la gestion d'un chargement dynamique. Pour finir, la classe `BitmapImage` n'est pas un objet de type `UIElement`. Les instances de ce type ne peuvent donc pas être ajoutées à la liste d'affichage de notre application `Silverlight`. Vous devrez utiliser un composant `Image` ou un pinceau d'image, `ImageBrush`, pour

afficher cette dernière. Le code ci-dessous illustre un exemple concret d'utilisation de la classe `BitmapImage` :

```
...
using System.Windows.Media.Imaging;

namespace SimpleBitmapLoader
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();

            Uri uri = new Uri("http://www.tweened.org/wp-content/files/livre/
                images/lapin-cretin.jpg", UriKind.RelativeOrAbsolute);

            BitmapImage bi = new BitmapImage(uri);

            Image img = new Image();

            img.Source = bi;

            LayoutRoot.Children.Add(img);
        }
    }
}
```

Comme nous l'avons évoqué plus haut, il est également possible d'afficher l'image en utilisant une instance de type `ImageBrush` :

```
//Chargement d'une image via l'utilisation d'un pinceau d'image
Uri uri = new Uri("http://www.tweened.org/wp-content/files/livre/images/
    lapin-cretin.jpg", UriKind.RelativeOrAbsolute);

BitmapImage bi = new BitmapImage(uri);

ImageBrush ib = new ImageBrush();

ib.ImageSource = bi;

LayoutRoot.Background = ib;
```

Il n'y a pas vraiment de différences flagrantes entre l'une ou l'autre de ces méthodes. Seul le mode d'étirement est différent : il est par défaut en mode `Fill` pour un pinceau d'image alors qu'il est de type `Uniform` par défaut pour les instances d'`Image`. Le rendu final est assez différent puisqu'en mode `Fill`, l'image est étirée afin de remplir la totalité de la surface mise à disposition. En outre, d'un point de vue réutilisation, le pinceau d'image est avantageux car il peut être affecté à n'importe quelle propriété acceptant une instance de type `Brush` tout en étant stocké une seule fois en mémoire. Le projet est dans l'archive *SimpleBitmapLoader.zip* du dossier *chap13* des exemples.

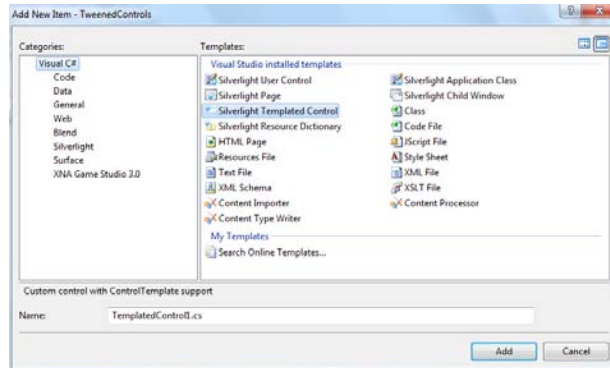
13.1.1.2 Conception du contrôle `ImageLoader`

Nous allons maintenant approfondir l'utilisation de la classe `BitmapImage` à travers la conception d'un contrôle personnalisé. Le but est de simplifier le chargement d'images à travers la réutilisation de ce composant. Nous implémenterons ainsi la quasi totalité des fonctionnalités possibles

dont le préchargement, la gestion des erreurs, la diffusion d'événements et la récupération des dimensions de l'image téléchargée. Au sein de Blend, générez un projet Silverlight + Website nommé `TestMediaCustomControls`. Ouvrez la solution dans Visual Studio, puis ajoutez dans la solution un autre projet de type "Silverlight Class Library", `MediaTweenedControls`. Supprimez la classe créée par défaut et insérez un nouvel élément de type "Silverlight Templated Control" (voir Figure 13.1).

Figure 13.1

La fenêtre de création du composant personnalisable.



Nommez ce composant `ImageLoader`, puis validez en cliquant sur le bouton `Add`. Dans la classe `ImageLoader`, définissez le contrat de modèle ainsi que les événements comme indiqué dans le code ci-dessous :

```
[TemplateVisualState(GroupName = "CommonStates", Name = "Normal")]
[TemplateVisualState(GroupName = "CommonStates", Name = "MouseOver")]
[TemplateVisualState(GroupName = "CommonStates", Name = "Pressed")]
[TemplateVisualState(GroupName = "CommonStates", Name = "Disabled")]

[TemplateVisualState(GroupName = "LoadingStates", Name = "Unloaded")]
[TemplateVisualState(GroupName = "LoadingStates", Name = "Loading")]
[TemplateVisualState(GroupName = "LoadingStates", Name = "Completed")]
public class ImageLoader : Control
{
    #region événements

    public event EventHandler<ExceptionRoutedEventArgs> ImageFailed;

    public event EventHandler<RoutedEventArgs> ImageOpened;

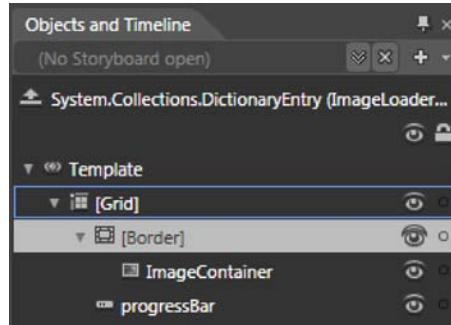
    public event EventHandler<DownloadProgressEventArgs> DownloadProgress;

    #endregion
    ...
}
```

Outre les états interactifs standard, nous en décrivons trois autres qui correspondent à l'état du chargement de l'image. Cela permettra au designer d'élaborer un visuel spécifique à chacun d'eux. Il pourra ainsi afficher ou faire disparaître la barre de progression, et générer des transitions de fondu de l'image. Modifiez le modèle du composant afin d'obtenir l'arbre visuel de la Figure 13.2.

Figure 13.2

Arbre visuel du composant ImageLoader.



Le composant ProgressBar hérite de RangeBase, comme Slider. Nous ne forçons pas sa création comme partie logique car sa propriété Value est liée la propriété de dépendance Progress que nous définissons sur ImageLoader :

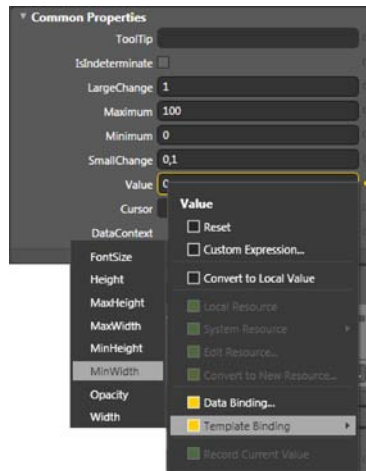
```
[Description("Progression du téléchargement de l'image de 0 à 100.")]
public double Progress
{
    get { return (double)GetValue(ProgressProperty); }
    private set { SetValue(ProgressProperty, value); }
}

public static readonly DependencyProperty ProgressProperty =
    DependencyProperty.Register
        ("Progress", typeof(double), typeof(ImageLoader),
        null);
```

Vous remarquez que nous n'autorisons pas le designer ou le développeur à modifier cette propriété sur l'instance de ImageLoader. Nous définissons pour cela son accesseur set en accès privé. Cette propriété est en fait liée à l'état du téléchargement, il ne serait donc pas logique de la définir manuellement. D'un autre côté, il peut être très utile d'y avoir accès en lecture. Comme nous l'avons indiqué plus haut, la propriété Value de notre instance de ProgressBar y est reliée *via* une liaison de modèles, toutefois celle-ci n'est pas visible dans l'interface de Blend car son écriture est en accès privé (voir Figure 13.3).

Figure 13.3

La propriété Progress est inaccessible via l'interface de Blend.



Il est toutefois possible de définir la liaison directement en XAML puisque la propriété est accessible en lecture :

```
<ProgressBar x:Name="progressBar" Value="{TemplateBinding Progress}" ... />
```

Vous pourriez être tenté d'utiliser l'attribut suivant pour résoudre cette problématique :

```
[EditorBrowsable( EditorBrowsableState.Always )]
```

Cela ne changera strictement rien, la propriété ne sera pas accessible pour autant dans l'interface de Blend. La conséquence directe de ce comportement est que le designer ne pourra pas créer une liaison de modèles simplement vers la propriété `Progress` car l'interface ne lui indique pas son existence. C'est au développeur de choisir l'implémentation idéale. Mais il est possible de trouver un compromis. Vous pouvez définir cette propriété en écriture tout en la cachant dans l'onglet des options avancées. Il suffit pour cela de définir l'attribut `EditorBrowsable` comme indiqué ci-dessous :

```
[EditorBrowsable( EditorBrowsableState.Advanced )]
```

Dans l'avenir, nous pourrions imaginer que les propriétés de dépendance pourraient être visibles dans l'interface lorsqu'elles sont en lecture seule... D'autres propriétés sont dans le même cas, comme la propriété `LoadedBitmapImage`, qui correspond à l'instance `BitmapImage` contenant les données de l'image chargée. Il en va de même pour `OriginalBitmapSize` qui permet de récupérer les dimensions natives de l'image chargée. Voici leur implémentation :

```
//Bitmap Data
[Description("Données de la bitmap chargée.")]
[EditorBrowsable(EditorBrowsableState.Advanced)]
[Category("Common Properties")]
public BitmapImage LoadedBitmapImage
{
    get { return (BitmapImage)GetValue(LoadedBitmapImageProperty); }
    set { SetValue(LoadedBitmapImageProperty, value); }
}
public static readonly DependencyProperty LoadedBitmapImageProperty =
    DependencyProperty.Register(
        "LoadedBitmapImage",
        typeof(BitmapImage),
        typeof(ImageLoader),
        null);

//Dimensions d'origine
[Description("Dimension d'origine de l'image chargée.")]
[EditorBrowsable( EditorBrowsableState.Advanced )]
[Category("Common Properties")]
public Size OriginalBitmapSize
{
    get { return (Size)GetValue(OriginalBitmapSizeProperty); }
    set { SetValue(OriginalBitmapSizeProperty, value); }
}
public static readonly DependencyProperty OriginalBitmapSizeProperty =
    DependencyProperty.Register(
```

```

        "OriginalBitmapSize",
        typeof(Size),
        typeof(ImageLoader),
        null);

```

Il nous faut également exposer une propriété de dépendance `Source` qui contiendra une URL pointant vers l'image à afficher. Cette propriété définit une méthode de rappel déclenchée lorsque sa valeur est modifiée :

```

//chemin d'accès à l'image
[Category("Common Properties")]
[Description("Url de l'image à charger.")]
public string Source
{
    get { return (string)GetValue(SourceProperty); }
    set { SetValue(SourceProperty, value); }
}

public static readonly DependencyProperty SourceProperty =
    DependencyProperty.Register("Source", typeof(string),
        typeof(ImageLoader), new PropertyMetadata(
            new PropertyChangedCallback(OnSourceUriChanged)));

private static void OnSourceUriChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e)
{
    ImageLoader il = d as ImageLoader;

    Uri uri = new Uri(e.NewValue as string, UriKind.RelativeOrAbsolute);

    il.LoadedBitmapImage = new BitmapImage(uri);

    il.LoadedBitmapImage.CreateOptions =
        BitmapCreateOptions.IgnoreImageCache;

    il.LoadedBitmapImage.DownloadProgress += new EventHandler<
        <DownloadProgressEventArgs>(il.LoadedBitmapImage_DownloadProgress);

    il.LoadedBitmapImage.ImageOpened += new EventHandler<RoutedEventArgs>
        (il.LoadedBitmapImage_ImageOpened);

    il.LoadedBitmapImage.ImageFailed += new EventHandler<
        <ExceptionRoutedEventArgs>(il.LoadedBitmapImage_ImageFailed);

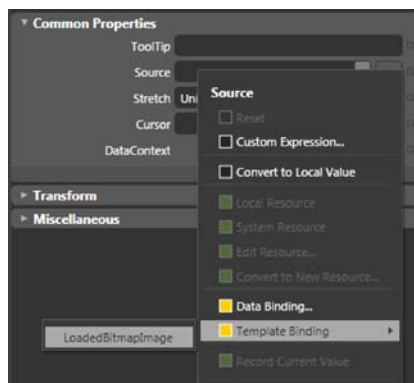
    VisualStateManager.GoToState(il, "Loading", true);
}

```

Comme vous le constatez, aucune partie logique n'est nécessaire pour afficher l'image. Nous n'avons pas besoin d'utiliser du code logique pour affecter `LoadedBitmapImage` au contrôle `Image` nommé `ImageContainer` présent dans l'arbre visuel et logique de notre modèle. Nous n'avons qu'à définir une liaison de modèles entre la propriété `Source` de l'objet `ImageContainer` et la propriété de dépendance `LoadedBitmapImage` (voir Figure 13.4).

Figure 13.4

Liaison de modèles de la propriété Source vers LoadedBitmapImage.



En procédant ainsi, le designer reste libre et peut modifier plus facilement le contrôle. L'utilisation du contrôle est au final très simple :

```
public MainPage()
{
    InitializeComponent();

    il.ImageFailed += new EventHandler<ExceptionRoutedEventArgs>
        (il_ImageFailed);

    il.ImageOpened += new EventHandler<RoutedEventArgs>(il_ImageOpened);

    il.Source = "http://www.tweneed.org/wp-content/files/livre/images/
        lapin-cretin.jpg";
}

void il_ImageOpened(object sender, RoutedEventArgs e)
{
    //on peut récupérer la taille originale de l'Image par exemple
    //il.Width = il.OriginalBitmapSize.Width+4;
    //il.Height = il.OriginalBitmapSize.Height+4;
}

void il_ImageFailed(object sender, ExceptionRoutedEventArgs e)
{
    throw e.ErrorException;
}
```

Le projet complet est dans l'archive *TestMediaCustomControl.zip* du dossier *chap13* des exemples. Le composant finalisé est également téléchargeable sur le blog www.tweneed.org. Il possède des capacités supplémentaires qui faciliteront vos développements.

13.1.2 Formats vidéo et modes de diffusion

Le chargement de contenu vidéo est l'un des principaux enjeux commerciaux des plateformes de diffusion plurimédia telles que Silverlight. Ce dernier fournit un support natif de divers formats d'encodage, ce qui le place en tête des solutions de diffusion vidéo pour le Web. Ainsi, lorsque vous concevez un lecteur vidéo pour Silverlight, vous bénéficiez d'un certain nombre de puissants formats optimisés pour l'environnement Internet. Cela évite à l'internaute de fastidieuses mises à

jour logicielles tout en simplifiant l'élaboration de lecteurs vidéo sophistiqués. Au sein de la plateforme Silverlight, plusieurs conteneurs et formats sont disponibles. Un conteneur est un fichier qui contient des données vidéo et/ou audio encodées dans divers formats. Il a pour rôle de référencer ces données en décrivant leur mode de stockage. Il contient également des métadonnées de description des médias. Il est ainsi possible de stocker des informations concernant le chapitrage, le copyright, le nom de l'auteur ou toute autre information liée aux médias contenus. Chaque conteneur possède des capacités propres et n'accepte qu'un nombre limité de formats d'encodage.

Vous devrez toutefois faire la différence entre conteneur et extensions de fichiers, l'extension ne joue au final pas de rôle à proprement parler alors que le format du conteneur structure de manière spécifique les métadonnées. Le conteneur MOV en est le parfait exemple : ce conteneur est polymorphe. Il n'est théoriquement pas supporté par Silverlight, toutefois si ce dernier contient des données encodées au bon format, il sera lu, interprété et affiché correctement par ce dernier. Voici la liste des conteneurs et de quelques formats supportés par Silverlight :

- Windows Media Video et Audio (wmv et wma) ;
- "353" – Microsoft Windows Media Audio v7 v8 and v9.x Standard (*WMA Standard*) ;
- "354" – Microsoft Windows Media Audio v9.x and v10 Professional (*WMA Professional*) ;
- WMV1 (Windows Media Video 7) ;
- WMV2 (Windows Media Video 8) ;
- WMV3 (Windows Media Video 9) ;
- MP4 ;
- H.264 (ITU-T H.264 / ISO MPEG-4 AVC), AAC-LC ;
- MP3 ;
- "85" – ISO MPEG-1 Layer III (MP3).

INFO

Certaines associations d'encodage ne sont pas supportées par Silverlight. Par exemple, il ne supporte pas l'association d'un format vidéo WMV (version 1, 2 et 3) avec un encodage de bande sonore de type MP3 ou AAC. Vous trouverez une liste complète des formats supportés à l'adresse : <http://msdn.microsoft.com/en-us/library/cc189080%28VS.95%29.aspx>.

En résumé, il est fortement conseillé d'encoder vos vidéos au sein d'un conteneur de type MP4. Ce conteneur vous apporte le meilleur rapport qualité d'image / bande passante à travers le format de compression vidéo H.264 et le format audio AAC. Il est toutefois très fréquent de récupérer des formats de ce type contenus dans une extension MOV ou un conteneur MOV. Vérifiez simplement que le type d'encodage correspond à celui d'un conteneur de type MP4 (soit H.264 et AAC). Lorsque vous utilisez la propriété `Source` de l'objet `MediaElement`, celui-ci ne vous propose pas les extensions `.mov`. Il est toutefois possible de renseigner le chemin d'accès au fichier manuellement, dans ce cas le fichier MOV est lu sans aucun problème. Silverlight 3 supporte différents standard du H.264, qui correspondent chacun à autant de résolutions listées ci-dessous. Attention au fait que Silverlight est capable de lire des vidéos dans n'importe quelles résolutions. Celles indiquées ci-dessous correspondent uniquement à des formats d'enregistrement et de diffusion :

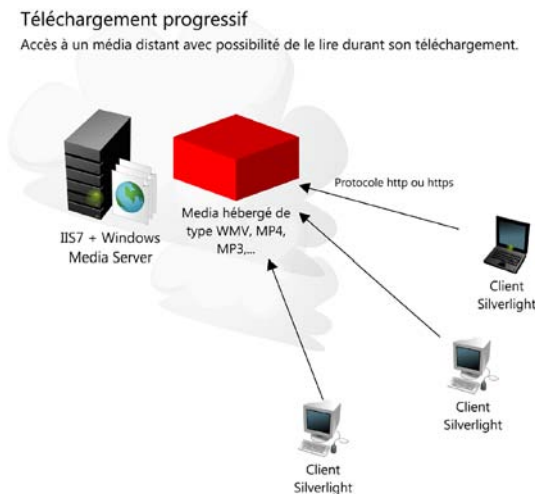
- 480p n'est pas vraiment utilisé et cible les résolutions 4/3 et 16/9 ayant un total de 480 lignes est de 480.
- 720p correspond à la norme HD Ready d'un format 16/9 en 1 280 C 720 pixels.
- 1 080p fait référence à la norme Full HD d'un format 16/9 en 1 920 C 1 080 pixels

Une fois que vous aurez choisi un type d'encodage, il vous faudra déterminer un mode de diffusion adapté à vos besoins et à vos ressources. Vous devrez choisir entre une diffusion de type progressif ou continu. Cette décision technologique impactera directement vos budgets ainsi que votre production.

Le mode de téléchargement progressif est le plus couramment rencontré sur Internet. Il consiste à placer un média sur votre serveur et à le lire en cours de téléchargement sur le disque local. Dans ce cas, le protocole utilisé est HTTP, et aucune technologie côté serveur n'est nécessaire mise à part un simple serveur web (voir Figure 13.5). Cette solution est peu coûteuse en termes d'argent et de temps de développement mais diminue la qualité de l'expérience utilisateur. À l'origine, il n'était pas possible de déplacer la tête de lecture jusqu'à un temps donné si la vidéo n'avait pas été téléchargée jusqu'à cet instant. Ce comportement a évolué avec l'amélioration des serveurs web et des plateformes de diffusion côté client.

Figure 13.5

Téléchargement progressif.



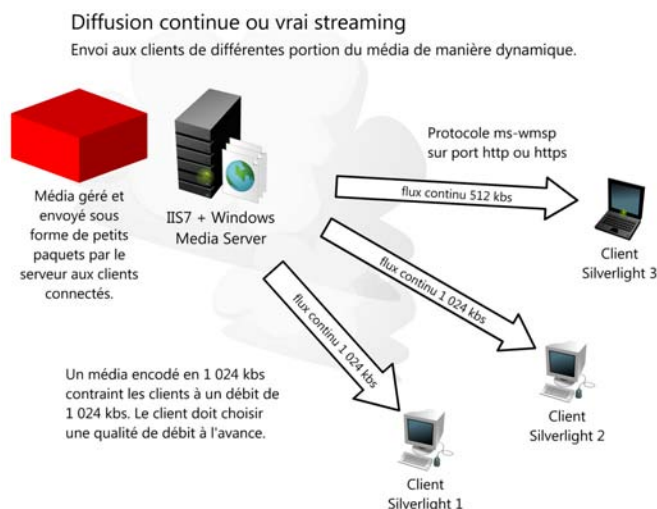
La diffusion en continu, également appelée *streaming*, apporte de nombreux avantages mais nécessite *Windows Media Server* – un module de *Internet Information Service 7*. IIS7 est disponible sur Windows Server 2008, Vista et Windows 7. Dans ce mode de diffusion, il est possible pour l'utilisateur de déplacer la tête de lecture à n'importe quel instant de la vidéo et d'éviter ainsi un téléchargement fastidieux. Dans ce cas, la vidéo n'est pas stockée dans la mémoire cache du navigateur, mais dynamiquement en RAM. Cela est avantageux en termes de sécurité et de droits d'auteur.

Grâce au streaming, il est également possible de retransmettre un flux live pour des émissions en direct ce qui n'est pas réalisable avec un téléchargement progressif. Plusieurs protocoles d'échanges sont disponibles dans ce cas. Le protocole de diffusion temps réel, nommé *Real Time Streaming Protocol* (RTSP), est le dernier en date utilisé par défaut sur Windows Media Server et

IIS7 pour échanger des flux de données continus entre le client Silverlight et le serveur. Il opère sur le port 554. Cela pose certaines problématiques si vous êtes dans une configuration réseau contraignante avec firewall, ce qui est le cas de nombreuses entreprises. Silverlight prend cela en considération et utilise par défaut le protocole d'échange *WMSP*, signifiant *Windows Media HTTP Streaming Protocol*. Ce dernier est basé sur une spécification HTTP et utilise par conséquent les ports 80 et 443 pour HTTPS, ces derniers sont généralement disponibles. *A contrario* de HTTP, WMSP est capable de maintenir des connexions persistantes client/serveur qui sont nécessaires à la diffusion continue (voir Figure 13.6). Côté Windows Media Server, il vous suffira de cocher l'option de diffusion pour le Web, vous n'aurez rien à faire, il vous suffira de placer vos fichiers à l'emplacement adéquat pour qu'ils soient accessibles.

Figure 13.6

Le streaming vidéo.

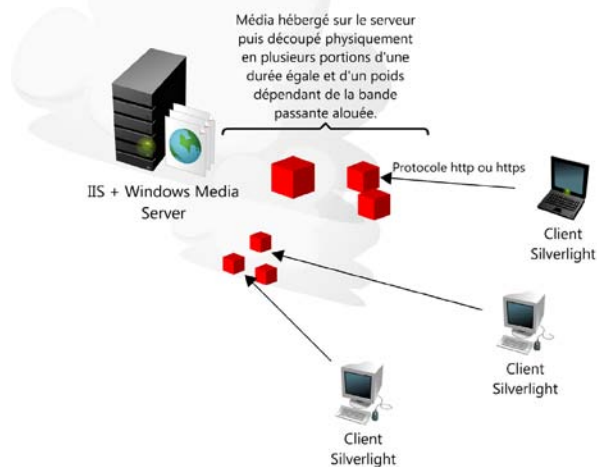


Le streaming est une solution élégante, mais assez coûteuse en terme de ressources côté serveur. Pour les jeux olympiques de Pékin en 2008, Microsoft utilise une nouvelle technologie nommée *Adaptive Streaming*. Cette technologie est également présente pour des technologies gratuites telles que PHP. Il s'agit en réalité d'un mode de téléchargement progressif simulant une lecture en streaming. Dans cette configuration, chaque média est scindé physiquement sur le serveur en de nombreux fragments d'une durée égale. Il est possible de préparer le média pour différentes qualités de bande passante. Sur le serveur, cela se traduit par plusieurs groupes de fragments qui sont organisés selon la bande passante qu'ils ciblent et leur poids qui correspond à un débit spécifique. Du côté client cette configuration est idéale car il suffit de télécharger le fragment qui correspond à un instant (*time code*) spécifique du flux (voir Figure 13.7).

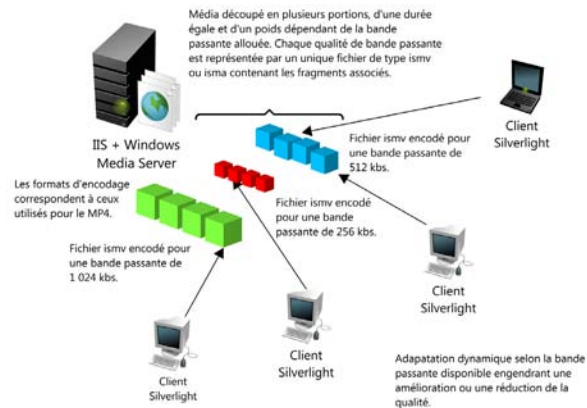
Cette solution, bien que très pratique, recèle toutefois un inconvénient de taille : elle force le serveur à gérer des centaines de fragments sur son disque. Dans le cas des jeux olympiques, les fragments représentaient deux secondes de vidéos. Pour 1 heure de vidéo on obtient donc le calcul : 60 minutes x 60 secondes / 2, ce qui représente 1 800 portions. Si nous recréons ces fragments pour du 512 kbs et du 1 024 kbs, nous obtenons 3 600 fragments à gérer – ce qui est assez conséquent. Le Smooth Streaming est né de cette problématique. L'idée est simple : il s'agit de représenter l'ensemble des fragments d'un même poids au sein d'un seul fichier (voir Figure 13.8).

Figure 13.7*Adaptive Streaming.***Adaptive streaming**

Pseudo diffusion continue simulée côté client. Chaque client télécharge à la demande la portion du média qui correspond à l'instant donné.

**Figure 13.8***Smooth Streaming.***Smooth streaming**

Pseudo diffusion continue simulée côté client. Chaque client télécharge à la demande un fragment au sein d'un fichier média unique.



Cette méthodologie limite les accès disque et améliore les performances côté serveur. Vous devrez déployer différents types de fichiers sur ce dernier :

- Le premier, nommé ismv, contient de la vidéo et de l'audio, ou de la vidéo uniquement. Les fragments qu'il détient sont au format MP4.
- Les fichiers de type isma sont dédiés au contenu audio uniquement.
- Un document déclaratif de type ism est nécessaire, il décrit les relations entre les fichiers et les qualités de bande passante associées. Il est dédié au serveur et ne sera pas lu par le client.

- Le document .ismc décrit les flux disponibles, leur qualité, le type d'encodage utilisé, le cha-pitrage et les informations utiles pour que le poste client puisse lire les médias.

La grande différence avec les précédents systèmes est que l'utilisateur n'a plus besoin de choisir la bande passante qu'il souhaite utiliser. Le type de fragment téléchargé, correspondant à une bande passante spécifique, est choisi de manière dynamique et transparente, selon les capacités de la connexion durant la lecture du média. Générer une vidéo pour Silverlight est une opération assez simple réalisée *via* Expression Encoder. L'utilisation de ce logiciel sort du cadre de ce livre, mais sachez qu'il vous permettra simplement de créer des montages simples, *via* l'ajout d'un générique de début et/ou de fin à la vidéo principale. Grâce à Expression Encoder, il devient très simple d'encoder de la vidéo et du son pour du téléchargement progressif, du streaming ou du Smooth Streaming dans l'ensemble des formats disponibles pour chacun de ces modes de diffusion. Un ensemble de paramètres prédéfinis vous permettront d'encoder la vidéo sans vraiment vous poser de questions dans un premier temps (voir Figure 13.9). Avec le temps il est conseillé de bien comprendre et d'assimiler les réglages de compression afin d'optimiser au maximum le trafic utile.

Figure 13.9

L'interface d'Expression Encoder.



Pour diffuser du Smooth Streaming, il suffit d'installer un serveur IIS 7, Windows Media Services 3 à travers l'application Web Platform Installer disponible gratuitement sur le portail de téléchargement Microsoft : <http://www.microsoft.com/downloads/>. Microsoft diffuse une librairie disponible à l'adresse <http://smf.codeplex.com>. Elle contient un lecteur capable de gérer les listes de lecture, du contenu préparé pour le Smooth Streaming et des vidéos standard. Elle apporte notamment un contrôle de type `MediaElement` qui possède la capacité de lire des médias préparés pour ce mode de diffusion. Il vous suffira de déposer les médias encodés pour le Smooth Streaming directement sur un répertoire du site de test local, d'activer le service de Smooth Streaming dans le gestionnaire IIS et d'y faire appel *via* le `MediaElement` amélioré.

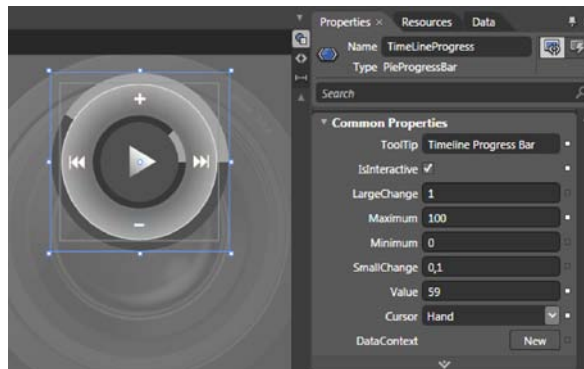
Maintenant que nous avons énuméré les différents types de diffusion, nous allons créer un lecteur vidéo simple à travers la reprise d'un projet réalisé précédemment.

13.1.3 Un lecteur vidéo simple

Afin de partir d'une base existante, vous pouvez télécharger le projet *PlayerVideo_base.zip* disponible dans le dossier *chap13* des exemples. Il est également nécessaire de télécharger le fichier compressé *Assets.zip*. Il contient une vidéo dans différents formats pour une diffusion standard et pour du Smooth Streaming. Ouvrez le projet dans Blend, le lecteur vidéo a été légèrement modifié depuis le Chapitre 7. Au sein de l'arbre visuel, vous trouverez un contrôle personnalisable, typé *PieProgressBar*, que j'ai réalisé pour l'occasion. Il s'agit d'une barre de progression circulaire qui hérite de *RangeBase* à l'instar des contrôles de type *ProgressBar* ou *Slider*. De ce fait, ce contrôle est hérité de nombreux comportements et propriétés inhérents à cette classe abstraite. Il n'est pas fourni en standard dans Silverlight. Sa conception est toutefois possible grâce à l'utilisation de figures géométriques de l'API de dessin Silverlight. Il possède en outre la capacité d'être directement utilisé comme jauge interactive. Il suffit pour cela de passer sa propriété *IsInteractive* à *true* (voir Figure 13.10). Ainsi, lorsque vous cliquerez n'importe où sur sa surface, sa propriété *Value* sera mise à jour.

Figure 13.10

Propriétés du contrôle
PieProgressBar.



Ajoutez une instance de type *MediaElement*, vous trouverez ce contrôle dédié à la lecture des médias dans le panneau *Assets*. Placez-le dans la grille juste au-dessus de l'image d'arrière-plan, en mode de redimensionnement étiré, sans lui donner de marges extérieures. Il est nécessaire de fournir l'adresse d'une vidéo à la propriété *Source* de notre *MediaElement*. Importez la vidéo *PetitPanOcéan.wmv* téléchargée depuis le fichier *Assets.zip* (ou une autre de votre choix) dans le projet. Celle-ci est désormais embarquée à la compilation, ce qui n'est pas l'idéal mais nous permet de débiter le code. Dans tous les cas, il est nécessaire de lire une vidéo chapitrée afin de suivre la totalité de ce tutorial. Nous allons maintenant créer le code logique nécessaire à la lecture de la vidéo. Dans la propriété *Source*, sélectionnez la vidéo *PetitPanOcéan.wmv* dans la liste déroulante.

INFO

Il est également possible de renseigner une URL distante mais nous aborderons le mode asynchrone ultérieurement. De plus, si vous essayez avec l'un des deux formats *mov* ou *mp4* contenus dans l'archive *Assets.zip*, ces derniers n'apparaissent pas dans la liste de choix car les extensions *mov* et *mp4* ne sont pas directement reconnues sous Blend. Ces fichiers ne sont donc pas embarqués dans le fichier *xap*, car étant de type inconnu pour Blend, ils ne sont pas référencés comme ressource de type média.

Commençons par les interactions simples : la gestion de la lecture et du volume. Nous passerons ensuite aux contrôles d'avance et de retour rapide. Ouvrez le projet dans Visual Studio, la classe `MediaElement` possède des méthodes assez simples permettant le contrôle de la tête de lecture. Il nous suffit d'écouter les événements des contrôles afin de les appeler :

```
public MainPage()
{
    InitializeComponent();

    ...

    playPauseBtn.Checked += new RoutedEventHandler(playPauseBtn_Checked);
    playPauseBtn.Unchecked += new RoutedEventHandler(playPauseBtn_Unchecked);
    volumeDownBtn.Click += new RoutedEventHandler(volumeDownBtn_Click);
    volumeUpBtn.Click += new RoutedEventHandler(volumeUpBtn_Click);
}

void playPauseBtn_Unchecked(object sender, RoutedEventArgs e)
{
    mediaElement.Pause();
}

void playPauseBtn_Checked(object sender, RoutedEventArgs e)
{
    mediaElement.Play();
}

void volumeUpBtn_Click(object sender, RoutedEventArgs e)
{
    mediaElement.Volume += 0.10;
    mediaElement.Volume = Math.Min(mediaElement.Volume, 1);
}

void volumeDownBtn_Click(object sender, RoutedEventArgs e)
{
    mediaElement.Volume -= 0.10;
    mediaElement.Volume = Math.Max(mediaElement.Volume, 0);
}
```

Décochez l'option `AutoPlay` du `MediaElement`, de cette manière la vidéo ne sera pas lue par défaut au lancement du projet. Nous pouvons maintenant gérer les marqueurs de temps qui permettent la conception d'un chapitrage. Il nous faut une variable indiquant le numéro du chapitre et récupérer la liste des marqueurs :

```
TimelineMarkerCollection tmc = new TimelineMarkerCollection();

int currentTimeLineMarker = -1;

public MainPage()
{
    InitializeComponent();

    ...

    backwardBtn.Click += new RoutedEventHandler(backwardBtn_Click);
```

```

        forwardBtn.Click += new RoutedEventHandler(forwardBtn_Click);

        mediaElement.MediaOpened += new RoutedEventHandler
            (mediaElement_MediaOpened);
    }

    void backwardBtn_Click(object sender, RoutedEventArgs e)
    {
        if (tmc.Count > 1)
        {
            currentTimeLineMarker = --currentTimeLineMarker >= 0 ?
                currentTimeLineMarker : tmc.Count-1;
            mediaElement.Position = tmc[currentTimeLineMarker].Time;
        }
    }

    void forwardBtn_Click(object sender, RoutedEventArgs e)
    {
        if (tmc.Count > 1)
        {
            currentTimeLineMarker = ++currentTimeLineMarker % mediaElement.
                Markers.Count;
            mediaElement.Position = tmc[currentTimeLineMarker].Time;
        }
    }

    void mediaElement_MediaOpened(object sender, RoutedEventArgs e)
    {
        tmc = mediaElement.Markers;
        if (tmc.Count < 2)
        {
            backwardBtn.Visibility = Visibility.Collapsed;
            forwardBtn.Visibility = Visibility.Collapsed;
        }
        else
        {
            backwardBtn.Visibility = Visibility.Visible;
            forwardBtn.Visibility = Visibility.Visible;
        }
    }
}

```

Le code ci-dessus est assez simple : nous modifions l'index du marqueur en cours, puis nous déplaçons la tête de lecture de la vidéo à la durée indiquée par le marqueur de temps ciblé. La propriété `Position` du `MediaElement` attend une instance de type `TimeSpan`. On peut donc directement lui affecter la propriété `Time` du nouveau marqueur cible. Les deux expressions incrémentant ou décrémentant l'index du marqueur en cours permettent de boucler dans la collection de marqueurs contenue dans le `MediaElement`. Pour finir, si ce dernier possède plus d'un marqueur de temps, nous rendons accessibles les boutons `forwardBtn` et `backwardBtn`. Dans le cas contraire, autant les cacher pour éviter que l'utilisateur ne clique dessus inutilement. Le mieux est de réaliser cette opération lorsque le média est complètement ouvert, et que la liste des marqueurs de temps est récupérée. Il est également indispensable de récupérer l'index du marqueur atteint lorsque la vidéo est en cours de lecture. Nous pourrions ainsi mettre à jour notre variable permettant de gérer les boutons `forwardBtn` et `backwardBtn`. Pour cela, nous pouvons écouter l'événement `MarkerReached` du `MediaElement` :

```

public MainPage()
{
    InitializeComponent();
}

```

```

...

mediaElement.MarkerReached += new TimelineMarkerRoutedEventHandler
    (media Element_MarkerReached);

}

void mediaElement_MarkerReached(object sender, TimelineMarkerRoutedEventArgs e)
{
    //l'instruction ci-dessous ne fonctionne pas
    //car la référence est perdue :
    //currentTimeLineMarker = tmc.IndexOf(e.Marker);

    //il est donc nécessaire d'utiliser une boucle
    //afin de trouver le bon marqueur
    foreach (TimelineMarker tm in tmc)
    {
        if (tm.Time == e.Marker.Time)
        {
            currentTimeLineMarker = tmc.IndexOf(tm);
            break;
        }
    }
}
}

```

Il peut être utile de récupérer le contenu de la propriété `Text` liée à chaque marqueur, ainsi que la vignette au format `jpeg` qui y est associée. De cette manière, il serait possible d'afficher les différents chapitres contenus dans la vidéo. Nous n'allons pas nous attarder sur cette fonctionnalité, à la place, nous allons voir comment contrôler la barre de progression circulaire nommée `timeLineProgress`. Celle-ci indique à l'utilisateur le temps écoulé depuis le début. Toute la problématique consiste à récupérer la position courante de la tête de lecture. Aucun événement n'existe à cette fin, il nous faut donc utiliser une instance de type `DispatcherTimer` pour rafraîchir cette valeur. Celle qui est déjà présente ne nous est d'aucune aide car elle concerne la gestion de la disparition des contrôles après un laps de temps d'inactivité. Nous devons modifier plusieurs éléments dans notre code. Dans un premier temps nous devons initialiser correctement les valeurs maximum est minimum de l'instance de `PieProgressBar`, cela peut être réalisé dans l'événement `MediaOpened` :

```

void mediaElement_MediaOpened(object sender, RoutedEventArgs e)
{
    tmc = mediaElement.Markers;

    if (tmc.Count == 0)
    {
        backwardBtn.Visibility = Visibility.Collapsed;
        forwardBtn.Visibility = Visibility.Collapsed;
    }
    else
    {
        backwardBtn.Visibility = Visibility.Visible;
        forwardBtn.Visibility = Visibility.Visible;
    }

    timeLineProgress.Minimum = 0;

    timeLineProgress.Maximum = mediaElement.NaturalDuration.TimeSpan.
TotalMilliseconds;
}

```


La propriété `NaturalDuration` permet de récupérer le temps total de la vidéo, nous pouvons le traduire sous forme de millisecondes ce qui est approprié si nous souhaitons obtenir une définition relativement bonne. Dans un second temps, nous devons ajouter une instance de `DispatcherTimer` comme champ privé de notre classe principale, puis écouter l'événement `Tick` :

```
//on crée un métronome pour espionner la position de la tête de lecture
DispatcherTimer dtPos = new DispatcherTimer()
{ Interval = TimeSpan.FromMilliseconds(250) };

public MainPage()
{
    InitializeComponent();

    ...

    dtPos.Tick += new EventHandler(dtPos_Tick);

    ...
}

void dtPos_Tick(object sender, EventArgs e)
{
    timelineProgress.Value = mediaElement.Position.TotalMilliseconds;
}
```

Dans le code ci-dessous, nous définissons un métronome qui diffusera l'événement `Tick` quatre fois par seconde. Au sein de la méthode d'écoute, nous affectons le laps de temps traduit en millisecondes à la propriété `Value` de notre `PieProgressBar`. Il est ensuite nécessaire d'arrêter le métronome lorsque la vidéo est en pause et de le redémarrer lorsqu'elle est lue :

```
void playPauseBtn_Unchecked(object sender, RoutedEventArgs e)
{
    mediaElement.Pause();
    dtPos.Stop();
}

void playPauseBtn_Checked(object sender, RoutedEventArgs e)
{
    mediaElement.Play();
    dtPos.Start();
}
```

Il nous reste encore une tâche à accomplir. La propriété `IsInteractive` de `timelineProgress` est égale à `true`. Ceci signifie que nous pouvons changer sa valeur en cliquant n'importe où sur la barre. Nous pourrions ainsi modifier la position de la tête de lecture dynamiquement. Nous n'avons qu'à écouter l'événement `ValueChanged` de cette dernière pour obtenir ce résultat :

```
public MainPage()
{
    InitializeComponent();

    ...

    timelineProgress.ValueChanged += new RoutedPropertyChangedEventHandler<double>(timelineProgress_ValueChanged);

    ...
}
```

```

void timeLineProgress_ValueChanged(object sender,
    RoutedEventArgs<double> e)
{
    if (mediaElement.Position.TotalMilliseconds != e.NewValue)
    {
        mediaElement.Position = TimeSpan.FromMilliseconds(e.NewValue);
    }
}

```

Comme vous le constatez, il faut faire attention à ne pas réaffecter deux fois la propriété `Position` entre les événements écoutés `Tick` et `ValueChanged`. Ce dernier ne doit être réellement utilisé que pour les changements importants qui sont dus au clic de l'utilisateur sur la ligne de temps. C'est pour cette raison que nous comparons les deux valeurs au sein d'une condition avant de réaffecter la position de la tête de lecture. Pour bien faire, il nous faut encore une fois boucler au sein de la liste des marqueurs afin de récupérer l'index de celui situé juste avant la position de la tête de lecture :

```

void timeLineProgress_ValueChanged(object sender,
    RoutedEventArgs<double> e)
{
    if (mediaElement.Position.TotalMilliseconds != e.NewValue)
    {
        mediaElement.Position = TimeSpan.FromMilliseconds(e.NewValue);

        foreach (TimelineMarker tm in tmc)
        {
            if (mediaElement.Position.TotalMilliseconds
                < tm.Time.TotalMilliseconds )
            {
                currentTimeLineMarker = tmc.IndexOf(tm)-1;
                break;
            }
        }
    }
}

```

Le lecteur est presque terminé, nous devons encore gérer le chargement de la vidéo de manière dynamique. Vous trouverez le lecteur en dans le dossier *chap13 : PlayerVideo_baseControl.zip*.

13.1.4 Chargement dynamique de vidéos

Créer un lecteur de flux vidéo utilisant le téléchargement progressif est une opération assez simple : il faut alimenter le `MediaElement` avec un flux binaire adapté. Il suffit pour cela d'affecter sa propriété `Source` avec une instance de type `Uri` pointant vers un fichier vidéo existant. Avant de commencer, nous allons modifier légèrement notre application afin de pouvoir la réutiliser ultérieurement. Après tout, une application Silverlight étant un composant de type `UserControl`, nous pouvons imaginer qu'il peut être pratique d'en instancier plusieurs exemplaires dans un autre projet.

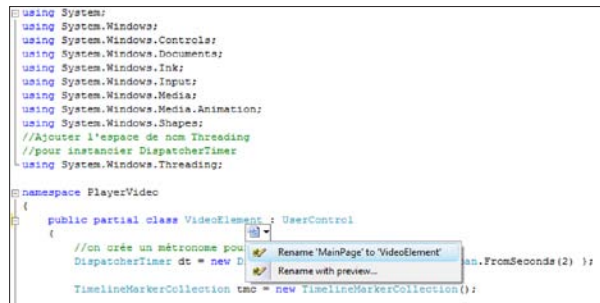
Cela est particulièrement vrai dans notre cas, car notre `UserControl` (`MainPage`) est dédié à une tâche bien spécifique : la lecture de médias vidéo. Pour arriver à nos fins, nous allons faire un peu de *refactoring* sous Visual Studio. Ce concept très puissant, complètement intégré dans Visual Studio, permet de modifier des définitions ou des noms d'objets logiques de manière intelligente. Il

ne s'agit pas d'un simple chercher/remplacer mais plutôt d'une opération très élaborée capable de réinterpréter le code logique et de le modifier en conséquence. Dans le fichier `MainPage.xaml.cs`, remplacez le nom de la classe `MainPage` par `VideoElement`. Une petite barre rouge apparaît juste en dessous du nom modifié. Cliquez dessus, puis sélectionnez le menu `Rename 'Video.MainPage' to 'Video.VideoElement'` (voir Figure 13.11).

Une fois cette opération accomplie, recompilez *via* le raccourci `Ctrl+Maj+B` pour vérifier qu'aucune erreur n'est levée. Dans l'explorateur de projet situé à droite dans l'interface de Visual Studio, modifiez également le nom du fichier `MainPage.xaml` par `VideoElement.xaml`. Le fichier C# associé est également modifié. Redéfinir le nom de l'application nous permettra par la suite d'instancier autant de lecteurs vidéo que nous en avons besoin. Testez le projet afin de vérifier son bon fonctionnement. Définissez une nouvelle propriété de dépendance nommée `StringUriSource`. Lorsque celle-ci sera modifiée, la méthode de rappel associée nous permettra de redéfinir la propriété `Source` du `MediaElement`.

Figure 13.11

*Renommer la classe
MainPage sous
Visual Studio.*



Le code ci-dessous décrit son implémentation au sein de la classe `VideoElement` :

```
public string StringUriSource
{
    get { return (string)GetValue(StringUriSourceProperty); }
    set { SetValue(StringUriSourceProperty, value); }
}

public static readonly DependencyProperty StringUriSourceProperty =
    DependencyProperty.Register(
        "StringUriSource",
        typeof(string),
        typeof(VideoElement),
        new PropertyMetadata( string.Empty, new PropertyChangedCallback
            (OnUriChanged)));

private static void OnUriChanged( DependencyObject d,
                                   DependencyPropertyChangedEventArgs e)
{
    VideoElement ve = d as VideoElement;

    ve.mediaElement.Source =
        new Uri(e.NewValue as string, UriKind.RelativeOrAbsolute);
}
```

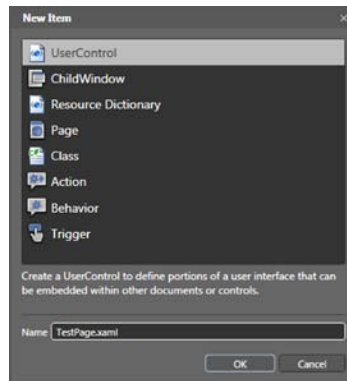
Afin de tester cette propriété, ajoutez un nouveau `UserControl` de test (voir Figure 13.12).

Au sein du fichier de code logique App.xaml.cs, il est nécessaire de remplacer le type du UserControl chargé, VideoElement, par TestPage. La ligne à modifier est centralisée dans la méthode Application_Startup :

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new TestPage();
}
```

Figure 13.12

Ajout d'une page de test.



Au sein de VideoElement.xaml, sélectionnez le MediaElement, et videz l'affectation de sa propriété Source. Il est nécessaire de créer une instance de la classe VideoElement dans la grille principale de TestPage. Vous pouvez accomplir cette opération grâce au panneau Assets *via* la catégorie Project. Sélectionnez l'instance de VideoElement, ouvrez le panneau Properties et dans la catégorie Miscellaneous entrez PetitPanOcéan.wmv. Recompilez le projet, cette fois, la vidéo est lue *via* l'instance de VideoElement présente dans la page de test. Nous pouvons ajouter sereinement l'ensemble du code permettant la gestion du téléchargement, puis écouter les événements MediaFailed, DownloadProgressChanged et MediaEnded. Ceux-ci sont respectivement diffusés en cas d'échec de la lecture, durant la progression du téléchargement, et lorsque la lecture du média est terminée :

```
public VideoElement()
{
    InitializeComponent();

    ...

    mediaElement.MediaFailed += new EventHandler<ExceptionRoutedEventArgs>
        (mediaElement_MediaFailed);

    mediaElement.MediaEnded += new RoutedEventHandler
        (mediaElement_MediaEnded);

    mediaElement.DownloadProgressChanged += new
        RoutedEventHandler(mediaElement_DownloadProgressChanged);
}

void mediaElement_DownloadProgressChanged(object sender, RoutedEventArgs e)
{
```

```
        downloadProgress.Value = mediaElement.DownloadProgress;
    }

    void mediaElement_MediaEnded(object sender, RoutedEventArgs e)
    {
        mediaElement.Stop();
    }

    void mediaElement_MediaFailed(object sender, ExceptionRoutedEventArgs e)
    {
        Debug.WriteLine
            ("Le format de la vidéo et/ou son adresse sont incorrects.");
    }
}
```

Assurez-vous que la propriété `Maximum` de l'instance `downloadProgress` est égale à 1. Cela est important car le média élément renvoie la progression du téléchargement sous forme d'un chiffre exprimé en pourcentage de 0 à 1 (à travers sa propriété `DownloadProgress` qu'il ne faut pas confondre avec l'instance `PieProgressBar`).

Pour tester tous ces comportements, vous pouvez utiliser l'adresse de la vidéo suivante qui est en ligne : http://www.tweened.org/wp-content/uploads/videos/14_advanceVisualStateManager.wmv. Il s'agit d'un webcast que j'ai réalisé durant l'été 2008 et dont le but était de donner les bases de Silverlight. Il est également possible de donner à l'internaute une idée du remplissage de la mémoire tampon allouée. Ainsi, un texte pourrait apparaître lorsque le buffer se remplit. Le projet finalisé *PlayerVideo_final.zip* est dans le dossier *chap13*.

13.2 Conception MVVM

Jusqu'à présent, nous avons fortement couplé les vues au code logique gérant nos interfaces. Nous allons éviter cet écueil, dans cette section, grâce à l'utilisation du modèle de conception *Modèle Vue Vue-Modèle*. Ce motif est un dérivé de MVC (*Modèle Vue Contrôleur*) propre à Silverlight et WPF. Il permet de séparer complètement la logique régissant les vues de celle régissant la fonctionnalité pure. Une fois cette architecture mise en place, nous y intégrerons facilement le chargement de données externes. Téléchargez le projet *MVVM_PatternBase.zip* du dossier *chap13* des exemples.

13.2.1 Principes

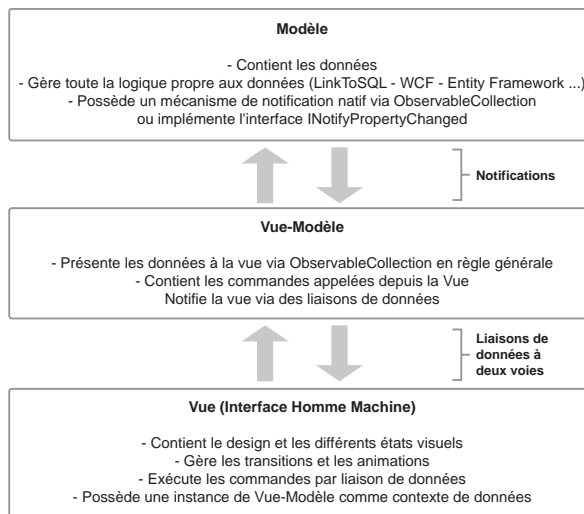
Concrètement, le modèle de conception MVVM sépare le code en trois sous-ensembles, *Modèle*, *Vue* et *Vue-Modèle*, qui possèdent une responsabilité spécifique. Une vue représente une interface graphique (ou IHM) liée à un flux d'utilisation. Dans MVVM, le *Modèle* est représenté par l'ensemble des classes et collections contenant les données et le code logique assurant la gestion du chargement de données distantes. La partie *Vue-Modèle* regroupe les classes dont le but est d'établir une passerelle entre *Vue* et *Modèle* et de gérer la fonctionnalité côté *Vue*. *Vue-Modèle* a donc deux responsabilités :

- Appeler des méthodes ou récupérer des données contenues par le *Modèle* lorsque la vue en a besoin.
- Rendre accessible et présenter les données à la vue de manière efficace en notifiant cette dernière lorsque les données changent.

La vue gère toutes les interactions utilisateur et exécute les commandes par un mécanisme de liaison. Elle possède un minimum de code logique qui se traduit souvent par une affectation de sa propriété `DataContext` à une instance de `Modèle-Vue`. Finalement, `Vue-Modèle` est pleinement responsable de la communication entre les couches basses et hautes de l'application (voir Figure 13.13). La liaison de données est très importante dans ce contexte car cela permet à `Modèle-Vue` de notifier la vue sans effort, et inversement dans le cas d'une liaison à deux voies.

Figure 13.13

Principes du modèle de conception MVVM.



Les principaux avantages de cette architecture sont de faciliter la maintenance, la mise à jour, la réutilisation et de simplifier la conception. En interne MVVM utilise le modèle de conception `Command`. Une commande est une instruction connue par la vue et dont l'exécution est définie sur `Vue-Modèle`. Au final, la Vue sait ce qu'elle souhaite accomplir mais n'a pas besoin de connaître les détails d'implémentation – ce principe fait référence au concept d'encapsulation.

C'est en suivant ces principes que nous allons finaliser le projet `MVVM_PatternBase` que vous avez téléchargé. Dans ce dernier, vous trouverez un répertoire `SLExtensionsMVVM` reprenant une infime partie des classes proposées par la bibliothèque `SLExtensions` que vous trouverez sur *CodePlex*. Vous trouverez également deux répertoires correspondant aux classes propres au `Modèle` et à `Vue-Modèle` (voir Figure 13.14).

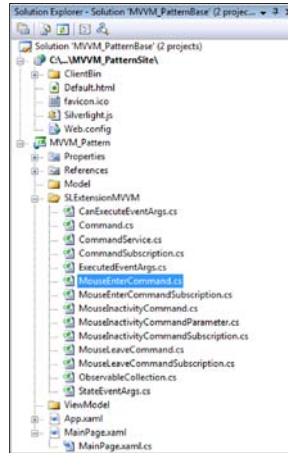
Il n'est pas forcément nécessaire de stocker les vues dans un répertoire dédié puisque la racine du projet peut les centraliser. Ce projet est une base adéquate pour apprendre MVVM.

INFO

L'une des critiques que l'on pourrait faire consiste à dire que la Vue connaît la commande alors qu'elle pourrait diffuser des événements qui exécutent les commandes. Toutefois, cette remarque n'est pas pertinente car le code logique gérant l'écoute de l'événement est forcément situé sur le processus (*Thread*) graphique donc la vue.

Figure 13.14

Arborescence du projet
MVVM_PatternBase.



13.2.2 Modèle

En règle générale, les collections de données fictives (voir Chapitre 11) sont conçues de manière éclairée par le développeur. Les éléments de ces collections correspondent très souvent à des objets métiers. Un objet métier est une classe spécifique à une problématique métier. Les informations relatives aux clients d'une entreprise de services seront ainsi stockées grâce une classe métier dédiée : *Customer*. Celle-ci possède diverses propriétés comme le nom du client, son titre, son numéro de compte client ou son e-mail. La problématique consiste souvent à transformer (sérialiser) un enregistrement récupéré depuis une source de donnée quelconque, en un objet métier côté client utilisable par la vue. En règle générale, on sépare logique métier et logique valeur.

Un objet valeur (*Value Object*), quant à lui, a pour objectif de gérer l'une de ces propriétés. Par exemple, la classe *Email* aurait pour but de stocker la valeur de l'e-mail client fournit, de vérifier sa validité et de lever les exceptions le cas échéant. Nous n'entrerons pas dans ces considérations car cela est un peu éloigné du sujet traité.

Nous allons nous contenter de créer un objet dont le but sera de centraliser toutes les propriétés d'un *Media*. Dans le contexte MVVM, cet objet fait partie du *Modèle*. Ouvrez le projet téléchargé à la fois dans Blend et dans Visual Studio, puis créez les classes *Author* et *Media* dans le répertoire *Model*. La première correspond à la description d'un auteur, la seconde contient l'ensemble des données utiles à la lecture du média. Voici l'implémentation de la classe *Author* :

```
public class Author
{
    public string Nom { get; set; }

    public string Prenom { get; set; }

    public string Blog { get; set; }

    public Author(string nom, string prenom)
    {
        this.Nom = nom;

        this.Prenom = prenom;
    }
}
```

```

    }

    public Author(string nom, string prenom, string blog)
    {
        this.Nom = nom;

        this.Prenom = prenom;

        this.Blog = blog;
    }

    public override string ToString()
    {
        return this.Prenom + " " + this.Nom;
    }

    public string ToString(bool displayBlog)
    {
        if (this.Blog == string.Empty || !displayBlog)
        {
            return this.Prenom + " " + this.Nom;
        }
        else
        {
            return this.Prenom + " " + this.Nom + " - " + this.Blog;
        }
    }
}

```

La classe `Author` est très simple et possède deux constructeurs selon que vous souhaitiez fournir ou non le titre du blog de l'auteur. Si vous utilisez C# 4, il est possible d'utiliser un paramètre optionnel ce qui est plus élégant dans ce cas. La première méthode `ToString()` surcharge celle héritée de `Object`. La seconde est propre à l'objet `Author` et accepte un paramètre permettant d'indiquer si vous souhaitez renvoyer le titre du blog. Le code suivant expose la classe `Media` :

```

public class Media
{
    public string Titre { get; set; }
    public string Description { get; set; }
    public TimeSpan Duree { get; set; }
    public string Date { get; set; }
    public string BaseUrl { get; set; }
    public string FichierMedia { get; set; }
    public string Url
    {
        get { return BaseUrl + FichierMedia; }
    }

    public string FichierVignette { get; set; }
    public string UrlVignette
    {
        get { return BaseUrl + FichierVignette; }
    }

    public Author Auteur { get; set; }
    public override string ToString()
    {
        return Titre;
    }

    public string ToString(bool displayAuthor)

```



```

    {
        if (Auteur.ToString() == string.Empty || !displayAuthor)
        {
            return Titre;
        }
        else
        {
            return Titre + " - " + Auteur.ToString(false);
        }
    }
}

```

Là encore, on surcharge la méthode `ToString()` afin d'avoir une représentation simple d'un `Media`. Cela sera particulièrement utile lorsque nous souhaiterons afficher la liste des médias au sein d'une instance de `ListBox`.

INFO

La bonne pratique consiste à centraliser les classes dans des espaces de nom dédiés à chaque ensemble défini par MVVM. Les classes du Modèle seront donc stockées dans l'espace de noms `MVVM_Pattern.Model`, celles correspondantes à l'ensemble Vue-Modèles seront quant à elles dans `MVVM_Pattern.ViewModel`.

Il nous reste maintenant à créer une classe qui centralisera nos données et s'occupera des appels distants. Dans un premier temps, nous allons juste créer le squelette de cette classe et gérer des données en dur. Nous pouvons appeler cette classe `MediasDatas`, celle-ci est assez simple :

```

public static class MediasDatas
{
    public static ObservableCollection<Media> Datas { get; set; }

    static MediasDatas()
    {
        Datas = new ObservableCollection<Media>();
    }

    public static void GetAllMedias()
    {
        Datas.Add(new Media()
        {
            BaseUrl = "http://www.tweened.org/wp-content/uploads/videos/",
            FichierMedia = "15_createJsonStream.wmv",
            Auteur = new Author ("Ambrosi", "Eric", "Tweened.org"),
            Description="C'est une jolie vidéo",
            Duree=TimeSpan.FromSeconds(312),
            Titre="Créer un flux JSON",
            Date="10 Juillet 2008"
        });
    }
}

```

En premier lieu la classe `MediasDatas` est statique, il n'est pas nécessaire d'en créer des instances puisqu'elle symbolise une source de données unique. Elle possède une propriété de type générique `ObservableCollection<object>`. Contrairement à une collection `List` standard, celle-ci notifie les objets qui y font référence à travers la liaison de données lorsqu'elle est modifiée d'une quelconque façon (élément ajouté ou supprimé par exemple). Ces derniers sont donc mis à jour

dynamiquement grâce à une liaison de modèles OneWay, la vue peut modifier ce type de collection à travers une liaison à deux voies (TwoWay). Dans ce contexte, elle implémente les interfaces `INotifyPropertyChanged` et `INotifyCollectionChanged` nativement.

Nous avons ajouté un constructeur statique qui affecte une instance de `ObservableCollection<Media>`. Cela évite par la suite de tester si le membre statique `Datas` est `null`, et de lui affecter l'instance le cas échéant. Dans la méthode statique `GetAllMedias`, nous simulons la réception de données en ajoutant un élément en dur. Nous avons accompli la première partie de la conception MVVM, nous allons maintenant nous occuper des parties Vue-Modèle et Vue.

13.2.3 Vue-Modèle

Comme nous l'avons précisé plus haut, Vue-Modèle possède deux facettes différentes. Elle doit tout d'abord présenter les données à la vue et notifier celle-ci lorsque les données évoluent. Les classes qui font partie de Vue-Modèle implémentent à cette fin l'interface `IPropertyNotifyChanged` permettant de gérer la liaison de données. C'est une bonne pratique à respecter, et cela devient indispensable si vous utilisez des types ne possédant pas de mécanismes de notification par défaut. Le plus simple est d'étendre une classe, par exemple `ViewModel`, qui implémente l'interface par défaut :

```
namespace MVVM_Pattern.ViewModel
{
    public class ViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        public void NotifyPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged
                    (this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

Pour l'instant, rien de très difficile, nous avons déjà implémenté cette interface dans le chapitre précédent. Voyons maintenant la classe `MediaViewModel` qui joue un rôle concret dans notre application :

```
namespace MVVM_Pattern.ViewModel
{
    public class MediaViewModel : ViewModel
    {
        public ObservableCollection<Media> Medias { get; set; }
        public ICommand LoadMedias { get; private set; }
        public MediaViewModel()
        {
            LoadMedias = new Command(DoGetMedias);

            Medias = MediasDatas.Datas;
        }
    }
}
```

```
    }

    public void DoGetMedias()
    {
        MediasDatas.GetAllMedias();

        //dans ce cas, Notifier la vue n'est pas utile puisque
        //Medias est une collection observable implémentant
        //nativement IPropertyNotifyChanged
        //NotifyPropertyChanged("Medias");
    }
}
}
```

Celle-ci est très simple, elle expose deux membres importants : une collection observable nommée *Datas*, ainsi qu'une commande *LoadMedias*. La propriété *ItemsSource* de la *ListBox* présente dans notre vue (*MainPage*) sera liée à la collection *Datas*. La commande est, quant à elle, liée à une propriété attachée *CommandServices.Command* que nous allons définir sur le bouton.

INFO

La commande est exécutée sur l'événement *Click* mais il est possible de la déclencher à travers n'importe quel événement en utilisant des comportements. La bibliothèque *SLExtensions* fournit tout ce qu'il faut à cette fin.

Lorsque nous affectons la commande, nous passons dans le constructeur la référence de la méthode de *MediaViewModel* que celle-ci (la commande) doit déclencher. Dans notre cas, il s'agit de la méthode *DoGetMedias* qui fait un appel de la méthode statique *GetAllMedias*. Comme nous l'avons vu précédemment dans la classe *MediasDatas*, *GetAllMedias* modifie la collection contenue par le membre statique *Datas*. Comme la propriété statique *Datas* est affectée à la propriété *Medias* de *MediaViewModel*, la collection *Medias* est bien mise à jour. Celle-ci implémente tout ce qu'il faut pour notifier la vue. Dans ce cas, il n'est donc pas nécessaire d'utiliser *NotifyPropertyChanged*.

A contrario, si nous avions utilisé une liste dans *MediaViewModel* en lieu et place d'une instance de type *ObservableCollection*, cela aurait été indispensable. Il nous reste à modifier légèrement le code logique et déclaratif de notre vue, *MainPage*. La première chose que nous pouvons faire consiste à affecter une instance de *MediaViewModel* comme contexte de données :

```
public MainPage()
{
    InitializeComponent();

    this.DataContext = new MediaViewModel();
}
```

Dans l'idéal, c'est le seul code logique qui doit être présent dans la vue.

ATTENTION

Il ne faut pas privilégier une certaine pureté de code au détriment de la productivité ou des contraintes liées au design et à l'expérience utilisateur. Cela peut faire l'objet d'un long et douloureux débat. Dans tous les cas, il ne faut pas perdre de vue le résultat, ce que voit et ce que ressent le client et l'utilisateur final. Il vaut mieux faire, puis critiquer et corriger à partir d'une première base, qu'essayer de faire une application parfaite d'un seul coup. Cela vous permettra également de faire participer le client à divers niveaux.

Modifions maintenant le code déclaratif. La première chose à faire consiste à référencer l'espace de noms `SLExtensions` dans le `UserControl` racine :

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:Slei="clr-namespace:SLExtensions.Input;assembly=MVVM_Pattern"
  x:Class="MVVM_Pattern.MainPage"
  Width="640" Height="480">
```

Vous remarquez que la bibliothèque référencée correspond à celle de notre projet. C'est tout à fait logique car nous utilisons les sources dont nous avons besoin directement dans notre projet au lieu de référencer des bibliothèques (dll). Il nous reste maintenant à référencer la commande `LoadMedias` dans notre bouton, ainsi qu'à créer une liaison de données entre la `ListBox` et la propriété `Medias` contenue par `MediaViewModel` :

```
<Grid x:Name="LayoutRoot" Background="White">
  <ListBox ItemsSource="{Binding Path=Medias}" ... />
  <Button Slei:CommandService.Command="{Binding LoadMedias}" ... />
</Grid>
```

Vous pouvez compiler l'application et effectuer un premier test. Lorsque vous cliquez sur le bouton, la commande est invoquée, les données sont récupérées et la `ListBox` reçoit les données par la liaison de données. Comme la classe `Media` surcharge `ToString()`, qui renvoie la propriété `Titre`, le titre du média ajouté dans `GetAllMedias` est affiché. Vous pourriez créer un modèle de données (`DataTemplate`) afin d'afficher toutes les propriétés du média. Il est également possible de passer un paramètre à la commande invoquée, celui-ci sera typé `object` au sein de la méthode déclenchée par la commande. Le projet complet *MVVM_Pattern.zip* se trouve dans le dossier *chap13* des exemples.

Dans la prochaine section, nous allons profiter de cette architecture pour nous concentrer sur la classe `MediasDatas` qui va gérer le chargement dynamique de contenu XML.

13.3 Chargement de données

Microsoft apporte de nombreuses solutions répondant à cette problématique. Ainsi la bibliothèque *RIA Services* pour Silverlight, *Windows Communication Foundation* ou *ADO.NET Entity Framework* sont quelques-unes des solutions vous permettant de gérer ces échanges de données. De nombreux livres abordent l'échange de données entre clients et serveurs, toutefois, le livre de John Papa, *Data Driven Services* aux éditions O'Reilly Media est une référence dans ce domaine. Je vous conseille vivement sa lecture si vous souhaitez apprendre de manière approfondie les différentes technologies proposées par Microsoft. De notre côté, nous nous placerons du point de vue Silverlight uniquement, nous n'étudierons pas le code logique côté

serveur. Nous aborderons l'objet asynchrone `WebClient`, le modèle de conception MVVM, LINQ et les expressions lambda. Pour finir, nous apprendrons à consommer des flux au format XML et JSON et nous ferons un rapide tour d'horizon des contraintes de sécurité liées à l'échange de données interdomaines.

13.3.1 L'objet *WebClient*

Nous allons récupérer du contenu distant au format XML. Silverlight est une technologie ouverte, qu'il est facile d'intégrer dans un environnement libre. Le XML est donc largement supporté à travers les services web ou la récupération de flux RSS. Dans cette section, nous traiterons le chargement de données XML distantes à travers l'objet simplifié `WebClient`. Nous envisagerons ensuite différents moyens de consommer et de sérialiser ces données *via* l'utilisation d'objets métier et d'attributs de sérialisation.

L'objet `WebClient` est contenu dans l'espace de noms `System.Net`. Il possède différentes méthodes asynchrones dont le but est de télécharger ou d'envoyer des données. Le contenu peut être perçu de deux manières différentes : en tant que chaîne de caractères ou en tant que flux binaire. Cela dépendra de la méthode asynchrone que vous utiliserez, elles sont au nombre de quatre : `DownloadStringAsync`, `UploadStringAsync`, `OpenReadAsync` ou `OpenWriteAsync`. Généralement le fichier distant à télécharger est l'un de ceux listés ci-dessous :

- un fichier de texte brut, une extension `txt`, par exemple ;
- un document au format XML, XAML, etc. ;
- une chaîne de caractères présentée sous forme de JSON – nous aborderons JSON ultérieurement ;
- un script côté serveur renvoyant l'un des trois types précédents.

Dans tous les cas, la chaîne de caractères contenue doit être encodée, de préférence, au format UTF8. Cet encodage permet à Silverlight d'interpréter les caractères spéciaux (français par exemple) correctement. Il supporte également les données texte encodées en UTF16 (*Big Endian unicode*) et en Unicode. Les différents types de données récupérées ou envoyées sous forme de flux binaire, sont listés ci-dessous :

- Tous types de formats texte listés précédemment. Il n'y a pas de différence de performances, qui peut le plus peut le moins.
- Les médias (sons, vidéos, les images). Pour affecter le flux binaire récupéré à l'instance de type `Image` ou `MediaElement`, il est nécessaire d'utiliser leur méthode `SetSource`. L'objet `WebClient` offre plus de capacités que `BitmapImage` ou `MediaElement`. Il peut, par exemple, annuler le téléchargement du média *via* la méthode `CancelAsync`.
- Des applications ou des bibliothèques Silverlight compilées. Silverlight 4 offre de ce point de vue de nombreuses possibilités que nous n'abordons pas dans ce chapitre.
- Des données transmises directement dans un format binaire compressé *via* des services de type *Windows Communication Foundation* ou SOAP, par exemple. Dans ce cas, les échanges client-serveur sont largement optimisés car Silverlight est capable d'interpréter le format binaire directement.

- Les fichiers compressés de type zip sont également téléchargeables. Silverlight est capable de les parcourir pour récupérer les documents qu'ils contiennent. Télécharger un unique fichier compressé est toujours plus efficace que d'en charger des centaines en termes de bande-passante.

INFO

L'objet `WebClient` possède la capacité de ne pas bloquer le processus (*Thread*) à l'intérieur duquel la méthode asynchrone est appelée. Cela peut paraître évident sur d'autres plateformes, toutefois Silverlight étant une technologie multithread, Microsoft aurait pu fournir une classe moins conviviale et déléguer cette gestion au développeur.

En interne, `WebClient` utilise la classe `WebRequest` dont le fonctionnement s'articule autour de méthodes de rappel asynchrones. Son principal avantage est d'être simple d'utilisation d'un point de vue code car le code complexe est encapsulé. Toutefois, la classe `WebClient` n'offre pas autant de possibilités que peut le faire une classe comme `HttpWebRequest` (héritant de `WebRequest`). `HttpWebRequest` fournit un contrôle plus fin du déroulement des appels.

Téléchargez la solution *MVVM_Pattern.zip* ou réutilisez le projet finalisé de la dernière section. Placez le fichier XML *videos.xml.zip* dans le répertoire `ClientBin` du projet `MVVM_PatternSite`. Le code logique assurant l'appel se situe dans la classe `MediasDatas`, voici comment récupérer le document XML distant :

```
namespace MVVM_Pattern.Model
{
    public static class MediasDatas
    {
        protected static int Percent = 0 ;

        public static Uri URL_MEDIA =
            new Uri("videos.xml", UriKind.RelativeOrAbsolute);

        private static WebClient wc = new WebClient();

        public static ObservableCollection<Media> Datas { get; set; }

        static MediasDatas()
        {
            Datas = new ObservableCollection<Media>();

            wc.OpenReadCompleted +=
                new OpenReadCompletedEventHandler(wc_OpenReadCompleted);

            wc.DownloadProgressChanged +=
                new DownloadProgressChangedEventHandler(wc_DownloadProgressChanged);

            wc.Encoding = Encoding.UTF8;
        }

        static void wc_DownloadProgressChanged(object sender,
            DownloadProgressChangedEventArgs e)
        {
            //throw new NotImplementedException();
            Percent = e.ProgressPercentage;
        }
    }
}
```

```

static void wc_OpenReadCompleted(object sender,
                                OpenReadCompletedEventArgs e)
{
    if (e.Error != null)
    {
        throw e.Error;
    }
    else if (e.Cancelled)
    {
        //l'appel asynchrone a été annulé
        //via la méthode CancelAsync
        //ou une erreur de sécurité est levée
    }
    else
    {
        //on reçoit les données contenues dans e.Result
    }
}

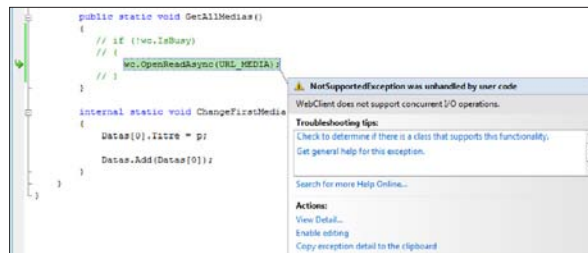
public static void GetAllMedias()
{
    if (!wc.IsBusy)
    {
        wc.OpenReadAsync(URL_MEDIA);
    }
}
...

```

Tout d'abord, nous définissons une instance de WebClient comme champ static de la classe MediasDatas. Au sein du constructeur static, nous écoutons les événements géant la progression et la réception des données. Il n'y a pas réellement de méthode dédiée aux erreurs de téléchargement. Tout se déroule dans OpenReadCompleted. L'objet événementiel reçu contient trois propriétés définissant le type de résultat : Result, Error et Cancelled. Nous pouvons traiter le résultat facilement à partir de là. La méthode GetAllMedias est légèrement changée au lieu de modifier la collection en dur : nous réalisons l'appel asynchrone *via* la méthode OpenReadAsync de l'objet WebClient. Vous remarquez toutefois que cet appel de méthode ne peut se faire que si l'objet WebClient est disponible. Imaginons que l'utilisateur clique deux fois très rapidement sur le bouton. Dans ce cas, l'instance WebClient n'a pas eu le temps de recevoir les premières données qu'un nouvel appel est déclenché. Cela lève automatiquement une erreur car WebClient ne possède pas cette faculté (voir Figure 13.15). Il est incapable de gérer plus d'un appel asynchrone à la fois.

Figure 13.15

L'appel de méthodes asynchrones concurrentes lève une erreur.



La propriété IsBusy résout cette problématique en renvoyant l'état d'occupation de l'objet. Une autre manière plus élégante de gérer cette problématique consiste à lier la propriété IsEnabled du bouton à la valeur opposée de la propriété IsBusy. De cette manière, le bouton devient inactif

durant le laps de temps correspondant à l'appel asynchrone. L'avantage de cette méthode est à la fois de tenir compte et d'utiliser le travail fourni par le designer. Ainsi, chacun y gagne sans effort ou presque. Dans la section 13.3.4, nous traitons les données reçues mais auparavant il nous faut découvrir la technologie LINQ.

13.3.2 Introduction à LINQ

LINQ (*Language INtegrated Query*) est un mini-langage phrasé directement utilisable en C#. Il permet d'écrire des requêtes très simples à partir de mots-clés équivalents à ceux existants pour les bases de données. LINQ est une technologie à utiliser de concert avec des ORM comme Entity Framework ou Link2Sql, il permet de sérialiser les enregistrements de bases de données échappant à une logique orientée objet, en objet fortement typé. LINQ est donc notamment utile lorsque vous souhaitez adresser des requêtes à une base SQL (notamment grâce à la bibliothèque *RIA Services*). Ce n'est toutefois pas son seul cadre d'utilisation. LINQ étant accessible en C#, d'autres sources de données peuvent être ciblées. Il est ainsi possible d'adresser des requêtes à des contenus de type XML, JSON ou Object. Ce contenu correspond à tous types C# implémentant les interfaces *IEnumerable* ou *IEnumerable<T>*. Ainsi tout objet de type *List*, *Array*, *Dictionary*, *Collection* ou *Queue*, est à même d'être parcouru par une requête LINQ. Cela est particulièrement utile pour les opérations de tri car LINQ est très simple d'utilisation. Dans un nouveau projet, commencez par définir une énumération ainsi qu'une classe *Author* comme indiqué ci-dessous :

```
public enum TypeAuteur : int
{
    Musicien=0,
    Ecrivain=1,
    Poete=2
}

public class Author
{
    public string Nom { get; set; }

    public string Prenom { get; set; }

    public string Oeuvre { get; set; }

    public TypeAuteur Type { get; set; }

    public Author(string nom, string prenom)
    {
        this.Nom = nom;
        this.Prenom = prenom;
    }

    public Author(string nom, string prenom, string oeuvre, TypeAuteur ta)
    {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Oeuvre = oeuvre;
        this.Type = ta;
    }

    public override string ToString()
    {
        return this.Prenom + " " + this.Nom;
    }
}
```



```

public string ToString(bool displayBlog)
{
    if (this.Oeuvre == string.Empty || !displayBlog)
    {
        return this.Prenom + " " + this.Nom;
    }
    else
    {
        return this.Prenom + " " + this.Nom + " - " + this.Oeuvre;
    }
}
}

```

Créez trois exemplaires de `ListBox` : le premier affiche une liste d'auteurs non filtrée, les deux autres vont représenter la liste filtrée à partir de deux requêtes différentes. Placez également un champ de saisie nommé `filtreTxt`, dans `LayoutRoot` – la valeur saisie va jouer le rôle de filtre. L'idéal serait de définir un modèle de donnée (`DataTemplate`) afin d'afficher le détail de chaque élément contenu. Voici le code initial de la classe `MainPage` :

```

public partial class MainPage : UserControl
{
    List<Author> auteurs;

    public MainPage()
    {
        InitializeComponent();

        auteurs = new List<Author>()
        {
            new Author("Mozart", "Amadeus", "La flute enchantée", TypeAuteur.
                Musicien),
            new Author("Melville", "Herman", "Moby dick", TypeAuteur.Ecrivain),
            new Author("Wilde", "Oscar", "Le portrait de Dorian Gray", TypeAuteur.
                Poete),
            new Author("Dickens", "Charles", "Un chant de Noël", TypeAuteur.
                Ecrivain),
            new Author("Hugo", "Victor", "Les travailleurs de la mer", TypeAuteur.
                Ecrivain),
            new Author("Van Beethoven", "Ludwig", "L'hymne à la joie", TypeAuteur.
                Musicien),
            new Author("Gordon Byron", "George", "Don Juan", TypeAuteur.Poete),
            new Author("Shelley", "Marie", "Frankenstein", TypeAuteur.Ecrivain)
        };

        liste.ItemsSource = auteurs;
    }
    ...
}

```

Affectez la liste d'auteurs à la propriété `ItemsSource` de la première instance de `ListBox` comme montré dans le code ci-dessus. Écoutez l'événement `TextChanged` diffusé par le champ de saisie. Dans la méthode d'écoute, nous allons créer notre requête LINQ. Il nous faut toutefois référencer l'espace de noms `System.Linq`. Voici un exemple de requête simple défini dans la méthode d'écoute :

```

private void FilterChanged(object sender, TextChangedEventArgs e)
{
    string filtre = filtreTxt.Text;
}

```

```

var query = from auteur
            in auteurs
            where auteur.Oeuvre.Contains(filtre)
            select auteur;

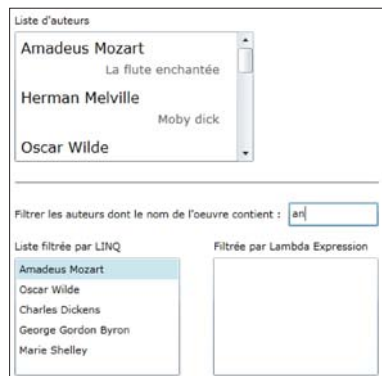
listeFiltrees.ItemsSource = query ;
}

```

Vous remarquez tout d'abord l'utilisation du mot-clé `var`, l'inférence de type nous évite de préciser le type de l'objet récupéré par la requête. Dans tous les cas, elle est de type `IEnumerable` ce qui est bien suffisant pour la réutiliser ou l'affecter par la suite. Vous pouvez envisager une requête LINQ un peu comme une boucle `foreach` d'un point de vue interprétation. La requête ci-dessus peut être traduite par la phrase suivante : pour chaque objet récupéré dans la liste `auteurs`, lorsque la propriété `Oeuvre` de cet objet contient la chaîne de caractères saisie, alors on récupère cet objet comme nouvel enregistrement de la requête. Cela peut paraître assez étrange de prime abord car le mot-clé `select` est situé à la fin de la requête. Cette notation est toutefois légitime car LINQ est un langage de requêtes intégré au code logique, cette écriture est vraiment proche des principes d'une boucle. Une fois la requête formalisée, vous pouvez l'affecter à la propriété `ItemsSource` de la seconde `ListBox`. Testez et compilez votre application : le filtre est pleinement fonctionnel (voir Figure 13.16).

Figure 13.16

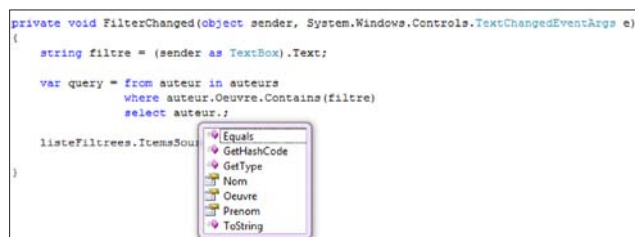
Contenu de la liste filtré puis affiché dans la seconde `ListBox`.



Contrairement aux apparences, ce n'est pas parce que les types ne sont pas précisés qu'il n'y a pas de typage fort. Ainsi, vous pourriez ne récupérer que le nom ou le prénom de l'auteur en précisant l'une de ces deux propriétés en fin de requête. L'IntelliSense de Visual Studio reste disponible et vous aide dans cette démarche puisque le type `Author` est récupéré dynamiquement (voir Figure 13.17).

Figure 13.17

L'IntelliSense de Visual Studio récupère le type dynamiquement.



Finalement, l'utilisation d'une inférence de type couplée à LINQ est adéquate car elle apporte beaucoup de flexibilité au développeur. Mise à part cette formalisation un peu spécifique, vous retrouverez de nombreuses clauses et mots-clés similaires à ceux trouvés dans une syntaxe SQL classique. Ainsi, `where` est la clause par excellence puisqu'elle permet de filtrer les éléments simplement. Elle peut être suivie de plusieurs expressions qui seront séparées par les opérateurs logiques `||`, `&&` et `!` (signifiant respectivement ou, et, opposé de) comme montré ci-dessous :

```
var query = from auteur in auteurs
            where auteur.Oeuvre.Contains(filtre)
              || auteur.Nom.Contains(filtre)
            select auteur;
```

Placez une instance de type `ComboBox` sur `LayoutRoot` et nommez-la `typeCombo`. Elle va nous permettre de filtrer les artistes en fonction de leur domaine respectif. Ajoutez trois éléments à cette liste déroulante par un clic-droit depuis l'interface de Blend. Nommez les objets `ComboBoxItem` dans l'ordre suivant : Musicien, Écrivain, Poète. Nous allons nous servir de l'index de chaque élément dans la liste comme correspondance avec l'une des valeurs de l'énumération. Écoutez l'événement `SelectionChanged` de la liste déroulante. Nous pouvons maintenant réécrire notre requête dans les deux méthodes d'écoute :

```
var query = from auteur in auteurs
            where auteur.Oeuvre.Contains(filtreTxt.Text)
              && (
                  typeCombo.SelectedIndex == (int)auteur.Type
                  || typeCombo.SelectedIndex == -1
                )
            select auteur;

listeFiltres.ItemsSource = query;
```

Lorsqu'aucun type n'est choisi dans la liste déroulante, l'index sélectionné correspond à une valeur de `-1`. C'est pour cette raison que nous utilisons l'opérateur logique `||` qui permet de renvoyer `true` lorsqu'une des deux expressions à droite ou à gauche est vérifiée. Nous pourrions également appliquer un tri à la liste des auteurs sélectionnés :

```
var query = from auteur in auteurs
            where auteur.Oeuvre.Contains(filtreTxt.Text)
              && (
                  typeCombo.SelectedIndex == (int)auteur.Type
                  || typeCombo.SelectedIndex == -1
                )
            orderby auteur.Prenom
            select auteur;
```

Par défaut le tri est réalisé dans un ordre ascendant, il est possible de renverser l'ordre grâce au mot-clé `descending` à placer après le champ utilisé pour le tri. Dans un tout autre registre, la clause `group by` permet de regrouper les éléments filtrés selon un critère. La requête ci-dessous récupère le nombre d'artistes répartis par domaines artistiques, dont l'œuvre contient la chaîne de caractère renseignée :

```
var query = from auteur in auteurs
            where auteur.Oeuvre.Contains(filtreTxt.Text)
            orderby auteur.Prenom
```

```
group auteur by auteur.Type into tmpGroup
select tmpGroup.Count();
```

Il est également possible de récupérer la totalité des auteurs triés par ordre alphabétique et par domaine artistique :

```
var query = from auteur in auteurs
            where auteur.Oeuvre.Contains(filtreTxt.Text)
            && (
                typeCombo.SelectedIndex == (int)auteur.Type
                || typeCombo.SelectedIndex == -1
            )
            orderby auteur.Prenom
            group auteur by auteur.Type into tmpGroup
            from artiste in tmpGroup
            select artiste;
```

Dans ce cas, on opère une nouvelle requête à partir du résultat de la première. Pour finir, la clause `join` permet d'opérer des jointures entre diverses sources de données, nous ne verrons pas son utilisation qui déborde un peu du sujet de ce livre. Sachez simplement que celle-ci est utile au même titre que son homologue côté SQL mais que la nature des différentes données reliées peuvent être diverses et variées. C'est l'un des grands avantages procurés par LINQ.

13.3.3 Expression lambda

Une autre manière de créer des requêtes avec LINQ consiste à utiliser les expressions *lambda*. Celles-ci sont apparues avec C# 3, vous pouvez les considérer comme des fonctions anonymes. L'opérateur `=>` définit une expression lambda. Les délégations couplées aux fonctions anonymes offrent de nombreux avantages. Le code ci-dessous expose leur utilisation dans ce contexte ainsi que deux manières différentes de les coder :

```
//définition de la délégation
delegate string del(Author i);

public partial class MainPage : UserControl
{
    //une première écriture possible
    del nomParOeuvre = x => x.Nom + "est l'auteur de :: " + x.Oeuvre;
    //une seconde délégation avec lambda expression
    del prenomNom = x => { return x.Prenom + " " + x.Nom; };

    List<Author> auteurs;

    public MainPage()
    {
        InitializeComponent();

        auteurs = new List<Author>()
        {
            new Author("Mozart", "Amadeus", "La flute enchantée", TypeAuteur.
                Musicien),
            new Author("Melville", "Herman", "Moby dick", TypeAuteur.Ecrivain),
            new Author("Wilde", "Oscar", "Le portrait de Dorian
                Gray", TypeAuteur.Poete),
            new Author("Dickens", "Charles", "Un chant de Noël", TypeAuteur.
                Ecrivain),
```

```

new Author("Hugo","Victor","Les travailleurs de la
            mer",TypeAuteur.Ecrivain),
new Author("Van Beethoven","Ludwig","L'hymne à la
            joie",TypeAuteur.Musicien),
new Author("Gordon Byron","George","Don Juan",TypeAuteur.Poete),
new Author("Shelley","Marie","frankenstein",TypeAuteur.Ecrivain)
};

//on récupère la chaîne de caractères du premier délégué
string chaine1 = nomParOeuvre(auteurs[3]);
//on récupère la chaîne de caractères du premier délégué
string chaine2 = prenomNom(auteurs[7]);

Debug.WriteLine
    ("chaîne 1 :: {0}\nchaîne 2 :: {1}",chaine1,chaine2);

```

Lorsque vous avez référencé l'espace de noms `System.Linq`, les objets de type `IEnumerable` ont reçu de nouvelles capacités par le biais de méthodes d'extension correspondantes aux clauses utilisées pour écrire une requête. Il devient dès lors possible de coupler l'utilisation de fonctions anonymes à LINQ, ce qui engendre un nombre impressionnant de possibilités. Il est ainsi facile de créer des conditions et des instructions élaborées. L'expression ci-dessous en est un exemple simple :

```

LambdaListeFiltrees.ItemsSource =
    auteurs.Where(a => a.Oeuvre.Contains(filtre)).Select(a=>a.Prenom);

```

La variable notée arbitrairement *a*, correspond à chaque élément parcouru au sein de la liste auteurs. L'expression placée à droite de l'opérateur `=>` est appelée *prédicat*. Ce terme désigne une fonction qui renvoie un booléen. Grâce à la seconde écriture possible d'une expression lambda `((a)=>{return a;})`, vous pouvez réécrire entièrement la requête LINQ :

```

LambdaListeFiltrees.ItemsSource =
    auteurs.Where(
        (a) =>
        {
            if (a.Oeuvre.Contains(filtre))
            {
                if (typeCombo.SelectedIndex == -1)
                {
                    return true;
                }
                else
                {
                    return typeCombo.SelectedIndex == (int)a.Type;
                }
            }
            else
            {
                return false;
            }
        }
    ).Select(a => a.Prenom);

```

La fonction anonyme retourne bien un booléen, ce qui est attendu par la clause `Where`. Le code ci-dessus peut être amélioré en renvoyant directement l'expression booléenne :

```

LambdaListeFiltrees.ItemsSource =
    auteurs.Where(

```

```
(a) =>
{
    if (a.Oeuvre.Contains(filtre))
    {
        return typeCombo.SelectedIndex == -1
            || typeCombo.SelectedIndex == (int)a.Type;
    }
    else
    {
        return false;
    }
}).Select(a => a.Prenom);
```

Comme nous l'avons expliqué précédemment, un prédicat correspond à une fonction renvoyant un booléen. Dès lors, ce qui fait d'une fonction un prédicat est tout simplement son contexte d'utilisation. Lorsque les conditions de la clause `where` deviennent complexes, il convient de formaliser le prédicat sous cette forme. Le code est plus lisible de cette manière, c'est ce qui est réalisé ci-dessous :

```
...
LambdaListeFiltrees.ItemsSource =
    auteurs.Where( (a) =>FiltreAuteurParType(filtre, a))
        .Select(a => a.Prenom);
}

private bool FiltreAuteurParType(string filtre, Author a)
{
    if (a.Oeuvre.Contains(filtre))
    {
        return typeCombo.SelectedIndex == -1
            || typeCombo.SelectedIndex == (int)a.Type;
    }
    else
    {
        return false;
    }
}
```

Le projet finalisé *LinqToObject.zip* se trouve dans le dossier *chap13* des exemples.

Maintenant que nous avons étudié les différents outils nous permettant de créer des requêtes, nous allons revenir sur notre précédent projet MVVM et traduire le contenu XML en objets C#.

13.3.4 Consommer du XML avec LINQ

Le XML est le format de prédilection lorsqu'il s'agit de diffuser des données facilement interprétables. Il permet, d'une part de structurer l'information simplement et d'autre part, il peut être parcouru par des fonctions récursives même lorsque l'on ne connaît pas son contenu à l'avance. L'exemple ci-dessous expose un extrait de données au format XML récupérées depuis l'adresse : <http://www.tweened.org/wp-content/uploads/videos/videos.xml> :

```
<VIDEOS>
...
<Video visible="true" id="13" titre="Gestionnaire d'états visuels"
    duree="00:07:13" date="08/15/2008">
    <Auteur nom="Ambrosi" prenom="Eric" Blog="http://www.tweened.org" />
    <description><![CDATA[Utilisation avancée du Visual State Manager]]>
```

```

        </description>
        <PreRequis>
            <PR>12</PR>
        </PreRequis>
        <file>14_advanceVisualStateManager.wmv</file>
        <baseurl>http://www.tweened.org/wp-content/uploads/videos/</baseurl>
        <filethumb>14_advanceVisualStateManager_0.000.jpg</filethumb>
    </Video>
    <Video visible="false" id="14" titre="Créer un flux Json"
        duree="00:05:09" date="08/16/2008">
        <Auteur nom="Ambrosi" prenom="Eric" Blog="http://www.tweened.org" />
        <description><![CDATA[Qu'est-ce que le Json et comment créer un flux
            Json avec php 5 et mysql]]></description>
        <PreRequis>
            <PR></PR>
        </PreRequis>
        <file>15_createJsonStream.wmv</file>
        <baseurl>http://www.tweened.org/wp-content/uploads/videos/</baseurl>
        <filethumb>15_createJsonStream_0.000.jpg</filethumb>
    </Video>
    ...
</VIDEOS>

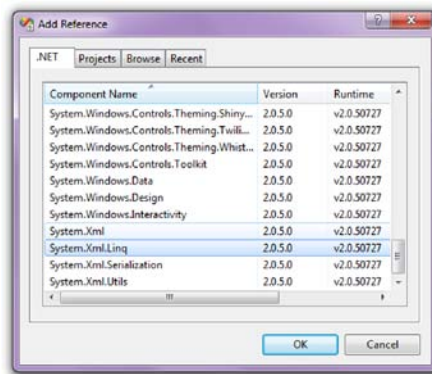
```

Notre objectif est de récupérer ce flux et d'alimenter le composant `ListBox` du précédent projet. Récupérez le précédent projet MVVM ou téléchargez le projet *MVVM_PatternWebClient.zip* présent dans le dossier *chap13* des exemples.

La première chose à faire est de référencer les espaces de noms adéquats. Nous avons besoin de `System.Linq` mais cela ne suffit pas, il nous faut également utiliser la bibliothèque C# (fichier dll) `System.Xml.Linq` et l'espace de noms correspondant (voir Figure 13.18).

Figure 13.18

Contenu de la liste filtré puis affiché dans la seconde `ListBox`.



À partir de là, nous pouvons commencer à parcourir le document XML dans la méthode de réception `wc_OpenReadCompleted`. Tout est contenu dans la propriété `Result` de l'objet événementiel. Il nous faut un objet XML contenant les données, l'espace de noms `System.Xml.Linq` donne accès à de nouveaux objets préfixés d'un `X`. Ainsi deux d'entre eux, `XDocument` et `XElement`, ont la capacité d'écrire du XML à partir du flux binaire récupéré. Nous utiliserons plutôt la méthode `Load` de `XElement` afin de simplifier le chemin d'accès aux nœuds XML enfants. Vous pouvez vérifier que le document est correctement interprété en le traçant comme montré dans le code qui suit :

```

static void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (e.Error != null)
    {
        throw e.Error;
    }
    else if (e.Cancelled)
    {
        //l'appel asynchrone a été annulé
    }
    else
    {
        ParseXMLResult(e);
    }
}

private static void ParseXMLResult(OpenReadCompletedEventArgs e)
{
    XElement xmlResult = XElement.Load(e.Result);

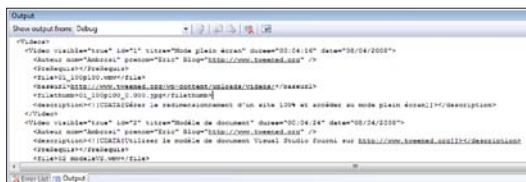
    Debug.WriteLine(xmlResult.ToString());
}

```

Si tout se passe bien, la fenêtre de sortie affiche le contenu du document (voir Figure 13.19).

Figure 13.19

Contenu du document XML.



Nous allons maintenant sérialiser chaque élément en objet C# Media. Pour cela, nous allons parcourir les collections XML fournies par LINQ avec une simple requête. Pour récupérer une collection contenant un type de nœud XML précis, le plus simple consiste à utiliser la méthode Descendants. Elle prend une chaîne de caractères comme argument, qui correspond au nom du nœud que vous souhaitez récupérer dans la collection. Pour chaque élément récupéré dans cette collection, nous pouvons instancier la classe Media. Il reste ensuite à affecter la requête à notre collection observable de type ObservableCollection<Media>. Toutefois, même si le parseur de Visual Studio ne détecte aucune erreur lors de l'affectation de Datas, le code ci-dessous ne fonctionnera pas à l'exécution :

```

private static void ParseXMLResult(OpenReadCompletedEventArgs e)
{
    XElement xmlResult = XElement.Load(e.Result);

    var query = from media in xmlResult.Descendants("Video")
                select new Media();

    Datas = (ObservableCollection<Media>)query;
}

```

Il n'est pas possible de transformer un type IEnumerable en ObservableCollection. De plus, la variable query ne possède aucune méthode équivalente à ToList ou ToArray permettant de renvoyer une collection observable. Cette problématique étant assez standard, la bibliothèque SL-

Extensions fournit une classe *ObservableCollection* implémentant de nombreuses fonctionnalités supplémentaires. Elle possède notamment la méthode *AddRange* que nous allons utiliser. C'est pour cette raison que nous ne faisons pas référence à l'espace de noms standard *System.Collections.ObjectModel* pour utiliser des collections de ce type, mais plutôt à celui de *SL-Extensions* *SLExtensions.Collections.ObjectModel*. La méthode *AddRange* récupère d'un seul coup tous les éléments contenus dans query :

```
private static void ParseXMLResult(OpenReadCompletedEventArgs e)
{
    XElement xmlResult = XElement.Load(e.Result);

    var query = from media in xmlResult.Descendants("Video")
                select new Media();

    Datas.AddRange(query);
}
```

Si vous compilez tout se déroule bien, mais rien n'apparaît en apparence dans la liste. En apparence seulement, car si vous cliquez dans la liste, vous sélectionnez une ligne d'enregistrement. Celle-ci est vide car nous n'avons pas réellement affecté les propriétés de chaque *Media*. Pour cela nous pouvons utiliser un objet d'initialisation et les méthodes XML fournies par LINQ :

```
private static void ParseXMLResult(OpenReadCompletedEventArgs e)
{
    XElement xmlResult = XElement.Load(e.Result);

    var query = from media in xmlResult.Descendants("Video")
                select new Media()
                {
                    Titre = (string)media.Attribute("titre"),
                    Description = (string)media.Element("description"),
                    BaseUrl = (string)media.Element("baseurl"),
                    FichierMedia = (string)media.Element("file"),
                    FichierVignette = (string)media.Element("filethumb"),
                    Auteur = new Author(
                        (string)media.Element("Auteur").Attribute("nom"),
                        (string)media.Element("Auteur").Attribute("prenom"),
                        (string)media.Element("Auteur").Attribute("blog")
                    ),
                    Date = (string)media.Element("Auteur").Attribute("date"),
                    Duree = TimeSpan.Parse((string)media.Attribute("duree"))
                };

    Datas.AddRange(query);
}
```

Recompilez l'application et cliquez sur le bouton afin de charger les données dans la liste. Comme vous le constatez, celle-ci est désormais remplie correctement. Le projet finalisé et légèrement complété, *MVVM_PatternXML.zip*, est disponible dans le dossier *chap13* des exemples.

13.3.5 Charger un flux JSON

JSON est un format de présentation des données au même titre que XML à la différence près que cet héritage direct de JavaScript. JSON signifie *JavaScript Object Normalisation*. Silverlight possède la capacité de consommer du JSON par différents moyens que nous allons étudier.

13.3.5.1 Pourquoi utiliser JSON

XML est très puissant lorsqu'il s'agit de présenter des données de manière structurée car il repose sur un principe de relations familiales. Il est donc assez légitime de savoir en quoi JSON peut être compétitif. Tout d'abord, au même titre que XML, JSON ne conserve pas les types. Toute donnée contenue est de type string. Les données sont, par exemple, formalisées à travers des objets anonymes. L'objet anonyme ci-dessous décrit une œuvre :

```
{"peintre": "Vassily Kandinski", "titre": "Jaune-rouge-bleu",
  "annee": "1925", "type": "abstraction lyrique"}
```

Il est également possible de décrire des listes d'objets anonymes :

```
[
  {"peintre": "Vassily Kandinski", "titre": "Jaune-rouge-bleu",
    "annee": "1925", "type": "abstraction lyrique"},
  {"peintre": "Vassily Kandinski", "titre": "Dans le gris",
    "annee": "1919", "type": "abstraction lyrique"},
  {"peintre": "Vassily Kandinski", "titre": "Avec l'arc noir",
    "annee": "1912", "type": "abstraction lyrique"}
]
```

Ainsi, les médias que nous avons récupérés précédemment au format XML pourraient être traduits en JSON de cette façon :

```
[
  {"titre": "Gestionnaire d'état visuel", "duree": "00:07:13",
    "baseurl": "http://www.tweened.org/wp-content/uploads/videos/",
    "file": "14_advanceVisualStateManager.wmv",
    "filethumb": "14_advanceVisualStateManager_0.000.jpg",
    "description": "Utilisation avancée du Visual State Manager",
    "date": "08/15/2008"},
  {"titre": "Créer un flux Json", "duree": "00:05:09", "baseurl": "http://www.tweened.org/wp-content/uploads/videos/",
    "file": "15_createJsonStream.wmv",
    "filethumb": "15_createJsonStream_0.000.jpg",
    "description": "Qu'est-ce que le Json et comment créer un flux Json avec php 5 et mysql",
    "date": "08/15/2008"}
]
```

JSON est en réalité très proche d'une représentation plate de type base de données, les objets du tableau représentent l'équivalent des enregistrements contenus dans une table SQL. Comme vous l'aurez compris, le premier avantage de JSON est d'être moins verbeux que XML. On peut considérer un gain entre 30 et 40 % de poids en moins comparé à XML. C'est autant de performances de bande passante gagnée. Toutefois, le but n'est pas ici d'établir une quelconque compétition entre ces deux formats. Il faut simplement concevoir que XML est là, non seulement pour transmettre des données, mais également pour leur donner un sens et une structure logique.

Ainsi, XML vous oblige à présenter vos données à un niveau plus élevé que JSON puisque les relations familiales entre chaque nœud déterminent une certaine logique entre chacun d'eux. Avec XML, vous prenez parti sur la manière dont les données sont présentées. Pas avec JSON : ce dernier est une représentation de table SQL qui n'a d'autre objectif que transférer des données. Les 30 ou 40 % que vous gagnerez le seront donc forcément au détriment de liens logiques définis entre les données tels que vous les trouvez dans XML. C'est pourquoi il faut choisir l'un ou l'autre de ces formats selon le cas de figure. Alors que XML est largement répandu quel que soit l'environnement de développement, à travers des services web, des flux RSS ou au sein d'applications

bureautiques, JSON est réellement lié à Internet et aux environnements de développement tels que PHP ou MySQL. Il est toutefois intéressant de savoir que *Windows Communication Foundation* est capable de transmettre des données dans ce type de format. De nombreuses bibliothèques existent sous PHP, mais la plus efficace en terme de sérialisation est sans doute celle fournie nativement par PHP 5 car elle est codée en langage C. Vous trouverez de nombreux tests de performance à ce sujet sur Internet. Encoder un enregistrement de base de données en JSON est réellement très simple avec PHP. C'est en partie ce qui fait le succès de ce format, le script ci-dessous le démontre :

```
$maConnexion = new MySqlConnection("localhost","root","", "catalogue");

$req = "SELECT titre as Titre, commentaire as Commentaire, url_image as
        UriImage, date as Date, artiste.nom as Nom, prenom as Prenom, style.
        nom as NomStyle
FROM disque, artiste, style
WHERE disque.idArtiste = artiste.idArtiste AND disque.idStyle =
        style.idStyle";

//on supprime le fichiers mis en cache
header("Cache-Control: no-cache, must-revalidate");
header("Content-Type: text/plain");

$res = mysql_query($req) or die (mysql_error());

$tabDisque = array();

while($ligne = mysql_fetch_array($res) )
{
    foreach( $ligne as $key => $value )
    {
        $ligne[$key] = utf8_encode($value);
    }
    array_push ($tabDisque, $ligne);
}

echo json_encode($tabDisque);
```

En quelques lignes, nous encodons la totalité des enregistrements renvoyés par une requête sans effort. Il faudra toutefois faire attention à certains détails sous PHP. Il est tout d'abord impossible d'encoder un retour de requête MySQL directement. Vous devrez parcourir l'objet MySQL réceptionné et réaffecter chaque ligne contenue dans un nouveau tableau, grâce à une boucle par exemple. De plus, comme nous l'avons déjà précisé, Silverlight fonctionne par défaut en UTF-8, il est donc nécessaire d'encoder chaque enregistrement en UTF-8 avant de renvoyer le tableau à Silverlight.

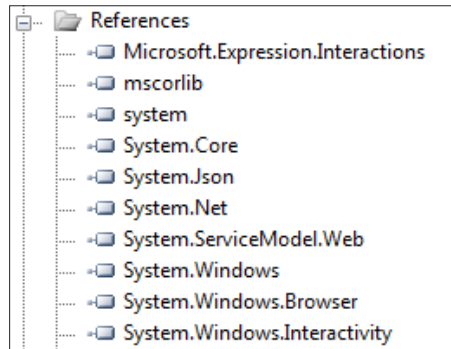
13.3.5.2 Contrat de sérialisation

Commencez par télécharger le projet *MVVM_Pattern.JSON.zip* depuis les exemples du livre. Nous allons récupérer le flux JSON situé dans le répertoire *ClientBin* du projet correspondant au site web. Il est tout d'abord important de référencer deux bibliothèques externes nommées *System.Json.dll* et *System.ServiceModel.Web.dll* (voir Figure 13.20). La première va nous permettre de créer des objets JSON *via* C#, la seconde sera utile dans un second temps afin de sérialiser en seulement deux lignes la totalité du contenu JSON récupéré. Ces dernières permettent respecti-

vement l'importation des espaces de noms `System.Json` et `System.Runtime.Serialization.Json`, via l'instruction `using`.

Figure 13.20

Bibliothèques à importer.



Le membre statique de notre classe `MediasDatas` nommé `URL_MEDIA` doit être modifié pour cibler notre fichier JSON :

```
public static Uri URL_MEDIA =
    new Uri("json.txt", UriKind.RelativeOrAbsolute);
```

Nous avons plusieurs techniques pour consommer un flux JSON. La première consiste à parcourir notre tableau d'objets anonymes et à transformer chaque enregistrement dynamiquement en objets typés `Media`. Attention, bien que portant le même nom que dans le projet dédié au chargement XML, cette classe est très différente :

```
public class Media
{
    public string Titre { get; set; }
    public string UrlImage { get; set; }
    public string Commentaire { get; set; }
    public string Date { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string NomStyle { get; set; }
    public override string ToString()
    {
        return Titre + " - Date :: " + Date;
    }
}
```

Notre classe `MediasDatas` n'est que très peu modifiée au final. L'écouteur de réception des données déclenche la méthode `ParseJSONResult` au lieu de la méthode `ParseXMLResult` :

```
static void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
```

```

        if (e.Error != null)
        {
            throw e.Error;
        }
        else if (e.Cancelled)
        {
            //l'appel asynchrone a été annulé
        }
        else
        {
            ParseJSONResult(e);
        }
    }

    private static void ParseJSONResult(OpenReadCompletedEventArgs e)
    {
        //ici on traite le flux de données contenu dans
        //l'objet événementiel
    }

```

À ce stade, il existe différentes manières de gérer les données au format JSON, en parcourant les objets JSON de type `JsonArray` ou `JsonObject` grâce à une boucle, *via* LINQ pour JSON (disponible grâce à l'espace de noms `System.Linq`) ou grâce à un contrat de sérialisation. Les deux premières méthodologies sont un peu rébarbatives et fastidieuses lorsqu'on les compare à la troisième. Le code ci-dessous illustre la première solution :

```

/*Méthode 1 avec boucle foreach*/
//on commence par récupérer le tableau d'objets
JsonArray arr = (JsonArray)JsonArray.Load(e.Result as Stream);

//et on le parcourt
foreach (JsonObject media in arr)
{
    Datas.Add(new Media()
    {
        Commentaire = media["Commentaire"],
        Date = media["Date"],
        Nom = media["Nom"],
        NomStyle = media["NomStyle"],
        Prenom = media["Prenom"],
        Titre = media["Titre"],
        UrlImage = media["UrlImage"],
    });
}

```

Comme vous le constatez, on accède à la valeur de chaque propriété d'objet grâce à une clé d'accès de type `string`. Le terme à gauche du signe égal correspond à une propriété de la classe `Media`, le terme à droite correspond à la valeur récupérée côté JSON. Vous remarquez qu'il n'est pas nécessaire de transtyper la valeur récupérée dans notre cas. Cela est valable car toutes les propriétés de la classe `Media` qui sont affectées sont de type `String`, ce qui est renvoyé nativement par les propriétés d'un objet JSON. Une autre méthode consiste à utiliser LINQ avec les collections JSON, cela est assez similaire mais permet en plus de filtrer les résultats plus efficacement :

```

JsonArray arr = (JsonArray)JsonArray.Load(e.Result as Stream);
//LINQ à faire
var query = from media in arr
             where ((string)media["Nom"]).Contains("a")

```

```
select new Media()  
{  
    Commentaire = media["Commentaire"],  
    Date = media["Date"],  
    Nom = media["Nom"],  
    NomStyle = media["NomStyle"],  
    Prenom = media["Prenom"],  
    Titre = media["Titre"],  
    UriImage = media["UriImage"],  
};  
Datas.AddRange(query);
```

Cela est facile à faire car LINQ possède la capacité de parcourir n'importe quel type de liste implémentant `IEnumerable` ce qui est le cas des objets JSON de type `JsonArray` ou `JsonObject`. Une dernière méthode est bien plus efficace et rapide. Elle consiste à créer un contrat de sérialisation. Cela est également réalisable avec XML mais légèrement plus contraignant. Côté JSON, il suffira de s'arranger pour que les propriétés des objets JSON et de la classe ciblée possèdent également les mêmes noms. Une fois que vous vous êtes assuré de cela, le reste est trivial :

```
//on crée un contrat de sérialisation  
DataContractJsonSerializer contrat =  
    new DataContractJsonSerializer(typeof(List<Media>));  
  
//on parcourt l'objet grâce à la méthode ReadObject qui se charge  
//des correspondances et on appelle la méthode AddRange  
//de la collection observable  
Datas.AddRange( (List<Media>)contrat.ReadObject(e.Result as Stream) );
```

INFO

Le code réel prend deux lignes en tout et pour tout ce qui est assez appréciable. Le contrat de sérialisation JSON est un dérivé simplifié des bibliothèques existantes pour d'autres types de données. Cela pour la bonne raison que la structure même du JSON est tellement simple qu'il n'y a qu'un seul type de propriété. Au contraire, dans XML une valeur peut être contenue dans un nœud élément, dans un attribut ou dans une balise CDATA. Il est du coup nécessaire de modifier la classe en ajoutant des métadonnées en en-tête de propriété et de classe.

Le projet *MVVM_PatternJSONFinal.zip* est dans le dossier *chap13* des exemples de cet ouvrage.

Nous allons maintenant évoquer les mécanismes de sécurité du lecteur Silverlight dans un contexte de chargement de données.

13.3.6 Sécurité interdomaines

Tout au long de ce chapitre, nous avons chargé des médias ou des données sans réellement nous préoccuper des contraintes liées à la sécurité. Nous sommes parti du fait que nous chargions des données présentes sur le même site, dans une sous-arborescence de ce dernier ou à sa racine. Ce n'est toutefois pas représentatif des applications riches qui ont souvent pour vocation de charger du contenu très varié à partir de sources très éclatées. Les notions de *cloud computing* et de services web impliquent directement le partage de ressources interdomaines. Toutefois, pour des raisons de sécurité, la politique par défaut des plateformes de développement comme Silverlight est d'encadrer ce modèle afin de limiter les abus, les fraudes et le piratage en tout genre.

ATTENTION

Depuis Silverlight, il est impossible de télécharger des données présentes dans un répertoire parent de celui où est situé le fichier xap, en utilisant la syntaxe d'accès relatif `../`. Ceci pour la bonne raison que le lecteur Silverlight pourrait atteindre des données en dehors de ce qui est accessible à la racine du site, donc dans l'arborescence non public du serveur.

La méthodologie permettant de renforcer la sécurité est assez simple. Concrètement, chaque site est responsable du type de contenu et de connexion qu'il partage et qu'il laisse à disposition des autres sites ou des services web externes. Dans la grande majorité des cas, les sites n'autorisent pas le chargement de leur contenu externe car lorsqu'aucune configuration n'a été mise en place, le comportement par défaut est de refuser tous chargements extérieurs. Chaque administrateur de site web peut décider de mettre à disposition ou non des données à l'extérieur *via* la création d'un fichier dédié que nous allons étudier. Dans tous les cas la machine cliente n'a aucune prise sur ce comportement. Reprenez le projet *MVVM_PatternJSON* et modifiez l'adresse de chargement du fichier *json.txt* comme exposé ci-dessous :

```
public static Uri URL_MEDIA =
    new Uri("http://referencesilverlight.tweened.org/json.txt",
UriKind.RelativeOrAbsolute);
```

Compilez, puis chargez les données. Comme vous le constatez, l'application lève une erreur de sécurité dont le détail n'est pas vraiment explicite. Vous avez essayé de charger du contenu externe à Silverlight depuis le site <http://referencesilverlight.tweened.org>, qui ne le permet pas d'emblée (voir Figure 13.21).

Figure 13.21

Erreur de sécurité.



En réalité, Silverlight ne s'autorise pas ce type de téléchargement sans en avoir expressément la permission du site contenant les données distante. Il va essayer de trouver et d'interpréter l'un des deux fichiers qui décrivent les règles d'accès. Si aucun d'eux n'est présent, le contenu distant n'est pas autorisé à être chargé. Le premier est nommé `crossdomain.xml` et fut apporté il y a quelques années par la plateforme Flash. Celle-ci connaît des contraintes équivalentes à Silverlight dans ce domaine. Pour des raisons de comptabilité et afin de faciliter le travail de chacun, Silverlight est capable d'interpréter la balise `allow-access-from` contenue dans ce type de fichiers.

Afin de concrétiser ce concept, téléchargez le fichier `crossdomain.xml` présent à la racine du site de Twitter : <http://www.twitter.com/crossdomain.xml> :

```
<cross-domain-policy xsi:noNamespaceSchemaLocation="http://www.adobe.com/
    xml/schemas/PolicyFile.xsd">
  <allow-access-from domain="twitter.com"/>
  <allow-access-from domain="api.twitter.com"/>
  <allow-access-from domain="search.twitter.com"/>
  <allow-access-from domain="static.twitter.com"/>
</cross-domain-policy>
```

```
<site-control permitted-cross-domain-policies="master-only"/>
<allow-http-request-headers-from domain="*.twitter.com" headers="*"
    secure="true"/>
</cross-domain-policy>
```

INFO

Pour ceux qui ne connaissent pas Twitter, sachez qu'il s'agit d'un site et d'un réseau de micro-blogging. Son principe repose sur la diffusion d'informations concentrées et limitées en chaîne de caractères. Il suffit de s'abonner au compte d'une personne sur Twitter pour recevoir les informations que ce dernier propage en temps réel sous la forme d'un tchat multiutilisateur. De nombreuses plateformes proposent des applications Twitter et reposent sur l'API qu'il fournit.

Ce genre de fichiers déclaratifs est accessible à tous mais cela ne pose aucun problème. Il est important que les règles de sécurité soient exposées très clairement, et permettent à tous les développeurs de connaître leurs droits et contraintes d'accès au serveur. Ce document déclaratif est assez simple à lire, il autorise de nombreux sous-domaines de Twitter à accéder aux données. Vous remarquez à ce propos que le sous-domaine **api.twitter.com** fait partie de cette liste. Il est nécessaire de spécifier les droits pour chaque domaine et sous-domaine. Le protocole, le port ou le serveur utilisé, sont autant de paramètres qu'il faudra également gérer d'un point de vue accès. Un deuxième type de fichier propre à Silverlight est recherché par défaut, et cela avant le `cross-domain.xml`. Il se nomme `clientaccesspolicy.xml`. Ce fichier est appelé en premier, s'il n'est pas trouvé ce sont les réglages du `crossdomain.xml` qui s'appliquent. Ce fichier ajoute quelques fonctionnalités comme la capacité de gérer chaque sous-dossier de manière indépendante. Modifiez l'URL du fichier JSON afin de cibler le site **www.tweened.org** :

```
public static Uri URL_MEDIA =
    new Uri("http://www.tweened.org/json.txt",
        UriKind.RelativeOrAbsolute);
```

Cette fois-ci tout se passe bien à la compilation car un fichier `clientaccesspolicy.xml` est présent à la racine de **www.tweened.org**. Ce fichier laisse la porte ouverte car il n'a pas de contenu réellement sensible. Voici ce qu'il contient :

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Concrètement, les droits définissent un accès total quel que soit le domaine à l'origine de la demande et cela pour la totalité des répertoires et des sous-répertoires du site **www.tweened.org**.

Voici donc le dernier chapitre qui se termine, vous retrouverez tous les composants et projets téléchargeables du livre sur le blog <http://referencesilverlight.tweened.org>. Certains des composants et des bibliothèques seront insérés ou fondus dans la bibliothèque *SLExtensions* disponible sur le portail *CodePlex*.

Nous avons abordé la conception d'applications riches tant du point de vue d'un designer que de celui d'un développeur. J'espère que ce livre vous aura aidé dans votre apprentissage de Silverlight et je vous remercie pour votre lecture.

Éric Ambrosi, le 21-12-2009 à Paris.

Index

Symboles

3D 245. *Voir* **Caméra, Matrices et**

Projection 3D

conteneurs 260
environnement 245
moteurs de rendu 246
point de fuite 262
propriétés
 modifier en C# 253
 rotation 249
 translation 251

[] (tableaux) 25

A

Accélération 121

 courbes 135
 équations 135
 gérer 132
 principes 132

ActualHeight 258

ActualWidth 258

Adaptative Streaming 441, 442

Add 96

Affectation dynamique

 animations 219
 Storyboard 144

Affichage

 détecter le changement 71
 mode plein écran 69

Afficher un visuel XAML 18

Ajouter des enfants à la liste d’affichage 91

Alignement étiré 60

allow-access-from 478

Angle d’ouverture 259

Animations 113

 affectation dynamique 219
 avec C# 138
 mise à jour dynamique 141
 cadence 115
 classes 119
 clé 117
 contrôler 119
 créer 116
 décaler une séquence 130
 déclencher 128
 définition 114
 de rebond 133
 dupliquer 137
 exemple 125
 horizontales 166
 méthodes de contrôle 123
 modes 132
 pied de page 131
 transparence 129
 types 118
 verticales 165

Voir aussi **Expression Blend et Particules**

AppendLine 148

Application

 affichage 46
 redimensionner 46

Applications riches 315

AppManifest.xaml 48

App.xaml 41

App.xaml.cs 41

Arbre visuel et logique 73

- d'une page 42
- objets
 - ajouter 92
 - désactiver 104
 - supprimer 99
- parcourir 89

Array 25**Auto, mode** 55**Aveuglement d'informations** 306**B****Background** 61, 329**BackgroundColor** 327**Base** 188**Begin** 123**BeginTime** 129**Behavior** 232**Behaviors** 198**Bibliothèque de contrôles** 56**BindingValidationError** 222**BitmapImage** 430

- description 432

Boîte de connexion 390**bool** 24**Border** 126**BorderBrush** 327**Boucles** 87**Bounces** 135**Bounciness** 135**Bouton** 61

- générique 177
- interrupteur 201
- menu 58
- personnalisé 169
- rotation 178
- survol 176

Brush 332**Button** 55, 74**ButtonBase** 169**byte** 23**C****C#**

- animation 138
- API Silverlight 20
- comparaison avec XAML 17
- fondamentaux 20

Caméra 247, 257

- angle d'ouverture 259
- perspective 260
- position 258

Canvas 79**Canvas.Left** 165**Canvas.Top** 165**CaptureMouse** 230**CenterOfRotationX** 250**CenterOfRotationY** 250**CenterOfRotationZ** 250**Champs d'objets** 23

- vs. propriétés 28

Champ texte

- créer 42
- Voir aussi* Texte

Chargement de médias 429

- images 430
- vidéos 449

CheckStates 203**Child** 91, 97**Children** 91**Classes** 119

- abstraites 64
- graphiques, héritage 75
- partielles 49

Clé d'animation 117**clientaccesspolicy.xml** 479

- Code behind** 3
 - CodePlex** 453
 - Color** 120
 - ColorAnimation** 122
 - ColorChooser** 410
 - ColorPicker** 410
 - ComboBox** 74
 - CommonStates** 188
 - Compilation** 44
 - fichiers générés 47
 - processus 49
 - Comportements** 67, 198, 232
 - action 233
 - personnalisés 234
 - simples 233
 - types de 232
 - vs. programmation événementielle 232
 - Composants** 73
 - anonymes, accès 84
 - famille de 73
 - héritage 386
 - interface utilisateur 74
 - liste de données 74
 - nommés, accès 83
 - personnalisables 78
 - personnalisés 385
 - recopie bitmap 239
 - Conteneurs** 74
 - 3D 260
 - de mise en forme 79
 - primitifs 77
 - racine 42
 - redimensionnable 53
 - vide et mode Auto 54
 - Content** 64, 91, 97, 177
 - ContentPresenter** 174, 178
 - Contexte de données** 371
 - Contrat**
 - de modèle 413
 - de sérialisation 474
 - Control** 332
 - Contrôles**
 - code logique 415
 - créer 411
 - gestion et affichage de texte 74
 - personnalisables 409
 - utilisateur 385
 - controlsToolkit** 60
 - ControlStoryboardAction** 128
 - ControlTemplate** 353
 - Couleur**
 - arrière-plan 61
 - code hexadécimal 61
 - Couplage faible** 207
 - principe 215
 - crossdomain.xml** 478
 - CubicOut** 191
 - Curseur de survol** 63
 - C# 63
 - Cursors** 63
- D**
- DataContext** 373
 - DataGrid** 74
 - DataTemplate** 379
 - vs. ListBoxItem 379
 - Débogueur** 86
 - decimal** 23
 - Décoration** 232
 - Decorator** 232
 - Dégradé** 335
 - radial 134
 - DependencyProperty** 423
 - Désactiver des objets** 103
 - Descendants** 471
 - DesignComponent** 311
 - DesignFocus** 311
 - Design patterns** 206

Dictionary 318

Dictionnaires de ressources 319

organiser 351

stockage et portée 319

Diffuseurs 207

Diffusion

Adaptative Streaming 441

format 438

progressive 440

Smooth Streaming 442

streaming 440

Discrete 121, 132

DispatcherTimer 156, 162

DockPanel 80

Données 429

charger 459

fictives 367

Voir aussi JSON, LINQ et XML

Double 119

DoubleAnimation 122

DoubleAnimationUsingKeyFrames 118,
167

DoubleKeyFrame 168

DownloadProgressChanged 451

DragDelta 421

E

Easing 120, 121, 132

Écoute d'un événement, supprimer 211

Écouteurs 207

Effet antagoniste 160

Effondrement 100

Ellipse 74

Embarquer les polices 345

EnableCounterFrameRate 115

enum 24

Énumérations 24

Équation d'accélération 191

Erreurs

d'accès 85

levées 94

Espaces de noms 19

États visuels

au niveau de l'application 196

créer 197

classe Button

afficher 190

animer 190

focus utilisateur 195

groupes 188

modifier 189

gestionnaire 200

SketchFlow 303

Événement

désinscription et souscription 213

diffusé 98, 104

éviter 223

écoute

supprimer 211

interaction 222

personnalisé 403

propagation événementielle 221

routé 221

souris 228

souscrire 208

Events, panneau 69

EventTrigger 312

Expression Blend 37

animer avec 124

compiler 44

fichiers 47

interface 39

Expressions lambda 467

Expression Studio 6

Expression Blend 7

Expression Encoder 7

Expression Media 7

Expression Web 7

F

Filigrane 234
couleur 237
Fill 77, 327
FillBoxClosed 410
FillBoxOpened 410
Flux de production 282
FontStretch 341
FontStyle 341
FontWeight 341
Forever 165
Formes primitives 77
FrameworkElement 131, 348

G

Garbage Collector 104, 211
Gestion des dégradés 134
Gestionnaire d'états visuels 183
C# et 200
Gestionnaires de médias 74
GetAllMedias 458
GetCurrentState 123
GetCurrentTime 123
GetValue 424
Glyphs 343
GotFocus 221
GoToStateAction 199, 234
Grid 60, 80, 171

H

Handled 228
Height 58
HelloWorld 37
HelloWorld.dll 48
HelloWorld.xap 48

Héritage 34
classes graphiques 75
HttpRequest 461
HuePickerPanel 416, 422

I

IDictionary 141
Image 74
arrière-plan 170
bitmap, afficher 171
charger dynamiquement 430
déformation, éviter 171
ImageFailed 431
ImageLoader 433
ImageOpened 431
ImageSource 430
Imbrication 81
Implémentations 34
Index, échange 105
Inférences de type 30
Info bulle 62
Inline 342
INotifyPropertyChanged 424
Insert 94
Instance
d'objet, afficher 177
nommer 62
Instanciation dynamique de ressources
Storyboard 139
Instancier des objets graphiques 91
int 23
Interaction, définir 228
Interactivité 205
Interfaces, implémenter 35
Interpolations 120
par clés d'animations 120
sans clé d'animation 122

types 121
choisir 121

Invoke 240

PropertyChanged 457

IsEnabled 414

IsolatedStorage 21

J

JSON 472

JsonArray 476

JsonObject 476

Just In Time Compiler 50

K

Kaxaml 18

KeyDown 221

KeyTime 120

KeyUp 221

L

Langages de développement 9

LayoutUpdate 104

Lecteur SketchFlow 290

Lecteur vidéo 444

chapitrage 445
couplage faible 217
créer 169
insérer image de fond 170
tête de lecture 447

Liaison 360

conversion des valeurs 363
créer 360
de données 393
 rafraîchir 398
de modèles 178
 contraintes 181
 créer 180

supprimer 182
paramètres 374

Ligne de temps 117

Line 74

Linear 121

LineBreak 342

LINQ 463

exemple de requête 463
expressions lambda 467
XML et 469

ListBox 74

composant personnalisé 382
modèle 375
structure 376

ListBoxItem 379

Loaded 98, 104

LoadedBitmapImage 438

LoginBox 390

états visuels 391

LostFocus 222

M

Main 22

MainPage.xaml 40

MainPage.xaml.cs 40

Math, animation 157

Matrices 263

3D 267
 opérations 269
DirectX 267
multiplication 269
OpenGL 267
perspective 272
principes 263
rotation 271
transformations affines et non affines 266

Matrix 266

Matrix3D 268

Matrix3DProjection 250, 257, 268

MatrixTransform 266

MediaElement 74, 445

MediaEnded 451

MediaFailed 451

MediaOpened 447

Médias 429

charger 429

dynamiquement des images 430

MediaViewModel 457

Menu

créer 55

dynamique 95

Méthodes

appel 31

contrôle d'animation 123

déclarer 31

extension, d' 33, 106

paramètres 32

récurives 89

surcharger 35

**Mise à jour dynamique d'une
animation** 141

Mode d'animations 132

Modèle 174

composants, de 348, 353

créer 356

Slider 355

Slider, personnaliser 358

conception 206

événementiel 205

bases 205

modifier 176

Voir aussi Liaison de modèles

Modificateurs d'accès 26

**Modifier l'apparence d'une liste
d'éléments** 377

Moonlight 2

MouseLeftButtonDown 139, 221, 229

MouseLeftButtonUp 221, 229

MouseMove 221, 229

MouseOver 189

MouseWheel 222

Mouvement circulaire 157

MultiScaleImage 74

MVVM 452

modèle 454

principes 452

MVVM_Pattern.Model 456

MVVM_Pattern.ViewModel 456

N

Name 62, 148

NaturalDuration 448

Navigate 389

Navigation page par page 386

.Net 3 3

new 93

Nom d'exemplaire 61

NoWrap 341

O

Object 52, 120

Objets

accès 83

ajouter 92

désactiver 103

graphiques, instancier 91

nommer 83

supprimer 99

Observateur 206

OnAttached 235

OnDetaching 235

Ordre d'imbrication 81

override 35

P**Panel** 79**PanelBackground** 330**params** 32**Parcourir la liste des enfants** 87**Particules** 160

créer 162

fond sous-marin 161

Path 74, 77**Pause** 123**PersoComponent** 311**Pied de page** 53**PieProgressBar** 447**Pile** 29**Pinceaux** 327

couleurs 328, 331

ressources 329

de dégradé 334

de vidéos 336

d'images 336

PlaneProjection 250, 253, 254, 272**Point** 120**PointAnimation** 122**Point d'arrêt** 87**Point de fuite** 262**Polices** 340

embarquer 345

gestionnaire 346

par défaut 344

personnalisées 344

propriétés 341

POO 206**Portée de variables** 33**Première application en mode console** 22**Primitives vectorielles** 74**Programmation événementielle** 210**Programmation orientée objet** 206**ProgressBar** 74**Projection** 109, 250, 253**Projection 3D** 245

plan 248

propriétés, modifier en C#

253

ProjectionMatrix 268**Projects, onglet** 37**Projets**

famille de 38

Silverlight 38

site Internet 51

Propagation événementielle 220

arrêter 225

exemple 222

phase de remontée 221

Properties, répertoire 41**PropertyPath** 155**Propriétés**

affecter 97

de dépendance 423

notifier un changement 399

vs. champs 28

Prototypage 277dynamique. *Voir* SketchFlow**ProxyRenderTransform** 154, 421

principes 154

public 24, 31**R****Ramasse-miettes** 104, 211**Random** 163**Rebond** 133

créer la balle 133

en C# 139

Rectangle 74**Refactoring** 449**References, répertoire** 41**Register** 424**ReleaseMouseCapture** 230

RenderTransform 109, 131

 affecter 153

 ciblage 130

RepeatBehavior 165

ResourceDictionary 319, 322

Resources 319, 333

Resources (propriété) 141

Ressources 118. *Voir aussi* **Polices**

 affecter à une propriété 319

 appliquer 324

 avec C# 325

 clé de 174

 dictionnaires 141, 318

 externaliser 322

 graphiques 315, 317. *Voir aussi* Pinceaux

 principes 316

 libérer 211

 logiques 317

 médias 317

 stocker 319

Resume 123

Rotation 3D 249

 axe 250

RoutedEvent 221

Run 342

S

sbyte 23

ScaleX 260

ScaleY 260

ScrollBar 74

 logique 378

Sécurité 477

 fichiers de 479

Seek 123

SelectedColor 425

SetTarget 123, 124

SetTargetName 123, 124

SetTargetProperty 123, 124

SetValue 424

Silverlight

 avantages 4

 définition 2

 langages de développement 9

 navigateurs et systèmes supportés 2

 objectifs 3

 positionnement métier 7

 première application 37

 système d'agencement (SLS) 103

Silverlight Layout System 103

Silverlight Toolkit 57

Site plein écran en 2 minutes 51

SketchFlow 277

 annotation 293

 carte de navigation 287

 comportements 285

 composants

 créer 308

 transitions 309

 configurateur riche 306

 écrans 287

 créer 288

 vs. composants 304

 états visuels 303, 311

 feedback 293

 flux de production 282

 importer

 croquis 290

 psd 298

 interactivité 298

 lecteur 290

 navigation 291

 utilisateur 300

 options de mise en forme 286

 principes 279

 projet 284

 partager 296

 prototype simple 283

 simuler des actions 302

 transition animée 289

 utilisation 282

SketchFlow Animation, panneau 302

SketchFlow Map 287

SLExtensions 453

Slider 355

Smooth Streaming 442

Solution, architecture 40

Souris, événements 228

Souscrire à un événement 208
arrêt 211

Spline 122, 132

StackPanel 53, 80
créer et configurer 53

StaticResource 325

stockage 23

Stop 123, 146

Storyboard 116, 119, 123, 192
dupliquer 146
propriétés 123

Streaming 440

Stretch 60, 171

string 24

StringUriSource 450

Stroke 77, 327

struct 24

Structures 24

Style 172, 348
accéder à 175
affecter 350
liaison de modèles 178
personnalisé 172
propriété prédéfinie 173

StyleTypedProperty 427

Survол 63

System.Linq 464

System.Text 147

System.Windows.Controls.dll 60

System.Windows.Controls.Toolkit.dll 60

System.Windows.Markup 148

System.Windows.Threading 162

T

Tableaux 25

Tag 219

TargetedTriggerAction 233

Tas 29

TestComponent 311, 314

TestPage.html 47

Tête de lecture 117

TextBlock 43, 53, 341

TextDecoration 341

Texte

ajouter 42
aligner 43
créer un champ 42
mise en forme 340
Voir aussi Champ texte

TextWrapping 341

TimeLine 165

timeLineProgress 448

TimeSpan 446

ToggleButton 201

Transformations 129

affines et non affines 266
Projection 109
relatives 129, 150, 189
types 150
vectorielles 109

TransformGroup 151

tester la présence 151

Transitions 183

comportements interactifs 198
Control (dans) 187
importance 186
spécifiques 191
Storyboard 192
système d'agencement fluide 203
tester 192
Voir aussi États visuels

Translation, axes 251

Transparence, animation 129

TriggerAction 232

Types

- anonymes 30
- complexes 24
- conversion 27
- primitifs 23

U

UniformToFill 171

UpdateSourceTrigger 402

UserControl 42, 74

UserCredentials 395

using 147

V

Value 120

var 30

Variables 23

- locales à la méthode 33

Vidéo

- charger dynamiquement 449
- codec 439
- format 438, 439

VideoElement 450

VisibleStates 196

VisualStateManager 183, 200

Visual Studio 7

VisualTransition 192

void 31

Vue perspective 272

W

WatermarkTextBox 234

WCF 3

WCS 3

WebClient 460

WebRequest 461

Width 58

Windows CardSpace 3

Windows Communication Foundation 3

Windows Presentation Foundation 4

**Windows Presentation Foundation
Everywhere** 2

Windows Workflow Foundation 4

WMSP 441

WPF 4

WPF/E 2

Wrap 341

WrapPanel 53, 55, 80
créer et configurer 56
options de mise en forme 58

WritableBitmap 241

WWF 4

X

XAML 4, 16
comparaison avec C# 17
convertir en chaîne de caractères 148
espaces de noms 19
utilité 16
visuel 18

XML 13
grammaire 14
JSON et 473
LINQ et 469
objectif 13

xmlns 19

Z

ZIndex 110

Pratique de Silverlight

Conception d'applications interactives riches

Le Web et son évolution ont rapproché développeurs et designers, et Silverlight offre l'environnement idéal pour leur permettre de communiquer et travailler de concert. Eric Ambrosi a ainsi fait le pari de s'adresser à ces deux profils.

Cet ouvrage s'articule autour des deux axes de conception que sont le design et le développement, et les met en perspective l'un par rapport à l'autre. Si vous êtes designer interactif, ce livre vous donnera une meilleure compréhension des contraintes liées au développement d'applications. Il vous immergera progressivement dans l'univers du développement orienté objet tout en vous apportant les clés de la conception graphique propre à Silverlight. Si vous êtes développeur, il vous sensibilisera aux contraintes nouvelles en matière de design, de prototypage et d'expérience utilisateur. Il facilitera le dialogue et la communication avec les autres acteurs de la production. Dans les deux cas, il est conçu pour fluidifier vos productions et établir les bases d'une réflexion inter-métier respectueuse de chaque profil.

Vous vous familiariserez progressivement, grâce à des exemples concrets et à une mise en pratique sous forme d'exercices téléchargeables, aux bonnes pratiques de Silverlight, et développerez rapidement des applications interactives riches de qualité.

TABLE DES MATIÈRES

- Introduction
- Le couple XAML / C#
- HelloWorld
- Un site plein écran en 2 minutes
- L'arbre visuel et logique
- Animations
- Boutons personnalisés
- Interactivité et modèle événementiel
- Les bases de la projection 3D
- Prototypage dynamique avec SketchFlow
- Ressources graphiques
- Composants personnalisés
- Médias et données

À propos de l'auteur :



Éric Ambrosi, MVP Silverlight, est responsable pédagogique et directeur technique chez Regart.net pour les plateformes Microsoft Silverlight et WPF. Infographiste dans les métiers de la postproduction et du multimédia depuis 1996, il est également cofondateur de l'école Européenne Supérieure d'Animation, rebaptisée école Méliès, dont il fut le responsable pédagogique de 1999 à 2003.

Développement
web

Niveau : Intermédiaire / Avancé
Configuration : Multiplate-forme,
Silverlight 3 et ultérieur

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4125-9

