Оглавление

[Общая информация о курсе 9](#_Toc310253242)

[Введение 11](#_Toc310253243)

[Об авторах 12](#_Toc310253244)

[Лекция 1. Краткое знакомство с моделью программирования XAML. 13](#_Toc310253245)

[Краткий экскурс в историю графических интерфейсов пользователя 13](#_Toc310253246)

[Windows Presentation Foundation – три в одном 15](#_Toc310253247)

[Декларативное программирование UI и XAML 15](#_Toc310253248)

[Основы модели UI и XAML 16](#_Toc310253249)

[Элементы XAML 17](#_Toc310253250)

[Панели 17](#_Toc310253251)

[Элементы управления 17](#_Toc310253252)

[Графические примитивы 18](#_Toc310253253)

[Службы документов 18](#_Toc310253254)

[Преобразования и анимация 18](#_Toc310253255)

[Ресурсы и стили 19](#_Toc310253256)

[Разделение труда дизайнера и разработчика 20](#_Toc310253257)

[XAML – текущая поддержка 21](#_Toc310253258)

[Модель программирования XAML 22](#_Toc310253259)

[Ключевые термины 26](#_Toc310253260)

[Краткие итоги 27](#_Toc310253261)

[Набор для практики 28](#_Toc310253262)

[Литература 28](#_Toc310253304)

[Лабораторная работа 1. Размещение элементов управления. 29](#_Toc310253305)

[Задания для самостоятельного выполнения 29](#_Toc310253306)

[Учебный элемент. Создание нового проекта для Silverlight в VisualStudio 2010. 29](#_Toc310253307)

[Шаг 1. Создание проекта 29](#_Toc310253308)

[Шаг 2. Добавляем элементы интерфейса для XAML-страниц 32](#_Toc310253309)

[Учебный элемент. Silverlight: размещение элементов. 34](#_Toc310253310)

[Шаг 1. Принципы разметки 35](#_Toc310253311)

[Шаг 2. Создаем Twitter-приложение 38](#_Toc310253312)

[Шаг 3. Навигация 40](#_Toc310253313)

[Шаг 4. Размещаем элементы для страницы поиска 40](#_Toc310253314)

[Шаг 5. Изменяем UriMapper для Search.xaml 42](#_Toc310253315)

[Краткие итоги 43](#_Toc310253316)

[Литература 43](#_Toc310253317)

[Лекция 2. Сведения о WPF и Silverlight: Введение и архитектура платформ. 44](#_Toc310253318)

[Windows Presentation Foundation 44](#_Toc310253319)

[Независимость от разрешения 44](#_Toc310253320)

[Архитектура Silverlight 46](#_Toc310253321)

[Состав ядра представления: 47](#_Toc310253322)

[В подмножество .NET Framework входят: 48](#_Toc310253323)

[Архитектура WPF 49](#_Toc310253324)

[Иерархия классов 51](#_Toc310253325)

[Элементы управления WPF 53](#_Toc310253326)

[Компоновочные элементы управления в системе WPF 55](#_Toc310253327)

[Ключевые термины 58](#_Toc310253328)

[Краткие итоги 59](#_Toc310253329)

[Набор для практики 59](#_Toc310253330)

[Литература 59](#_Toc310253364)

[Лекция 3. Стили и шаблоны элементов управления WPF. 60](#_Toc310253365)

[Стили 60](#_Toc310253366)

[Создание стиля 61](#_Toc310253367)

[Использование стиля 62](#_Toc310253368)

[*Размещение стилей* 63](#_Toc310253369)

[Основы шаблонов 64](#_Toc310253370)

[Создание шаблона 65](#_Toc310253371)

[Повторное использование шаблона Control’а 66](#_Toc310253372)

[ContentPresenter 67](#_Toc310253373)

[Связывание в Шаблонах 68](#_Toc310253374)

[Установка шаблонов через стили 69](#_Toc310253375)

[Повторное использование настроек цвета 71](#_Toc310253376)

[Ключевые термины 72](#_Toc310253377)

[Краткие итоги 72](#_Toc310253378)

[Набор для практики 72](#_Toc310253379)

[Литература 72](#_Toc310253409)

[Лабораторная работа 2. Стили и ресурсы. 73](#_Toc310253410)

[Задания для самостоятельного выполнения 73](#_Toc310253411)

[Учебный элемент. Стили и ресурсы в WPF 73](#_Toc310253412)

[Шаг 1. Применение стиля в XAML документе 73](#_Toc310253413)

[Шаг 2. Динамическое применение стилей в коде приложения 77](#_Toc310253414)

[Краткие итоги 81](#_Toc310253415)

[Литература 81](#_Toc310253416)

[Лекция 4. Silverlight и WPF в Visual Studio 2010. 82](#_Toc310253417)

[Новые возможности WPF 4. 82](#_Toc310253418)

[WPF история становления 82](#_Toc310253419)

[WPF 3.0. 82](#_Toc310253420)

[WPF 3.5. 82](#_Toc310253421)

[WPF 3.5 SP1. 82](#_Toc310253422)

[WPF 4. 82](#_Toc310253423)

[Visual Studio 2010 и WPF. 83](#_Toc310253424)

[Поддержка множества целевых платформ 83](#_Toc310253425)

[Клиентский профиль .NET 84](#_Toc310253426)

[Визуальный конструктор Visual Studio 85](#_Toc310253427)

[Новые средства WPF 4 86](#_Toc310253428)

[Новые элементы управления. 86](#_Toc310253429)

[Диспетчер визуального состояния. 86](#_Toc310253430)

[Поддержка Multitouch 86](#_Toc310253431)

[Windows 7 Shell Intergration 87](#_Toc310253432)

[Updated File Dialogs 88](#_Toc310253433)

[Custom Dictionary 88](#_Toc310253434)

[Графика 88](#_Toc310253435)

[Текст 90](#_Toc310253436)

[XAML 91](#_Toc310253437)

[Ключевые термины 94](#_Toc310253438)

[Краткие итоги 95](#_Toc310253439)

[Набор для практики 95](#_Toc310253440)

[Литература 96](#_Toc310253442)

[Лекция 5. Silverlight и WPF в Visual Studio 2010 (продолжение). 97](#_Toc310253443)

[Новые возможности Silverlight 4. Silverlight 5: основные нововведения ожидаемого релиза. 97](#_Toc310253444)

[Хронология Silverlight 97](#_Toc310253445)

[Обзор Silverlight 5 100](#_Toc310253446)

[Поддержка 64-битной платформы 101](#_Toc310253447)

[Улучшенное управление питанием 101](#_Toc310253448)

[Копирование изображений 101](#_Toc310253449)

[Поддержка 3D 101](#_Toc310253450)

[Поддержка Behaviors и Triggers 101](#_Toc310253451)

[Layout transition 101](#_Toc310253452)

[Связывание с данными 102](#_Toc310253453)

[Поддержка Full-Trust внутри браузера 102](#_Toc310253454)

[Обработка множественного нажатия кнопки мыши 102](#_Toc310253455)

[Поддержка выбора элементов в ItemsControl с клавиатуры 103](#_Toc310253456)

[Повышение полномочий для приложений в браузере 104](#_Toc310253457)

[Поддержка нескольких окон 106](#_Toc310253458)

[Доступ к файловой системе для приложений с повышенными привилегиями 107](#_Toc310253459)

[Установка имени файла по умолчанию в SaveFileDialog 108](#_Toc310253460)

[Улучшение работы с аудио 108](#_Toc310253461)

[Изменение скорости проигрывания медиа 108](#_Toc310253462)

[Поддержка пульта дистанционного управления 109](#_Toc310253463)

[Работа с текстом 109](#_Toc310253464)

[Печать 110](#_Toc310253465)

[P/Invoke 110](#_Toc310253466)

[Отладка при связывании с данными 111](#_Toc310253467)

[Улучшенная поддержка шаблона MVVM 113](#_Toc310253468)

[Использование связывания в стилях 115](#_Toc310253469)

[Неявные шаблоны 115](#_Toc310253470)

[Другие изменения 117](#_Toc310253471)

[Краткие итоги 118](#_Toc310253472)

[Набор для практики 118](#_Toc310253473)

[Литература 118](#_Toc310253499)

[Лекция 6. Привязка данных в технологиях WPF и Silverlight 119](#_Toc310253500)

[Принципы работы с данными 119](#_Toc310253501)

[Модель данных в .NET 119](#_Toc310253502)

[Всепроникающее связывание 120](#_Toc310253503)

[Преобразование данных 121](#_Toc310253504)

[Ресурсы 121](#_Toc310253505)

[Основные принципы связывания 125](#_Toc310253506)

[Привязка к объектам CLR 127](#_Toc310253507)

[Редактирование 129](#_Toc310253508)

[Ключевые термины 132](#_Toc310253509)

[Краткие итоги 133](#_Toc310253510)

[Набор для практики 133](#_Toc310253511)

[Литература 134](#_Toc310253541)

[Лекция 7. Пользовательские элементы управления. 135](#_Toc310253542)

[UserControl или CustomControl? 135](#_Toc310253543)

[UserControl 136](#_Toc310253544)

[Шаг 1. Создание повторно используемой функциональности 136](#_Toc310253545)

[Шаг 2. Добавление обработки событий клавиатуры 139](#_Toc310253546)

[Шаг 3. Инкапсуляция разметки и логики в элемент управления UserControl 140](#_Toc310253547)

[Использование пользовательского элемента управления 143](#_Toc310253548)

[CostomControl 144](#_Toc310253549)

[Шаг 1. Рефакторинг кода адресной формы 144](#_Toc310253550)

[Шаг 2. Рефакторинг кода разметки указателя цвета 147](#_Toc310253551)

[Шаг 3. Оптимизация шаблона элемента управления 149](#_Toc310253552)

[Ключевые термины 151](#_Toc310253553)

[Краткие итоги 152](#_Toc310253554)

[Набор для практики 152](#_Toc310253555)

[Литература 152](#_Toc310253585)

[Лабораторная работа 3. Пользовательские элементы управления. 153](#_Toc310253586)

[Задания для самостоятельного выполнения 153](#_Toc310253587)

[Учебный элемент. Создание пользовательского элемента управления средствами дизайнера VisualStudio 153](#_Toc310253588)

[Шаг 1. Создание Silverlight проекта 153](#_Toc310253589)

[Шаг 2. Создание пользовательского элемента управления 154](#_Toc310253590)

[Шаг 3. Class CustomControlDemo 156](#_Toc310253591)

[Шаг 4. Template CustomControlDemo 156](#_Toc310253592)

[Шаг 5. Использование Custom Control 156](#_Toc310253593)

[Учебный элемент. Создание пользовательского элемента управления средствами дизайнера Expression Blend 158](#_Toc310253594)

[Шаг 1. Инкапсуляция UI в User Control 158](#_Toc310253595)

[Шаг 2. Привязка адресов к нашему AddressUserControl – контроллеру. 161](#_Toc310253596)

[Краткие итоги 164](#_Toc310253597)

[Литература 164](#_Toc310253598)

[Лекция 8. Использование событий, команд и триггеров в технологиях WPF и Silverlight 165](#_Toc310253599)

[Принципиальные основы действий 165](#_Toc310253600)

[Композиция элементов 166](#_Toc310253601)

[Слабая связь 166](#_Toc310253602)

[Декларативные действия 168](#_Toc310253603)

[События 169](#_Toc310253604)

[Команды 171](#_Toc310253605)

[Команды и привязка к данным 176](#_Toc310253606)

[Триггеры 178](#_Toc310253607)

[Добавление триггеров к данным 179](#_Toc310253608)

[Добавление триггеров к элементам управления 181](#_Toc310253609)

[Краткие итоги 182](#_Toc310253610)

[Набор для практики 182](#_Toc310253611)

[Литература 182](#_Toc310253645)

[Лабораторная работа 4. Использование команд в технологиях WPF и Silverlight 183](#_Toc310253646)

[Задания для самостоятельного выполнения 183](#_Toc310253647)

[Учебный элемент. Использование стандартных и нестандартных команд 183](#_Toc310253648)

[Шаг 1. Использование стандартных команд. 183](#_Toc310253649)

[Шаг 2. Использование нестандартных команд. 184](#_Toc310253650)

[Шаг 3. Перевод команд из неактивного состояния в активное и обратно. 185](#_Toc310253651)

[Краткие итоги 186](#_Toc310253652)

[Литература 186](#_Toc310253653)

[Лекция 9. Применение паттерна MVVM как оптимального при проектировании WPF и Silverlight приложений 187](#_Toc310253654)

[Паттерн Model-View-ViewModel (MVVM) 187](#_Toc310253655)

[Обязанности и характеристики классов. 187](#_Toc310253656)

[Класс представления (View). 188](#_Toc310253657)

[Класс модели представления (View Model). 189](#_Toc310253658)

[Представление или Модель Представления? 190](#_Toc310253659)

[Класс модели (Model). 191](#_Toc310253660)

[Взаимодействие классов. 192](#_Toc310253661)

[Привязка данных. 192](#_Toc310253662)

[Реализация INotifyPropertyChanged. 193](#_Toc310253663)

[Реализация INotifyCollectionChanged. 193](#_Toc310253664)

[Реализация ICollectionView. 194](#_Toc310253665)

[Команды. 195](#_Toc310253666)

[Реализация Command Objects. 196](#_Toc310253667)

[Подтверждение правильности данных и сообщение об ошибках. 196](#_Toc310253668)

[Реализация IDataErrorInfo. 197](#_Toc310253669)

[Создание и соединение. 198](#_Toc310253670)

[Создание модели представления через XAML. 199](#_Toc310253671)

[Создание модели представления программно. 199](#_Toc310253672)

[Создание представления, определенного как шаблон данных. 200](#_Toc310253673)

[Краткие итоги 201](#_Toc310253674)

[Набор для практики 201](#_Toc310253675)

[Литература 201](#_Toc310253705)

[Лабораторная работа 5. Реализация WPF проекта с помощью MVVM toolkit’а 202](#_Toc310253706)

[Задания для самостоятельного выполнения 202](#_Toc310253707)

[Учебный элемент. Реализация паттерна MVVM средствами Model-View-ViewModel Toolkit 0.1 202](#_Toc310253708)

[Шаг 1. Постановка задачи 203](#_Toc310253709)

[Шаг 2. Model 204](#_Toc310253710)

[Шаг 3. ViewModel 204](#_Toc310253711)

[Шаг 4. View 205](#_Toc310253712)

[Шаг 4. Редактирование 206](#_Toc310253713)

[Шаг 5. Команды 208](#_Toc310253714)

[Краткие итоги 210](#_Toc310253715)

[Литература 210](#_Toc310253716)

[Лекция 10. Проектирование приложения с учетом использования единого опыта разработки для настольных и Web-проектов. 211](#_Toc310253717)

[Подходы к решению задачи 211](#_Toc310253718)

[Разработка кроссплатформенного проекта с самого начала 211](#_Toc310253719)

[Разработка для одной платформы с последующим портированием на другую 211](#_Toc310253720)

[Инструментарий 212](#_Toc310253721)

[Создание ссылок на файлы в Visual Studio 212](#_Toc310253722)

[Создание ссылок на XAML файлы 213](#_Toc310253723)

[Директивы препроцессора 216](#_Toc310253724)

[Разделяемые классы 217](#_Toc310253725)

[Синхронное использование XAML 218](#_Toc310253726)

[Решение проблем недостающего функционала 218](#_Toc310253727)

[Отсутствие FrameworkPropertyMetadata 218](#_Toc310253728)

[Отсутствие приведения значения 219](#_Toc310253729)

[Отсутствие метода OverrideMetadata() 219](#_Toc310253730)

[Отсутствие зависимых свойств только для чтения 221](#_Toc310253731)

[Отсутствие класса Brushes 221](#_Toc310253732)

[Stroke и Pen 222](#_Toc310253733)

[Конструкторы по умолчанию для геометрий 222](#_Toc310253734)

[Отсутствие свойства No PathSegment.IsStroked 222](#_Toc310253735)

[Краткие итоги 225](#_Toc310253736)

[Набор для практики 225](#_Toc310253737)

[Литература 226](#_Toc310253759)

[Лекция 11. Реализация паттерна MVVM с использованием IoC-контейнера, как метод избавления от зависимости между компонентами системы. 227](#_Toc310253760)

[Принцип инверсии зависимостей 227](#_Toc310253761)

[Формы инверсии зависимостей 229](#_Toc310253762)

[IoC контейнер 231](#_Toc310253763)

[MEF 231](#_Toc310253764)

[MVVM и IoC 234](#_Toc310253765)

[Определение моделей представления 234](#_Toc310253766)

[Определение представлений 235](#_Toc310253767)

[Сопоставление модели представления и представления 235](#_Toc310253768)

[Реализация INotifyPropertyChanged средствами аспектно-ориентированного программирования 238](#_Toc310253769)

[Краткие итоги 241](#_Toc310253770)

[Набор для практики 241](#_Toc310253771)

[Литература 242](#_Toc310253793)

[Лекция 12. Особенности отображения диалоговых окон в WPF и Silverlight версиях приложения. 243](#_Toc310253794)

[Понятие ICloseableViewModel и IChildViewModel 243](#_Toc310253795)

[Понятие IChildViewModelManager 244](#_Toc310253796)

[ChildViewManager 246](#_Toc310253797)

[Реализация ChildViewModelBase 251](#_Toc310253798)

[Реализация ModalChildViewModelBase 253](#_Toc310253799)

[Реализация MessageViewModel 254](#_Toc310253800)

[Краткие итоги 257](#_Toc310253801)

[Литература 257](#_Toc310253803)

[Лабораторная работа 6. Построение кроссплатформенного Silverlight/WPF приложения. 258](#_Toc310253804)

[Создание проектов 258](#_Toc310253805)

[Краткие итоги 269](#_Toc310253806)

[Список литературы 270](#_Toc310253807)

# Разделы дисциплины и виды занятий в часах

|  |  |  |  |
| --- | --- | --- | --- |
| **№ раздела** | **Наименования лекции (лабораторной работы)** | **Кол-во часов лек. / л. р.** | |
|  | Лекция 1. Краткое знакомство с моделью программирования XAML | 2 ч. |  |
|  | *Размещение элементов управления* |  | 2 ч. |
|  | Лекция 2. Сведения о WPF и Silverlight: Введение и архитектура платформ | 2 ч. |  |
|  | Лекция 3. Стили и шаблоны элементов управления WPF | 2 ч. |  |
|  | *Лабораторная работа 2. Стили и ресурсы* |  | 2 ч. |
|  | Лекция 4. Silverlight и WPF в Visual Studio 2010 | 2 ч. |  |
|  | Лекция 5. Silverlight и WPF в Visual Studio 2010 (продолжение) | 2 ч. |  |
|  | Лекция 6. Привязка данных в технологиях WPF и Silverlight | 2 ч. |  |
|  | Лекция 7. Пользовательские элементы управления | 2 ч. |  |
|  | *Лабораторная работа 3. Пользовательские элементы управления* |  | 2 ч. |
|  | Лекция 8. Использование событий, команд и триггеров в технологиях WPF и Silverlight | 2 ч. |  |
|  | *Лабораторная работа 4. Использование команд в технологиях WPF и Silverlight* |  | 2 ч. |
|  | Лекция 9. Применение паттерна MVVM как оптимального при проектировании WPF и Silverlight приложений | 2 ч. |  |
|  | *Лабораторная работа 5. Реализация WPF проекта с помощью MVVM toolkit’а* |  | 4 ч. |
|  | Лекция 10. Проектирование приложения с учетом использования единого опыта разработки для настольных и Web-проектов | 2 ч. |  |
|  | Лекция 11. Реализация паттерна MVVM с использованием IoC-контейнера, как метод избавления от зависимости между компонентами системы | 2 ч. |  |
|  | Лекция 12. Особенности отображения диалоговых окон в WPF и Silverlight версиях приложения | 2 ч. |  |
|  | *Лабораторная работа 6. Построение кроссплатформенного Silverlight/WPF приложения* |  | 6 ч. |
| **Итого** | | **24ч.** | **18ч.** |

# Общая информация о курсе

**Название курса:** Методология синхронной разработки Windows и Web версий бизнес-приложения средствами Microsoft Visual Studio 2010

**Название курса на английском языке:** Methodology of Windows and Web cross-platform business application development using Microsoft Visual Studio 2010.

**Краткая аннотация:** Разработка корпоративной системы все чаще подразумевает единовременное создание двух версий одного продукта – Windows и Web приложений, функциональность которых идентична. Оптимизацией решения задачи может служить синхронная разработка двух версий продукта средствами WPF и Silverlight технологий.

**Подробное описание:** Использование единого опыта разработки настольных и Web-проектов кажется сложновыполнимой задачей даже после выхода их последних версий. Но при правильном проектировании архитектуры проекта разработчику представится возможность большую часть кода (а в большинстве случаев – весь код) использовать одновременно в обеих версиях корпоративного бизнес-приложения. WPF и Silverlight создают новую парадигму в разработке приложений, используя единую базовую технологию декларативного программирования. Они идеально накладываются на шаблон проектирования MVVM в качестве основного архитектурного решения, упрощающего использование разделенного кода и дальнейшее расширение приложения.

**Цель курса:** Целью данного курса является ознакомление слушателей с методологией синхронной разработки двух версий информационной системы – Windows и Web приложений, средствами технологий WPF и Silverlight.

**Предварительные знания:** Слушатель должен иметь хорошее представление об объектно-ориентированном программировании (желательно на языке C#), знаком с архитектурой Microsoft.NET Framework и навыками работы в одной из версий Microsoft Visual Studio.

**Дополнительные курсы**

* **Основы XAML·**

https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=8199

* **Введение в разработку приложений на Windows Presentation Foundation**

http://www.intuit.ru/department/se/devwpf/

* **Методика создания интерактивных приложений Silverlight**

https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=8482

* **Разработка приложений на базе WPF и Silverlight**

https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=8822

**Авторы:** Виденин Сергей Александрович, Гризан Сергей Алексеевич.

## Введение

Сегодня многим разработчикам необходимо писать код, который должен выполняться как Silverlight так и .Net Runtime’мами. Хорошим примером является проверка данных, когда вы проверяете их сначала на стороне клиента, используя Silverlight, а затем на стороне сервера, используя .Net. До недавних пор нужно было компилировать один и тот же код в сборки для разных runtime’мов (Silverlight и .Net). Эта модель работоспособна, но не идеальна. Помимо этого, в последнее время разработка корпоративной системы все чаще подразумевает единовременное создание двух версий одного продукта – Windows и Web приложений, функциональность которых идентична. Оптимизацией решения задачи может служить синхронная разработка двух версий продукта средствами WPF и Silverlight технологий.

Несмотря на то, что Microsoft в последней версии .NET 4 и Silverlight 4 приложила большие усилия для поддержания совместимости кода WPF и Silverlight (теперь Silverlight использует новую CLR как основу для своей CoreCLR и тем самым вводит множество усовершенствований настольной версии в веб-версию.), по-прежнему остается множество тонкостей, которые нужно учитывать при разработке WPF\Silverlight приложений содержащих портируемый код.

При объединении сборок, Microsoft стояла перед большим выбором, какой сценарий выбрать. В итоге было принято решение позволить портировать сборки из Silverlight в .Net, а не наоборот. Мотивацией для такого решения послужил факт того, что Silverlight реализует подмножество сборок .Net и становится очевидно, что сборки использующее это подмножество будут работать и в .Net. Другим серьезным вопросом для разработчиков компании Microsoft был вопрос выбора тех самых сборок, которые будут безопасно работать и в Silverlight и в .Net. Было рассмотрено большинство сценариев, где может пригодиться портируемость. Как результат, для .Net 4 и Silverlight 4 выбрали только набор низкоуровневых сборок.

Различия между двумя наборами библиотек создает проблемы при параллельной разработке под две платформы. Одна и та же сборка не может быть использована на обеих платформах из-за различия ссылок на базовые сборки. Библиотека может быть ориентированна только на одну платформу. Тем не менее, существует ряд техник позволяющих переносить исходный код между платформами, о которых и будет идти речь в данном курсе.

Представленные в пособии примеры разрабатывались для Visual Studio 2010, C# 4, Net.Framework 4.0, Silverlight 4, SQL Server 2008.

## Об авторах

**Виденин Сергей Александрович.** Доцент, канд. пед. наук. Окончил Красноярский Государственный Педагогический Университет им. В. П. Астафьева. Работает доцентом кафедры «Информационные системы» Института космических и информационных технологий Сибирского федерального университета. Читает курс лекций по проектированию информационных систем. Наряду с учебно-методической работой продолжает практику программиста в софтверной компании Digital Mind Development г. Красноярск, где занимается разработкой широкого спектра бизнес-приложений с использованием современных программных платформ.

**Гризан Сергей Алексеевич.** Аспирант кафедры «Вычеслительная техника» Института космических и информационных технологий Сибирского федерального университета. Наряду с учебно-методической работой и написанием диссертации продолжает практику в должности архитектора софтверной компании Digital Mind Development г. Красноярск, где занимается построением архитектурных решений бизнес-приложений с использованием современных программных платформ.

# Лекция 1. Краткое знакомство с моделью программирования XAML.

**Краткая аннотация лекции:** в лекции рассматривается технология декларативного программирования при помощи языка XAML.

**Цель лекции:** показать читателям предпосылки возникновения модели декларативного программирования XAML, разобрать основополагающие элементы XAML (контейнеры, элементы управления, службы документов, графические примитивы), а так же кратко познакомить с текущей поддержкой языка XAML, и моделью программирования.

## Краткий экскурс в историю графических интерфейсов пользователя

Человечество проделало огромный путь в технологиях создания пользовательского интерфейса. Когда-то, на заре компьютерной эры, единственный интерфейс пользователя составляла колода перфокарт, которые программист загружал в компьютер, и бумажная лента, которую компьютер ему выдавал. Первым революционным шагом стало появление консоли, что автоматически (хотя и далеко не сразу) привело к смене парадигмы общения компьютера и пользователя с пакетной обработки к интерактивной. Довольно долго консоли были текстовыми, и пользовательский интерфейс, в основном, сводился к вопросам и ответам. Однако и в консоли со временем появились довольно богатые интерфейсы, использовавшие возможности псевдографики.

В то же время становится популярной Windows, что в очередной раз меняет подходы к созданию интерфейсов. Windows предложила набор стандартных элементов управления (или, по-русски, «контролов»), а также API для работы с ними. Это позволило приложениям выглядеть одинаково с системными, не говоря уже о существенном облегчении в программировании UI, т.к. большую часть труда взяла на себя система.

Поначалу набор контролов был недостаточен, а API вызывало много нареканий, но с развитием Windows положение вещей улучшалось. Кроме того, появилось много альтернатив низкоуровневому API Windows. В частности, можно упомянуть «родные» Microsoft’овские технологии MFC и ATL, которые были интегрированы со средством разработки Microsoft – Visual Studio. С помощью этих технологий можно разрабатывать интерфейсы в объекто-ориентированном стиле, кроме того, в них реализован шаблон («паттерн») model-view-controller, позволяющий разделить представление и данные. Стоит упомянуть и другие технологии, такие как кроссплатформенная Qt и, конечно же, Java c ее AWT и Swing.

Параллельно с появлением и развитием технологий, подобных MFC, появились и стали развиваться технологии для работы с продвинутой графикой. Так, в 1995-м году в одноименной системе Windows 95 появилась технология DirectX, облегчающая работу по созданию игр и мультимедиа-приложений. За 3 года до Microsoft’овского DirectX появилась технология OpenGL компании Silicon Graphics. Точнее сказать, это стандарт, выработанный несколькими компаниями (и, в частности, Microsoft), выросший из разработки Silicon Graphics. В отличие от DirectX, OpenGL является кросс-платформенным и стандартным. Нас же интересует то, что развитие графических технологий с этого времени шло параллельно с развитием технологий создания «обычного» UI.

Наконец, надо упомянуть и третью ветвь развития UI, появившуюся примерно в то же время – UI сначала веб-страниц, потом и веб-приложений. Поначалу это был чистый HTML, потом как грибы после дождя стали появляться многочисленные технологии вроде Java-апплетов, flash-приложений и т.п. Наконец, с выходом Microsoft.NET появились технологии Windows Forms и ASP.NET.

Итак, с точки зрения UI у нас существует три разных типа приложений, каждый из которых имеет свой набор технологий для реализации. И одной из предпосылок создания **WPF** является идея интеграции подобных технологий в одной. Идея интеграции всего со всем вообще является доминирующей последние годы. Кроме того, текущей графической платформе Windows уже 20 лет. Это прекрасно, но, учитывая ошеломительное развитие аппаратных графических средств (видеокарт и разнообразных графических ускорителей), потенциально мы могли бы использовать всю их мощь не только в приложениях типа игр, но и в обычных приложениях. Мешает нам только то, что с помощью привычных технологий типа MFC или Windows Forms крайне трудно сделать изощренный интерфейс, а тратить время на создание оного с помощью DirectX или OpenGL представляется малоосмысленным. Таким образом, одной из целей создания WPF было предоставить преимущества современных аппаратных графических средств более широкой аудитории разработчиков.

Наконец, еще одной предпосылкой появления WPF стала неудовлетворенность текущей схемой общения дизайнера и разработчика при создании UI. Это мы обсудим чуть позже в данной лекции.

## Windows Presentation Foundation – три в одном

WPF – графическая подсистема, основанная на управляемом коде. WPF является надстройкой над DirectX, что дает приложениям, написанным с её помощью, богатые возможности по отрисовке. Эта технология объединяет в себе три мира, перечисленные ранее, – UI для настольных и веб-приложений, и UI для игр и мультимедиа-приложений. WPF содержит в себе множество элементов управления, поддержку видео, анимации, трехмерных изображений и т.п.

Надо отметить, что WPF – это не только диалоги, картинки и видео. Кроме прочего, WPF включает в себя также синтез и распознавание речи. Более полный список того, что поддерживается WPF:

* 2D
* 3D
* Работа с текстами и шрифтами
* Работа с изображениями
* Эффекты
* Аудио
* Видео
* Анимация и работа с временными интервалами

## Декларативное программирование UI и XAML

Одной из идей WPF является то, что программирование UI хочется сделать как можно более декларативным. В самом деле, почему размеры и расположение элементов управления, реакцию на всякие события и т.п. надо реализовывать в коде, если эта информация известна заранее? Эти соображения привели разработчиков WPF к идее использовать XML для описания пользовательского интерфейса. Стандарт называется **XAML** (eXtensible Application Markup Language), и файл на нем выглядит примерно так:

<Button Width="200px" Click="onHelloClick">  
 Hello, XAML!  
 <Button.Background>  
 LightBlue  
 </Button.Background>  
</Button>

Вы могли бы подумать, что вам предлагается всего лишь описать раскладку элементов управления и обработчики событий, но это не так. Например, вы можете описать различные трансформации ваших элементов управления, их анимацию и т.п.

Идея использовать XML для описания UI не нова. Достаточно популярен язык разметки **XUL** (XML User Interface Language). XUL является частью среды разработки кросс-платформенных интерфейсов, известной как XPFE. Это полнофункциональный язык разметки, на объекты приложений, такие как окна, метки и кнопки.

C помощью XAML описывается, прежде всего, пользовательский интерфейс. Логика приложения по-прежнему управляется процедурным кодом (С#, VB и т.д.).

XAML может использоваться как для браузер-базированных приложений, так и для локальных настольных приложений. Немаловажной деталью является легкость XAML для локализации UI. Если раньше для этих целей использовали специальные механизмы вроде таблицы строк, загрузки строк из ini-файлов, ресурсные dll или же специальных утилит «вытаскивания» строк, то с появлением XAML, хранящего все строчки UI в текстовом виде, локализация значительно упрощается.

Наконец, формат XAML, основанный на XML, очень легко генерировать программно, что открывает более широкие возможности для третьесторонних разработчиков, реализующих различные инструменты работы с UI, по сравнению с текущей ситуацией, когда приходится генерировать UI в закрытых проприетарных форматах.

## Основы модели UI и XAML

Интерфейс пользователя WPF-приложения задается XAML. C XAML может быть связан скомпилированный code-behind код (что может вызвать легкое ощущение дежавю у знакомых с моделью ASP.NET). Например, если страница вашего приложения описывается файлом page.xaml, то code-behind, как правило, будет храниться в page.xaml.cs. Codebehind может содержать реакцию на различные события, генерируемые пользовательским интерфейсом (такие как нажатие клавиши мыши или «наезд» указателя мыши на элемент управления). Одной из светлых целей такого разделения является написание одного кода для всех типов приложений (то есть, ваш код можно было бы скомпилировать и как настольное приложение, и как приложение, просматриваемое с помощью браузера, и как smart client-приложение).

Т.к. XAML – это «нормальный» XML, то он подчиняется всем правилам wellformed XML, в частности, содержит ровно одну корневую вершину и является деревом. На вершине иерархии находится один из контейнерных объектов. Внутри этих объектов располагаются знакомые нам элементы управления и другие контейнеры. Набор элементов управления, без сюрпризов, очень похож на «старый добрый» набор контролов Win32. Это кнопки, меню, текст, картинки и т.п. Но на самом деле каждый тег XAML соответствует классу модели, который, в свою очередь, имеет набор свойств, методов и событий. В соответствие с этими членами класса вы можете настраивать ваш XAML-код. Во время исполнения именно экземпляры этих классов создаются рантаймом для того, чтобы отобразить то, что вы указали в XAML, – очень похоже на ASP.NET и его серверные контролы.

## Элементы XAML

Основное деление элементов XAML таково:

* Контейнеры (панели)
* Элементы управления
* Службы документов (document services)
* Графические примитивы

### Панели

С помощью панелей вы можете располагать содержащиеся внутри них элементы. Среди стандартных панелей есть Canvas (дочерние элементы размещаются с использованием относительных координат), DockPanel (панель, в которой дочерние элементы стыкуются), FlowPanel (напоминает джавский FlowLayout, где элементы выводятся в ряд друг за другом), GridPanel (табличная организация сыновей) и некоторые другие. Естественно, вы можете определять свои типы панелей. Пример панели:

<DockPanel xmlns=”http://schemas.microsoft.com/2003/xaml>  
 <Button DockPanel.Dock=”Left” Width=”50px” Height=”30px”>One</Button>  
 <Button DockPanel.Dock=”Left” Width=”50px” Height=”30px”>Two</Button>  
 <Button DockPanel.Dock=”Left” Width=”70px” Height=”30px”>Three</Button>  
</DockPanel>

На этой панели будет создано три кнопки, состыкованные друг с другом по горизонтали. Обратите внимание на указание соответствующего пространства имен в элементе верхнего уровня.

### Элементы управления

Элементы управления, как уже было сказано, во многом уже знакомы программистам Win32 и WinForms. Все они унаследованы от типа Control, среди них есть старые добрые комбобоксы, листбоксы, меню, скроллбары, слайдеры, и т.п., а есть и новые контролы наподобие RadioButtonList. XAML однако же дает вам невероятную гибкость в настройке свойств этих контролов (то, чего ранее можно было добиться только нелегким трудом ручной отрисовки owner draw). Так, например, чтобы нарисовать кнопку с изображением (что, правда, стало несложно и во второй версии WinForms), в WPF надо всего лишь написать

<Button>  
 <Image Source=”myphoto.jpg”>  
</Button>

Вы можете задавать различные цвета, кисти, градиентные заливки и даже сделать кнопку с чек-боксом на ней!

### Графические примитивы

С ними все понятно, набор весьма стандартен – Ellipse, Line, Rectangle, Path, Polygon, Polyline, для которых можно использовать заливки (Fill) и штрихи (stroke), причем всё это можно рисовать разными кистями, использовать градиенты, устанавливать толщину линий и т.п. Кто работал в векторных редакторах наподобие Macromedia Flash или Adobe Illustrator, тому эти возможности знакомы.

Графические примитивы не имеют дочерних элементов и обычно содержатся внутри панели Canvas, например:

<Canvas xmlns = ”http://schemas.microsoft.com/2003/xaml”>  
 <Line X1 = ”0” Y1 = ”0” X2 = ”100” Y2 = ”200” Stroke = ”red” />  
</Canvas>

нарисует обычную линию красного цвета из 0,0 в 100, 200 (в локальных координатах панели).

### Службы документов

В то время как традиционная веб-страница имеет непрерывное представление, отображая весь контент в одном едином блоке, сервисы управления документами предоставляют вам удобные возможности разбивать контент на страницы, масштабировать содержимое, адаптировать документы под специфические особенности дисплея пользователя и т.д. Еще один шаг, приближающий веб-интерфейс к привычному интерфейсу настольных приложений.

## Преобразования и анимация

С помощью XAML вы можете определить различные преобразования (трансформации) над элементами интерфейса. Среди стандартных преобразований есть:

* RotateTransform – поворот на указанный угол
* TranslateTransform – сдвиг на указанные смещения
* ScaleTransform – растягивание или сужение в указанное количество раз
* SkewTransform – искажение на указанные углы относительно указанного центра
* MatrixTransform – любое аффинное преобразование
* TransformCollection – суперпозиция нескольких преобразований

Например, вы можете повернуть обычную надпись с помощью единственного преобразования:

<Canvas xmlns = "http://schemas.microsoft.com/2003/xaml/">  
<TransformDecorator>  
 <TransformDecorator.Transform>  
 <RotateTransform Angle="45"/>  
 </TransformDecorator.Transform>  
 <Text>Hello!</Text>  
</TransformDecorator>  
</Canvas>

или отобразить список (ListBox) с помощью нескольких преобразований, объединенных TransformCollection, так, что он будет растянут и перевернут:

<Canvas xmlns=”http://schemas.microsoft.com/2003/xaml”>  
 <TransformDecorator>  
 <TransformCollection>  
 <RotateTransform Angle=”100”/>  
 <ScaleTransform ScaleX=”2”, ScaleY=”1”/>  
 </TransformCollection>  
 <ListBox>  
 <ListBoxItem> One </ListBoxItem>  
 <ListBoxItem> Two </ListBoxItem>  
 <ListBoxItem> Three </ListBoxItem>  
 </ListBox>  
 </TransformDecorator>  
</Canvas>

Кроме преобразований XAML предлагает вам также возможности анимации (нечто похожее на преобразования, но что можно повторять и применять многократно). Например, вы можете сделать кнопку, которая при наведении на нее мышки будет постепенно растягиваться до некоторых пределов (не скачком) и так же постепенно сужаться после того, как мышка будет отведена. Анимировать можно все свойства элементов управления, что позволяет вам создавать сколь угодно изощренные эффекты анимации. Правда сказать, создавать эти эффекты руками – не самое благодарное занятие. Например, взгляните на доступные для скачивания примеры XAML на сайте http://xamlshare.com. Конечно же, для создания более-менее сложных UI лучше использовать продукты семейства Expression (рассматриваемые ниже).

## Ресурсы и стили

Важный вопрос – каким образом можно управлять всем этим великолепием с целью улучшения повторного использования внешнего вида элементов. Тут XAML предлагает нам идеи, схожие с идеями CSS. Для начала вводится понятие ресурса: **ресурс – это именованное значение некоторого элемента XAML**. Например:

<Canvas xmlns=”http://schemas.microsoft.com/2003/xaml”>  
 <Canvas.Resources>  
 <SolidColorBrush def:Name=”MyFavoriteColor” Color=”Magenta”/>  
 </Canvas.Resources>  
</Canvas>

определяет ресурс под именем MyFavoriteColor типа SolidColorBrush с заданным значением цвета кисти, который мы можем переиспользовать в любом контексте, в котором ожидается кисть, например:

<Button Background=”{MyFavoriteColor}”/>

Коллекция ресурсов есть у каждого элемента XAML. В случае, если какой-либо элемент ссылается на ресурс, но в нем самом ресурсы не определены (или такой ресурс не найден), то поиск производится вверх по дереву вложенности. Важным типом ресурса является стиль, который описывается так:

<Style def:Name=”MagentaButtonWithLargeFont”>  
 <Button Background=”{MyFavoriteColor}” FontSize=”100”>  
</Style>

и который затем можно применить к кнопке:

<Button Style=”{MagentaButtonWithLargeFont}”>My Magenta Button</Button>

Интересным применением стилей является изменение внешнего вида контролов по событиям. Так, например, вы можете изменить стиль кнопки по наведению на нее мышки, что может быть описано в теле стиля с помощью тега PropertyTrigger.

Каждый тэг XAML соответствует классу объектной модели WPF. Тэг обычно имеет набор атрибутов, которые соответствуют свойствам этого класса. Во время компиляции парсер по XAML-описанию порождает частичный (partial) класс, который содержит эквивалентный код. Каждый тэг XAML становится экземпляром соответствующего класса объектной модели, значения атрибутов тэга присваиваются соответствующим свойствам этого объекта. Затем частичный класс, порожденный из XAML, объединяется с code-behind кодом, написанным программистом. Все это дает нам возможность, кроме всего прочего, порождать интерфейс на лету.

## Разделение труда дизайнера и разработчика

Как выглядит общение дизайнера и разработчика в настоящий момент? Как правило, дизайнер создает макет пользовательского интерфейса в графическом редакторе, на бумаге или с помощью какого-либо другого средства – любого, кроме средства разработки приложения. Как следствие, конкретную реализацию мечтаний дизайнера все равно создает программист. Это ведет к тому, что, во-первых, дизайнер может создать макет UI, который в принципе нереализуем или труднореализуем, а во-вторых, программист может ошибиться в реализации или же по-своему интерпретировать те или иные пожелания дизайнера. В итоге дизайнер вынужден пересматривать получившееся творение и высказывать пожелания к улучшению интерфейсов. Таким образом, разработка UI превращается в многоэтапный итеративный процесс, тормозящий основную разработку. С помощью WPF станет возможным исключить такие ситуации. Наличие в нем общего знаменателя – языка описания интерфейсов XAML – позволит разделить работу дизайнера и программиста. В самом деле, программисту от XAML в основном нужны только имена элементов управления и обработчики их событий, которые он использует в своей программе. Дизайнер же может спокойно разрабатывать UI в инструментах, более приспособленных для этой задачи, а не в непонятных ему средах разработки приложений.

## XAML – текущая поддержка

Итак, для разделения труда программиста и разработчика нам нужны соответствующие интегрированные средства для работы с XAML в IDE и в графических редакторах. Таким средства есть:

* **Visual Designer for the Windows Presentation Foundation** (ранее известный как Cider) – это то, с чем работает программист в Visual Studio при разработке. Visual Designer для WPF станет частью Visual Studio в ее следующей версии, Orcas, поддерживающей Vista. Orcas планируется выпустить в 2007 году
* **Expression Graphic Designer** (бывшая Acrylic), программа для работы с 2D-графикой, – это то, с чем работает дизайнер вне всяких IDE. В Graphic Designer сделан упор на внешний вид приложений
* **Expression Interactive Designer** (ранее известный как Sparkle) – инструмент для создания 3D-анимации и графики. Interactive Designer уделяет внимание "поведению и взаимодействию"
* **Expression Web Designer** (бывший Quartz) – инструмент для разработки веб-страниц. По сути новая и улучшенная версия FrontPage, предназначенная для разработки пользовательского интерфейса и верстки веб-страниц и Windows-приложений. Тесно интегрируется с ASP .NET 2.0 (в частности, «знает» о контролах ASP.NET, смарт-тэгах и т.п.)
* **XAMLPAD** – названный, видимо, по аналогии с NotePad, несложный инструмент редактирования XAML.

Семейство Expression-редакторов работает как с векторной графикой (напомним, что движок WPF – векторный), так и с растровыми изображениями.

## Модель программирования XAML

Язык XAML обладает дополнительной по сравнению с XML семантикой, которая допускает единую интерпретацию. Слегка упрощая, можно сказать, что XAML - это основанный на XML сценарный язык для создания объектов CLR. Имеется соответствие межу XML-тегами и типами CLR, а также между XML-атрибутами и свойствами и событиями CLR. В следующем примере показано, как создать объект и присвоить значение его свойству на языках XAML и C#:

<!— версия на XAML —>  
<MyObject SomeProperty='1' />

// версия на C#  
MyObject obj = new MyObject();   
obj.SomeProperty = 1;

XML-теги всегда определяются в контексте некоторого пространства имен, которое и описывает, какие теги являются допустимыми. В XAML пространства имен в смысле XML отображаются на наборы пространств имен и сборок в смысле CLR. Чтобы приведенный выше простой пример заработал, необходимо установить соответствие между требуемыми пространствами имен. В XML для определения новых пространств имен применяется атрибут xmlns:

<!— версия для XAML —>  
<MyObject xmlns='clr-namespace:Samples' SomeProperty='1' />

// версия для C# using Samples;  
MyObject obj = new MyObject();   
obj.SomeProperty = 1;

В C# перечень сборок, в которых находятся используемые типы, всегда задается в файле проекта или с помощью аргументов командной строки для запуска компилятора csc.exe. В XAML можно определить местоположение исходной сборки для каждого пространства имен:

<!— версия для XAML —>  
<MyObject xmlns='clr-namespace:Samples; assembly=samples.dll'  
 SomeProperty='1'/>

// версия для C#  
csc /r:samples.dll test.cs  
using Samples;  
MyObject obj = new MyObject();  
obj.SomeProperty = 1;

В XML мир разбит на две половины: элементы и атрибуты. Модель XAML более тесно связана с CLR, поскольку апеллирует к объектам, свойствам и событиям. Значения свойств можно представлять в виде атрибутов или дочерних элементов. Предыдущий пример допустимо записать и так:

<MyObject xmlns='clr-namespace:Samples, assembly=samples.dll'>  
 <MyObject.SomeProperty>  
   
 </MyObject.SomeProperty>  
</MyObject>

Каждый элемент, соответствующий свойству, квалифицируется типом, которому это свойство принадлежит. Предположим, например, что есть еще одно свойство, значением которого является объект типа Person со свойствами FirstName и LastName. На XAML можно было бы легко выразить это соотношение, воспользовавшись элементами для описания свойств:

<MyObject xmlns='clr-namespace:Samples, assembly=samples.dll'>  
 <MyObject.Owner>  
 <Person FirstName='Chris' LastName='Anderson' />  
 </MyObject.Owner>  
</MyObject>

XAML проектировался как язык разметки, тесно интегрированный с CLR и обеспеченный развитой инструментальной поддержкой. Дополнительно ставилась цель создать такой формат, который было бы легко читать и записывать. Может показаться, что проектировать свойство платформы, которое оптимизировано прежде всего для инструментов, а лишь потом для людей, не слишком вежливо, но команда WPF полагала, что приложения для WPF как правило будут создавать с помощью таких программ визуального конструирования, как Microsoft Visual Studio или Microsoft Expression. Чтобы граница между инструментами и людьми не была непроходимой, WPF позволяет автору типу определить одно свойство как контентное.[[1]](#footnote-1)

В примере выше, если сделать свойство Owner типа MyObject контентным[[2]](#footnote-2), то в разметке можно будет опустить тег элемента, соответствующего этому свойству:

<MyObject xmlns='clr-namespace:Samples; assembly=samples.dll'>  
 <Person FirstName='Megan' LastName='Anderson' />   
</MyObject>

Чтобы воспринимать текст было еще удобнее, в XAML есть возможность расширения разметки. Это общий способ расширить синтаксический анализа­тор языка с целью создания более простой разметки. Расширения реализуются в виде типов CLR и работают почти так же, как атрибуты CLR. Они заключаются в фигурные скобки { }. Например, чтобы присвоить свойству специаль­ное значение null, можно воспользоваться встроенным расширением разметки Null:

<MyObject xmlns='clr-namespace:Samples; assembly=samples.dll'>  
 <Person FirstName='Megan' LastName='{x:Null}'/>   
</MyObject>

В таблице 1.1 перечислены все встроенные расширения XAML.

Таблица 1.1. Встроенные расширения XAML

|  |  |  |
| --- | --- | --- |
| **Пространство имен XAML** | **Назначение** | **Пример** |
| x:Array | Создает массив CLR | <x:Array Type='{x:Type Button}'>  <Button />  <Button />  </x:Array> |
| x:Class | Задает имя определяемого типа (используется только при компиляции разметки) | <Window  x:Class = 'MyNamespace.MyClass '>  ...  </Window> |
| X:ClassModifier | Задает модификаторы определяемого типа («public» «intemal» и т.д.) (используется только при компиляции разметки) | <Window x:Class='...'  x:ClassModifier='Public'>  </Window> |
| x:Code | Ограничивает блок встроенного кода (используется только при компиляции разметки) | <Window x:Class='...'>  <x:Code>  public void DoSomething()  {  }  </x:Code>  </Window> |
| x:Key | Задает ключ элемента (поддерживается только для элементов, содержащихся в словарях) | <Button>  <Button.Resources>  <Style x:Key='Hi'>...</Style>  </Button.Resources>  </Button> |
| x:Name | Задает имя элемента для ссылки на него в программе (обычно используется, когда у элемента нет встроенного свойства name) | <sys:Int32  xmlns:sys = 'clr-namespace: System;...'  x:Name = '\_myIntegerValue'> 5 </sys:Int32> |
| x:Null | Создает значение | null. <Button Content = '{x:Null}' /> |
| x:Static | Создает значение путем доступа к статическому полю или свойству типа. | <Button  Command = '{x:Static  ApplicationCommands.Close}' /> |
| x:Subclass | Предоставляет базовый тип для компиляции разметки на язык, в котором не поддерживаются частичные типы. |  |
| x:Type | Предоставляет тип CLR (эквивалент  Type.GetType). | <ControlTemplate  TargetType='{x:Type Button}'>  </ControlTemplate> |
| x:TypeArguments | Задает обобщенные аргументы типа для создания экземпляра обобщенного типа. | <gc:List xmlns:gc='clrnamespace:  System.Collections.Generic;...' |
| x:TypeArguments ='{x:Type Button}'/> x:XData | Ограничивает блок встроенного XML; может использоваться только для свойств типа IXmlSerializable. | <XmlDataSource>  <x:XData>  <Book xmlns=" Title='...' />  </x:XData>  </XmlDataSource> |

Расширения разметки ищутся точно так же, как теги объектов, то есть необходи­мо объявить XML-префикс «x», иначе синтаксический анализатор выдаст ошибку. В языке XAML определено специальное пространство имен для встроенных типов:

<MyObject   
 xmlns:x='<http://schemas.microsoft.com/winfx/2>006/xaml'   
 xmlns='clr-namespace:Samples,-asserrbly=samples.dll'>  
  
 <Person FirstName='Megan' LastName='{x:Null}' />  
</MyObject>

Кроме того, для любой сборки CLR (или набора таких сборок) можно опреде­лить имя, построенное по принципу URI и соответствующее пространствам имен и сборок CLR. Это можно считать эквивалентом старого доброго предложения #include 'windows.h', которое хорошо известно программистам на C/C++. В сбор­ках WPF этот механизм применяется, поэтому для импорта WPF в XAML-файл можно использовать любой формат:

<!— вариант 1: импорт по пространству имен CLR —>   
<Window  
 xmlns:x='<http://schemas.microsoft.com/winfx/2> 0 06/xaml'  
 xmlns='clr-namespace:System.Windows;assembly=presentationframework.dll'>  
</Window>  
  
<!— вариант 2: импорт по URI —>  
<Window  
 xmlns:x='<http://schemas.microsoft.com/winfx/2> 0 06/xaml'   
 xmlns='<http://schemas.microsoft.com/winfx/2> 006/xaml/presentation'>   
</Window>

Метод с применением синтаксиса URI хорош тем, что импортируются сразу несколько пространств имен и сборок CLR, а, значит, разметка получается более компактной и работать с ней проще.

В завершении лекции, нужно сказать о такой возможности XAML, как способность расширять типы за счет свойств, предоставляемых другими типами. Такие присоединенные свой­ства - это просто безопасный относительно типов вариант добавленных свойств (expando properties) в языке JavaScript. В версии XAML, предназначенной для WPF, присоединенные свойства работают только, если и тип, в котором свойство определено, и тип, к которому оно присоединяется, наследуют классу DependencyObject. Однако в общей спецификации XAML такого требования нет.

В следующем примере свойство Dock определено в типе DockPanel. Присое­диненному свойству всегда предшествует имя предоставляющего его типа, даже если такое свойство употребляется в качестве атрибута:

<Window  
 xmlns:x='<http://schemas.microsoft.com/winfx/2> 0 06/xaml'  
 xmlns='<http://schemas.microsoft.com/winfx/2> 006/xaml/presentation'>  
 <DockPanel>  
 <Button DockPanel.Dock='Top'>Top</Button>  
 <Button>  
 <DockPanel.Dock>Left</DockPanel.Dock>  
 Left  
 </Button>  
 <Button>Fill</Button>  
 </DockPanel>  
</Window>

XAML - довольно простой язык, в нем не очень много правил. В версии, ко­торая поставляется в составе .NET Framework 3.0 (4.0), все определения тегов XAML реализованы в виде типов CLR. Поэтому все, что можно сделать с помощью раз­метки, можно написать и в виде компилируемой программы.

## Ключевые термины

**API** (Windows API англ. application programming interfaces) — общее наименование целого набора базовых функций интерфейсов программирования приложений операционных систем семейств Windows и Windows NT корпорации «Майкрософт». Является самым прямым способом взаимодействия приложений с Windows. Для создания программ, использующих Windows API, «Майкрософт» выпускает SDK, который называется Platform SDK и содержит документацию, набор библиотек, утилит и других инструментальных средств.

**CLR** (Common Language Runtime) — «общеязыковая исполняющая среда» — компонент пакета Microsoft .NET Framework, исполняющий программы, написанные на .NET-совместимых языках программирования.

**CSS** (англ. Cascading Style Sheets — каскадные таблицы стилей) — формальный язык описания внешнего вида документа, написанного с использованием языка разметки.

**Silverlight** — это программная платформа, включающая в себя плагин для браузера, который позволяет запускать приложения, содержащие анимацию, векторную графику и аудио-видео ролики, что характерно для RIA (Rich Internet application).

**UI** (англ. user interface) — разновидность интерфейсов, в котором одна сторона представлена человеком (пользователем), другая — машиной/устройством. Представляет собой совокупность средств и методов, при помощи которых пользователь взаимодействует с различными, чаще всего сложными, машинами, устройствами и аппаратурой.

**URI** (англ. Uniform Resource Identifier) — унифицированный (единообразный) идентификатор ресурса. URI — это последовательность символов, идентифицирующая абстрактный или физический ресурс. Ранее назывался Universal Resource Identifier — универсальный идентификатор ресурса.

**WPF** (Windows Presentation Foundation, кодовое название — Avalon) — система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая (презентационная) подсистема в составе .NET Framework (начиная с версии 3.0), имеющая прямое отношение к XAML.

**XAML** (англ. eXtensible Application Markup Language — расширяемый язык разметки приложений — основанный на XML язык разметки для декларативного программирования приложений, разработанный Microsoft.

**XML** (англ. eXtensible Markup Language — расширяемый язык разметки) — рекомендованный Консорциумом Всемирной паутины язык разметки, фактически представляющий собой свод общих синтаксических правил. XML — текстовый формат, предназначенный для хранения структурированных данных (взамен существующих файлов баз данных), для обмена информацией между программами, а также для создания на его основе более специализированных языков разметки (например, XHTML).

## Краткие итоги

В данной теме были рассмотрены основные синтаксические конструкции языка XAML и приведены простые примеры их реализации. Проектирование WPF-приложений требует от разработчика понимания структуры и содержания XAML-документа, описывающего интерфейс создаваемой системы. В последующих темах курса будут рассматриваться различные аспекты построения WPF- и Silverlight-приложений и необходимым условием эффективной работы создателя таких систем является знание основ языка XAML.

## Набор для практики

Вопросы:

1. Имеется ли возможность вложенности друг в друга у элементов XAML?
2. Сколько корневых элементов может иметь XAML-документ?
3. Поясните назначение пространства имен x:Static.
4. Имеется ли возможность реализовать в виде типов CLR теги XAML?
5. Приведите синтаксис присоединения обработчиков событий.

## Литература

1. XAML в WPF <http://msdn.microsoft.com/ru-ru/library/ms747122.aspx>
2. XUL, XML <http://www.xulplanet.com> <http://xul.ru> <http://ru.wikipedia.org/wiki/XUL>
3. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011, стр. 48 – 81.
4. Нейгел, К. C# 4.0 и платформа .NET 4 для профессионалов. / К. Нейгел, Б. Ивьен, Дж. Глинн, К. Уотсон.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2011, стр.834 – 853.

# Лабораторная работа 1. Размещение элементов управления.

**Краткая аннотация лабораторной работы:** в лабораторной работе будут даны задания для самостоятельного выполнения на закрепление пройденной теоретической части лекции 1, а так же рассмотрен пример, нацеленный показать пути решения поставленных в лабораторной работе задач.

**Цель лабораторной работы:** показать способы размещения элементов управления (Canvas, StackPanel, Grid). Рассмотрим каждый из этих элементов, чтобы получить о них более полное представление. Закрепить знания, полученные в лекции 1.

## Задания для самостоятельного выполнения

Разместить на экране Grid с двумя колонками и двумя строками. Нижньь строку объеденить и положить в неё StackPanel. В StackPanel разместить несколько элементов управления. Задать им расположение по вертикали, выравнивание по центру.

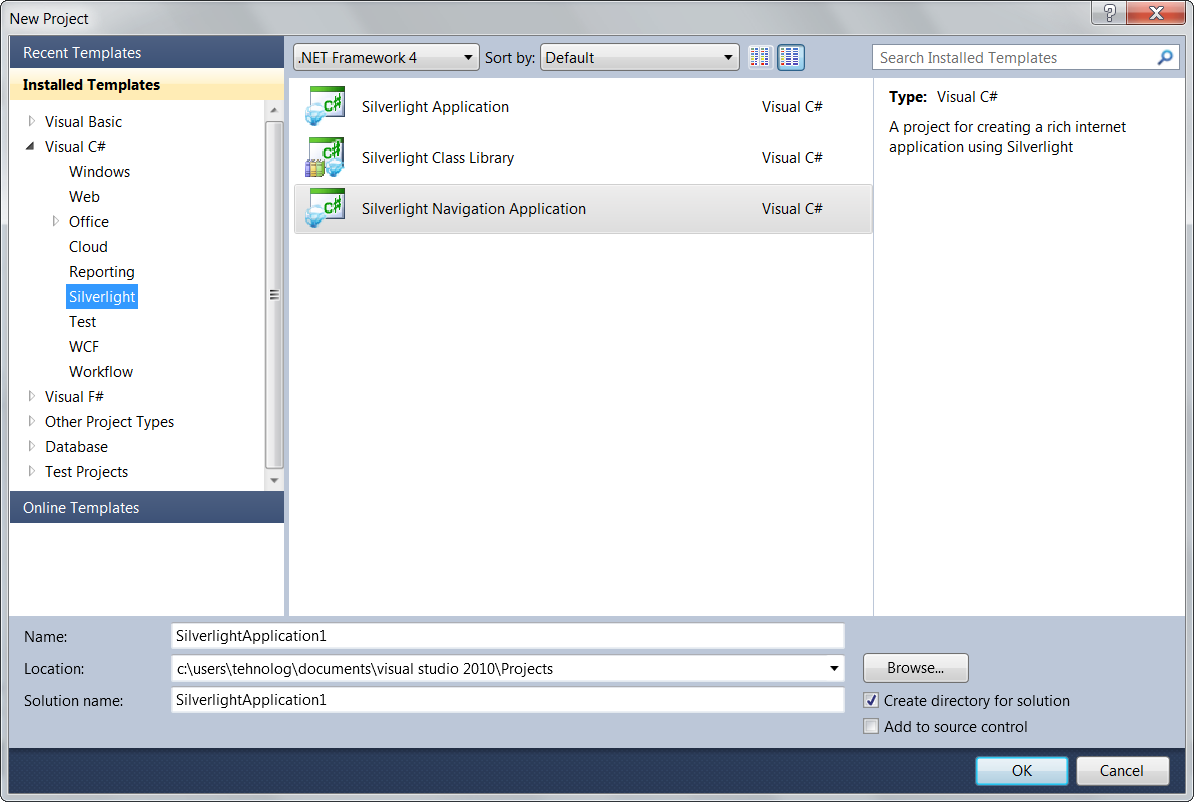
## Учебный элемент. Создание нового проекта для Silverlight в VisualStudio 2010.

Silverlight и WPF имеют общие корни и используют одинаковый подход к созданию приложений при помощи XAML. В данном учебном элементе максимально просто и понятно будут рассмотрены основные особенности при программировании на Silverlight. По традиции в качестве примера будет использован стандартный проект «Здравствуй, мир!».

Проект будет разрабатываться на Visual Studio 2010, но в данном случае это не принципиально. Поэтому, если вы еще не перешли на новую версию среды разработки, то можете использовать и VS2008.

### Шаг 1. Создание проекта

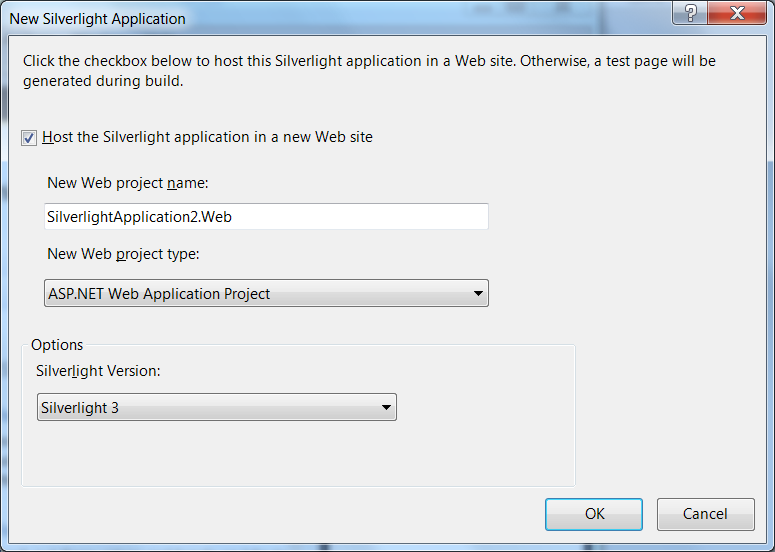
Запустим Visual Studio и посмотрим, как создавать новые проекты для Silverlight (рисунок л. р. 1.1).

  
Рис. л. р. 1.1. Создание нового проекта Silverlight.

Если вы раньше выбирали подгруппу Windows для разработки стандартных приложений, то теперь вам нужно выбирать подгруппу Silverlight, в которой находятся несколько типов шаблонов: Silverlight Application, Silverlight Class Library и Silverlight Navigation Application.

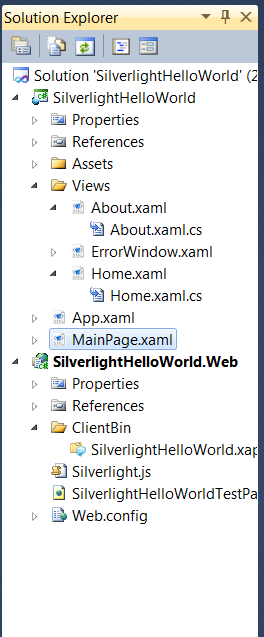
Для нашего первого примера возьмем шаблон *Silverlight Navigation Application* и присвоим своему проекту имя SilverlightHelloWorld.

В следующем окне (рисунок л. р. 1.2) Вам нужно выбрать тип проекта.

  
Рис. л. р. 1.2. Выброр типа проекта Silverlight

Используем настройки по умолчанию, и выбираем ASP.NET Web Application Project. В этом случае Вам будет проще отлаживать код и не придется использовать различные веб-сервисы.

Итак, Вы присвоили имя новому проекту и выбрали его тип. Откроется заготовка для проекта в среде разработки (рисунок л. р. 1.3). Для начала изучим структуру стандартного проекта, которую можно видеть в правой части экрана:

  
Рис. л. р. 1.3. Структура проекта Silverlight

Вы можете видеть множество различных файлов, которые были созданы без нашего участия. Рассмотрим их.

Итак, наибольший интерес для нас представляют следующие файлы:

* App.xaml – очень важный для приложения, в котором описаны ресурсы и различные глобальные события. Также в этом файле содержится точка входа в программу. А также содержатся инструкции для плагина Silverlight.
* MainPage.xaml – это страница, которая является часть выбранного вами шаблона. По умолчанию, страница имеет имя MainPage, но вы можете выбрать свое имя при желании. При помощи кода в этом файле мы можем настроить интерфейс будущей программы.
* Папки Assets/Views – в папках содержатся вспомогательные файлы: картинки, стили и т.п.

XAML-файлы позволяют визуализировать интерфейс программы. Файлы XAML основаны на XML и имеют свой язык разметки, который мы будем менять в этом учебном элементе.

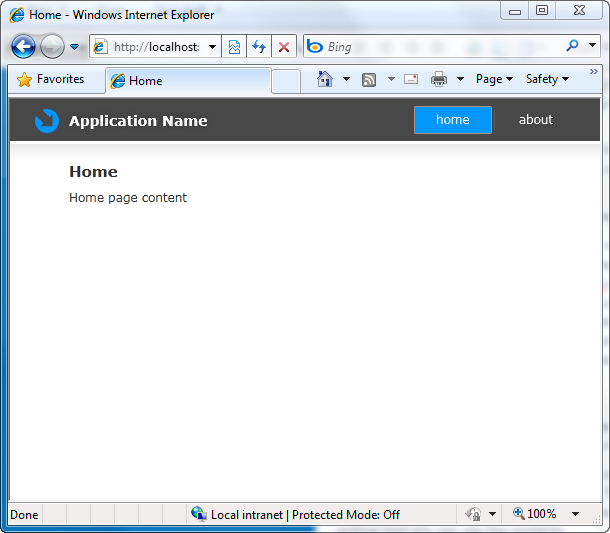
После компиляции проекта в папке ClientBin появится также новый файл с расширением XAP. На самом деле это обычный архивный ZIP-файл, имеющий другое расширение. Вы можете переименовать его в ZIP и открыть любым архиватором, чтобы просмотреть содержимое файла. XAP-файл предназначен для вашего браузера. Не будем углубляться пока в детали.

Для просмотра приложения используются две тестовые страницы для ASP.NET, и HTML. Содержание страниц одинаково, поэтому один из этих файлов Вы можете безболезнено удалить. Если удалите тестовую страницу для ASP.NET, то нужно сделать оставшуюся тестовую страницу SilverlightHelloWorldTestPage.html стартовой (через контекстное меню выберите команду Set As Start Page).

### Шаг 2. Добавляем элементы интерфейса для XAML-страниц

В нашем шаблоне, который мы выбрали для примера, имеются некоторые возможности для навигации (это видно по названию шаблона). В папке Views имеются файлы About.xaml, ErrorWindow.xaml и Home.xaml. Сосредоточимся на файле Home.xaml, который первым выводится на экран при запуске. Откройте файл двойным щелчком и Вы увидите код, который хранится в файле.

Рассматривая XAML-код, можно увидеть, что содержимое страницы содержится в контейнере Grid, например, такие стандартные элементы, как StackPanel и TextBlock. Элемент TextBlock предназначен для вывода текста и является аналогом элемента Label, знакомого Вам по WinForm. Запустите приложение (F5), не внося никаких изменений, чтобы увидеть, что из себя представляет программа. Вы увидите следующее окно:

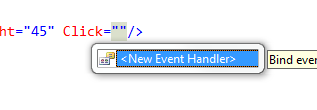
  
Рис. л. р. 1.4. Стартовая страницы созданного Silverlight приложения.

Обратите внимание на кнопки-ссылки в верхнем правом углу. Они отвечают за навигацию. Давайте теперь добавим свой код, чтобы понять, как работать с XAML.

Предположим, мы хотим добавить кнопку. В файле Home.xaml, который у Вас открыт, напишите после второго TextBlock следующее:

<Button Content="Click me"   
 x:Name="MyButton" FontSize="18" Width="150" Height="45" />

Вы уже заметили, что XAML-редактор состоит из двух частей. И когда вы пишете код, то в окне Design в интерактивном режиме сразу появляются описываемые элементы управления. После того, как Вы напечатали указанный код, то увидите кнопку с заданными параметрами после текста Home page content**.** Обратите внимание на атрибут x:Name. Это уникальный идентификатор для элемента и, именно, по этим идентификатором мы сможем обращаться к элементам из фонового кода. Любая уважающая себя кнопка позволяет пользователю щелкать по ней мышью. Вот как это делается в XAML. Достаточно при описании кнопки добавить атрибут Click, далее VS Intellisense услужливо спросит Вас, хотите ли сгенерировать обработчик для события:

  
Рис. л. р. 1.5. Intellisense.

Впрочем, Вы можете по старинке написать код для обработчика события вручную не в самом XAML, а в файле Home.xaml.cs:

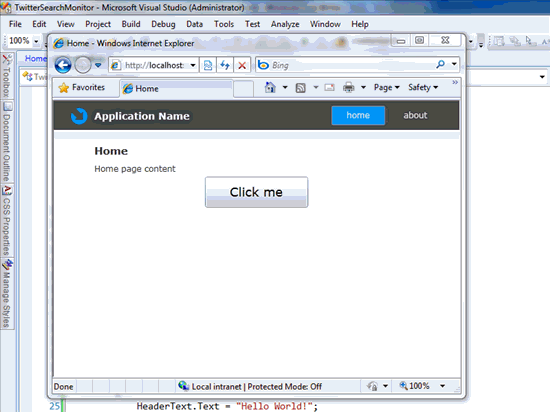
public Home()

{  
 InitializeComponent();  
 MyButton.Click += new RoutedEventHandler(MyButton\_Click);  
}

Когда мы определили функцию MyButton\_Click, то можем написать управляемый код в функции. Для первого примера давайте поменяем текст в элементе TextBlock под именем HeaderText (HeaderText является значением атрибута x:Name и мы можем обратиться к нему через это имя). Щелкнем дважды на созданной кнопке, чтобы сразу оказаться в нужном месте кода в редакторе и напишем следующее:

void MyButton\_Click(object sender, RoutedEventArgs e)  
{  
 HeaderText.Text = "Hello World!";  
}

Если теперь запустить приложение снова, то увидим нашу кнопку. Щелкнув по ней, мы изменим текст:

  
Рис. л. р. 1.5. Результат работы функции MyButton\_Click

## Учебный элемент. Silverlight: размещение элементов.

В первом учебном элементе мы рассмотрели базовые приемы программирования. Теперь разберем вопрос размещения элементов в Silverlight-приложениях. Размещение элементов на форме в Windows Forms было простым занятием. Мы просто перетаскивали нужный элемент с панели инструментов на форму и задавали ему нужные свойства. В приложениях, основанных на XAML, используется другой подход. Важно разобраться с основными принципами размещения элементов в XAML, так как это является основой для дальнейшего успешного программирования на Silverlight (и WPF).

### Шаг 1. Принципы разметки

Silverlight использует гибкую систему для размещения видимых элементов управления на поверхности. Разметка в коде поддерживает два стиля размещения: динамический и абсолютный. Основными элементами, которые помогают размещать элементы являются:

* Canvas
* StackPanel
* Grid

Рассмотрим каждый из этих элементов, чтобы получить о них более полное представление. Для демонстрации воспользуемся привычными нам кнопками, которые будет размещать на поверхности нашего приложения. Вы можете заново создать новый проект или продолжить работу над предыдущим примером. Будем вносить изменения в коде файла Home.xaml.

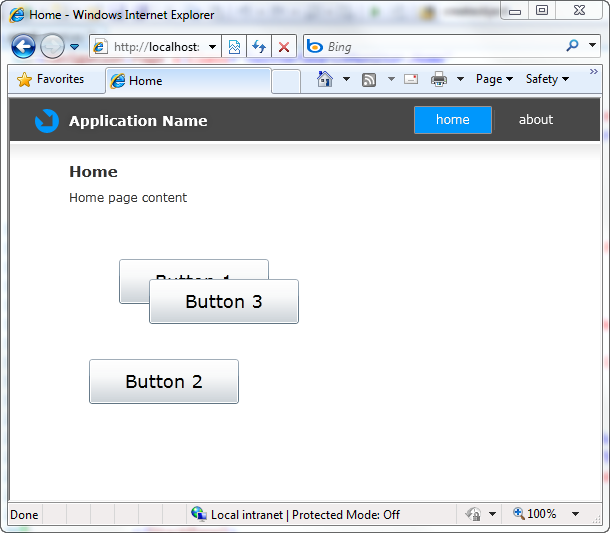
#### Canvas

Элемент Canvas является одним из базовых элементов разметки и позволяет размещать элементы привычным нам способом при помощи задания нужных координат (именно такой способ мы использовали в Windows Forms. Например, попробуем разместить три кнопки следующим образом:

<Canvas>  
<Button Canvas.Top="50" Canvas.Left="50" Content="Button 1" FontSize="18"   
 Width="150" Height="45" />  
  
<Button Canvas.Top="150" Canvas.Left="20" Content="Button 2" FontSize="18"   
 Width="150" Height="45" />  
  
<Button Canvas.Top="70" Canvas.Left="80" Canvas.ZIndex="99"  
 Content="Button 3" FontSize="18" Width="150" Height="45" />

</Canvas>

На экране мы увидим следующую картинку:

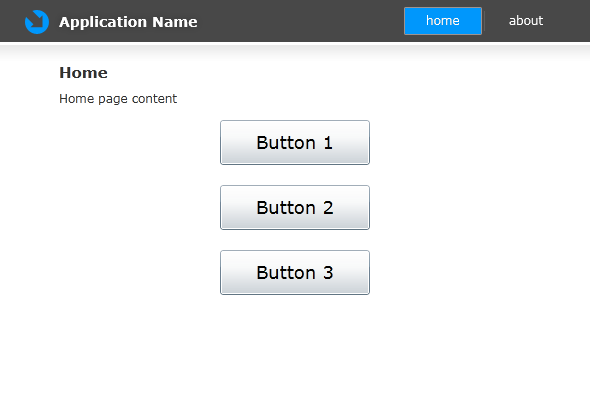
  
Рис. л. р. 1.6. Результат размещения контролов на элементе разметки Convas.

Как Вы видите, мы просто задаем координаты для кнопок. В случае необходимости мы также можем указать порядок ZIndex, если одна кнопка перекрывает другую. Подобная модель размещения элементов вполне годится при создании игр или демонстрации каких-то физических процессов. Также Canvas подойдет в тех случаях, когда элементы статичны и хорошо смотрятся в различных ситуациях. Но, на самом деле, существуют более удобные контейнеры, которые вытесняют Canvas, и избавляют программиста от множества проблем, возникаемых при разработке удобных приложений. К числу таких элементов относятся StackPanel или Grid.

#### StackPanel

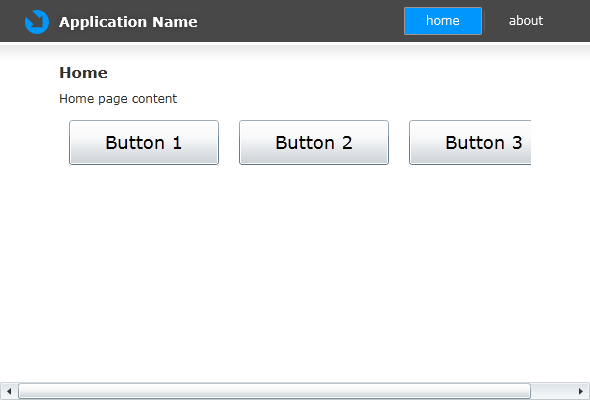
StackPanel - элемент-контейнер, который размещает элементы в ряд по вертикали или горизонтали (по умолчанию используется вертикальное размещение). Вот как можно расположить три кнопки по вертикали:

<StackPanel>  
 <Button Margin="10" Content="Button 1" FontSize="18"  
 Width="150" Height="45" />  
 <Button Margin="10" Content="Button 2" FontSize="18"   
 Width="150" Height="45" />  
 <Button Margin="10" Content="Button 3" FontSize="18"  
 Width="150" Height="45" />  
</StackPanel>

  
Рис. л. р. 1.6. Результат размещения контролов на элементе разметки StackPanel.

Веб-разработчик может увидеть здесь некоторое сходство с блоковыми элементами HTML (например, DIV), которые размещаются с новой строки. Если Вы хотите разместить кнопки по вертикали, то достаточно присвоить атрибуту Orientation элемента StackPanel значение Horizontal:

<StackPanel Orientation="Horizontal">  
 <Button Margin="10" Content="Button 1" FontSize="18"  
 Width="150" Height="45" />  
 <Button Margin="10" Content="Button 2" FontSize="18"  
 Width="150" Height="45" />  
 <Button Margin="10" Content="Button 3" FontSize="18"  
 Width="150" Height="45" />  
</StackPanel>

  
Рис. л. р. 1.6. Orientation элемента StackPanel в значении Horizontal.

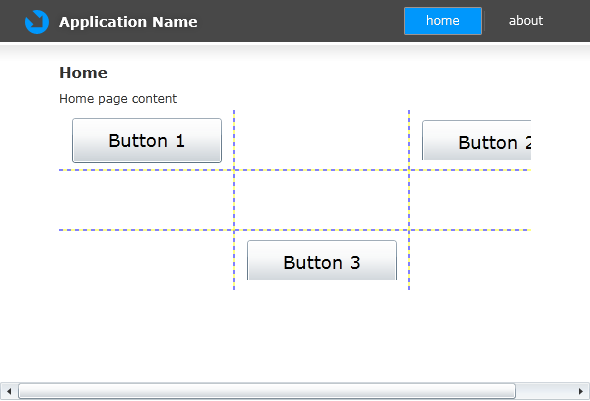
StackPanel позволяет легко управлять расположением элементов друг за другом не думая о вычислениях координат, так как все будет сделано автоматически.

#### Grid

Grid (сетка) используется очень часто. Возвращаясь к сравнению с элементами HTML, можно сказать, что данный контейнер похож на элемент TABLE (таблица), так как тоже использует столбцы и ряды. Небольшое отличие состоит в том, что в XAML мы сначала задаем структуру для Grid, а уже потом начинаем описывать элементы, которые будут находиться в сетке. Очень удобно, что у Grid есть атрибут ShowGridLines, который показывает линии сетки, что позволит нам наглядно увидеть пример с расположением кнопок (в реальных примерах вы можете не выводить сетку).

<Grid ShowGridLines="True">  
<Grid.RowDefinitions>  
 <RowDefinition Height="60" />  
 <RowDefinition Height="60" />  
 <RowDefinition Height="60" />  
</Grid.RowDefinitions>  
  
<Grid.ColumnDefinitions>  
 <ColumnDefinition Width="175" />  
 <ColumnDefinition Width="175" />  
 <ColumnDefinition Width="175" />  
</Grid.ColumnDefinitions>  
  
<Button Grid.Column="0" Grid.Row="0" Content="Button 1" FontSize="18"   
 Width="150" Height="45" />  
<Button Grid.Column="2" Grid.Row="0" Margin="10" Content="Button 2"   
 FontSize="18" Width="150" Height="45" />  
<Button Grid.Column="1" Grid.Row="2" Margin="10" Content="Button 3"   
 FontSize="18" Width="150" Height="45" />  
</Grid>

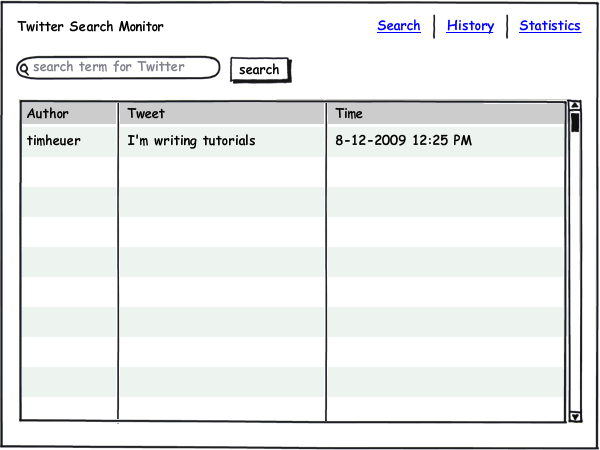
Если внимательно изучить код, то видим, что мы определили три колонки и три ряда и указали их размеры. Далее мы размещаем кнопки, указывая их положение при помощи Grid.Column и Grid.Row:

  
Рис. л. р. 1.7. Результат размещения контролов на элементе разметки Grid.

В Expression Blend размещать элементы удобнее, так как в нем реализован удобный редактор размещения. Поэтому Вам стоит изучить данную программу и использовать ее в своих разработках.

### Шаг 2. Создаем Twitter-приложение

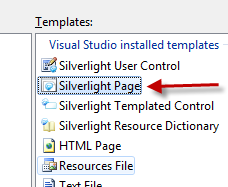
Теперь мы вполне можем приступить к разработке настоящего приложения.

  
Рис. л. р. 1.8. Вот как выглядит наше будущее приложение в эскизе.

Основными элементами приложения являются строка для поиска, кнопки и поля для вывода результатов поиска. Также у нас будут задействованы элементы навигации в виде ссылок, с помощью которых мы можем просматривать историю запросов и статистику.

Мы поступили дальновидно, выбрав шаблон с навигацией. Внесем некоторые изменения в файле MainPage.xaml. Около строки 29 удалим логотип приложения (<ContentControl Style="{StaticResource LogoIcon}"/>), а также изменим текст для ApplicationnameTextBlock на **Twitter Search Monitor**.

Оставим на минутку наш проект и займемся его структурой. В окне Solution Explorer найдите папку Views, и при помощи контекстного меню создайте новую страницу под именем Search.xaml. Для этого щелкните правой кнопкой на папке Views, далее выберите команды Add | New Item... и в следующем диалоговом окне выберите шаблон Silverlight Page:

  
Рис. л. р. 1.9. Выбор шаблона Silverlight Page

У нас появится пустая XAML-страница с сеточной разметкой Grid (по умолчанию). На этой странице мы создадим окно поиска, эскиз которого был показан выше.

### Шаг 3. Навигация

Изучим основны особенности навигации, используемой в нашем шаблоне. Мы видим, что шаблон состоит из файла MainPage.xaml и дополнительных элементов Home, About. Навигация состоит из трех важных состаляющих: UriMapper, Frame и Page.

#### UriMapper

В двух словах сложно объяснить, что такое UriMapper. Считайте, что это удобный способ сокращения ссылок. Например, вместо использования строки /Views/Home.xaml, вы можете настроить приложение таким образом, что ссылка будет выглядеть как /Home. Разработчики PHP могут найти тут сходство с настройкой файла .htaccess. UriMapper является частью элемента Frame в MainPage.xaml:

<navigation:Frame x:Name="ContentFrame"  
 Style="{StaticResource ContentFrameStyle}"   
 Source="/Home" Navigated="ContentFrame\_Navigated"   
 NavigationFailed="ContentFrame\_NavigationFailed">  
 <navigation:Frame.UriMapper>  
 <uriMapper:UriMapper>  
 <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml"/>  
 <uriMapper:UriMapping Uri="/{pageName}"   
 MappedUri="/Views/{pageName}.xaml"/>  
 </uriMapper:UriMapper>  
 </navigation:Frame.UriMapper>  
</navigation:Frame>

#### Frame

Frame - это область для навигации. Вы сначала настраиваете эту область по умолчанию, а затем она меняется при навигации. Если посмотреть на код, приведенный выше, то увидите, что значением атрибута Source для элемента Frame является /Home.

#### Page

И, наконец, последний важный элемент навигации - это элемент Page, который мы создали сами. Данный элемент является контейнером для содержимого, который выводится в Frame. Можете рассматривать его как элемент UserControl, который вы можете добавлять в проект.

### Шаг 4. Размещаем элементы для страницы поиска

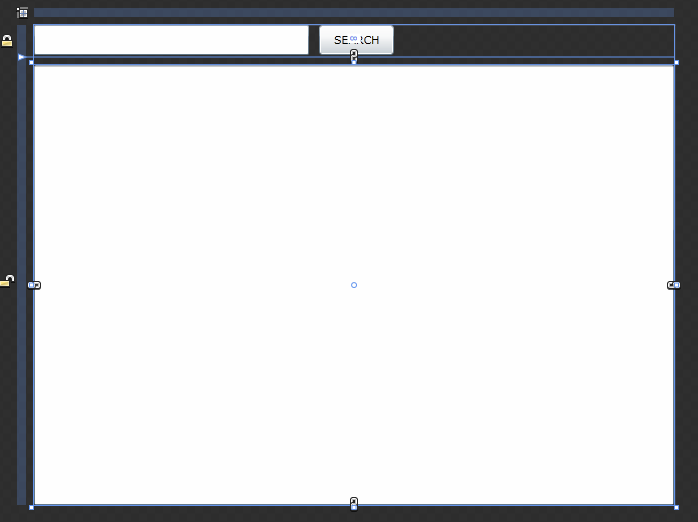
Вернемся к проекту и закончим с размещением элементов на созданной нами странице Search.xaml. В соответствии с нашим эскизом, нам нужно разместить текстовое поле, кнопку и сетку для отображения данных.

Поместим на страницу Grid с двумя рядами: первый ряд для поисковой строки и кнопки, а второй ряд для вывода результатов. В верхнем ряду расположим элемент StackPanel и добавим на него текстовое поле и кнопку. Чтобы элементы располагались друг за другом, присвоим у StackPanel значение Orientation=Horizontal.

Далее добавим элемент DataGrid, предназначенный для работы с данными, во второй ряд сетки. Код для страницы будет выглядеть следующем образом:

<navigation:Page x:Class="SilverlightHelloWorld.Views.Search"   
 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"  
 d:DesignWidth="640" d:DesignHeight="480"  
 Title="Twitter Search Page"   
 xmlns:data="clr-namespace:System.Windows.Controls;  
 assembly=System.Windows.Controls.Data">  
 <Grid x:Name="LayoutRoot">  
 <Grid.RowDefinitions>  
 <RowDefinition Height="32"/>  
 <RowDefinition/>  
 </Grid.RowDefinitions>  
 <StackPanel HorizontalAlignment="Left" Margin="0,-32,0,0"   
 VerticalAlignment="Top" Grid.Row="1" Orientation="Horizontal">  
 <TextBox x:Name="SearchTerm" FontSize="14.667" Margin="0,0,10,0"  
 Width="275" TextWrapping="Wrap"/>  
 <Button x:Name="SearchButton" Width="75" Content="SEARCH"/>  
 </StackPanel>  
 <data:DataGrid x:Name="SearchResults" Margin="0,8,0,0" Grid.Row="1" />  
 </Grid>  
</navigation:Page>

Обратите внимание на запись xmlns:data в начале кода. Так можно добавлять свои элементы управления в XAML, после добавления ссылки на сборку. Сейчас приложение выглядит следующим образом (в Blend):

  
Рис. л. р. 1.10. Вид приложения в Blend.

Для идентификации мы дали имена x:Name для текстового поля (SearchTerm), кнопки (SearchButton) и DataGrid (SearchResults). Можно приступать к написанию кода.

### Шаг 5. Изменяем UriMapper для Search.xaml

Мы практически закончили работать над интерфейсом страницы поиска. Так как она является лицом нашего приложения, то неплохо бы сделать ее главной. Внесем некоторые изменения в файле MainPage.xaml. Найдите в нем Frame и поменяйте в нем строчку таким образом, чтобы по умолчанию открывалась наша поисковая страница, а не Home.xaml:

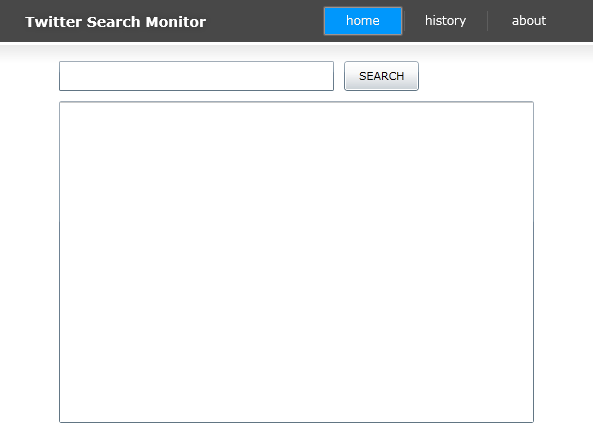
<navigation:Frame x:Name="ContentFrame"   
 Style="{StaticResource ContentFrameStyle}"   
 Source="/Search" Navigated="ContentFrame\_Navigated"   
 NavigationFailed="ContentFrame\_NavigationFailed">  
 <navigation:Frame.UriMapper>  
 <uriMapper:UriMapper>  
 <uriMapper:UriMapping Uri="" MappedUri="/Views/Search.xaml"/>  
 <uriMapper:UriMapping Uri="/{pageName}"   
 MappedUri="/Views/{pageName}.xaml"/>  
 </uriMapper:UriMapper>  
 </navigation:Frame.UriMapper>  
</navigation:Frame>

Нам больше не нужен файл Home.xaml, поэтому смело удаляем его из проекта. Далее добавим новый файл History.xaml и изменим в MainPage.xaml область LinksBorder, чтобы включить ссылку на него:

<Border x:Name="LinksBorder" Style="{StaticResource LinksBorderStyle}">  
 <StackPanel x:Name="LinksStackPanel" Style="{StaticResource   
 LinksStackPanelStyle}">  
  
 <HyperlinkButton x:Name="Link1" Style="{StaticResource LinkStyle}"   
 NavigateUri="/Search" TargetName="ContentFrame" Content="home"/>  
  
 <Rectangle x:Name="Divider1" Style="{StaticResource DividerStyle}"/>  
  
 <HyperlinkButton x:Name="Link2" Style="{StaticResource LinkStyle}"   
 NavigateUri="/History" TargetName="ContentFrame" Content="history"/>

<Rectangle x:Name="Divider2" Style="{StaticResource DividerStyle}"/>  
  
 <HyperlinkButton x:Name="Link3" Style="{StaticResource LinkStyle}"   
 NavigateUri="/About" TargetName="ContentFrame" Content="about"/>  
  
 </StackPanel>  
</Border>

Сейчас программа выглядит следующим образом:

  
 Рис. л. р. 1.11. Вид приложения в браузере.

## Краткие итоги

Мы создали первое приложение на Silverlight. Разобрали способы размещения элементов управления на таких панелях как Canvas, StackPanel и Grid. Познакомились с составляющими навигациеи: UriMapper, Frame и Page.

## Литература

1. XAML в WPF

<http://msdn.microsoft.com/ru-ru/library/ms747122.aspx>

# Лекция 2. Сведения о WPF и Silverlight: Введение и архитектура платформ.

**Краткая аннотация лекции:** в лекции вы ознакомитесь с архитектурой WPF и Silverlight, узнаете, как платформы справляются с различными разрешениями экрана, и получите общее представление о сборках и классах, а так же основных элементах управления.

**Цель лекции:** получить знания об архитектуре WPF и Silverlight, освоить назначение и декларативное описание элементов контроля.

## Windows Presentation Foundation

Windows Presentation Foundation (WPF) — это графическая система отображения для Windows. Платформа WPF спроектирована для .NET под влиянием таких современных технологий отображения, как HTML и Flash, и использует аппаратное ускорение. Она также является наиболее радикальным изменением в пользовательском интерфейсе Windows со времен Windows 95. Даже если бы единственным достоинством WPF было аппаратное ускорение через DirectX, это уже стало бы значительным усовершенствованием, хоть и не революционным. Однако WPF на самом деле включает целый набор высокоуровневых служб, ориентированных на прикладных программистов. Существует одна область, для которой WPF не слишком хорошо подходит — создание приложений с требованиями к графике реального времени, таких как эмуляторы сложных физических процессов или современные интерактивные игры. Поскольку для такого рода приложений нужна максимально возможная видеопроизводительность, необходимо программировать на более низком уровне и использовать DirectX напрямую.

## Независимость от разрешения

Традиционные Windows-приложения связаны определенными предположениями относительно разрешения экрана. Обычно разработчики рассчитывают на стандартное разрешение монитора (вроде 1024x768 пикселей) и проектируют свои окна с учетом этого, стараясь обеспечить разумное поведение при изменении размеров в большую и меньшую сторону.

Проблема в том, что пользовательский интерфейс в традиционных Windows- приложениях не является масштабируемым. В результате, если вы используете монитор с высоким разрешением, который располагает пиксели более плотно, окно приложения становится меньше и читать текст в нем труднее. Эта проблема особенно актуальна для новых мониторов, которые имеют высокую плотность пикселей и соответственно работают с более высоким разрешением. Например, легче встретить мониторы (особенно на портативных компьютерах), которые имеют плотность пикселей в 120 dpi или 144 dpi (точек на дюйм), чем более традиционные 96 dpi. При их встроенном разрешении эти мониторы располагают пиксели более плотно, создавая напрягающие глаз мелкие элементы управления и текст.

В идеале приложения должны использовать более высокую плотность пикселей, чтобы отобразить больше деталей. Например, монитор с высоким разрешением может отображать одинакового размера значки панели инструментов, но использовать дополнительные пиксели для отображения мелкой графики. Подобным образом можно сохранить некоторую базовую компоновку, но обеспечить более высокую четкость деталей.

По разным причинам такое решение было невозможно в прошлом. Хотя можно изменять размер графического содержимого, нарисованного в GDI/GDI+, компонент User32 (который генерирует визуальное представление распространенных элементов управления) не поддерживает реального масштабирования.

WPF не страдает от этой проблемы, потому что самостоятельно визуализирует все элементы пользовательского интерфейса — от простых фигур до таких распространенных элементов управления, как кнопки. В результате если вы создаете кнопку шириной в 1 дюйм на обычном мониторе, она останется шириной в 1 дюйм и на мониторе с высоким разрешением. WPF просто визуализирует ее более детализировано, с большим количеством пикселей.

Так выглядит картина в целом, но нужно уточнить еще несколько деталей. Самое важное, что следует осознать — WPF базирует свое масштабирование на системной установке DPI, а не на DPI физического дисплейного устройства. Это совершенно логично — в конце концов, если вы отображаете приложение на 100-дюймовом проекторе, то, скорее всего, отойдете подальше на несколько шагов и будете ожидать увидеть огромную версию окон. Конечно, не желательно, чтобы WPF масштабировал приложение, уменьшая его до "нормального" размера. Аналогично, если вы используете портативный компьютер с дисплеем высокого разрешения, то хотите увидеть несколько уменьшенные окна это цена, которую приходится платить за то, чтобы уместить всю информацию на маленьком экране. Более того, у разных пользователей разные предпочтения на этот счет. Некоторым нужны расширенные подробности, в то время как другие хотят увидеть больше содержимого. Так каким же образом WPF определяет, насколько большим должно быть окно приложения? Краткий ответ состоит в том, что при вычислении размеров WPF использует **системную установку DPI**.

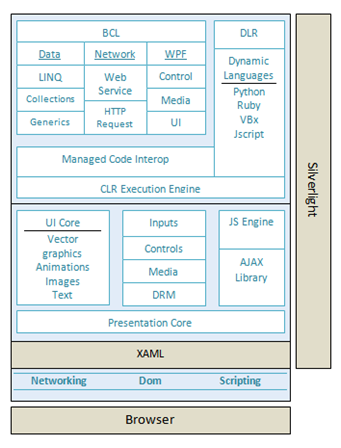
## Архитектура Silverlight

Технология Microsoft Silverlight — это библиотека классов для создания интернет-приложений с богатым интерфейсом (Rich Internet Applications, RIA), поддерживающих использование мультимедиа, графики и анимации. Silverlight поставляется в виде расширения для веб­браузера (размером всего 4 Мбайт), содержащего среду исполнения кода. Поддерживаются платформы Microsoft Windows, Mac OS X, Linux и FreeBSD (для последних требуется разработанная совместно с компанией Novell версия Silverlight под названием Moonlight). Поддерживаются браузеры Microsoft Internet Explorer 5.5+, Safari, Opera и FireFox. В версии Silverlight 4 также планируется поддержка браузера Chrome.

Разработка Silverlight-приложений возможна в Visual Studio начиная с версии 9.0 с пакетом обновлений Service Pack 1, Microsoft Expression Design и Microsoft Expression Blend 3 + SketchFlow. Также потребуется загрузка и установка Microsoft Silverlight Software Development Kit.

Silverlight поддерживает графическую модель, схожую с Windows Presentation Foundation, использует язык XAML для описания интерфейсов приложений и поддерживает подмножество .NET Framework. Помимо создания интернет-приложений с богатым интерфейсом, Silverlight поддерживает отображение мультимедийных файлов в форматах WMV, WMA и MP3 через Windows Media Player, соответствующий компонент ActiveX или расширение для веббраузера, помимо этого обеспечивается поддержка видео в формате VC-1, H.264 и MP4, а также аудио в формате AAC. В версию Silverlight 3 включено множество новинок, среди которых поддержка работы вне браузера, навигация по приложениям, множество новых элементов управления, 3D-трансформации и проекции, поддержка аппаратных ускорителей, шейдерные эффекты и адаптивное вещание с помощью технологии Smooth Streaming.

На рис. 2.1 показана архитектура Silverlight.

  
Рис. 2.1. Архитектура платформы Silverlight 4

Как платформа, Silverlight состоит из трех основных компонентов: ядра представления (Presentation Core), подмножества .NET Framework for Silverlight (clr execution core) и компонентов для установки и обновления.

Ядро представления содержит компоненты и сервисы, поддерживающие пользовательские интерфейсы, взаимодействие с пользователями, включая ввод, набор компонентов для вебприложений, средства воспроизведения мультимедиа, управления цифровыми правами и связи с данными, а также поддержку графики, включая векторную графику, текст, анимацию и графические изображения. Также сюда входит поддержка языка XAML для описания расположения элементов.

### Состав ядра представления:

* компоненты ввода — обрабатывают ввод с таких устройств, как клавиатура и мышь, диджитайзеры и другие устройства ввода;
* средства отрисовки — обеспечивают отображение векторной и растровой графики, анимации и текста;
* мультимедийные компоненты — отвечают за воспроизведение различных аудио и видеофайлов, таких как WMV и MP3;
* поддержка Deep Zoom — позволяет увеличивать размеры изображений в высоком разрешении;
* набор интерфейсных компонентов — поддерживает настройки с применением стилей и шаблонов;
* средства расположения — отвечают за динамическое позиционирование интерфейсных элементов;
* средства связи с данными — поддерживают связь объектов данных с интерфейсными элементами;
* механизм DRM — обеспечивает управление цифровыми правами для мультимедийных файлов;
* поддержка XAML — заключается в обработке разметки на этом языке.

Подмножество .NET Framework представляет собой набор классов для интеграции данных, поддержки сетевых коммуникаций, сборки мусора, расширяемый набор Windows-компонентов, а также общие классы для CLR. Некоторые части .NET Framework развертываются вместе с приложением. Эти библиотеки не входят в состав Silverlight Runtime и поставляются в составе Silverlight SDK. Сюда относятся библиотеки поддержки новых интерфейсных элементов, XLINQ, Syndication (RSS/Atom), XML-сериализации и Dynamic Language Runtime (DLR).

### В подмножество .NET Framework входят:

* средства работы с данными — обеспечивают поддержку технологий Language-Integrated Query (LINQ) и LINQ to XML, облегчающих интеграцию с различными источниками данных. Помимо этого поддерживается использование данных в формате XML и работа с сериализационными классами;
* базовая библиотека классов (BCL) — обес-печивает набор стандартных функций, таких как обработка строк, работа с регулярными выражениями, функции вводавывода, механизм reflection, работа с коллекциями и средства многоязыковой поддержки (глобализация);
* Windows Communication Foundation (WCF) — обеспечивает доступ к удаленным сервисам и данным. Сюда относится поддержка объектной модели браузера, обработка HTTP-запросов и ответов, поддержка кроссдоменных HTTP-запросов, поддержка RSS/Atom, JSON, POX и сервисов на основе протокола SOAP;
* CLR (Common Language Runtime) — предоставляет средства для управления памятью, механизм сборки мусора (garbage collection), проверку типов и обработку исключений;
* компоненты WPF (Windows Presentation Foundation) — обеспечивают богатый выбор элементов, среди которых такие элементы, как Button, Calendar, CheckBox, DataGrid, DatePicker, HyperlinkButton, ListBox, RadioButton и ScrollViewer;
* DLR (Dynamic Language Runtime) — поддерживает динамическую компиляцию и выполнение сценарных языков, таких как JavaScript и IronPython для управления Silverlight-приложениями. Также обеспечивается расширяемая модель, позволяющая добавлять новые динамические языки для использования их в Silverlight.

Компоненты для установки и обновления упрощают процесс установки приложений и обеспечивают их автоматическое обновление.

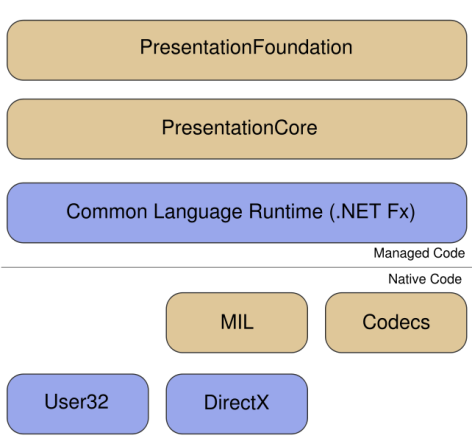
Помимо рассмотренных выше возможностей Silverlight следует отметить следующие функции:

* изолированное хранилище — безопасный доступ из клиентского приложения к файловой системе на локальном компьютере, позволяющий использовать локальное хранилище;
* асинхронное программирование — поддержку фоновых потоков, обрабатывающих логику приложений;
* управление файлами — возможность применения диалоговой панели File —> Open для упрощения создания безопасных загрузок файлов;
* интеграция с HTML-кодом — возможность управления интерфейсными элементами в составе вебстраницы на уровне HTML DOM для доступа к объектам, свойствам, событиям и методам;
* сериализация — поддержку сохранения CLR-типов в JSON и XML;
* упаковка — класс Application и средства для создания \*.xap-пакетов, содержащих само приложение и точку входа для его запуска из плагина Silverlight;
* XML-библиотеки — классы XmlReader и XmlWriter для работы с XML-данными из вебсервисов. Поддержка XLinq позволяет разработчикам обращаться к XML-данным непосредственно из кода.

## Архитектура WPF

Технология WPF использует многоуровневую архитектуру. На вершине ваше приложение взаимодействует с высокоуровневым набором служб, которые полностью написаны на управляемом коде С#. Действительная работа по трансляции объектов .NET в текстуры и треугольники Direct3D происходит "за кулисами", с использованием низкоуровневого неуправляемого компонента по имени milcore.dll. Библиотека milcore.dll реализована в неуправляемом коде потому, что ей требуется тесная интеграция с Direct3D, и вдобавок для нее чрезвычайно важна производительность.

На рис. 2.2 показаны уровни, на которых построена работа приложения WPF.

  
Рисунок 2.2. Архитектура WPF

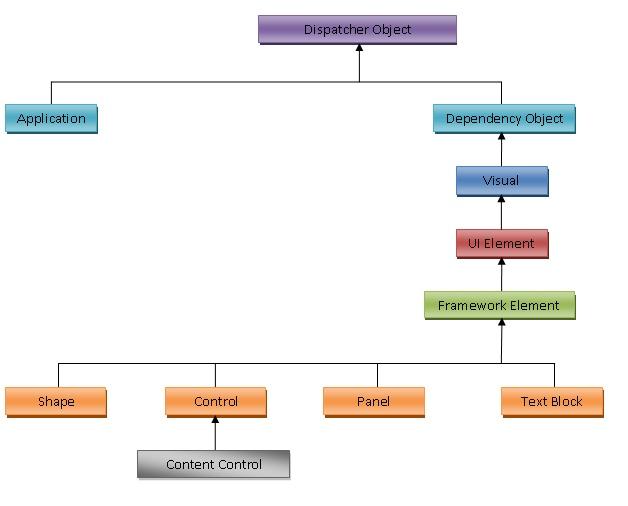
Ниже описаны ключевые компоненты, присутствующие на рис. 2.2.

* **PresentationFramework** (PresentationFramework.dll) содержит типы WPF верхнего уровня, включая те, что представляют окна, панели и прочие виды элементов управления. Также он реализует высокоуровневые программные абстракции, такие как стили. Большинство классов, которые вы будете использовать, находятся непосредственно в этой сборке.
* **PresentationCore** (PresentationCore.dll) содержит базовые типы, такие как UIElement и Visual, от которых унаследованы все фигуры и элементы управления. Если вам не нужен полный уровень абстракции окон и элементов управления, можете опуститься ниже, на этот уровень, и продолжать пользоваться преимуществами механизма визуализации WPF.
* **Common Language Runtime** (.NET Fx) а именно WindowsBase.dll — содержит еще более базовые ингредиенты, которые потенциально могут применяться вне WPF, такие как Dispatcher Object и Dependency Object, поддерживающие механизм свойств зависимости
* **MIL** (milcore.dll) — ядро системы визуализации WPF и фундамент уровня медиа-интеграции (Media Integration Layer — MIL). Его составной механизм транслирует визуальные элементы в треугольники и текстуры, которых ожидает Direct3D. Хотя milcore.dll считается частью WPF, это также важнейший компонент операционных систем Windows Vista и Windows 7. В действительности DWM (Desktop Window Manager — диспетчер окон рабочего стола) использует milcore.dll для отображения рабочего стола.
* **Codecs** (WindowsCodecs.dll) — низкоуровневый API-интерфейс, обеспечивающий поддержку изображений (например, обработку, отображение и масштабирование растровых изображений и файлов JPEG).
* **DirectХ** (Direct3D) — низкоуровневый API-интерфейс, через который визуализируется вся графика в WPF.
* **User32** используется для определения того, какое место на экране к какой программе относится. В результате он по-прежнему вовлечен в WPF, но не участвует в визуализации распространенных элементов управления.

Наиболее важный факт, который потребуется осознать, состоит в том, что Direct3D визуализирует все рисование в WPF. При этом не важно, установлена на компьютере видеокарта со скромными возможностями или же более мощная, используются базовые элементы управления или рисуется более сложное содержимое, запускается приложение в Windows ХР, Windows Vista или Windows 7. Даже двумерные фигуры и обычный текст трансформируются в треугольники и проходят по трехмерному конвейеру. Какие-либо обращения к GDI или User32 отсутствуют.

### Иерархия классов

На рис. 2.3 показан базовый обзор некоторых ключевых ветвей иерархии классов.

Рисунок. 2.3. Ключевые ветви иерархии классов WPF.

#### Dispatcher Object

Приложения WPF используют знакомую однопоточную модель (single-thread affinity — STA), а это означает, что весь пользовательский интерфейс принадлежит единственному потоку. Будучи унаследованным от DispatcherObject, каждый элемент пользовательского интерфейса может удостовериться, выполняется ли код в правильном потоке, и обратиться к диспетчеру, чтобы направить код в поток пользовательского интерфейса.

#### Dependency Object

В WPF центральный путь взаимодействия с экранными элементами пролегает через свойства. За счет наследования от DependencyObject, классы WPF получают поддержку свойств зависимости.

#### Visual

Каждый элемент, появляющийся в WPF, в основе своей является Visual. Класс Visual можно воспринимать как единственный объект рисования, инкапсулирующий в себе инструкции рисования, дополнительные подробности рисования и базовую функциональность.

#### UI Element

Класс UIElement добавляет поддержку таких сущностей WPF, как компоновка (layout), ввод (input), фокус (focus) и события (events) — все, что команда разработчиков WPF называет аббревиатурой LIFE. Здесь же щелчки кнопками мыши и нажатия клавиш трансформируются в более удобные события, такие как MouseEnter.

#### Framework Element

Класс FrameworkElement — конечный пункт в центральном дереве наследования WPF. UIElement устанавливает фундамент для системы компоновки WPF, но FrameworkElement включает ключевые свойства (HorizontalAlignment и Margin), которые поддерживают его.

#### Shape

От этого класса наследуются базовые фигуры, такие как Rectangle, Polygon, Ellipse, Line и Path. Эти фигуры могут использоваться наряду с более традиционными графическими элементами Windows вроде кнопок и текстовых полей.

#### Control

Элемент управления (control) — это элемент, который может взаимодействовать с пользователем. К нему очевидным образом относятся такие классы, как Text Box, Button и ListBox.

#### Content Control

Это базовый класс для всех элементов управления, которые имеют отдельный фрагмент содержимого. Сюда относится все — от скромной метки Label до окна Window.

#### ItemsControl

Это базовый класс для всех элементов управления, которые отображают коллекцию каких-то единиц информации, вроде ListBox и TreeView.

#### Panel

Это базовый класс для всех контейнеров компоновки — элементов, которые содержат в себе один или более дочерних элементов и упорядочивают их в соответствии с определенными правилами компоновки.

### Элементы управления WPF

Система WPF содержит множество элементов управления, предназначенных для использования в пользовательском интерфейсе. Эти элементы управления примерно соответствуют стандартным элементам управления в системе Windows Forms. Если взглянуть на предыдущие версии системы WPF, то можно заметить много элементов управления (таких как Calendar, DatePicker, DataGrid и др.), которые были включены в стандартный набор для системы Windows Forms, но не вошли в стандартный набор для системы WPF.

Для того чтобы воспользоваться этими элементами управления, пользователь должен подключить свободно распространяемый пакет WPF Toolkit, размещенный на сайте CodePlex. Этот пакет был разработан компанией Microsoft, чтобы компенсировать указанный недостаток первой версии системы WPF за счет недостающих компонентов. Однако с появлением системы WPF 4.0 многие элементы управления из пакета WPF Toolkit уже вошли в стандартный набор. Разумеется, пользователь может использовать пакеты элементов управления, созданные другими разработчиками в тех случаях, когда стандартных средств недостаточно.

Несмотря на то, что набор элементов управления в системе WPF сопоставим с набором элементов управления в системе Windows Forms, по своим свойствам они сильно отличаются от своих аналогов. Например, у многих элементов управления больше нет свойства Text, а вместо него появилось свойство Content, которое используется для присвоения элементу управления определенного содержимого (а следовательно, и имени). В большинстве случаев это свойство можно интерпретировать как прямой аналог свойства Text в элементах управления системы Windows Forms и просто присваивать ему какую то строку, которая должна прорисовываться на экране. Однако на самом деле свойство Content может принимать не только текстовое значение, но и любой элемент системы WPF. Это открывает практически безграничные возможности для настройки существующих элементов управления и освобождает пользователя от необходимости создавать свои собственные элементы управления. При разработке сложных пользовательских интерфейсов это очень полезно. Кроме того, многие элементы управления больше не имеют удобных свойств, которые существовали в системе Windows Forms. Это может показаться несколько странным. Например, у элемента управления Button в системе WPF нет свойства Image, позволявшего присваивать кнопке определенное изображение, как это было в системе Windows Forms. На первый взгляд, это сильно ограничивает возможности системы WPF, но это впечатление ошибочно, поскольку теперь у кнопки есть свойство Content. Так как свойство Content позволяет присваивать элементу управления системы WPF определенное содержимое, пользователь может присвоить ему элемент StackPanel (который будет рассмотрен далее), содержащий как элемент управления Image, так и элемент управления TextBlock, обеспечивающие тот же самый эффект. Это требует от пользователя чуть больше усилий, чем при работе с системой Windows Forms, но позволяетему проще моделировать содержимое кнопки на любой форме (а не подбирать элементы управления, которые он может реализовать) и обеспечивает невероятную гибкость системы WPF и языка XAML. Код на языке XAML для кнопки, показанной на рис. 2.4, имеет следующий вид.

<Button  
 HorizontalAlignment="Left"  
 VerticalAlignment="Center"  
 Width="100"  
 Height="30">  
 <Button.Content>  
 <StackPanel Orientation="Horizontal">  
 <Image  
 Source="save.png"  
 Width="16"  
 Height="16" />  
 <TextBlock  
 Margin="5,0,0,0"  
 Text="Отмена"  
 VerticalAlignment="Center" />  
 </StackPanel>  
 </Button.Content>  
</Button>



Рисунок. 2.4

Другими замечательными свойствами, которые отличаются от свойств системы Windows Forms, являются свойства IsEnabled (которое в системе Windows Forms называлось Enabled) и Visibility (которое в системе Windows Forms называлось Visible). На примере свойства IsEnabled видно, что имена большинства булевых свойств имеют префикс Is (например, IsTabStop, IsHitTestVisible и т.д.), что соответствует стандартной схеме именования. Однако свойство Visibility больше не имеет булевого значения — теперь оно представляет собой перечисление, принимающее значение Visible, Hidden или Collapsed.

### Компоновочные элементы управления в системе WPF

В системе Windows Forms для размещения элементов управления на рабочей поверхности использовались абсолютные координаты (т.е. каждый элемент управления имел явно заданные координаты x и y). Со временем появились элементы управления TableLayoutPanel и FlowLayoutPanel, которые могут содержать другие элементы управления. Это позволило создавать более сложные схемы размещения элементов управления на форме. Однако концепция позиционирования элементов управления в системе WPF немного отличается от системы Windows Forms. Наряду с элементами управления, имеющими специальное предназначение (например, кнопками, полями ввода и т.п.), в системе WPF есть множество элементов управления, используемых специально для определения компоновки пользовательского интерфейса.

Компоновочные элементы управления являются невидимыми и предназначены для позиционирования других элементов управления на своей поверхности. В системе WPF нет поверхности, по умолчанию предназначенной для позиционирования элементов управления, — поверхность, на которой работает пользователь, определяется компоновочными элементами управления, образующими иерархию. Компоновочные элементы управления, как правило, находятся в этой иерархии непосредственно под корневым узлом XAML файла и определяют метод компоновки, принятый по умолчанию для данного XAML файла. Наиболее важными компоновочными элементами управления в системе WPF являются Grid, Canvas и StackPanel, поэтому мы рассмотрим каждый из них, но сначала заглянем в код XAML, чтобы увидеть, как используются элементы управления.

Если пользователь хочет размещать элементы управления, используя абсолютную систему координат (как в системе Windows Forms), то он может в качестве поверхности выбрать элемент управления Canvas, предусмотренный в системе WPF. Определение элемента Canvas в языке XAML выглядит очень просто.

<Canvas>  
 …  
</Canvas>

Для того чтобы разместить элемент управления (например, TextBox) на этой поверхности, используя координаты x и y (по отношению к левому верхнему углу элемента Canvas), необходимо использовать концепцию присоединенных свойств (attachedproperties), принятую в языке XAML. Элемент управления TextBox на самом деле не имеет свойств, определяющих его местоположение, поскольку его позиция в компоновочном элементе управления полностью зависит от типа этого элемента. Следовательно, свойства, которых недостает элементу управления TextBox, чтобы определить его положение в компоновочном элементе, должны быть позаимствованы у самого компоновочного элемента, поскольку именно он определяет, где в нем должен находиться элемент TextBox. Именно в этот момент на первый план выходят **присоединенные свойства**. Коротко говоря, присоединенными называют свойства, которые присваивают элементу управления какое-то значение, но определяются и принадлежат другому элементу управления, стоящему выше в иерархии элементов. При использовании такого свойства его имя уточняется именем элемента управления, в котором оно на самом деле определено, за которым следует точка, а затем имя элемента управления, в котором оно используется (например, Canvas.Left). Присвоив это значение другому элементу управления, который содержится внутри (как, например, наше поле ввода TextBox), элемент Canvas хранит это значение в себе и с его помощью управляет позицией поля ввода. Например, следующий фрагмент кода на языке требует разместить поле ввода в точке 15, 10, используя свойства Left и Top, определенные в элементе управления Canvas.

<Canvas>  
 <TextBox Text="Hello" Canvas.Left="15" Canvas.Top="10" />  
</Canvas>

В то время как абсолютное позиционирование в системе Windows Forms принято по умолчанию, в системе WPF для компоновки лучше всего использовать элемент управления Grid. Элемент управления Canvas следует использовать редко и только при крайней необходимости, поскольку элемент управления Grid представляет собой намного более мощное средство для определения компоновки форм и лучше работает в большинстве сценариев. Одним из самых заметных преимуществ элемента управления Grid является возможность автоматически изменять размер его содержимого при изменении его собственного размера. Поэтому пользователь может легко проектировать формы, автоматически изменяющие свои размеры, чтобы заполнить всю доступную область, — иначе говоря, размер и положение элементов управления в этой форме определяются динамически. Элемент управления Grid позволяет разделять свою область на подобласти (ячейки), в которые можно поместить другие элементы управления. Эти ячейки определяются путем задания строк и столбцов сетки с помощью значений свойств RowDefinitions и ColumnDefinitions. Пересечения строк и столбцов становятся ячейками, в которые можно помещать элементы управления. Для определения строк и столбцов пользователь должен знать только, как определять сложные значения в языке XAML. До сих пор мы могли присваивать элементам управления лишь простые значения, которые отображались либо в элементарные типы данных платформы .NET, либо в имя элемента перечисления. Кроме того, мы могли конвертировать строку в соответствующий объект. Значения этих простых свойств использовались как атрибуты при определении элемента управления. Однако сложные значения нельзя присваивать таким способом, потому что они отображаются в объекты (т.е. необходимо выполнить присваивание нескольких свойство объекта) и должны определяться с помощью синтаксиса “свойство–элемент” (property element syntax). Поскольку свойства RowDefinitions и ColumnDefinitions элемента Grid представляют собой коллекции, они принимают сложные значения, которые должны быть определены с помощью синтаксиса “свойство–элемент”. Например, рассмотрим сетку, состоящую из двух строк и трех столбцов, определенных с помощью синтаксиса “свойство–элемент”.

<Grid>  
 <Grid.RowDefinitions>  
 <RowDefinition />  
 <RowDefinition />  
 </Grid.RowDefinitions>  
 <Grid.ColumnDefinitions>  
 <ColumnDefinition Width="100" />  
 <ColumnDefinition Width="150" />  
 <ColumnDefinition />  
 </Grid.ColumnDefinitions>  
</Grid>

Для того чтобы задать свойство RowDefinitions с помощью синтаксиса “свойство–элемент”, необходимо создать и определить дочерний элемент компонента Grid. Ставя имя Grid перед именем свойства, мы указываем, что оно принадлежит элементу управления, который относится к более высокому уровню иерархии (как и в случае присоединенных свойств), а задание свойства в виде элемента XAML файла означает, что мы присваиваем сложное значение указанному свойству элемента управления Grid. Свойство RowDefinitions получает коллекцию объектов RowDefinitions. Таким образом мы создаем несколько экземпляров объектов RowDefinition, образующих эту коллекцию. Соответственно свойству ColumnDefinitions присваивается коллекция объектов ColumnDefinition. Для того чтобы продемонстрировать, что ColumnDefinition (как и RowDefinition) — это действительно объект, свойство Width объекта ColumnDefinition было задано в двух первых строках определения столбца. Для того чтобы поместить элемент управления в заданную ячейку, необходимо снова использовать присоединенные свойства, на этот раз сообщив контейнеру, какие столбец и строку он должен разместить внутри себя.

<CheckBox  
 Grid.Column="0"  
 Grid.Row="1"  
 Content="A check box"   
 IsChecked="True" />

Еще одним важным контейнерным элементом управления, используемым для компоновки, является StackPanel. Он расставляет элементы управления, содержащиеся в нем, в горизонтальном или вертикальном направлении (в зависимости от значения своего свойства Orientation). Например, если в одной и той же ячейке сетки определены две кнопки (без элемента StackPanel), сетка может разместить вторую кнопку прямо поверх первой. Если же поместить эти кнопки в элемент управления StackPanel, то он выровняет их и разместит одну после другой.

<StackPanel Orientation="Horizontal">  
 <Button Content="OK" Height="23" Width="75" />  
 <Button Content="Cancel" Height="23" Width="75" Margin="10,0,0,0" />  
</StackPanel>

## Ключевые термины

**Attached properties:** является понятием Extensible Application Markup Language (XAML). Attached properties предназначено для использования в качестве типа глобального свойства, которое может быть задано для любого объекта. В Windows Presentation Foundation attached properties обычно определяются как особая форма свойства зависимости, не имеющая "оболочки" традиционного свойства.

**ContentPresenter:** отображает содержимое элемента управления [ContentControl](http://msdn.microsoft.com/ru-ru/library/system.windows.controls.contentcontrol.aspx).

[**ContentControl**](http://msdn.microsoft.com/ru-ru/library/system.windows.controls.contentcontrol.aspx)**:** Представляет элемент управления с отдельным содержимым любого типа.

**Direct3D** (D3D): интерфейс вывода трёхмерных примитивов.dpi

**Rich Internet application** (RIA, «Насыщенное [Интернет-приложение](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)») — это [приложение](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5), доступное через [Интернет](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82), насыщенное функциональностью традиционных [настольных приложений](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), которое предоставляется либо уникальной спецификой [браузера](http://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80), либо через [плагин](http://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B3%D0%B8%D0%BD), либо путём «[песочницы](http://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%81%D0%BE%D1%87%D0%BD%D0%B8%D1%86%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5))» (виртуальной машины).

**WPF Toolkit** - набор визуальных элементов и компонентов для WPF.

## Краткие итоги

Архитектура WPF включает уровни API интерфейса, медиа-интеграции и визуализации. Каждый уровень имеет пространства имен, в которых сформированы иерархии классов WPF. В основу всех элементов управления в WPF положены три основных принципа: композиция, повсеместное использование развитого содержимого и простая модель программирования. Мы рассмотрели фундаментальные концепции, которые способствуют претворению этих принципов в жизнь: модель содержимого и шаблоны. Не упуская их из виду, мы дали краткий обзор элементов управления, поставляемых вместе с WPF.

## Набор для практики

Вопросы:

1. Назовите и коротко охарактеризуйте основные компоненты архитектуры WPF?
2. Назовите способы компоновки элементов управления в WPF?
3. Поясните назначение и возможности класса Grid.
4. Поясните назначение и отличия классов StackPanel и WrapPanel.
5. Приведите пример необходимости использования класса ContentControl.

## Литература

1. WPF <http://msdn.microsoft.com/ru-ru/library/ms746927.aspx>
2. Элементы управления <http://msdn.microsoft.com/ru-ru/library/ms752069.aspx>.
3. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 25 – 45, 81-120.
4. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008 — стр. 38-60.

# Лекция 3. Стили и шаблоны элементов управления WPF.

**Краткая аннотация лекции**: очень часто при разработке графического интерфейса пользователя на WPF программист сталкивается с необходимостью создать элемент управления, который бы отличался по виду и/или набору возможностей от уже имеющихся в Microsoft .NET Framework. Видя разнообразные сложные элементы управления, хочется сразу приступить к работе, создать свой User Control и таким образом решить задачу. Правильный ли это подход мы будем разбираться в этой лекции.

**Цель лекции:** получить знания о стилях и шаблонах. Научить создавать и повторно использовать шаблон Control’а. Задавать шаблон через стиль.

В Windows Presentation Foundation существует очень четкое разделение между поведением Control'а и тем, как он выглядит. К примеру, поведение объекта класса Button состоит в том, чтобы реагировать на различные события по клику, но его вид может быть любым — вы можете сделать кнопку в виде стрелки, рыбы, или чего-либо еще, что подходит для вашего приложения. Переопределение отображения Control'а очень просто сделать стилями и шаблонами. В WPF стилизация и использование шаблонов относятся к набору функций (стилей, шаблонов, триггеров и раскадровок), позволяющих разработчикам и дизайнерам создавать визуально привлекательные эффекты, а также создавать целостный внешний вид своих продуктов. Несмотря на то, что разработчики и дизайнеры могут настроить внешний вид в масштабе приложений, для обслуживания и синхронное использования внешнего вида внутри приложений и между приложениями необходима строгая модель стилей и шаблонов.  WPF предоставляет такую модель.

Другая функция модели стилизации WPF состоит в разделении представления и логики. Это означает, что дизайнеры могут создавать внешний вид приложения, используя только XAML, в то же самое время, когда разработчики работают над логикой программы, используя C# или Visual Basic.

## Стили

***Стиль – это совокупность значений свойств, которые можно все сразу применить к нужному элементу***. В Silverlight стили позволяют разгрузить вашу XAML разметку путем вынесения деталей форматирования элемента в отдельный блок. Система Silverlight стилей играет ту же роль, что и стандарт каскадных таблиц стилей (CSS) в HTML-верстке. Как и CSS, стили в Silverlight позволяют вам определять базовый набор характеристик форматирования и использовать их в вашем приложении для обеспечения согласованности. Но есть и несколько важных ограничений. Например, вы не сможете применить один стиль к элементам разных типов или назначить автоматическое использование стиля. По этой причине стили кажутся несколько неудобными, хотя и являются одной из ключевых возможностей.

### Создание стиля

Представьте, что вам нужно стандартизировать шрифт и его цвет для всех кнопок страницы. Первым делом опишите объект Style содержащий в себе все необходимые вам свойства. Поместите этот объект как ресурс (например, в секции UserControl.Resources, в которой можно хранить ресурсы, относящиеся к данной странице):

<UserControl.Resources>  
 <Style x:Key="BigButtonStyle" TargetType="Button">  
 ...  
 </Style>  
</UserControl.Resources>

У стиля, как и у всех ресурсов, имеется имя-ключ, по которому вы можете к нему обращаться. В нашем примере имя ключа – BigButtonStyle. (Обычно принято добавлять к именам стилей окончание «Style»). Кроме того, для любого Silverlight стиля должно быть задано поле TargetType определяющее тип элементов, к которым может применяться данный стиль. В нашем случае стиль создается для форматирования кнопок (Button).

Наш объект Style содержит коллекцию сеттеров (setter) состоящую из 6-ти Setter-объектов (по одному на каждое свойство). Каждый сеттер описывает только одно свойство элемента. Единственным ограничением является то, что сеттер способен изменять только зависимые свойства (dependency properties), и никакие другие. Но, как показывает практика, это не такое уж и большое ограничение, поскольку почти все свойства Silverlight элементов – зависимые свойства. Сеттеры свойств могут влиять на любые зависимые свойства, даже на те, что управляют поведением объекта, а не его внешним видом. К примеру, если вы применяете стиль к текстовому полю, можете выставлять параметры AcceptsReturn и IsReadOnly непосредственно в самом стиле.

Перед вами стиль большой кнопки с белым текстом шрифта Georgia на темном фоне:

<UserControl.Resources>  
 <Style x:Key="BigButtonStyle" TargetType="Button">  
 <Setter Property="FontFamily" Value="Georgia" />  
 <Setter Property="FontSize" Value="40" />  
 <Setter Property="Foreground" Value="SlateGray" />  
 <Setter Property="Background" Value="Black" />  
 <Setter Property="Padding" Value="20" />  
 <Setter Property="Margin" Value="10" />  
 </Style>  
</UserControl.Resources>

В некоторых случаях вы не сможете задать значение свойства, используя простой формат записи атрибута. Например, простая запись не позволяет создавать неоднородную кисть LinearGradientBursh или ImageBrush. В такой ситуации вы можете применить уже знакомый для XAML прием замещения атрибута каким-либо вложенным элементом. Вот пример:

<Style x:Key="BigButtonStyle" TargetType="Button">  
 <Setter Property="Background">  
 <Setter.Value>  
 <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">  
 <GradientStop Color="Blue"></GradientStop>  
 <GradientStop Color="Yellow" Offset="1"></GradientStop>  
 </LinearGradientBrush>  
 </Setter.Value>  
 </Setter>  
 ...  
</Style>

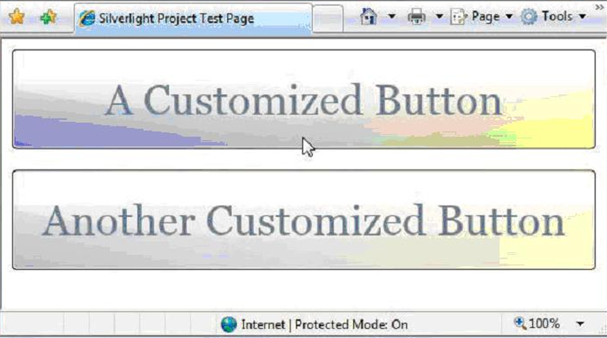
### Использование стиля

Каждому элементу Silverlight можно задать только один стиль (или ни одного). Стиль встраивается в элемент через свойство стиля элемента (которое определено в базовом классе FrameworkElement). Например, чтобы применить к кнопке заранее созданный стиль, вы должны указать ресурс стиля, как в этом случае:

<Button  
 Style="{StaticResource BigButtonStyle}"  
 Content="A Customized Button"/>

Стили задают исходный внешний вид элемента, но вы вправе перекрыть параметры заданные в стиле. Допустим, вы используете стиль BigButtonStyle, а также явно устанавливаете другое значение свойству FontSize, тогда значение свойства FontSize указанное внутри тега кнопки перекрывает это же свойство, описанное в стиле. Конечно, в идеале вы не должны полагаться на такое поведение. Вместо этого рекомендуется создать больше стилей, чтобы вы могли описать при помощи стилей как можно больше деталей элемента. Это даст вам большую гибкость с расчетом на последующие изменения настроек пользовательского интерфейса с меньшими проблемами.

На рис. 3.1 показана страница с двумя кнопками, использующими стиль BigButtonStyle.

  
Рисунок 3.1. Повторное использование настроек с применением стиля

Система стилей имеет много плюсов. Она не только позволяет вам создавать группы настроек с четкими связями, но и делает вашу XAML разметку более компактной благодаря простому способу подключения этих настроек. Но самое главное, вы можете применять стили, не заботясь об их внутреннем содержимом. В предыдущем примере настройки шрифта были описаны внутри стиля BigButtonSyle. Если спустя какое-то время вы решите увеличить величину значений свойств Padding и Margin, вы сможете просто добавить к описанию стиля соответствующие сеттеры свойств. Тогда новые настройки стиля автоматически вступят в силу для всех кнопок, использующих этот стиль.

Замечание. Технически допустимо задавать стиль программно. Тем не менее, этот действие может быть выполнено только один раз. Если вы попытаетесь задать стиль копке, которая уже его получила, получите исключение.

### Размещение стилей

В предыдущем примере стиль описан на уровне определенной страницы и затем использован для двух кнопок той же страницы. Хотя такая практика наиболее распространена, этот вариант размещения стиля не единственно возможный.

Собственно говоря, вы не обязаны объединять стили с ресурсами. Вы можете, к примеру, определить стиль отдельной кнопки, описав набор атрибутов стиля прямо в самой кнопке, как показано здесь:

<Button Content="A Customized Button">  
 <Button.Style>  
 <Style TargetType="Button">  
 <Setter Property="FontFamily" Value="Georgia" />  
 <Setter Property="FontSize" Value="40" />  
 <Setter Property="Foreground" Value="White" />  
 <Setter Property="Background" Value="Black" />  
 </Style>  
 </Button.Style>  
</Button>

И эта, очевидно, мало чем полезная конструкция, будет работать. В этом случае вы уже не сможете повторно применить стиль к другим кнопкам.

Будет более разумно, если вы решите описать стили в отдельном ресурсе. Если вы пожелаете создать более строго специализированные стили, вы можете описать их в ресурсах контейнера, например в StackPanel или в Grid (такие стили можно будет применять только к элементам, вложенным в этот контейнер). Более того, один и тот же стиль можно описать на нескольких уровнях сразу (в контейнере StackPanel, содержащем кнопку, и внутри страницы, содержащей этот StackPanel). В такой ситуации Silverlight следует стандартному процессу поиска имен: сначала он ищет в ресурсах текущего элемента, затем в контейнере содержащем этот элемент, затем в следующем контейнере уровнем выше, и т.д., пока не найдет стиль с соответствующим именем. Если вы хотите, чтобы стиль был доступен в любой части кода вашего приложения, опишите его в ресурсах приложения (в файле App.xaml), поиск в котором происходит в последнюю очередь.

## Основы шаблонов

Стили (Styles) позволяют изменять внешний вид элементов. Однако у стилей имеются ограничения на изменение свойств описанных в классе элемента. Например, в обычной кнопке присутствуют различные видимые детали, которые вы не в состоянии изменить, поскольку они не представлены через свойства. Одна из них – тень кнопки, возникающая в момент нажатия мышью.

В Silverlight имеется куда более радикальный инструмент для настройки подобных вещей, называемый **шаблонами** (templates). Тогда как стили можно применять к любому Silverlight элементу, использование шаблонов ограничено Silverlight Control’ами, т.е. элементами которые наследуются от класса Control принадлежащего пространству имен System.Windows.Controls. У таких элементов имеется свойство Template, через которое вы можете задавать свой шаблон, и тем самым, перекрыть стандартное визуальное представление Control’а.

Например, изменив шаблон, используемый объектом Button, вы можете создать множество экзотических кнопок, что было бы немыслимым при использовании одних только стилей. Вы сможете создавать кнопки округлой или другой нестандартной формы, а также кнопки со специальным эффектом, привлекающим взгляд в момент наведения мышкой (свечения, увеличения или мерцания).

Шаблоны – это одна из наиболее сложных деталей в WPF, поэтому не удивительно, что в Silverlight все еще отсутствуют некоторые возможности. Более неожиданно то, что факт отсутствия этих возможностей создает условия, в которых Silverlight Control’ы вынуждены опираться на новый комплекс стандартов и использовать более продуманные правила проектирования шаблонов.

### Создание шаблона

Каждый Control содержит встроенный набор правил, определяющий его отрисовку (в виде набора более простых элементов). Этот набор правил называется шаблоном Control’а (control template). Описывается он как блок XAML-разметки и применяется к Control’у через свойство «Template». Для примера давайте рассмотри простую кнопку. Предположим, создавая пользовательский Control, вы пожелаете получить больше контроля над эффектами затенения и анимации кнопки. В этом случае первым делом нужно заменить существующий стандартный шаблон кнопки на свой собственный. Для того чтобы создать шаблон кнопки вам понадобится нарисовать свой бордюр кнопки, ее фон, а также предусмотреть размещение контента кнопки. На роль бордюра имеется несколько кандидатов, тут все зависит от того, какой корневой элемент вы выберите:

* **Бордюр** (Border). Данный элемент решает две задачи: может содержать один элемент внутри себя (скажем TextBlock с заголовком кнопки), и отображать окаймляющий бордюр.
* **Таблица** (Grid). Расположив несколько элементов в одном месте, вы можете создать кнопку с каемкой. Воспользуйтесь элементом формы (таким как Rectangle или Path) и в той же ячейке разместите TextBlock. Удостоверьтесь, что описание TextBlock’а в XAML идет после описания фигуры, поскольку текст должен быть наложен на фоновую фигуру, а не наоборот. Одно из достоинств контейнера Grid в том, что он поддерживает автоматический контроль размера, и вы можете быть уверены, что ваш Control будет всегда иметь размер, соответствующий размеру своего содержимого.
* **Канва** (Canvas). В Canvas элементы могут размещаться строго по указанным координатам. В обычной ситуации это излишне, но может быть полезным, если вам требуется разместить несколько фигур особым образом относительно друг друга, например, при создании кнопки со сложным рисунком.

В следующем примере используется класс Border для сочетания закругленного оранжевого контура и броского красного фона с белым текстом:

<Button Content="A Custom Button Template">  
 <Button.Template>  
 <ControlTemplate TargetType="Button" >  
 <Border BorderBrush="Orange" BorderThickness="3"   
 BornerRadius="10" Background="Red">  
 <TextBlock Foreground="White" Text="A CustomTemplate"/>  
 </Border>  
 </ControlTemplate>  
 </Button.Template>  
</Button>

На рисунке 3.2 вы видите результат:

  
Рис. 3.2. Очень простая кнопка

Если вы попробуете использовать эту кнопку, то обнаружите, что данный шаблон слишком примитивен. Он лишен большей части признаков нормальной кнопки (например, визуального эффекта вдавливания в момент нажатия), и фактически игнорирует любое задаваемое свойство кнопки, даже самое основное - «Content». (Наш пример содержит жестко «вшитый» текст в TextBlock, так что команда «Content="A Custom Button Template"» тут только для вида.) Тем не менее, этот шаблон имеет все шансы стать настоящим шаблоном кнопки, и мы будем заниматься его доводкой в следующих параграфах.

Примечание: возможно как раз сейчас вы задумались, почему мы начали создание пользовательского шаблона, даже не взглянув на стандартный шаблон кнопки. Дело в том, что стандартные шаблоны содержат слишком много деталей: шаблон Control’а простой кнопки занимает целых четыре печатных листа. Поэтому только после того, как вы разберетесь в устройстве простого шаблона, вы сможете понять все детали стандартного шаблона.

### Повторное использование шаблона Control’а

В предыдущем примере описание шаблона расположено внутри элемента. Однако гораздо чаще шаблон Control’а размещают отдельно, т.к. часто вы будете применять шаблон сразу к нескольким Control’ам. Шаблон можно поместить в ресурсы:

<UserControl.Resources>  
 <ControlTemplate x:Key="ButtonTemplate" TargetType="Button" >  
 <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="10"   
 Background="Red">  
 <TextBlock Foreground="White" Text="A Custom Template"/>  
 </Border>  
 </ControlTemplate>  
</UserControl.Resources>

Затем в теге кнопки дописать ссылку на StaticResource:

<Button Template="{StaticResource ButtonTemplate}"/>

Такой подход не только упростит создание множества специализированных кнопок, но и позволит вам в будущем свободно модифицировать шаблон Control’а без необходимости изменять каждую кнопку, тем самым, уменьшая риск что-то испортить.

Есть еще один вариант – вы можете вложить описание шаблона в описание стиля. Преимущество такого подхода в том, что стиль может сочетать сеттеры, устанавливающие другие свойства, с сеттером, определяющим новый шаблон Control’а. Применяя стиль к кнопке, вы задействуете все сеттеры. В результате ваша кнопка получит новый шаблон, а также все упомянутые в стиле значения свойств.

## ContentPresenter

Кнопка из предыдущего примера довольно бесполезна, т.к. отображает только «вшитый» текст. Вы, конечно же, захотите иметь возможность задавать контент кнопки через свойство Button.Content. Для этого вам понадобится метка-заполнитель специального назначения – «ContentPresenter».

**ContentPresenter** присутствует во всех Control’ах содержащих какой-либо контент. Это специальная метка означающая «вставь контент здесь» и говорящая Silverlight, куда следует поместить содержимое Control’а. Вот как вы можете применить ее в нашем примере:

<UserControl.Resources>  
 <ControlTemplate x:Key="ButtonTemplate" TargetType="Button" >  
 <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="10"   
 Background="Red">  
 <ContentPresenter/>  
 </Border>  
 </ControlTemplate>  
</UserControl.Resources>

Замечание: ContentPresenter – наиболее востребованная, но не единственная метка. Control’ы представляющие списки и использующие ItemsControl применяют в своих шаблона метку ItemsPresenter, определяющую место, где должна располагаться панель списка элементов. Прокручиваемый контент Control’а ScrollViewer представлен меткой ScrollContentPresenter.

## Связывание в Шаблонах

Несмотря на то, что наш улучшенный шаблон кнопки уже поддерживает свойство контента, он все еще не воспринимает большинство других свойств. Для примера давайте рассмотри следующий случай использования шаблона:

<Button Template="{StaticResource ButtonTemplate}"   
 Content="A Templated Button"  
 Margin="10"  
 Padding="20"/>

В этой разметке свойствам кнопки Margin и Padding заданы значения 10 и 20 соответственно. За учет свойства Margin отвечает контейнер, содержащий кнопку, поэтому с этим проблем не возникает. Но свойство Padding не берется в расчет, поэтому края кнопки вплотную прилегают к ее контенту. Дело в том, что свойство Padding не будет иметь никакого эффекта, пока вы сами не позаботитесь об этом. Иначе говоря, именно ваш шаблон должен позаботиться о дополнительных отступах вокруг контента кнопки, величина которых задается в Padding.

Для этой цели в Silverlight\WPF есть специальная возможность – связывание в шаблонах (template bindings). С помощью template binding шаблон Control’а сможет извлекать значения свойств Control’а использующего данный шаблон. В следующем примере вы можете использовать template binding для получения значения свойства Padding и создания отступа вокруг ContentPresenter:

<ControlTemplate x:Key="ButtonTemplate" TargetType="Button" >  
 <Border BorderBrush="Orange"  
 BorderThickness="3"  
 CornerRadius="10"  
 Background="Red">  
 <ContentPresenter Margin="{TemplateBinding Padding}"/>  
 </Border>  
</ControlTemplate>

Теперь желаемый эффект получен и установлен некоторый зазор между текстом и краями кнопки. Рис. 3.3 демонстрирует вашу новую скромную кнопку:

  
Рисунок 3.3. Кнопка пользовательского шаблона Control’а

Замечание: Связывание в шаблонах похоже на обычное связывание данных (data bindings), но весит гораздо меньше, поскольку предназначено специально для шаблонов и поддерживает только одностороннее связывание данных (другими словами, можно передать информацию от Control’а в шаблон, но не наоборот).

Стало быть, вам придется добавить в ContentPresenter приличное количество элементов, если вы, конечно, хотите получить полноценную поддержку свойств класса Button и иметь возможность задавать такие свойства как выравнивание, перенос текста, и т.д.. ControlPresenter в составе стандартного шаблона кнопок выглядит примерно так:

<ContentPresenter  
 HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"  
 Margin="{TemplateBinding Padding}"   
 VerticalAlignment="{TemplateBinding VerticalContentAlignment}"   
 Content="{TemplateBinding Content}"   
 ContentTemplate="{TemplateBinding ContentTemplate}"/>

Связывание в шаблонах очень важно для свойства Content. Благодаря связыванию содержимое извлекается из Control’а и отображается в ContentPresenter. Зачастую можно не включать связывание для некоторых свойств шаблона, если вы не намерены их использовать.

Замечание: Связывание в шаблонах поддерживает встроенную во все [зависимые свойства](http://silverlight.su/viewtopic.php?id=56) инфраструктуру мониторинга изменений. Это значит, что если вы изменяете свойство Control’а, шаблон автоматически применяет его новое значение. Это особенно полезно, когда вы используете анимацию, многократно изменяющую значение свойства.

## Установка шаблонов через [стили](http://silverlight.su/viewtopic.php?id=59)

Связывание в шаблонах не ограничивается классом ContentPresenter. На самом деле вы можете использовать его в любом месте шаблона Control’а. Давайте рассмотрим наш пример кнопки с «вшитым» красным цветом фона элемента Border. Вот как можно использовать шаблонное связывание для управления фоном:

<Border BorderBrush="Orange"   
 BorderThickness="3"   
 CornerRadius="10"  
 Background="{TemplateBinding Background}">

Этот пример затрагивает один из вечных вопросов проектирования. Стоит ли жестко «зашивать» цвет, чтобы иметь стандартный вид кнопок, или применить связывание, чтобы сделать шаблон более гибким? На этот раз у нас имеется отличный компромисс, позволяющий убить сразу двух зайцев: вы можете сочетать шаблоны и стили! Основная идея в том, чтобы шаблон позволял изменять цвет фона, а стиль содержал бы значения по умолчанию, на случай если цвет не будет указан явно:

<Style x:Key="ButtonStyle" TargetType="Button">  
 <Setter Property="Background" Value="Red"/>  
 <Setter Property="Template">  
 <Setter.Value>  
 <ControlTemplate TargetType="Button">  
 <Border BorderBrush="Orange"   
 BorderThickness="3"   
 CornerRadius="10"  
 Background="{TemplateBinding Background}">  
 <ContentPresenter Margin="{TemplateBinding Padding}"/>  
 </Border>  
 </ControlTemplate>  
 </Setter.Value>  
 </Setter>  
</Style>

Но можно оставить [стиль](http://silverlight.su/viewtopic.php?id=59) отдельным ресурсом:

<Style x:Key="ButtonStyle" TargetType="Button">  
 <Setter Property="Background" Value="Red"/>  
 <Setter Property="Template" Value="{StaticResource ButtonTemplate}"/>  
</Style>

Только учтите, что в таком случае объявление [стиля](http://silverlight.su/viewtopic.php?id=59) должно следовать после объявления шаблона.

Синхронное использование [стилей](http://silverlight.su/viewtopic.php?id=59) и шаблонов также бывает полезным, если вам нужно задать свойства ([зависимые свойства](http://silverlight.su/viewtopic.php?id=56)) отсутствующие в ContentPresenter или в элементах контейнера вашего шаблона. Обратите внимание, что в нашем примере нет связывания для цвета шрифта кнопки. А все потому, что эти свойства (Foreground, FontFamily, FontSize, FontWeight, и т.д.) учавствуют в наследовании свойств. Когда вы задаете эти значения в элементе более высокого уровня (например, в кнопке), они распространяются на вложенные элементы (например, TextBlock внутри кнопки). Сам же ContentPresenter не имеет этих свойств, и в этом нет необходимости. Значения этих свойств передаются от Control’а к вложенному контенту минуя ContentPresenter.

Иногда вам может понадобиться изменить значение наследуемого свойства, чтобы ограничить изменение вида вашего Control’а. Например, в нашем случае важно, чтобы цвет текста был белым, поскольку белый текст лучше выделяется на красном фоне кнопки. Но стандартный цвет шрифта наследуется от Silverlight страницы, на которой расположены наши элементы, и этот цвет – черный. К тому же вы не можете установить цвет через ContentPresenter, т.к. он не содержит свойства Foreground. Для решения этой проблемы нужно объединить шаблон Control’а со [стилем](http://silverlight.su/viewtopic.php?id=59), сеттер которого задает белый цвет текста:

<Style x:Key="ButtonStyle" TargetType="Button">  
 <Setter Property="Foreground" Value="White"/>  
 <Setter Property="Background" Value="Red"/>  
 <Setter Property="Template" Value="{StaticResource ButtonTemplate}"/>  
</Style>

Данный подход объединяет в себе удобство и гибкость. Без каких либо дополнительных действий вы автоматически получаете кнопку с красным фоном и белым текстом, но также можете создать другой [стиль](http://silverlight.su/viewtopic.php?id=59), задающий другую цветовую схему, но использующий тот же самый шаблон. Это сэкономит вам много сил.

## Повторное использование настроек цвета

Как вы уже знаете, гибкий шаблон Control’а можно настраивать через свойства Control’а, значение которых можно задавать в [стиле](http://silverlight.su/viewtopic.php?id=59). Однако Silverlight приложения редко изменяют свойства только одного Control’а за раз. Как правило, для изменения вида приложения изменяются настройки сразу у множества шаблонов. В таких ситуациях нужно уметь синхронно использовать определенные свойства Control’ов (например, настройки цвета). Для этого все «вшитые» значения вынесите из [стилей](http://silverlight.su/viewtopic.php?id=59) и шаблонов, описав их как отдельные ресурсы. Например:

<SolidColorBrush x:Key="BackgroundBrush" Color="Red"/>

Затем вы можете обращаться к этим ресурсам из [стилей](http://silverlight.su/viewtopic.php?id=59) и шаблонов:

<Style x:Key="ButtonStyle" TargetType="Button">  
 <Setter Property="Foreground" Value="White"/>  
 <Setter Property="Background" Value="{StaticResource BackgroundBrush}"/>  
 <Setter Property="Template" Value="{StaticResource ButtonTemplate}"/>  
</Style>

Простое добавление ресурса с определенным именем позволит вам использовать один и тот же шаблон, но различные цвета бордюра. Однако имеется и слабая сторона такого подхода, т.к. это несколько усложняет ваш проект.

Для большей гибкости вы можете вынести в отдельный ресурс даже настройки цветов, а затем использовать их как ресурсы кисти:

<Color  
 x:Key="BackgroundColor">#FF800000</Color>  
<SolidColorBrush  
 x:Key="BackgroundBrush" Color="{StaticResource BackgroundColor}"/>

Такой принцип позволяет повторно использовать цветовую схему для самых разных случаев (например, для заливки и для градиентной кисти) без необходимости дублировать информацию о цвете в вашей разметке. При правильном использовании такой модели вы сможете изменять цветовую схему всего приложения, манипулируя всего лишь одним набором ресурсов.

Замечание: при объявлении цвета как отдельного ресурса ему может быть задан цвет в виде имени цвета или в виде шестнадцатеричного HTML-кода цвета (как в примере). Но, к сожалению, нельзя объявить цвет в XAML, используя набор красного, зеленого, и синего составляющих цвета.

## Ключевые термины

**Styles:** совокупность значений свойств, которые можно все сразу применить к нужному элементу.

**Templates**: каждый Control содержит встроенный набор правил, определяющий его отрисовку (в виде набора более простых элементов). Этот набор правил называется шаблоном Control’а (control template). Описывается он как блок XAML-разметки и применяется к Control’у через свойство «Template».

## Краткие итоги

В этой лекции мы рассмотрели механизм применения некоторого набора свойств к одному или нескольким элементам управления - стили. С помощью стилей можно создавать однородные темы и применять их к разным приложениям. Однако у стилей имеются ограничения на изменение свойств описанных в классе элемента. В Silverlight имеется куда более радикальный инструмент для настройки подобных вещей, называемый **шаблонами** (templates). Тогда как стили можно применять к любому Silverlight элементу, использование шаблонов ограничено Silverlight Control’ами, т.е. элементами которые наследуются от класса Control принадлежащего пространству имен System.Windows.Controls. Более детально эти отличия мы рассмотри в следующей лекции.

## Набор для практики

Вопросы:

1. Приведите пример необходимости использования класса ContentPresenter.
2. Назначение ресурсов в WPF. Пример использования?
3. Назначение стилей в WPF. Пример использования?
4. Назначение шаблонов в WPF. Пример использования?
5. Поясните назначение установки шаблонов через стили.

## Литература

1. Стили и шаблоны <http://msdn.microsoft.com/ru-ru/library/bb613570.aspx>.
2. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 302-323.
3. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008 — стр. 123-214.

# Лабораторная работа 2. Стили и ресурсы.

**Краткая аннотация лабораторной работы:** в лабораторной работе будут даны задания для самостоятельного выполнения на закрепление пройденной теоретической части лекций 2 и 3, а так же рассмотрен пример, нацеленный показать пути решения поставленных в лабораторной работе задач.

**Цель лабораторной работы:** рассмотреть небольшой пример работы со стилями в WPF. Закрепить знания, полученные в лекциях 2 и 3.

## Задания для самостоятельного выполнения

Изменить стандартный стиль у контрола управдения TextBox. Задать цвет фона, рамку, изменить шрифт. Стиль вынести в файл Generic.xaml

## Учебный элемент. Стили и ресурсы в WPF

Стили – это одна из наиболее мощных и полезных особенностей WPF и XAML. Стили позволяют практически до неузнаваемости изменять внешний вид (и поведение) элементов управления и форм.

Предположим, что мы хотим иметь возможность менять внешний вид интерфейса на лету. В таком случае мы можем просто создать несколько различных стилей и менять их в ответ на какие-либо действия пользователя. Стили также очень полезны из-за того, что они позволяют избегать чрезмерного разрастания XAML кода страницы, так как стили могут быть вынесены в отдельный файл. Каждый элемент управления имеет множество свойств, каждое из которых может быть задано при помощи стиля.

В данном учебном элементе мы рассмотрим небольшой пример работы со стилями в WPF.

### Шаг 1. Применение стиля в XAML документе

Итак, пускай мы хотим задать цвет фона и размер шрифта для кнопки. Самый простой и очевидный способ сделать это – это просто задать все нужные стили прямо в теге кнопки:

<Button Name="button1" Width="150" FontSize="12" Background="AliceBlue"   
 Content="Click Me!" Margin="3" Click="OnResources" />

Есть еще один способ задать стиль элемента прямо в самом элементе:

<Button Width="150" Content="Click Me!" Margin="3">  
 <Button.Style>  
 <Style TargetType="Button">  
 <Setter Property="Background" Value="Yellow" />  
 <Setter Property="FontSize" Value="14" />  
 <Setter Property="FontWeight" Value="Bold" />  
 </Style>  
 </Button.Style>  
</Button>

Понятно, что использование такого способа задания стиля не является оптимальным. Например, часто нужно применить один и тот же стиль к нескольким элементам управления, в этом случае нам пришлось бы копировать код.

К счастью, стили можно создавать отдельно от самих элементов управления и хранить их в коллекциях ресурсов. Например, вот так:

<Window.Resources>  
 <Style TargetType="{x:Type Button}">  
 <Setter Property="Background" Value="LemonChiffon" />  
 <Setter Property="FontSize" Value="18" />  
 </Style>  
</Window.Resources>

Такой стиль станет стилем по умолчанию для кнопок на текущей странице.

<Button Width="200" Content="Uses default style" Margin="3" />

Давайте зададим еще несколько стилей.

<Window.Resources>  
 <Style TargetType="{x:Type Button}">  
 <Setter Property="Background" Value="LemonChiffon" />  
 <Setter Property="FontSize" Value="18" />  
 </Style>  
 <Style x:Key="ButtonStyle">  
 <Setter Property="Button.Background" Value="Red" />  
 <Setter Property="Button.Foreground" Value="White" />  
 <Setter Property="Button.FontSize" Value="18" />  
 </Style>  
  
 <Style x:Key="FancyButtonStyle">  
 <Setter Property="Button.FontSize" Value="22" />  
 <Setter Property="Button.Foreground" Value="White" />  
 <Setter Property="Button.Background">  
 <Setter.Value>  
 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">  
 <GradientStop Offset="0.0" Color="LightCyan" />  
 <GradientStop Offset="0.14" Color="Cyan" />  
 <GradientStop Offset="0.7" Color="DarkCyan" />  
 </LinearGradientBrush>  
 </Setter.Value>  
 </Setter>  
 </Style>  
</Window.Resources>

Применим их к нескольким новым кнопкам.

<Button  
 Width="200"   
 Content="Uses named style"  
 Style="{StaticResource ButtonStyle}"  
 Margin="3" />  
<Button  
 Width="200"  
 Content="Fancy button style"  
 Style="{StaticResource FancyButtonStyle}"  
 Margin="3" />

Еще одной особенностью стилей в WPF является то, что стили могут наследовать свойства других стилей. Вот пример:

<Window.Resources>  
 <Style TargetType="{x:Type Button}">  
 <Setter Property="Background" Value="LemonChiffon" />  
 <Setter Property="FontSize" Value="18" />  
 </Style>  
 <Style x:Key="ButtonStyle">  
 <Setter Property="Button.Background" Value="Red" />  
 <Setter Property="Button.Foreground" Value="White" />  
 <Setter Property="Button.FontSize" Value="18" />  
 </Style>  
 <Style x:Key="FancyButtonStyle">  
 <Setter Property="Button.FontSize" Value="22" />  
 <Setter Property="Button.Foreground" Value="White" />  
 <Setter Property="Button.Background">  
 <Setter.Value>  
 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">  
 <GradientStop Offset="0.0" Color="LightCyan" />  
 <GradientStop Offset="0.14" Color="Cyan" />  
 <GradientStop Offset="0.7" Color="DarkCyan" />  
 </LinearGradientBrush>  
 </Setter.Value>  
 </Setter>  
 </Style>  
 <Style x:Key="AnotherButtonStyle"   
 BasedOn="{StaticResource FancyButtonStyle}"  
 TargetType="Button">  
 <Setter Property="Foreground">  
 <Setter.Value>  
 <LinearGradientBrush>  
 <GradientStop Offset="0.2" Color="White" />  
 <GradientStop Offset="0.5" Color="LightYellow" />  
 <GradientStop Offset="0.9" Color="Orange" />  
 </LinearGradientBrush>  
 </Setter.Value>  
 </Setter>  
 </Style>  
</Window.Resources>

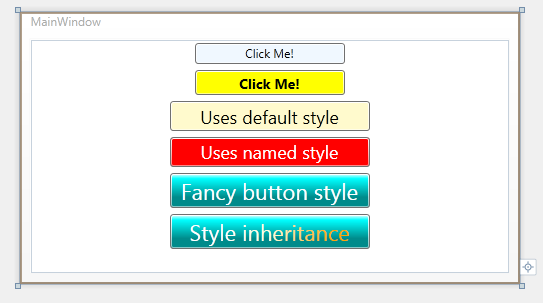
Стиль AnotherButtonStyle наследует свойства от FancyButtonStyle и переопределяет лишьсвойство ForeGround. Можно убедиться, что цвет фона и размер шрифта идентичен для двух этих стилей.

<Button  
 Width="200"  
 Content="Style inheritance"  
 Style="{StaticResource AnotherButtonStyle}"  
 Margin="3"/>

Так будет выглядеть весь код страницы:

<Window   
 x:Class="StylesAndResources.MainWindow"  
        xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
        xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
        Title="MainWindow" Height="240" Width="500">  
    <Window.Resources>  
        <Style TargetType="{x:Type Button}">  
            <Setter Property="Background" Value="LemonChiffon" />  
            <Setter Property="FontSize" Value="18" />  
        </Style>  
        <Style x:Key="ButtonStyle">  
            <Setter Property="Button.Background" Value="Red" />  
            <Setter Property="Button.Foreground" Value="White" />  
            <Setter Property="Button.FontSize" Value="18" />  
        </Style>  
  
        <Style x:Key="FancyButtonStyle">  
            <Setter Property="Button.FontSize" Value="22" />  
            <Setter Property="Button.Foreground" Value="White" />  
            <Setter Property="Button.Background">  
                <Setter.Value>  
                    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">  
                        <GradientStop Offset="0.0" Color="LightCyan" />  
                        <GradientStop Offset="0.14" Color="Cyan" />  
                        <GradientStop Offset="0.7" Color="DarkCyan" />  
                    </LinearGradientBrush>  
                </Setter.Value>  
            </Setter>  
        </Style>  
         
        <Style x:Key="AnotherButtonStyle"   
 BasedOn="{StaticResourceFancyButtonStyle}" TargetType="Button">  
            <Setter Property="Foreground">  
                <Setter.Value>  
                    <LinearGradientBrush>  
                        <GradientStop Offset="0.2" Color="White" />  
                        <GradientStop Offset="0.5" Color="LightYellow" />  
                        <GradientStop Offset="0.9" Color="Orange" />  
                    </LinearGradientBrush>  
                </Setter.Value>  
            </Setter>  
        </Style>  
  
  
    </Window.Resources>  
    <StackPanel>  
        <Button Name="button1" Width="150" FontSize="12"   
 Background="AliceBlue"   
 Content="Click Me!" Margin="3" Click="OnResources" />  
        <Button Width="150" Content="Click Me!" Margin="3">  
            <Button.Style>  
                <Style TargetType="Button">  
                    <Setter Property="Background" Value="Yellow" />  
                    <Setter Property="FontSize" Value="14" />  
                    <Setter Property="FontWeight" Value="Bold" />  
                </Style>  
            </Button.Style>  
        </Button>  
        <Button Width="200" Content="Uses default style" Margin="3" />  
        <Button Width="200" Content="Uses named style"  
 Style="{StaticResourceButtonStyle}" Margin="3" />  
        <Button Width="200" Content="Fancy button style"  
 Style="{StaticResourceFancyButtonStyle}" Margin="3" />  
        <Button Width="200" Content="Style inheritance"  
 Style="{StaticResourceAnotherButtonStyle}" Margin="3" />  
  
    </StackPanel>  
</Window>

В результате должно получиться вот такое окошко с шестью кнопками.

[](http://2.bp.blogspot.com/--RDfw_spvhY/Tg0FpbdlPSI/AAAAAAAABLE/neV-FVLKyKY/s1600/st.png)  
Рис. л. р. 2.1. Результат применения стилей.

### Шаг 2. Динамическое применение стилей в коде приложения

Стили также можно создавать и применять динамически в коде приложения. Давайте посмотрим на простой пример создания такого стиля.

using System;  
using System.Collections.Generic;  
using System.Linq;  
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.Navigation;  
using System.Windows.Shapes;  
  
namespace StylesAndResources  
{  
    /// <summary>  
    /// Interaction logic for MainWindow.xaml  
    /// </summary>  
    public partial class MainWindow : Window  
    {  
        public MainWindow()  
        {  
            InitializeComponent();  
        }

        private void OnResources(object sender, RoutedEventArgs e)  
        {  
            new ResourceDemo().Show();  
        }  
    }  
}

<Window x:Class="StylesAndResources.ResourceDemo"  
        xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
        xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
        Title="Resource Demo" Height="300" Width="300">  
<StackPanel x:Name="myContainer">  
        <StackPanel.Resources>  
            <LinearGradientBrush x:Key="MyGradientBrush"   
 StartPoint="0,0" EndPoint="0.3,1">  
                <GradientStop Offset="0.0" Color="LightCyan" />  
                <GradientStop Offset="0.14" Color="Cyan" />  
                <GradientStop Offset="0.7" Color="DarkCyan" />  
            </LinearGradientBrush>  
        </StackPanel.Resources>  
        <Button Width="200" Height="50" Foreground="White" Margin="5"   
 Background="{StaticResource MyGradientBrush}"  
 Content="Click Me!" />  
       <Button Name="button1" Width="220" Height="50" Margin="5"   
 Click="button1\_Click"/>  
        <Button Name="button2" Width="200" Height="50" Foreground="White"   
 Margin="5" Background="{DynamicResource MyGradientBrush}"  
 Content="Change Resource" Click="button2\_Click" />  
    </StackPanel>  
</Window>

Создадим еще одну страницу и реализуем обработчик первой кнопки так, чтобы при нажатии на нее открывалась новая страница:

using System;  
using System.Collections.Generic;  
using System.Linq;  
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.Navigation;  
using System.Windows.Shapes;  
  
namespace StylesAndResources  
{  
    /// <summary>  
    /// Interaction logic for MainWindow.xaml  
    /// </summary>  
    public partial class MainWindow : Window  
    {  
        public MainWindow()  
        {  
            InitializeComponent();  
        }  
        private void OnResources(object sender, RoutedEventArgs e)  
        {  
            new ResourceDemo().Show();  
        }  
    }  
}

<Window x:Class="StylesAndResources.ResourceDemo"  
        xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
        xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
        Title="Resource Demo" Height="300" Width="300">  
    <StackPanel x:Name="myContainer">  
        <StackPanel.Resources>  
            <LinearGradientBrush x:Key="MyGradientBrush"  
  StartPoint="0,0" EndPoint="0.3,1">  
  
                <GradientStop Offset="0.0" Color="LightCyan" />  
  
                <GradientStop Offset="0.14" Color="Cyan" />  
  
                <GradientStop Offset="0.7" Color="DarkCyan" />  
  
            </LinearGradientBrush>  
  
        </StackPanel.Resources>  
  
        <Button Width="200" Height="50" Foreground="White"   
 Margin="5" Background="{StaticResource MyGradientBrush}"  
 Content="Click Me!" />  
        <Button Name="button1" Width="220" Height="50" Margin="5"  
  Click="button1\_Click"/>  
 <Button Name="button2"  
  Width="200" Height="50" Foreground="White"  Margin="5"  
  Background="{DynamicResource MyGradientBrush}"  
  Content="Change Resource" Click="button2\_Click" />  
 </StackPanel>  
</Window>

Тут мы задали стиль MyGradientBrush и у нас есть три кнопки. Для первой мы просто используем этот стиль. А для остальных будем менять его динамически.

В обработчике нажатия на первую кнопку нам нужно найти стиль MyGradientBrush и применить его к текущей кнопке.

        private void button1\_Click(object sender, RoutedEventArgs e)  
        {  
            Control ctrl = sender as Control;  
            ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;  
        }

Для второй кнопки мы создадим несколько другой алгоритм. Сначала мы очистим всю коллекцию стилей для данной страницы. После этого мы создадим кисть и добавим ее в коллекцию стилей сстриницы.

        private void button2\_Click(object sender, RoutedEventArgs e)  
        {  
            myContainer.Resources.Clear();  
            var brush = new LinearGradientBrush {   
 StartPoint = new Point(0, 0), EndPoint = new Point(0, 1) };  
  
            brush.GradientStops = new GradientStopCollection()  
            {  
                new GradientStop(Colors.White, 0.0),  
                new GradientStop(Colors.Yellow, 0.14),  
                new GradientStop(Colors.YellowGreen, 0.7)  
            };  
  
            myContainer.Resources.Add("MyGradientBrush", brush);  
        }

Обратите внимание на то, что теперь новый стиль будет применяться и к кнопке button1, в то время как кпопка со статическим стилем не изменится.

Весь код этой страницы выглядит вот так:

using System;  
using System.Collections.Generic;  
using System.Linq;  
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;

namespace StylesAndResources  
{  
    /// <summary>  
    /// Interaction logic for ResourceDemo.xaml  
    /// </summary>  
    public partial class ResourceDemo : Window  
    {  
        public ResourceDemo()  
        {  
            InitializeComponent();  
        }

       private void button1\_Click(object sender, RoutedEventArgs e)  
        {  
            Control ctrl = sender as Control;  
            ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;  
        }  
  
        private void button2\_Click(object sender, RoutedEventArgs e)  
        {  
            myContainer.Resources.Clear();  
            var brush = new LinearGradientBrush { StartPoint =  
  new Point(0, 0, EndPoint = new Point(0, 1) };  
            brush.GradientStops = new GradientStopCollection()  
            {  
                new GradientStop(Colors.White, 0.0),  
                new GradientStop(Colors.Yellow, 0.14),  
                new GradientStop(Colors.YellowGreen, 0.7)  
            };  
  
            myContainer.Resources.Add("MyGradientBrush", brush);  
        }  
    }  
  
}

## Краткие итоги

Стили – это очень важный инструмент при работе с XAML, Windows Presentation Foundation и Silverlight. В данной лабораторной работе мы рассмотрели несколько возможных способов их использования.

## Литература

1. Стили и шаблоны

<http://msdn.microsoft.com/ru-ru/library/bb613570.aspx>.

# Лекция 4. Silverlight и WPF в Visual Studio 2010.

**Краткая аннотация лекции:** освещаются новые возможности Framework 4, WPF версии 4, а так же новые возможности Microsoft Visual Studio 2010 добавленные для поддержки реализованных нововведений.

**Цель лекции:** рассмотреть новые средства WPF версии 4. Показать какая поддержка появидась в Microsoft Visual Studio 2010 для удобства и эффективного программирования с применением технологии WPF.

## Новые возможности WPF 4.

## WPF история становления

WPF 4 — относительно новая технология. Частично она входила в несколько выпусков .NET и постепенно совершенствовалась.

### WPF 3.0.

Первая версия WPF вышла вместе с двумя другими технологиями: **Windows Communication Foundation (WCF)** и **Windows Workflow Foundation (WF)**. Все вместе это называлось .NET 3.0.

### WPF 3.5.

Год спустя, вышла новая версия WPF, как часть .NET Framework 3.5. Новые средства WPF в основном были слегка усовершенствованы, включая исправление ошибок и повышение производительности.

### WPF 3.5 SP1.

Когда вышел пакет обновлений .NET Framework Service Pack 1 (SP1), проектировщики WPF получили возможность добавить некоторые новые средства, подобные сглаженной графике (благодаря построителям текстуры) и изощренному элементу управления DataGrid.

### WPF 4.

В последнем выпуске WPF появилось множество улучшений, включая ценные новые средства, построенные на базе существующей инфраструктуры WPF. Среди некоторых наиболее заметных изменений — улучшенная визуализация текста, более естественная анимация и поддержка средств Windows 7, таких как сенсорные возможности и новая панель задач.

## Visual Studio 2010 и WPF.

Вся "обстановка" в **IDE** реализована на WPF. Буквально все рамки и украшения документа, и служебные окна являются частью "визуального дерева" (Visual Tree) WPF. В него включены главное меню, панели инструментов, контекстные меню и панель статуса. Документ в целом - по сути, вся система управления окном - тоже WPF, включая оверлейные слои, которые появляются при перетаскивании окон. Обновленная стартовая страница (Start Page) так же полностью реализована на WPF и, при желании, Вы можете написать свою собственную замену стартовой странице на XAML. Новый редактор текста, который был создан с нуля с использованием WPF. Использование WPF в редакторе открывает для разработчиков возможности усовершенствования редактора путем создания впечатляющих расширения написанием всего нескольких строк кода. Несколько других часто используемых текстовых окон, такие, как "Вывод даннах" (Output) и "Cписок найденного" (Find Results List) также базируются на новом текстовом редакторе.

Небольшой, но важной функцией для Visual Studio является возможность создания основного окна WPF, не показывая его. Он используется в ситуациях, когда Visual Studio создается программно, например, в режиме "Design Time Extensibility" или при сборке приложения в режиме командной строки.

Для взаимодействия с меню и панелями инструментов с клавиатуры в Visual Studio использует новую функцию WPF "режим меню" (MenuMode). Она позволяет перенаправить сообщения в меню или панель инструментов без "кражи" фокуса ввода у активного в данный момент элемента. Без этой функции некоторые сценарии в Visual Studio было бы невозможно осуществить на практике с использованием WPF.

### Поддержка множества целевых платформ

В прошлом каждая версия Visual Studio была тесно привязана к определенной версии .NET. Версия Visual Studio 2010 свободна от этого ограничения и позволяет проектировать приложения, ориентированные на любую версию .NET— от 2.0 до 4. Хотя очевидно невозможно создать приложение WPF для .NET 2.0, в версиях .NET 3.0 и 3.5 поддержка WPF имеется. Выбор в качестве целевой платформы .NET 3.0 обеспечивает наиболее широкую совместимость (т.к. приложения .NET 3.0 могут работать под управлением исполняющих сред .NET 3.0, 3.5 и 4). Выбор в качестве целевой платформы .NET 3.5 или .NET 4 открывает доступ к новейшим средствам WPF, имеющимся в .NET.

При создании нового проекта в Visual Studio можно выбирать целевую версию .NET Framework в раскрывающемся списке, который расположен в верхней части диалогового окна New Project (Новый проект) прямо над списком шаблонов проектов (рис. 4.1).

  
Рисунок 4.1. Выбор целевой версии .NET Framework.

Целевую версию можно изменить в любой момент позже, дважды щелкнув на узле Properties (Свойства) в окне Solution Explorer (Проводник решения) и изменив выбор в списке Target Framework (Целевая платформа).

Для обеспечения аккуратной поддержки множества целевых платформ Visual Studio 2010 включает ссылочные сборки для каждой версии .NET. Эти сборки содержат метаданные каждого типа, но ничего из кода, нужного для их реализации. Это значит, что Visual Studio 2010 может использовать ссылочную сборку для настройки средства

IntelliSense и проверки ошибок, гарантируя, что вы не сможете использовать элементы управления, классы или члены, которые не доступны в выбранной версии .NET. Эти метаданные также используются для определения того, что должно появиться в окне Properties (Свойства) и браузере объектов (Object Browser), и т.д., гарантируя, что вся IDE-среда будет ограничена выбранной версией .NET.

### Клиентский профиль .NET

Как ни странно, доступны два способа выбрать в качестве цели WPF 4. Первый способ — построить приложение, которое требует стандартной установки полной платформы .NET Framework 4. Второй способ — построить приложение, которому требуется .NET Framework 4 Client Profile (Клиентский профиль .NET Framework 4).

**Клиентский профиль *— это подмножество .NET Framework, которое требуется многофункциональным клиентским приложениями вроде WPF.*** Сюда не входят средства серверной стороны, такие как ASP.NET, отладчики, средства разработки, компиляторы кода и унаследованные средства (подобные поддержке баз данных Oracle). Более важно то, что клиент имеет меньший размер, требуя загрузки около 30 Мбайт, в то время как полный комплект распространения .NET Framework занимает около 100 Мбайт. Естественно, если приложение ориентировано на .NET Framework 4 Client Profile, оно без проблем будет работать под управлением полной версии .NET Framework. Концепция клиентского профиля появилась в .NET 3.5 SP1. Однако в ней по-прежнему присутствуют несколько моментов, которые мешают ей стать стандартом. В .NET 4 были проведены работы по тонкой настройке средств, включаемых в комплект клиентского профиля, предполагая сделать его стандартным выбором для любого приложения. В Visual Studio 2010 большинство проектов автоматически нацелены на .NET Framework 4 Client Profile. (Именно это вы получаете, выбирая .NET Framework 4 в диалоговом окне New Project.) Изменив настройку Target Framework (Целевая платформа) в свойствах проекта, можно увидеть более подробный список, который имеет отдельные опции для полной версии .NET Framework 4 и .NET Framework 4 Client Profile.

При выборе целевой версии .NET часто важно учитывать, насколько широко распространены различные исполняющие среды в настоящее время. В идеале пользователи должны иметь возможность запускать приложения, не требуя дополнительного шага по загрузке и установке. Ниже дано несколько советов, которые помогут принять правильное решение:

* Windows Vista включает .NET Framework 3.0.
* Windows 7 включает .NET Framework 3.5 SP1.
* .NET Framework 4 Client Profile является рекомендуемым обновлением (через службу Windows Update) для Windows Vista и Windows 7. Для компьютеров Windows XP оно является необязательным.

### Визуальный конструктор Visual Studio

Несмотря на тот факт, что Visual Studio является важнейшим инструментом для программирования с применением WPF, в предыдущих версиях был существенный пробел в доступных возможностях — они не предлагали графического визуального конструктора для создания пользовательского интерфейса. В результате разработчики были вынуждены писать код XAML вручную либо переключаться между Visual Studio и более ориентированным на дизайн инструментом Expression Blend. В Visual Studio 2010, наконец, этот недостаток был восполнен за счет появления мощного визуального конструктора для создания пользовательских интерфейсов WPF.

Однако тот факт, что Visual Studio 2010 позволяет легко перетаскивать окна WPF на поверхность проектирования, не означает, что это нужно делать прямо сейчас или вообще когда-либо. Visual Studio может помочь в этом, но будет намного легче, если первым делом освоить основы разметки XAML и компоновки WPF. Это позволит впоследствии просматривать код разметки, сгенерированный Visual Studio, и при необходимости модифицировать его вручную.

Часть профессиональных разработчиков используют Visual Studio, часть — Expression Blend, есть те, кто пишет код XAML вручную, а есть те, кто применяет комбинацию перечисленных методов с последующим конфигурированием в визуальном конструкторе Visual Studio.

## Новые средства WPF 4

### Новые элементы управления.

Семейство элементов WPF продолжает расти. Теперь оно включает профессиональный выглядящий DataGrid стандартные DataPicker и Calendar и встроенный WebBrowser для просмотра HTML-разметки и веб-серфинга. Отдельная загрузка также добавляет полезный элемент управления Ribbon, который придает приложениям современный вид.

### Диспетчер визуального состояния.

Впервые появившийся в Silverlight, диспетчер визуального состояния облегчает изменение обложек элементов управления без необходимости понимания их внутреннего устройства и работы.

### Поддержка Multitouch

С представлением мультитач ввода и поддержки манипулированием обработки, WPF 4 предоставляет отличный способ добавить пикантности в ваше клиентское приложение в Windows 7. Появились новые события манипуляции и инерции:

* Multitouch события Manipulation, Inertia (Pan, Zoom, Rotate) над UIElement
* Простые Multitouch события (Up, Move, Down) над UIElement, UIElement3D и ContentElement
* Захват нескольких элементов управления
* Поддержка Multitouch в ScrollViewer
* Расширяемость сенсорных устройств

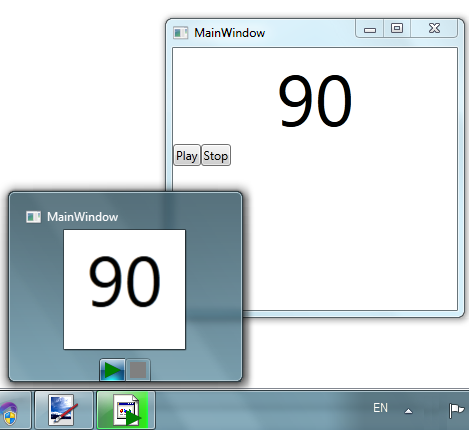
Чтобы начать работать с Multitouch нужно ознакомиться с новыми событиями, которые добавили в классы [UIElement](http://msdn.microsoft.com/en-us/library/system.windows.uielement(VS.100).aspx), [UIElement3D](http://msdn.microsoft.com/en-us/library/system.windows.uielement3d(VS.100).aspx) и [ContentElement](http://msdn.microsoft.com/en-us/library/system.windows.contentelement(VS.100).aspx).

### Windows 7 Shell Intergration

Теперь запросто можно добавлять поддержку всех красивостей и удобностей от Windows 7 в приложение. Для этого нам нужно обратиться к пространству имен [System.Windows.Shell](http://msdn.microsoft.com/en-us/library/system.windows.shell(VS.100).aspx) и посмотреть, что же он нам предлагает.

#### [TaskbarItemInfo](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo(VS.100).aspx)

Он предлагает управляемую обертку для Taskbar в среде Windows 7. У класса [Window](http://msdn.microsoft.com/en-us/library/system.windows.window(VS.100).aspx) появилось свойство (dependency) [TaskbarItemInfo](http://msdn.microsoft.com/en-us/library/system.windows.window.taskbariteminfo(VS.100).aspx), которому и необходимо установить объект типа TaskbarItemInfo (можно как в XAML - декларативно, так и программно). При помощи свойства [ThumbnailClipMargin](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo.thumbnailclipmarginproperty(VS.100).aspx) можно установить, какая часть окна будет отображена в Preview окна на taskbar (показывается при наведении, если включено Aero), то есть можно отображать не все окно, а, например, только значимую необходимую часть. Более того [TaskbarItemInfo](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo(VS.100).aspx) позволяет отобразить функциональные клавиши в том же Preview ([ThumbButtonInfos](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo.thumbbuttoninfos(VS.100).aspx)), а так же отобразить какой-нибудь свой элемент над иконкой на Taskbar ([Overlay](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo.overlay(VS.100).aspx)), и состояние длительного процесса ([ProgressValue](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo.progressvalue(VS.100).aspx) и [ProgressState](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo.progressstate(VS.100).aspx)). На рис. 4.2 вы можете видеть как будет это выглядеть (пример с [MSDN](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo(VS.100).aspx))

  
Рис. 4.2. Применение [TaskbarItemInfo](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo(VS.100).aspx).

#### [JumpList](http://msdn.microsoft.com/en-us/library/system.windows.shell.jumplist(VS.100).aspx)

Тут все просто, он может в себя включать либо [JumpTask](http://msdn.microsoft.com/en-us/library/system.windows.shell.jumptask(VS.100).aspx), либо [JumpPath](http://msdn.microsoft.com/en-us/library/system.windows.shell.jumppath(VS.100).aspx), первый может задавать программу с набором аргументов, второй путь до файла.

#### Windows 7 shell integration

Полезное дополнение, но так как в основном мы все программируем для корпоративных клиентов, то на него нужно смотреть как на возможность дополнения функциональности приложения, если у пользователя стоит Windows 7. Так, если мы как-то отображаем долгий процесс, то почему бы и не отобразить его при помощи [TaskbarItemInfo](http://msdn.microsoft.com/en-us/library/system.windows.shell.taskbariteminfo(VS.100).aspx) в дополнение: тогда играющий в косынку менеджер всегда сможет заметить, когда программа закончила формировать отчет или выполнила какую либо задачу. Но вот вынести часто используемые файлы просто в [JumpList](http://msdn.microsoft.com/en-us/library/system.windows.shell.jumplist(VS.100).aspx) не достаточно, так как мы ограничим пользователей, которые используют более ранние версии Windows, а о них тоже стоит подумать.

### Updated File Dialogs

В предыдущих версиях окна выбора файлов, директорий и т.п. выглядели в стиле WinXP, теперь же окна полностью соответствуют ОС, в которой запущено приложение, а точнее диалоги выбора файлов в Windows 7 выглядят как в Windows 7.

### Custom Dictionary

В WPF, как вы знаете, есть поддержка проверки правописания SpellCheck. А может и не знаете, потому что поддерживаются только 4 языка: английский, немецкий, испанский и французский, и расширить это никак нельзя. В WPF 4.0 появилась возможность заводить Custom Dictionary, чтобы добавить всяческие слова, вроде *lol*, в лексикон проверки. Но опять же, сделать проверку русского языка при помощи этого нельзя, даже если вы забьете всевозможные русские слова в такой словарь, то изначально вам WPF скажет, что данный язык не поддерживается. Более подробная информация об этом тут: <http://blogs.msdn.com/b/text/archive/2009/10/02/custom-dictionaries.aspx>.

### Графика

#### Шейдеры

Как и должно быть: новый WPF и поддержка новых **шейдеров**, теперь поддерживаются шейдеры версии 3.0. Опять же, нужно не забывать, что существуют компьютеры и без поддержки шейдеров версии 3.0 – для этого введен дополнительный набор методов для определения версии шейдеров, поддерживаемых системой, в классе [RenderCapability](http://msdn.microsoft.com/en-us/library/system.windows.media.rendercapability_members(VS.100).aspx). В сети достаточно примеров написанных эффектов, один из них был Grayscale эффект, который из цветной фотографии делал ч/б – использовали в проекте для toolbar’a – все иконки делали ч/б, и только при наведении они приобретали цвет (позаимствовали идею у Photoshop).

#### Кеширование графики

С графикой также связаны нововведения, которые позволят повысить производительность наших приложений. Теперь у класса [UIElement](http://msdn.microsoft.com/en-us/library/system.windows.uielement(VS.100).aspx) есть свойство [CacheMode](http://msdn.microsoft.com/en-us/library/system.windows.uielement.cachemode(VS.100).aspx), которому мы можем установить объект типа [BitmapCache](http://msdn.microsoft.com/en-us/library/system.windows.media.bitmapcache(VS.100).aspx). Полезно это будет, в основном, когда ваше приложение использует множество элементов, описанных декларативно при помощи Path, тогда вы можете задать кеш для этого элемента, с указанием масштаба при котором данный векторный рисунок прорендериться. Посмотреть пример можно на MSDN <http://msdn.microsoft.com/en-us/library/ee230083(VS.100).aspx>, где в векторе нарисована алюминиевая банка, у которой постоянно меняют масштаб. Основа этого примера в строках

<Canvas.CacheMode>

<BitmapCache EnableClearType="False"

RenderAtScale="1"

SnapsToDevicePixels="False"/>

</Canvas.CacheMode>

Проще некуда, осталось только найти места в приложениях, где это может повысить производительность. Так же знакомимся с классом [BitmapCacheBrush](http://msdn.microsoft.com/en-us/library/system.windows.media.bitmapcachebrush(VS.100).aspx), который также должен нам значительно повысить производительность приложения при должном использовании. Он нам понадобиться в случае, если мы какой-то элемент отображаем несколько раз в одном окне. Например, это может быть иконка в DataGridView: теперь ее один раз можно отрендерить, сделать кеш при помощи [BitmapCache](http://msdn.microsoft.com/en-us/library/system.windows.media.bitmapcache(VS.100).aspx) в ресурсах, а затем просто продублировать уже закешированную картинку при помощи [BitmapCacheBrush](http://msdn.microsoft.com/en-us/library/system.windows.media.bitmapcachebrush(VS.100).aspx). Опять же, на MSDN есть хороший пример на эту тему: <http://msdn.microsoft.com/en-us/library/ee230085(VS.100).aspx>.

#### Layout Rounding

Следующим нововведением в WPF является [LayoutRounding](http://msdn.microsoft.com/en-us/library/system.windows.frameworkelement.uselayoutrounding(VS.100).aspx). Новое в WPF, но уже старое в Silverlight. Сложно ответить чем [UseLayoutRounding](http://msdn.microsoft.com/en-us/library/system.windows.frameworkelement.uselayoutrounding(VS.100).aspx) лучше [SnapsToDevicePixels](http://msdn.microsoft.com/en-us/library/system.windows.uielement.snapstodevicepixels(VS.100).aspx). Вообще идея двух этих свойств разная, но эффект иногда получается один и тот же, но в разработке лучше привязываться к Layout Rounding и использовать только его, он более интуитивно понятен. Более подробную информацию можно получить здесь: <http://blogs.msdn.com/b/text/archive/2009/08/27/layout-rounding.aspx>.

#### Новые функции для анимации

Дискретная, линейная и сплайновая анимация уже поддерживается предыдущими версиями WPF. WPF 4 представляет новый концепт «Easing Functions», которые позволяют разработчикам создавать различную анимацию. Например, пружинистые движения или добавить упреждение в анимацию. Функции упрощения (Easing Functions) определяют способ анимации от начала и до конца. Встроенные функции упрощения предоставляют спектр режимов в анимации: круговой, экспоненциальной, эластичной и пружинистой. Функции упрощения спроектированы легко расширяемыми, позволяя разработчикам создавать собственные. Благодаря данному нововведения, дизайнеры могут без усилий создавать плавную и органичную анимацию.

#### Усовершенствования двухмерной графики.

Теперь визуальное представление каждого элемента может быть радикально изменено посредством эффектов в духе PhotoShop — через построители текстур. Разработчики, которые желают манипулировать индивидуальными пикселями вручную, могут также генерировать и модифицировать изображения с помощью класса WriteableBitmap.

### Текст

#### CaretBrush и SelectionBrush

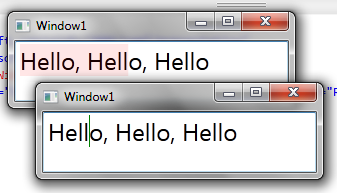
Теперь у каретки и для выделения можно задать кисти:

<TextBox FontSize="24" CaretBrush="Green" SelectionOpacity="0.1"   
 SelectionBrush="Red">

Hello, Hello, Hello

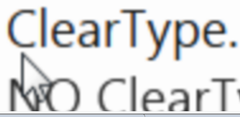
</TextBox>

И получить результат который изображен на рис. 4.3

  
Рисунок 4.3. Эффект применения CaretBrush и SelectionBrush.

#### ClearTypeHint

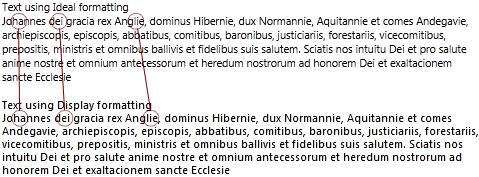
Очередная заклепка в WPF – свойство [RenderOptions.ClearTypeHint](http://msdn.microsoft.com/en-gb/library/system.windows.media.renderoptions.cleartypehint(VS.100).aspx). Идея в следующем, если в вашем визуальном дереве на каком то шаге используется, например, [Opacity](http://msdn.microsoft.com/ru-ru/library/system.windows.media.brush.opacity.aspx), то все визуальное дерево ниже уровнем будет прорисовывать текст без ClearType. Из-за этого текст будет выглядеть значительно иным (рис 4.4):

  
Рисунок 4.4. Эффект свойства [RenderOptions.ClearTypeHint](http://msdn.microsoft.com/en-gb/library/system.windows.media.renderoptions.cleartypehint(VS.100).aspx).

Другим примером может быть, если вы у Grid'а будете использовать свойство Clip. Свойством [RenderOptions.ClearTypeHint](http://msdn.microsoft.com/en-gb/library/system.windows.media.renderoptions.cleartypehint(VS.100).aspx) можно задать Enabled – для того, чтобы текст все таки прорисовался с ClearType (если он рисуется без Opacity). Хороший пример для ознакомления приводится тут: <http://blogs.msdn.com/b/llobo/archive/2009/10/28/new-wpf-features-cleartypehint.aspx>

#### TextFormattingMode и TextRenderingMode

Теперь мы можем действительно управлять тем, как будет отрисовываться текст. Для этого нам предоставили два свойства [TextOptions.TextFormattingMode](http://msdn.microsoft.com/en-us/library/system.windows.media.textoptions.textformattingmode(VS.100).aspx) и [TextOptions.TextRenderingMode](http://msdn.microsoft.com/en-us/library/system.windows.media.textoptions.textrenderingmode(VS.100).aspx). Первому можно установить *Ideal* – текст будет выглядеть как обычно в WPF, или *Display* – будет выглядеть как **текст GDI**. *Display* советуют применять только с мелким текстом, в остальных случаях оставить как есть. Разница в чтении мелкого текста заметна (рис. 4.5):

  
Рисунок 4.5. Демонстрация работы свойств [TextOptions.TextFormattingMode](http://msdn.microsoft.com/en-us/library/system.windows.media.textoptions.textformattingmode(VS.100).aspx)

Так же новый стек поддерживает явно выбираемые режимы прорисовки текста: aliased, grayscale, ClearType. Новый текстовый стек позволяет оптимизировать текстовую привязку для анимации и статики. Вдобавок, новый текстовый стек поддерживает шрифты с встроенными картами изображений, что позволяет многим восточно-азиатским шрифтам прорисовываться с чёткостью, к которой привыкли пользователи Win32.Новый текстовый стек позволяет оптимизировать текстовую привязку для анимации и статики. Вдобавок, новый текстовый стек поддерживает шрифты с встроенными картами изображений, что позволяет многим восточно-азиатским шрифтам прорисовываться с чёткостью, к которой привыкли пользователи Win32.

### XAML

#### Built in Types

В XAML добавилась возможность использовать стандартные типы .NET в разметке без указания лишних namespace, для чего это стало нужно видно в нижеописанных возможностях. Если раньше для описания в XAML объекта типа string приходилось писать:

<s:String xmlns:s="clr-namespace:System;assembly=mscorlib"> Foo </s:String>

Теперь же

<x:String > Foo </s:String>

#### Поддержка Generics

Этого давно не хватало, а именно с выпуска первого WPF. Как часто хотелось объявить в XAML ObservableCollection с типом вроде Person, а для этого нам приходилось создавать новый тип и наследовать его от коллекции, либо инкапсулировать ее:

class PersonCollection:ObservableCollection<Person>{}

и уже этот новый класс объявлять в ресурсах:

<l:PersonCollection>

<Person Name="Tom" />

</l:PersonCollection>

Теперь же этого можно избежать, теперь в XAML можно объявлять и инициализировать generics следующим образом:

<ObservableCollection   
 x:TypeArguments='local:Person'   
 xmlns='clr-namespace:System.Collections.ObjectModel;assembly=System' >  
 <local:Person Name='Tom' Age='21' />  
</ObservableCollection>

#### FactoryMethod\Arguments

Еще одна возможность, которую хотели уже давно. Возможность использовать методы для инициализации объектов, а так же конструкторы, которые принимают параметры. Для того чтобы создать объект такого типа:

public class Person

{

public Person (string name, int age)

{

Name = name;

Age = age;

}

public string Name {get;set;}

public int Age {get;set;}

}

В XAML теперь можно написать

<local:Person>

<x:Arguments>

<x:String>Tom</x:String>

<x:Int32>21</x:Int32>

</x:Arguments>

</local:Person>

Более того для создания объекта теперь можно использовать FactoryMethod, самый простой пример – это Guid.NewGuid():

<p:Guid x:FactoryMethod='NewGuid'/>

Или более сложный пример с передачей параметров:

<coll:List x:Key='list' x:TypeArguments='x:String'   
 x:FactoryMethod='local:Factory.CreateStringList'>

<x:Arguments>

<x:String>Mickey,Donald</x:String>

</x:Arguments>

</coll:List>

#### Named Object References

Тут все просто, такой биндинг:

<Label Target='{Binding ElementName=firstNameBox}' >\_Target</Label>

<TextBox Name='firstNameBox'>Uses Binding</TextBox>

Теперь можно записать при помощи x:Reference

<Label Target= '{x:Reference secondNameBox}'>\_Second Target</Label>

Или даже так:

<Label Target= 'thirdNameBox' >\_Third Target</Label>

#### Binding to Dynamic Objects

.NET 4 приходит вместе с dynamic, потому и в WPF добавили возможность биндить к свойствам dynamic объектов. Если мы установим такие свойства у объекта:

dynamic dynamicObj = BindPanel.DataContext ;

dynamicObj.A = "Simple Binding";

dynamicObj.B = new DynamicObjectClass();

dynamicObj.B.C = "Nested Prop Binding";

dynamicObj.AddItem("item 0");

dynamicObj[0] = "Indexer Binding";

То мы запросто можем установить такой биндинг:

<StackPanel Name="BindPanel" DataContext="{StaticResource MyDynamicObject}">

<TextBox Text="{Binding Path=A}"/>

<TextBlock Text="{Binding Path=B.C}" />

<TextBox Text="{Binding Path=[(x:Int32)0]}"/>

</StackPanel>

Эта возможность тоже даст пищу для размышлений. Теперь можно придумать о быстрой реализации биндинга к DataSet или XMLDocument.

#### Node Loop flexibility

Как известно в .NET 4 появилась отдельная библиотека System.Xaml, позволяющая нам обрабатывать Xaml разметку, считывать и превращать в граф объектов. Раньше у нас была возможность так же работать с Xaml разметкой при помощи XamlReader и XamlWriter, но все что они нам давали – это считывать и записывать Xaml, но не предоставляли нам возможность работать с ним. Теперь же набор классов для работы с Xaml пополнился, например, XamlXmlReader, который позволяет пробегаться еще и по элементам xaml дерева. Пример можно посмотреть здесь [http://blogs.msdn.com/b/llobo/archive/2009/11/09/  
xaml-2009-features-node-loop-flexibility.aspx](http://blogs.msdn.com/b/llobo/archive/2009/11/09/xaml-2009-features-node-loop-flexibility.aspx), в данном примере у считываемого Xaml файла заменяют родительский элемент Window на Page.

#### Key\Gesture Binding

Появилась возможность биндить Key и Modifiers в KeyBinding. Теперь можно создать свою DelegateCommand со свойствами

public Key GestureKey { get; set; }

public ModifierKeys GestureModifier { get; set; }

public MouseAction MouseGesture { get; set; }

Инициализировать команду следующим образом

public ICommand ExitCommand

{

get

{

if (exitCommand == null)

{

exitCommand = new DelegateCommand(Exit);

exitCommand.GestureKey = Key.X;

exitCommand.GestureModifier = ModifierKeys.Control;

exitCommand.MouseGesture = MouseAction.LeftDoubleClick;

}

return exitCommand;

}

}

А дальше забиндить клавиши:

<Window.InputBindings>

<KeyBinding Command="{Binding ExitCommand}"   
 Key="{Binding ExitCommand.GestureKey}"  
 Modifiers="{Binding ExitCommand.GestureModifier}"/>

</Window.InputBindings>

Без необходимости объявления ресурсов.

## Ключевые термины

**GDI** — это интерфейс Windows для представления графических объектов и передачи их на устройства отображения, такие как мониторы и принтеры.

**IDE** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Integrated Development Environment) — система программных средств, используемая программистами для разработки программного обеспечения.

**IntelliSense** — технология [автодополнения](http://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%B4%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) [Microsoft](http://ru.wikipedia.org/wiki/Microsoft), наиболее известная в [Microsoft Visual Studio](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio). Дописывает название функции при вводе начальных букв. Кроме прямого назначения IntelliSense используется для доступа к документации и для устранения неоднозначности в именах [переменных](http://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), [функций](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [методов](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), используя [рефлексию](http://ru.wikipedia.org/wiki/%D0%9E%D1%82%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

**Windows Communication Foundation** (WCF) — программный [фреймворк](http://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA), используемый для обмена данными между приложениями входящими в состав [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework). До своего выпуска в декабре 2006 года в составе .NET Framework 3.0, WCF был известен под кодовым именем Indigo.

**Windows Workflow Foundation** (WF) представляет собой технологию компании [Microsoft](http://ru.wikipedia.org/wiki/Microsoft) для определения, выполнения и управления [рабочими процессами](http://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D0%B1%D0%BE%D1%87%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81&action=edit&redlink=1) ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) workflow). Данная технология входит в состав [.NET Framework 3.0](http://ru.wikipedia.org/wiki/.NET_Framework_3.0), который изначально установлен в [Windows Vista](http://ru.wikipedia.org/wiki/Windows_Vista) и может быть установлен в [Windows 2003 Server](http://ru.wikipedia.org/wiki/Windows_2003_Server)и [Windows XP](http://ru.wikipedia.org/wiki/Windows_XP) SP2. WF ориентирована на визуальное программирование и использует [декларативную модель программирования](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

**Ше́йдер** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Shader) — это [программа](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) для одной из ступеней [графического конвейера](http://ru.wikipedia.org/w/index.php?title=%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80&action=edit&redlink=1), используемая в [трёхмерной графике](http://ru.wikipedia.org/wiki/%D0%A2%D1%80%D1%91%D1%85%D0%BC%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0) для определения окончательных параметров объекта или изображения. Она может включать в себя произвольной сложности описание поглощения и рассеяния света, наложения [текстуры](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D1%83%D1%80%D0%B0_(%D1%82%D1%80%D1%91%D1%85%D0%BC%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0)), отражение и преломление, затенение, смещение поверхности и эффекты пост-обработки.

## Краткие итоги

Как видно из вышеизложенного, появилось масса нововведений в WPF 4.0. Можно сказать, что WPF — это одно из ведущих направлений Windows-приложений. Со временем WPF превратится в систему, подобную User32 и GDI/GDI+, к которой будут добавляться новые расширения и высокоуровневые средства. В конечном итоге WPF позволит проектировать приложения, которые было бы невозможно (или, по крайней мере, непрактично) построить средствами Windows Forms.

## Набор для практики

**Вопросы:**

1. Назовите основные нововведения Visual Studio 2010 касающиеся разработки WPF/Silverlight приложений.
2. Перчислите новые средства WPF 4.
3. Назначение *клиентского профиля.*
4. Назовите новые и доработанные графикические возможности WPF 4
5. Ключевые улучшения WPF 4 для XAML

## Литература

1. Chipalo Street рассказывает, что нового и какие улучшения в WPF 4 Text <http://channel9.msdn.com/blogs/jaime+rodriguez/new-text-stack-in-wpf-4>
2. Jossef Goldberg углубляется в новую версию Client Profile <http://channel9.msdn.com/blogs/jaime+rodriguez/net-4-client-profile>
3. Channel 9: Пример для работы с графикой в WPF 4 от David Teitlebaum

<http://channel9.msdn.com/blogs/jaime+rodriguez/graphics-improvements-in-wpf-4>

1. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 40 – 45.

# Лекция 5. Silverlight и WPF в Visual Studio 2010 (продолжение).

**Краткая аннотация лекции:** освещаются новые возможности Silverlight 4. Рассматриваются новые возможности Silverlight версии 5

**Цель лекции:** показать историю развития платформы Silverlight и дать понимание дальнейшей тенденции. Изложить структуру и возможности Silverlight 4 по разработке интерактивных интернет-приложений нового поколения. Продемонстрировать новые возможности Silverlight 5 реализованные в версии Release Candidate.

## Новые возможности Silverlight 4. Silverlight 5: основные нововведения ожидаемого релиза.

## Хронология Silverlight

В настоящее время доступны четыре версии Microsoft Silverlight: Silverlight 1, Silverlight 2, Silverlight 3, и Silverlight 4.

**Silverlight 1.** Это выпуск Silverlight не имел общеязыковой среды выполнения. Он поддерживал небольшое подмножество XAML и множество возможностей, которые предвещали будущее Silverlight. Возможно самый очевидный аспект Silverlight 1 – то, что приложения написаны или полностью на XAML или в смеси с JavaScript и объектной моделью документов, чтобы управлять интерфейсом пользователя. Так как нет общеязыковой среды выполнения, нет фазы компиляции, и JavaScript интерпретируется на клиентской стороне.

**Silverlight 2.** Вскоре после того, как Silverlight 1 был выпущен предварительный релиз следующей версии Silverlight. Этот выпуск был известен как Silverlight 1.1, наиболее значительным аспектом которого является кроссплатформенная общеязыковая среда выполнения. Хотя Silverlight 1 мог использоваться, чтобы разработать некоторые впечатляющие и «богатые» медиа-приложения, его возможности значительно расширялись со способностью выполняться на платформе .NET. Самая большая недостающая возможность Silverlight 1.1 – это недостаточный набор стандартных элементов управления. Это делало разработку пользовательских интерфейсов трудной. Обработка входных событий была также трудной, так как события могли перехватываться только в корневом контейнере. Затем вручную приходилось распространять события к дочерним объектам. Использование фокуса ввода было также запутанным. В марте 2008, Microsoft объявила, что Silverlight 1.1 будет фактически выпущен как Silverlight 2, так как набор его возможностей значительно расширился.

**Silverlight 3.** В марте 2009 Microsoft выпустила бета-версию Silverlight 3. В июле 2009 появилась среда разработки Expression Blend 3. Silverlight 3 является расширением Silverlight 2 и главным образом обеспечивает усовершенствование возможностей графики, управления медиа, области разработки приложений (дополнительные элементы управления, усовершенствованная поддержка связывания элементов управления с данными, поддержка внебраузерных приложений), а также интеграцию с Expression Blend.

**Silverlight 4.** Microsoft главным образом фокусировалась на возможностях, управляемых медиа до версии Silverlight 3. Silverlight 3 представил ключевые возможности интеграции данных для легкой разработки приложений, управляемых данными. Однако если необходимо разработать приложение, управляемое данными, с функциональными возможностями, такими как печать, интеграция документов, создание отчетов, «богатые» офлайновые возможности и интеграция с локальными устройствами, то надо начинать с Silverlight 4.

Бета-версия Silverlight 4 была быстро выпущена в ноябре 2009, а окончательный релиз Silverlight 4 появился 12 апреля 2010 во время конференции DevConnection. Детальный разбор архитектура программной платформы Silverlight 4 был приведен во второй лекции.

Silverlight 4 имеет следующие ключевые особенности:

1. Возможность печати контента с использованием нового API печати, позволяет разработчикам создавать пользовательские визуальные представления печати, которые могут быть интегрированы с локальными принтерами или могут быть сохранены как файлы (XPS, PDF).
2. Поддержка контекстных меню является возможностью по умолчанию, обеспечивая интерфейс пользователя подобный имеющемуся в настольных приложениях.
3. Поддержка Drag and Drop и операций с буфером обмена допускают интеграции локальных файлов или данные с RIA Silverlight.
4. Новые элементы управления, такие как RichTextBox, который позволяет разработчикам предоставлять область редактирования текста с поддержкой текстового форматирования, гиперссылок и изображений в приложениях Silverlight. Новые элементы управления WebBrowser и WebBrowser-Brush позволяют интегрировать HTML-контент в приложения, которые являются внебраузерными.
5. Расширение возможностей существующих элементов управления, таких как DataGrid поддержкой сортировки и изменения размеров, а также операций с буфером обмена.
6. Улучшенная привязка к данным
7. Улучшенные возможности локализации добавляют поддержку для двунаправленного текста, языков, на которых пишут справа-налево и дополнительных 30 новых языков.
8. Введение окон уведомлений для обеспечения основанных на панели задач традиционных уведомлений на клиентской машине, улучшит интерфейс конечного пользователя, обеспечивая непротиворечивый подход, как многие другие традиционные приложения.
9. Поддержка клавиатурного доступа в полноэкранном режиме и простой реализации поддержки колесика мыши обеспечивает обработку события MouseWheel и соответственно значительное улучшение удобства и простоты использования приложений.
10. Возможность интеграции с Web-камерой и микрофоном позволяет разрабатывать интерактивные корпоративные и клиентские приложения, которые поддерживают голосовое взаимодействие и проведение видео-конференций.

Возможность для приложений Silverlight быть "доверенными" так же как внебраузерные приложения открывает многие возможности, которые обеспечивают традиционные WPF-приложения, которые могут вызывать «родной» код за пределами «песочницы» на клиентской машине:

1. Поддержка позднего связывания с объектами, полученными от HTML DOM или от Automation API. Это обеспечивает возможность интеграции с COM-приложениями, такими как приложения Microsoft Office (например, Word, Excel, и Outlook) на Windows-клиентах и интеграцию других подключенных устройств.
2. Доступ к Папкам “Мои…” из приложения позволяет доступ ко всем папкам «Мои документы», «Мои видеозаписи», «Изображения», и «Музыка» для Windows и Mac и читать и записывать файлы из приложения, работающего в среде «песочницы».
3. Легкое развертывание и управление, избавляющее от необходимости использования междоменных файлов политик доступа (ClientAccess.xml или CrossDomainAcess.xml), а также возможность создания групповых политик для управления доверенными приложениями.

Silverlight 4 действительно включает несколько ключевых расширений, чтобы защитить, обработать и доставить более дружественные с медиа RIA:

1. Поддержка офлайнового управления цифровыми правами поможет доставить медиа в офлайновом режиме в более защищенной и управляемой среде.
2. Возможности интеграции с Web-камерой и микрофоном и возможность записи аудио и видео клиентской стороны, делающая, обеспечивает многочисленные возможности для разработки интерактивных аудио/видео RIA.

Следующие усовершенствования основной платформы и компонентов поможет улучшить интерфейс конечного пользователя и разрабатывать корпоративные приложения в режиме быстрой разработки.

1. Официальная поддержка web-браузера Google Chrome - это вероятно конечный шаг к провозглашению Silverlight кроссбраузерной платформой и конечно осчастливит фанатов Google.
2. Введение поддержки широковещательного сетевого взаимодействия по протоколу UDP поможет улучшить потребительские свойства и стабильность приложений, более эффективно используя сетевые ресурсы.
3. WCF-службы RIA помогают разрабатывать многозвенные приложения в режиме быстрой разработки.
4. Полная оптимизация производительности улучшает производительность запуск и выполнения приложений Silverlight 4 по сравнению с приложениями Silverlight 3.
5. Visual Studio 2010 позволяет разработку пользовательского интерфейса для приложений Silverlight 4 (и Silverlight 3) и вводит улучшенную привязку данных и интеграцию WCF-служб RIA, а также другие расширения, чтобы улучшить опыт разработки.
6. Silverlight 4 и инструментальные средства разработки Windows Phone допускают разработку интерактивных приложений Silverlight для мобильного телефонов серии Windows 7. Вы можете разрабатывать высококачественные медиа-приложения (включая интеграцию с видео камерой и микрофоном) для игровых приложений для мобильных телефонов Windows.

## Обзор Silverlight 5

Рассмотрим ожидаемые возможности релиза.

### Поддержка 64-битной платформы

Даже Silverlight 4 не работает в 64-битных браузерах. Между тем, 32-бита уверенно уходят в прошлое. Именно поэтому, начиная с 5-й версии Silverlight, плагин будет доступен как в 32-битной, так и в 64-битной версиях. Предполагается, что пользователю не придется думать, какая из версий должна быть установлена.

### Улучшенное управление питанием

Речь идет о том, что при просмотре видео с помощью Silverlight (а, следовательно, при отсутствии взаимодействия с мышью и клавиатурой), Вас не застанет врасплох установленный на машине Screensaver и Вам не придется бросаться к мыши, чтобы продолжить смотреть видео.

### Копирование изображений

В Silverlight 4 появился класс Clipboard, который позволил копировать текст в буфер и обратно. Silverlight 5 расширяет возможности этого класса, позволяя работать с изображениями. Методы SetImage и GetImage будут работать с объектами типа BitmapSource. Также предусмотрено наличие метода ContainsImage, проверяющего наличие изображения в буфере.

### Поддержка 3D

Речь идет не о технологии псевдо 3D, появившейся в Silverlight 4, а о полноценном 3D API с поддержкой аппаратного ускорения. Последнее уже хорошо зарекомендовало себя при работе с видео.

### Поддержка Behaviors и Triggers

Поддержка Behaviors призвана перенести функциональность Поведений, реализованных в Blend на уровень ядра Silverlight. Речь идет о создании комплексных объектов, объединяющих в себе ряд различных преобразований, создавая тем самым уникальное Поведение. И если ранее подобное можно было реализовать, только подключив специальную сборку из Blend, то теперь это все будет в ядре. Введение триггеров призвано еще больше сблизить WPF и Silverlight, хотя триггера в Silverlight выглядят более универсальными.

### Layout transition

В Silverlight 5 любому контейнеру будет можно задавать эффекты, связанные с заполнением контейнера и добавлением нового элемента. Благодаря Layout transition разработчику не придется работать с каждым элементом отдельно, а достаточно будет указать механизм заполнения для всего контейнера.

### Связывание с данными

#### Отладка

У разработчиков теперь будет меньше головной боли, так как появилась возможность ставить точки останова в коде, описывающим связывание данных. Достаточно поставить точку останова в XAML файле, на строке, описывающей Binding, и разработчик может получить всю возможную информацию о связывании и проблемах, возникающих с этим. Поэтому найти такие ошибки, как неправильное написание имени свойства или имени преобразователя типов, станет теперь проще.

#### FindAncestor

FindAncestor позволяет выполнять связку. Выполняя поиск любого вышестоящего элемента и выполнять с ним связывание.

#### DataContextChanged

Как видно, Silverlight 5 все ближе и ближе приближается к WPF. Новое событие DataContextChanged позволит определить изменение источника данных при связывании.

#### Стили

Связывание теперь можно будет использовать в Setter элементах стилей. Это позволит быстро менять внешний вид окна, задавая различные темы.

### Поддержка Full-Trust внутри браузера

Silverlight 5 будет поддерживать правила, установленные групповыми политиками. Поддержка групповых политик позволит централизовано управлять развертыванием Silverlight-приложений. Кроме того, это даст возможность устанавливать список Publishers, разрешая избранным приложениям работать внутри браузера, но с использованием Full-Trust со всеми вытекающими последствиями (включая поддержку элемента WebBrowser). Это нововведение связано с отзывом от некоторых компаний, которые хотят запускать свои корпоративные приложения внутри браузера, но с повышенными привилегиями.

### Обработка множественного нажатия кнопки мыши

Обычно требуется реализовать обработчик события при двойном щелчке, но Silverlight позволяет обрабатывать и большее количество нажатий. Речь идет как о левой, так и о правой кнопках мыши. Тут разработчики немного схитрили. Чтобы не менять модель событий и не вводить новые события на двойной щелчок мыши, к параметру EventArgs уже существующих событий, возникающих при однократном нажатии кнопки мыши, было добавлено свойство ClickCount. Это свойство и содержит количество нажатий.

Интересно то, что события MouseLeftButtonDown и MouseRightButtonDown срабатывают для каждого щелчка, даже если вы хотите обрабатывать только двойной щелчок. При этом, если между щелчками проходит больше секунды, или другое событие (например, пользователь еще и потянул мышь), то счетчик обнуляется. Поэтому, если Вы хотите для одного и того же объекта обрабатывать одинарный и двойной (а может и тройной) щелчок, то необходимо, чтобы они не были взаимоисключающими. Иными словами, при одинарном щелчке Вы можете выбрать папку, а при двойном – открыть папку. Наоборот работать не будет, так как уже при первом щелчке папка будет открыта.

Ниже показан простой код, который позволяет отобразить количество щелчков:

<Grid x:Name="LayoutRoot" Background="White"   
 MouseLeftButtonDown="LayoutRoot\_MouseLeftButtonDown">  
 <sdk:Label Height="28" HorizontalAlignment="Left"   
 Name="label1" VerticalAlignment="Top" Width="120" />  
 </Grid>

### Поддержка выбора элементов в ItemsControl с клавиатуры

Данный функционал достаточно простой, но приятный. Вспомните бесконечные элементы ComboBox со списком стран, или элементы, позволяющие установить фильтр по чьей-то фамилии, выбрав ее из сотни других и т. д. Для всех этих случаев, имея элемент-наследник от ItemsControl, такой как ListBox, ComboBox и другие, Вы всегда должны были пользоваться скроллингом, чтобы найти нужный элемент (это утверждение относится к Silverlight). В WPF или даже в простом HTML, подобные элементы давно поддерживают возможность выбора с клавиатуры. Так, переходя на тот же выпадающий список стран, мне достаточно легко набрать на клавиатуре букву «Р», чтобы увидеть активным элементом слово «Россия». Это позволяет быстро заполнять сложные формы с большим количеством опций. Теперь такая возможность появилась и в Silverlight. Рассмотрим простой пример:

<Grid x:Name="LayoutRoot" Background="White">  
 <ComboBox Height="23" HorizontalAlignment="Left"   
 Name="comboBox1" VerticalAlignment="Top" Width="120">  
 <ComboBoxItem Content="Последний" />  
 <ComboBoxItem Content="Третий" />  
 <ComboBoxItem Content="Второй" />  
 <ComboBoxItem Content="Первый" />  
 </ComboBox>  
 </Grid>

На экране отобразится обычный ComboBox, который позволяет выбрать один из четырех элементов. Переключитесь на русскую клавиатуру и попробуйте понажимать символ «П». ComboBox будет последовательно выбирать один из двух элементов (Первый и Последний). Если Вы последовательно наберете два символа, например «Пе», то будет выбран «Первый» без альтернатив.

Понятно, что ComboBox в Silverlight, как и другие ItemsControl, позволяет отображать что угодно и не обязательно текст, особенно если речь идет о связывании со сложными объектами. Именно поэтому, для всех элементов ItemsControl ввели дополнительное свойство TextPath, которое задает имя свойства объекта, с которым мы реализуем связывание, используемое при выборе элемента с клавиатуры.

<ComboBox Height="23" TextSearch.TextPath="ImageTitle"   
 HorizontalAlignment="Left" Name="comboBox1"   
 VerticalAlignment="Top" Width="120">

### Повышение полномочий для приложений в браузере

В предыдущих версиях Silverlight, приложения, которые получали возможность работать вне браузера, могли претендовать на дополнительные (повышенные) привилегии. Это позволяло подобным приложениям получать доступ к диску, расширять изолированное хранилище без уведомления пользователя, реализовывать доступ к клавиатуре в полноэкранном режиме и многое другое. Между тем, многие разработчики желали бы иметь подобную функциональность и для приложений, работающих в браузере. Естественно речь идет о корпоративных приложениях, которые доступны только сотрудникам филиалов и при этом могут часто обновляться. В связи с этим в Silverlight 5 появилась возможность создавать приложения с повышенными полномочиями, работающими в окне браузера. Рассмотрим процедуру создания таких приложений.

#### Шаг 1.

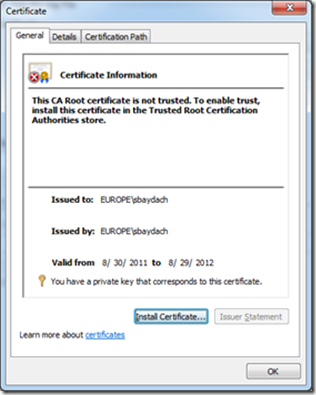
Приложение необходимо подписать сертификатом, что вполне можно сделать из оболочки Visual Studio в настройках проекта (рис. 5.1).

  
Рисунок 5.1. Вкладка Signing

Тут вы можете сгенерировать тестовый сертификат или выбрать его из хранилища (файла). Теоретически, если вы создаете приложение для работы внутри своей сети, сертификат можно и не покупать, а пользоваться тестовым.

#### Шаг 2.

Далее необходимо развернуть сертификат на машинах, где будет работать ваше Silverlight-приложение. Если вы хотите протестировать функциональность на рабочей машине, то это можно легко сделать, нажав кнопку More Details… в диалоге выше, перейдя к параметрам сертификата (рис 5.2):

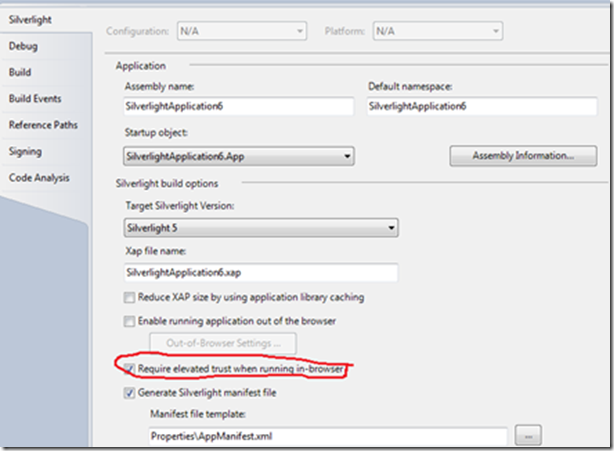
  
Рисунок 5.2. Вкладка General

#### Шаг 3.

На следующем этапе нужно у всех клиентов разрешить запуск приложений в браузере с повышенными полномочиями. Это можно сделать, установив значение AllowElevatedTrustAppsInBrowser (DWORD) в 0х00000000 (disabled) или 0×00000001 (enabled) в ключах HKEY\_LOCAL\_MACHINE\Software\Microsoft\Silverlight\ (для 32 бит) или HKEY\_LOCAL\_MACHINE\Software\Wow6432Node\Microsoft\Silverlight\ (для 64 бит).

#### Шаг 4.

Необходимо внести изменения в конфигурационный файл таким образом, чтобы ваше приложение при работе с браузером запрашивало дополнительные полномочия. Это легко можно сделать в настройках проекта, выбрав соответствующую галочку (рис. 5.3):

  
Рисунок 5.3. Вкладка Silverlight

После этих действий, Visual Studio сгенерирует дополнительный конфигурационный файл следующего содержимого:

<InBrowserSettings>  
 <InBrowserSettings.SecuritySettings>  
 <SecuritySettings ElevatedPermissions="Required" />  
 </InBrowserSettings.SecuritySettings>  
 </InBrowserSettings>

Теперь ваши приложения (подписанные сертификатом), будут запускаться в браузере с дополнительными полномочиями, включая полную поддержку клавиатуры в полноэкранном режиме, поддержку элементов WebBrowser и NotificationWindow и др.

Хочется отметить, что если вы попробуете тестировать эту процедуру, запуская приложение через localhost, то первые три шага игнорируются. Поэтому не попадите в заблуждение: если на вашей машине работает, то будет работать и на других.

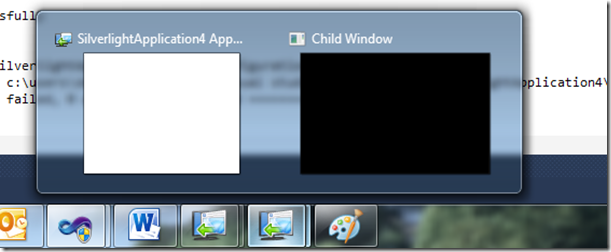
### Поддержка нескольких окон

Для приложений с повышенными привилегиями, работающих вне браузера, стало возможным отображать несколько немодальных окон, которые могут перемещаться за пределы родительского окна, а также интегрируются с панелью задач Windows 7.

Ниже приведен небольшой код, который, внутри конструктора главного окна, генерирует дочернее окно и делает его видимым:

public MainPage()  
 {  
 InitializeComponent();  
   
 Window childWindow = new Window();  
 childWindow.Height = 400;  
 childWindow.Width = 600;  
 childWindow.Top = 30;  
 childWindow.Left = 30;  
 childWindow.Title = "Child Window";  
 childWindow.Visibility = Visibility.Visible;  
 }

В данном примере дочернее окно создается с размерами 600 на 400, при этом верхний левый его угол будет находит в позиции (30, 30), относительно экрана системы (не родительского окна). При этом обратите внимание на то, что созданное окно поддерживается панелью задач Windows 7 (рис. 5.4):

  
Рисунок 5.3. Поддержка нескольких окон в Silverlight 5.

### Доступ к файловой системе для приложений с повышенными привилегиями

Технология Silverlight 4 поддерживала возможность приложениям с повышенными привилегиями получать доступ к файловой системе пользователя. Правда, доступ был ограничен лишь несколькими папками. Так, в режиме с повышенными привилегиями, приложения получают полный контроль к специальным папкам, ассоциированными с пользователем: MyDocuments, MyVideos, MyPictures, MyMusic. Пути к указанным папкам можно получить, используя статический класс Environment. При этом нужно отметить, что SilverLight-приложение работает только с папками, но не с библиотеками (Windows 7).

В Silverlight 5 появилась возможность взаимодействовать со всей файловой системой пользователя. Речь идет как о чтении файлов, так и о записи.

### Установка имени файла по умолчанию в SaveFileDialog

Теперь при вызове этого диалогового окна, можно установить имя файла (как и список расширений), которое будет отображаться по умолчанию, как имя файла, предлагаемого для записи. Вот как это выглядит:

SaveFileDialog dialog = new SaveFileDialog();  
dialog.DefaultFileName = "hello.txt";  
dialog.ShowDialog();

Кроме описанной возможности, появилось свойство InitialDirectory, позволяющее задать директорию, которая будет открыта в диалоге по умолчанию.

### Улучшение работы с аудио

В Silverlight 5 стала доступна библиотека по работе с Media. Классы этой библиотеки представлены пространством имен Microsoft.Xna.Framework.Audio. Нетрудно догадаться, что данный шаг выполнен в рамках интеграции с XNA библиотеками, которые можно использовать и в Silverlight (это аудио и 3D).

Новое пространство имен содержит два основных класса, позволяющих создавать аудио эффекты в процессе работы Silverlight приложения. Файлы, доступные для проигрывания, должны быть в формате WAV и могут быть загружены с помощью класса SoundEffect. Конструктор этого класса получает ссылку на набор данных и (или) настройки аудио. Создание конкретного экземпляра можно также выполнить с помощью статического метода FromStream, получающего ссылку на поток в качестве параметров. Проигрывание эффекта осуществляется с помощью метода Play. Каких-то других методов SoundEffect не имеет, но это вовсе не означает, что отдельно взятым объектом нельзя управлять. Так, если вы хотите не просто воспроизвести эффект, но и добавить элементы управления аудио (громкость, пауза и т .д.), то вместо метода Play следует вызывать метод CreateInstance, создающий конкретный экземпляр на базе SoundEffect, и возвращающий ссылку типа SoundEffectInstance, которая позволит управлять нашим аудио файлом.

SoundEffect sEffect=SoundEffect.FromStream(streamInfo.Stream);  
sEffect.Play();

### Изменение скорости проигрывания медиа

В Silverlight 5 произошли улучшения и в MediaElement элементе. Теперь этот элемент управления содержит такое свойство как PlaybackRate и событие RateChanged. Свойство позволяет задавать скорость проигрывания, синхронизировать видео и звук, а также выполнять «переметку» не только вперед, но и назад. Предполагается, что можно будет использовать следующие множители для скорости проигрывания: 0.2, 0.4, 0.6, 0.8, 1, 2, 4, 8, 16, 32. Событие RateChanged позволит реагировать на изменение свойства.

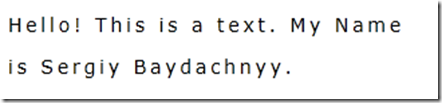
### Поддержка пульта дистанционного управления

Silverlight 5 поддерживает и работу с пультом. Данная функциональность стала доступна благодаря поддержке события MediaCommand, которое доступно у всех UIElement. Иными словами, если вы хотите обрабатывать нажатия кнопок пульта, то Вам следует обработать это событие у родительского контейнера (от дочерних элементов оно как раз туда и свалится). Чтобы понять, какая кнопка на пульте была нажата, обработчик события получает доступ к свойству в MediaCommandEventArgs, содержащему свойство перечислимого типа MediaCommand. Последнее и содержит информацию по нажатой кнопке.

### Работа с текстом

Silverlight 5 предлагает несколько интересных возможностей при работе с текстом. Так, элементы управления Control, TextBlock и TextElement, имеют новое свойство CharacterSpacing. Это свойство позволяет установить расстояние между символами внутри текста.

Свойство LineHeight позволяет установить ширину строки (можно интерпретировать, как расстояние между строками) и определено в таких элементах как Block, TextBlock, TextBox и RichTextBox. Для элемента RichTextBox можно также задать стратегию с помощью свойства LineStackingStrategy. На рис. 5.4 пример текста внутри элемента TextBlock с расстоянием между строками 30 и расстоянием между символами – 200:

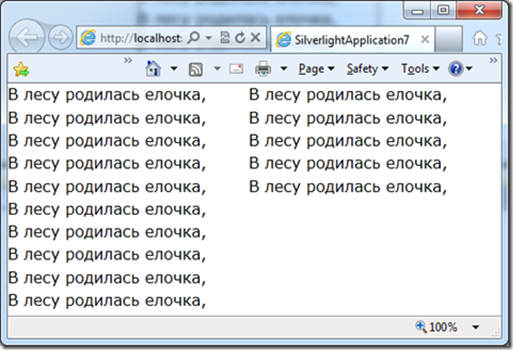
[](http://baydachnyy.files.wordpress.com/2011/09/image1.png)  
Рисунок 5.4. Пример возможности TextBlock в Silverlight 5.

Наконец, Silverlight 5 стал поддерживать два новых текстовых элемента, это RichTextBlock и RichTextBlockOverflow. Эти элементы аналогичны элементам RichTextBox и RichTextBoxOverflow, но позволяют отображать текст только на чтение.

Рассмотрим небольшой пример с элементом RichTextBlockOverflow:

<StackPanel x:Name="LayoutRoot" Background="White"  
 Orientation="Horizontal" VerticalAlignment="Top">  
 <RichTextBlock  
 Width="250" FontSize="16" OverflowContentTarget=  
 "{Binding ElementName=SecondBox}">  
 <Paragraph>  
 В лесу родилась елочка,  
 </Paragraph>  
 . . . . . . . . //повторить 20 разJ  
 </RichTextBlock>  
   
 <RichTextBlockOverflow Width="250" Name="SecondBox">  
 </RichTextBlockOverflow>  
 </StackPanel>

Результат работы этого кода показан на рис. 5.5:

[](http://baydachnyy.files.wordpress.com/2011/09/image2.png)  
Рисунок 5.5. Пример с элементом RichTextBlockOverflow в Silverlight 5.

Как видно, RichTextBlockOverflow может отображать текст, который не «вмещается» в основной элемент RichTextBlock. Благодаря таким элементам мы можем размещать текст в 2, 3 и более колонках (или реализовывать более сложные сценарии).

### Печать

Silverlight 5 позволяет теперь не только реализовать растровую, но и векторную печать. Для этих целей используется все тот же метод Print класса PrintDocument, который осуществляет попытку печати в векторном формате, а в случае неудачи (принтер не поддерживает), перейти к печати в растровом виде. Растровую печать можно инициировать с помощью нового метода PrintBitmap.

### P/Invoke

В документации эта возможность пока не описана, но она уже реализована – возможность вызова методов Windows API из Silverlight приложений, обладающих повышенными полномочиями. Причем речь идет о приложениях, которые работают как в браузере, так и вне браузера. Поскольку вызов нативных функций в .NET обычно осуществляется с помощью атрибута DllImport. Запустим приложение с повышенными привилегиями:

public partial class MainPage : UserControl  
 {  
 [DllImport("user32.dll", CharSet = CharSet.Unicode)]  
 public static extern int MessageBox(IntPtr hWnd,   
 String text, String caption, uint type);  
   
 public MainPage()  
 {  
 MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);  
 InitializeComponent();  
 }  
 }

На экране, поверх браузера вы увидите стандартный MessageBox.

### Отладка при связывании с данными

Современный отладчик в Visual Studio давно позволяет разработчику получать всю необходимую информацию, если речь идет о коде на C#, C++ или даже JavaScript. Но как только речь заходит о XAML, то тут механизмы отсутствуют. Казалось бы, зачем нужна отладка в XAML, если тут идет декларативное описание интерфейса приложения. Действительно, отладка в XAML не нужна, если речь не идет о связывании с данными. Небольшой кусок кода, связывающий наши данные и интерфейс, может вызвать массу проблем при отладке. Ведь причин для возникновения проблем при связывании может быть множество, это и отсутствие какого-либо свойства, и проблемы с преобразованием либо же несоответствие типа. Но если связывание с данными описывается в XAML, то механизма получить информацию о проблеме не было. В Silverlight 5 возможна отладка XAML кода, описывающего связывание элементов управления и данных.

Рассмотрим небольшой пример кода на XAML:

<UserControl x:Class="SilverlightApplication1.MainPage"  
 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>  
 Loaded="UserControl\_Loaded"  
 mc:Ignorable="d"  
 d:DesignHeight="300" d:DesignWidth="400">  
  
 <Grid x:Name="LayoutRoot" Background="White">  
 <Grid.Resources>  
 <Style TargetType="TextBox">  
 <Setter Property="Background" Value="AliceBlue"></Setter>  
 <Setter Property="Width" Value="200"></Setter>  
 <Setter Property="Margin" Value="5"></Setter>  
 </Style>  
 <Style TargetType="TextBlock">  
 <Setter Property="Margin" Value="5"></Setter>  
 </Style>  
 </Grid.Resources>  
 <Grid.RowDefinitions>  
 <RowDefinition Height="Auto"></RowDefinition>  
 <RowDefinition Height="Auto"></RowDefinition>  
 <RowDefinition Height="Auto"></RowDefinition>  
 </Grid.RowDefinitions>  
 <Grid.ColumnDefinitions>  
 <ColumnDefinition Width="Auto"></ColumnDefinition>  
 <ColumnDefinition Width="Auto"></ColumnDefinition>  
 </Grid.ColumnDefinitions>  
 <TextBlock Text="First Name:" Grid.Column="0" Grid.Row="0"/>   
 <TextBlock Text="First Name:" Grid.Column="0" Grid.Row="1"/>  
 <TextBlock Text="First Name:" Grid.Column="0" Grid.Row="2"/>  
 <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding FisrtName}"/>  
 <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding LastName}"/>  
 <TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Age}"/>  
 </Grid>  
 </UserControl>

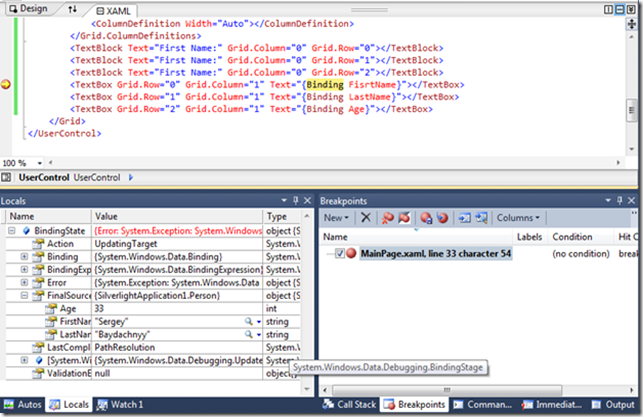
В результате мы получим небольшую формочку, которая позволяет заполнить три поля. Код, необходимый для работы формы может выглядеть так:

private void UserControl\_Loaded(object sender, RoutedEventArgs e)  
{  
 Person p = new Person();  
 p.FirstName = "Sergey";  
 p.LastName="Baydachnyy";  
 p.Age = 33;  
 LayoutRoot.DataContext = p;  
}

И класс Person:

public class Person  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public int Age { get; set; }  
}

Запустив это приложение, можно убедиться, что первое поле не было заполнено данными. Чтобы понять причину ошибки, достаточно установить **Breakpoint** в XAML файле на строку, описывающую связывание в первом поле и запустить приложение в режиме отладки. Вот что можно увидеть в окне **Locals**:

  
Рисунок 5.6. Отладка XAML в Silverlight 5.

Тут и информация об ошибке (ошибка намеренно сделана в имени поля) и данные об объекте и т. д. То есть, сгенерированный объект **BindingState** четко позволяет выявить проблему и проверить правильность данных.

### Улучшенная поддержка шаблона MVVM

Данная поддержка состоит в расширении перечислимого типа **UpdateSourceTrigger** значением **PropertyChanged**. Данное значение влияет на обновление источника данных, происходящее при обновлении цели (элемента, куда данные выгружаются). Ранее этот тип поддерживал лишь два значения, это **LostFocus** и **Explicit**. Иными словами, если Вы связываете данные с элементом управления TextBox, то в случае двунаправленного связывания, при изменении элемента TextBox, источник мог быть обновлен лишь при потере фокуса или явным образом. Значение **PropertyChanged** позволяет реагировать на любые изменения в текстовом поле «на лету». При этом нет необходимости обрабатывать нажатия клавиш и т .д. Вот пример кода:

1: <UserControl x:Class="SilverlightApplication1.MainPage"  
 2: xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
 3: xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
 4: xmlns:d=<http://schemas.microsoft.com/expression/blend/2008>  
 5: xmlns:mc=<http://schemas.openxmlformats.org/markup-ompatibility/2006>  
 6: Loaded="UserControl\_Loaded"  
 7: mc:Ignorable="d"  
 8: d:DesignHeight="300" d:DesignWidth="400">  
 9:    
 10: <Grid x:Name="LayoutRoot" Background="White">  
 11: <Grid.Resources>  
 12: <Style TargetType="TextBox">  
 13: <Setter Property="Background" Value="AliceBlue"/>  
 14: <Setter Property="Width" Value="200"/>  
 15: <Setter Property="Margin" Value="5"/>  
 16: </Style>  
 17: <Style TargetType="TextBlock">  
 18: <Setter Property="Margin" Value="5"/>  
 19: </Style>  
 20: </Grid.Resources>  
 21: <Grid.RowDefinitions>  
 22: <RowDefinition Height="Auto"></RowDefinition>  
 23: <RowDefinition Height="Auto"></RowDefinition>  
 24: </Grid.RowDefinitions>  
 25: <Grid.ColumnDefinitions>  
 26: <ColumnDefinition Width="Auto"></ColumnDefinition>  
 27: <ColumnDefinition Width="Auto"></ColumnDefinition>  
 28: </Grid.ColumnDefinitions>  
 29:    
 30: <TextBlock Text="First Name:" Grid.Column="0" Grid.Row="0"/>  
 31: <TextBox Grid.Row="0" Grid.Column="1" Text=  
 32: "{Binding FirstName, UpdateSourceTrigger=PropertyChanged,   
 Mode=TwoWay}">  
 33: </TextBox>  
 34: <TextBlock Text="Your First Name:" Grid.Column="0"   
 Grid.Row="1"/>  
 35: <TextBlock Text="{Binding FirstName}" Grid.Column="1"   
 Grid.Row="1"/ >  
 36:    
 37: </Grid>  
 38: </UserControl>

Ну и код модифицированного класса (чтобы работало связывание в двух направлениях):

1: public class Person: INotifyPropertyChanged   
 2: {  
 3: public string firstName;  
 4:    
 5: public string FirstName {  
 6: get { return firstName; }  
 7: set {   
 8: firstName = value;  
 9: OnPropertyChanged(new  
 PropertyChangedEventArgs("FirstName"));  
 10: }   
 11: }  
 12:    
 13: public event PropertyChangedEventHandler PropertyChanged;  
 14:    
 15: public void OnPropertyChanged(PropertyChangedEventArgs e)  
 16: {  
 17: if (PropertyChanged != null)  
 18: PropertyChanged(this, e);  
 19: }  
 20:    
 21: }

В результате, если вводить текст в поле редактирования, он тут же будет отображаться и в поле ниже. Это не всегда хорошо, так как если текст проходит валидацию, то в процессе набора нет необходимости требовать его корректность, но иногда такая возможность может быть полезна.

### Использование связывания в стилях

Теперь в стилях также можно применять связывание. Это упрощает «натягивание» уникальной оболочки для каждого пользователя, позволяя загружать стили динамически, а также предусматривать установку и модификацию характеристик интерфейса.

1: <Grid.Resources >  
 2: <Style TargetType="TextBox">  
 3: <Setter Property="Background" Value="{Binding background}"/>  
 4: <Setter Property="Width" Value="{Binding width}"></Setter>  
 5: <Setter Property="Margin" Value="{Binding margin}"></Setter>  
 6: </Style>  
 7: <Style TargetType="TextBlock">  
 8: <Setter Property="Margin" Value="{Binding margin}"></Setter>  
 9: </Style>  
 10: </Grid.Resources>

Класс модели

1: public class TextStyle  
 2: {  
 3: public string width { get; set; }  
 4: public string background { get; set; }  
 5: public string margin { get; set; }  
 6: }

И связывание:

1: LayoutRoot.DataContext = new TextStyle()   
 2: { width = "200", background = "Blue", margin = "5" };

### Неявные шаблоны

Шаблоны данных позволяют указать формат/разметку отображения не интерфейсных данных и использовать эту разметку в различных местах нашего приложения.

Неявные шаблоны позволяют формировать разметку, базируясь на типе данных. В более ранних версиях технологии необходимо было указывать ключ шаблона, что бы его использовать в нескольких местах приложения. Используя неявные шаблоны, нам необходимо лишь указать, для какого типа данных необходимо использовать шаблон.

Создадим проект и добавим несколько типов данных

public class Customer  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string Title { get; set; }  
 public string Company { get; set; }  
}  
  
public class Employee  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public int Room { get; set; }  
}

Добавим несколько шаблонов данных в ресурсы приложения(файл App.xaml)

<Application  
 xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"   
 x:Class="ImplicitDataTemplate.App"  
 xmlns:data="clr-namespace:ImplicitDataTemplate">  
 <Application.Resources>  
 <DataTemplate ***DataType="data:Customer"***>  
 <Grid Background="LightSkyBlue">  
 <Grid.RowDefinitions>  
 <RowDefinition/>  
 <RowDefinition/>  
 Grid.RowDefinitions>  
 <StackPanel Orientation="Horizontal">  
 <TextBlock Text="{Binding FirstName,   
 StringFormat='\{0\} '}"/>  
 <TextBlock Text="{Binding LastName}"/>  
 </StackPanel>  
 <StackPanel Grid.Row="1">  
 <TextBlock Text="{Binding Title}" FontStyle="Italic"/>  
 <TextBlock Text="{Binding Company,   
 StringFormat='Компания: \{0\}'}" FontWeight="Bold"/>  
 </StackPanel>  
 Grid>  
 </DataTemplate>  
 <DataTemplate ***DataType="data:Employee"***>  
 <Grid Background="LightGreen" >  
 <Grid.RowDefinitions>  
 <RowDefinition/>  
 <RowDefinition/>  
 Grid.RowDefinitions>  
 <StackPanel Orientation="Horizontal"   
 HorizontalAlignment="Right">  
 <TextBlock Text="{Binding FirstName,  
 StringFormat='\{0\} '}"/>  
 <TextBlock Text="{Binding LastName}"/>  
 </StackPanel>  
 <StackPanel Grid.Row="1" Orientation="Horizontal"   
 HorizontalAlignment="Right">  
 <TextBlock Text="{Binding Room,  
 StringFormat='Комната: \{0\}'}" FontStyle="Italic"/>  
 </StackPanel>  
 </Grid>  
 </DataTemplate>  
 </Application.Resources>  
</Application>

И последний момент, создание тестовых данных в приложении, связывание их с формой

this.DataContext = new List<object>  
{  
 new Customer{ FirstName="Иван", LastName="Копаткинский",   
 Company="Топинамбур", Title="Бухгалтер"},

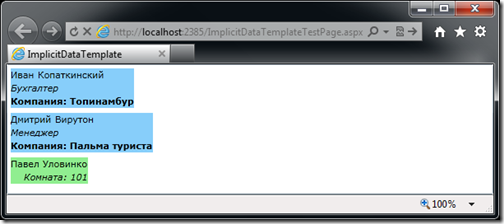
new Customer{ FirstName="Дмитрий", LastName="Вирутон",  
 Company="Пальма туриста", Title="Менеджер"},

new Employee{ FirstName="Павел", LastName="Уловинко", Room=101}  
};

и добавление компонента ListBox для отображения списка данных

<Grid x:Name="LayoutRoot" Background="White">  
 <ListBox ItemsSource="{Binding}"/>  
</Grid>

Результат работы:

[](http://lutay.uneta.com.ua/image.axd?picture=image_21.png)  
Рисунок 5.7. Результат применения неявных шаблонов в Silverlight 5.

### Другие изменения

Для полноты статьи приведем еще несколько нововведений, которые сами по себе интересны, но не стоят отдельного раздела из-за своей прозрачности (понятности).

Особое внимание стоит обратить на новое событие **DataContextChanged**, которое представлено в базовом классе **FrameworkElement**. Теперь элемент способен отреагировать на изменение источника данных.

Если связывание происходит с другими элементами в модели, то теперь можно использовать свойство **RelativeSource** совместно с **FindAncestor**. Это позволяет указать вместо имени элемента, его тип и уровень вложенности. В результате нужный элемент будет найден в реальном времени и выполнено связывание.

Последнее нововведение, это доступность интерфейса **ICustomTypeProvider**, описывающего метод **GetCustomType**. Данный интерфейс может быть полезен, когда Вы создаете объекты с динамически сгенерированными свойствами. В этом случае Вы генерируете и возвращаете объект типа Type, описывающий все свойства, которые Вы хотели бы использовать при связывании. Аналогичным образом можно поступить, когда Вы хотите скрыть некоторые свойства от элементов управления, генерирующих свою структуру автоматически. Например, **DataGrid**, генерирующий свои колонки автоматически.

## Краткие итоги

Silverlight – новая технология от Mirosoft, предназначенная для разработки насыщенных Web – приложений. Особенности версии Silverlight 4 – интенсивное использование графики, анимации, работа с медиа-файлами, а так же эффективное взаимодействие с данными и серверными компонентами. В декабре 2010 года в Microsoft сообщили, что Silverlight 5 будет включать 40 новых функций. Большая часть которых, так же как и в изменениях пришедших с 4 версией Silverlight, сосредоточена вокруг графики, анимации и работа с медиа-файлами. 1 сентября 2011 Silverlight 5 от Microsoft перешел в стадию Release Candidate. Выход финальной версии технологии по-прежнему намечен на конец 2011 года.

## Набор для практики

**Вопросы:**

1. Назовите ключевые особенности Silverlight 4
2. Охарактеризуйте основные нововведения Silverlight 5 связанные с текстом
3. Повышение полномочий для приложений в браузере
4. Поддержка нескольких окон в Silverlight 5

## Литература

1. Байдачный С.С. Silverlight 4: Создание насыщенных Web-приложений / С.С. Байдачный. – М.: СОЛОН-ПРЕСС, 2010. – 288 с.
2. Центр разработки Silverlight [Электронный ресурс]. – Режим доступа к ресурсу: http://msdn.microsoft.com/ru-ru/.
3. Сообщество пользователей Silverlight [Электронный ресурс]. – Режим доступа к ресурсу: http:// silverlighter.ru/.

# Лекция 6. Привязка данных в технологиях WPF и Silverlight

**Краткая аннотация лекции:** в лекции описываются принципы работы с данными, основные принципы связывания, разбираются интерфейсы INotifyPropertyChanged и INotifyCollectionChanged.

**Цель лекции:** рассмотреть основы привязки данных, узнать, как извлечь информацию из одного элемента и отобразить в другом, не написав ни единой строки кода. Научить реализовывать интерфейсы INotifyPropertyChanged, INotifyCollectionChanged.

Платформа Windows Presentation Foundation отличается от многих других каркасов для создания пользовательских интерфейсов в части обработки данных. Хотя WPF/Silverlight и позволяет смешивать данные и пользовательский интерфейс, модели, управляемые данными, все же обеспечивают большую гибкость и возможность совместной работы программистов и дизайнеров.

## Принципы работы с данными

Как правило, приложение создается для отображения или создания тех или иных данных. Что бы ни представляли собой данные – документ, базу данных или чертеж, главная задача приложения состоит в том, чтобы их отобразить, создать и отредактировать. Способов представления данных столько же, сколько приложений. Уже с момента возникновения на платформе .NET существовала стандартная модель, которая существенно изменила подходы к обработке данных.

### Модель данных в .NET

Модель данных описывает контракт между источником и потребителем данных. Исторически сложилось так, что каждый каркас нес с собой новую модель данных: В Visual Basic это сначала была **DAO** (Data Access Objects), потом **RDO** (Remote Data Objects) и, наконец, **ADO** (ActiveX Data Objects). На платформе .NET произошел переход от *API зависимых моделей* к единой для всего каркаса.

В .NET имеется некая объектная модель, содержащаяся классы, интерфейсы, структуры, перечисления, делегаты и т.д. Но сверх того включена очень простая модель данных. Списки в .NET представляются интерфейсами из пространства имен System.Collections: IEnumerable и IList. Свойства определяются либо с помощью встроенных в CLR свойств, либо путем реализации интерфейса ICustomTypeDescriptor. Эта базовая модель осталась неизменной, несмотря на появление новых технологий доступа к данным.

В .NET также имеется и несколько конкретных способов работы с данными, например: **ADO.NET** (пространство имен System.Data), **XML** (пространство имен System.Xml), **контракт о данных** (пространство имен System.Runtime.Serialization) и разметка (пространство имен System.Windows.Markup). И это далеко не все. Самое важное, что все они построены поверх базовой модели данных .NET.

Поскольку все операции с данными в WPF/Silverlight базируются на фундаменте модели данных .NET, то элементы управления WPF могут получать данные от любого объекта CLR:

ListBox listBox1 = new ListBox();  
listBox1.ItemsSource = new string[] { “Hello”, “World” };

### Всепроникающее связывание

Из всех нововведений, появившихся в WPF, для программиста, пожалуй, самым важным является полная интеграция связывания в систему. Шаблоны элементов, ресурсы и даже базовая «модель содержимого», лежащая в основе таких элементов управления, как Button и ListBox, – все может быть привязано к данным. Идея привязки свойств к источникам данным, отслеживания зависимостей и автоматического обновления экрана пронизывает все части WPF/Silverlight.

У связывания в WPF много названий. Например, для привязки к шаблонам в разметке применяется обозначение **TemplateBinding** вместо стандартной нотации **Binding**, используемой для привязки к данным. Привязка к шаблону работает лишь в контексте некоего шаблона, и при этом допускается привязывать лишь свойства шаблонного элемента управления. Это ограничение обеспечивает очень высокую эффективность привязки, что абсолютно необходимо, так как в WPF/Silverlight очень многие элементы управления привязываются к шаблонам. Привязка позволяет синхронизировать два представления данных. В WPF это означает, что некие данные (то есть свойство источника данных) привязываются к свойству элемента пользовательского интерфейса. Привязка прекрасно годится для поддержания синхронизации между двумя объектами, но, если типы данных этих объектов согласуются не идеально, начинают возникать проблемы.

Чтобы идея привязки действительно проникла во все части каркаса, необходимы способы преобразования данных.

### Преобразование данных

В любом каркасе с повсеместной привязкой доступ к произвольным данным возможен лишь тогда, когда эти данные могут быть преобразованы в любом месте, где привязка поддерживается (кроме привязки ресурсов).

WPF поддерживает два основных вида преобразований: **конвертацию значений** и **шаблоны данных**. Конвертеры значений просто преобразовывают данные из одного формата в другой (например, строку «Red» в Brushes.Red). Это весьма мощный механизм, поскольку конвертер может осуществлять двустороннее преобразование. Шаблоны данных, с другой стороны, позволяют «на лету» создавать элементы управления для представления данных. В модели шаблонов данных элементы управления могут с помощью привязок обращаться к данным, то есть читать и потенциально даже записывать данные обратно в источник.

С помощью преобразований можно получить очень выразительные представления данных. Рассмотрим работу с данными в WPF с одного из наиболее применяемых источников – **ресурсов**. В первой лекции мы уже столкнулись с данным вопросом, теперь рассмотрим его более подробно

## Ресурсы

Самое первое место, где программист сталкивается с необходимостью отделить отображение от самих данных – это ресурсы. У каждого элемента есть свойство **Resources**, представляющее собой обычный словарь, который позволяет искать значение по ключу. Эта простая техника применяется в механизмах поддержки тем, стилизации и привязки к данным.

В программе на языке C# можно определять переменные для последующего использования. Часто это делается ради удобства чтения кода, а иногда в целях обобществления:

public class Window1 : Window   
{  
 public Window1()  
 {  
 Title = “Resources”;  
 **Brush toShare = new SolidColorBrush(Colors.Yellow);**  
 Button b = new Button();  
 b.Content = “My Button”;  
 **b.Background = toShare;** Content = b;  
 }  
}

В разметке эта проблема не так проста. Модель грамматического разбора XAML требует, чтобы все создаваемые объекты были свойствами чего-либо. Захоти мы получить кисть, общую для двух кнопок, пришлось бы создавать какой-то вид привязки одного элемента к другому, а это, очевидно, куда сложнее, чем просто завести переменную. Поэтому при разработке WPF было решено ввести единое свойство *Resources*, в котором можно хранить список именованных объектов. Эти объекты стали бы тогда доступны любому дочернему элементу. В следующем фрагменте кисть определяется в свойстве Resources окна, а затем используется в кнопке:

<Window  
 Text=’Resources’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <Window.Resources>  
 <SolidColorBrush **x:Key=’toShare’**>Yellow</SolidColorBrush>  
 </Window.Resources>  
   
 <Button Background=’{**StaticResource toShare**}’>  
 My Button  
 </Button>  
</Window>

Путь поиска ресурса несколько сложнее, чем просто проход вверх по иерархии. Просматривается также объект приложения, системная тема и тема по умолчанию для типов. Порядок просмотра таков:

1. Иерархия элементов.
2. Application.Resources.
3. Тема типа.
4. Системная тема.

Ранее мы говорили, что у класса типа Application есть свойство Resources, позволяющее определить ресурсы, глобальные для всего приложения. Рассмотрим на примере:

<Application x:Class=’EssentialWPF.MyApp’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <Application.Resources>  
 <SolidColorBrush x:Key=’toShare’>Purple</SolidColorBrush>  
 </Application.Resources>  
</Application>

Эта техника позволяет создавать ресурсы, общие для всех страниц, окон и элементов управления. На любом уровне приложения можно переопределить глобальное значение ресурса. В общем случае рекомендуется определять ресурс на самом нижнем возможном уровне. Если некий ресурс используется только в одной панели, то для этой панели его и надо определить. Если же ресурс используется в нескольких окнах, то определить его следует на уровне приложения. Определяя ресурс, важно помнить, что использовать его можно в разных местах. Поскольку каждый элемент WPF может присутствовать только в одном месте дерева отображения, мы не можем надежно использовать элемент в качестве ресурса:

<Window  
 x:Class=’EssentialWPF.Resources’  
 Title=’Resources’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <Window.Resources>  
 <TextBox x:Key=’sharedTextBox’ />  
 </Window.Resources>  
 <Button Content=’{StaticResource sharedTextBox}’/>  
</Window>

Эта разметка будет работать только, если на элемент sharedTextBox есть только одна ссылка. Попытайся мы воспользоваться этим ресурсом еще раз, приложение завершится с ошибкой:

<Window  
 x:Class=’EssentialWPF.Resources’  
 Title=’Resources’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <Window.Resources>  
 <TextBox x:Key=’sharedTextBox’ />  
 </Window.Resources>  
 <StackPanel>  
 **<!— это ошибка! —>** <Button Content=’{StaticResource sharedTextBox}’/>  
 <Button Content=’{StaticResource sharedTextBox}’/>  
 </StackPanel>  
</Window>

Когда ресурс требуется использовать более одного раза, необходимо прибегнуть к классу **FrameworkElementFactory**. Для элементов, принадлежащих шаблонам, мы создаем фабрику, а не сами элементы. Большинству визуальных объектов (кистей, перьев, сеток и т.д.) фабрика не нужна, так как многократное использование обеспечивается наследованием классу Freezable.

Почему в лекции о привязке к данным мы говорим о ресурсах? Дело в том, что, используя ссылки на статические ресурсы, мы по существу выполняем присваивание переменной, как в приведенном выше фрагменте на C#. Когда эта переменная используется, никакой связи с исходной переменной уже нет. Рассмотрим следующий код:

Brush someBrush = Brushes.Red;  
Button button1 = new Button();  
button1.Background = someBrush;  
someBrush = Brushes.Yellow;  
  
// button1.Background здесь будет красным

При работе с ресурсами можно либо выполнить аналогичное **статическое связывание** в форме присваивания, либо организовать **динамическое связывание:**

<Window  
 Title=’Resources’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <Window.Resources>  
 <SolidColorBrush x:Key=’toShare’>Yellow</SolidColorBrush>  
 </Window.Resources>  
 <Button Background=’{**DynamicResource toShare**}’>  
 My Button  
 </Button>  
</Window>

Поскольку на этот раз мы воспользовались динамическим связыванием, то можем изменить цвет кнопки, присвоив новое значение свойству Resources окна:

<!— window1.xaml —>  
  
...  
<Button Background=’{DynamicResource toShare}’ Click=’Clicked’>  
 My Button  
</Button>  
...  
  
// window1.xaml.cs  
...  
void Clicked(object sender, RoutedEventArgs e)  
{  
 Brush newBrush = new SolidColorBrush(Colors.Blue);  
 this.Resources[“toShare”] = newBrush;  
}  
...

Это очень полезный механизм. В сочетании с иерархической областью видимости ресурсов он позволяет обновить сразу все окна или страницы приложения.

Чтобы выполнить динамическое связывание ресурса программно, нам понадобится метод FrameworkElement.SetResourceReference:

button1.SetResourceReference(Button.BackgroundProperty,«toShare»);

Конечно, такие динамические ссылки не обходятся без накладных расходов, поскольку система должна отслеживать изменения ресурса.

При изменении любого ресурса обновляется все дерево. Поэтому статические и динамические ссылки на ресурсы можно использовать во многих местах, стоимость операции от количества ссылок не зависит. А вот часто изменять ресурсы для обновления пользовательского интерфейса не стоит. Зато и тревожиться по поводу большого числа ссылок на ресурсы не надо.

***Ресурсы – это особая форма привязки к данным, оптимизированная в расчете на большое число привязок, которые редко обновляются.*** В общем случае, механизм привязки к данным оптимизирован в предположении умеренного числа привязок (в том числе и двусторонних) с высокой частотой обновления. Этот более общий вид привязки получил и более простое название; в WPF он называется просто **связыванием** или **привязкой**.

## Основные принципы связывания

**Связывание *– это просто способ синхронизации двух элементов данных***. **Элемент данных (data point) *– абстрактное понятие, выражающее идею «точки» в пространстве данных.*** Описать элемент данных можно разными способами; чаще всего он представляется источником данных и запросом. Например, элемент данных «свойство» состоит из объекта и имени свойства. Имя свойства определяет само свойство, а объект служит источником данных для этого свойства.

В WPF элемент данных представлен классом **Binding**. Для конструирования привязки мы указываем **источник** (данных) и **путь** (запрос). В следующем примере создается элемент данных, ссылающийся на свойство Text объекта TextBox:

Binding bind = new Binding();  
bind.Source = textBox1;  
bind.Path = new PropertyPath(“Text”);

Нужен еще второй элемент данных, который будет синхронизован с первым. Поскольку связывание в WPF ограничивается только деревом элементов, то для определения какого-либо элемента данных нужно вызвать метод SetBinding. Этот метод вызывается для источника данных, а данные привязываются к запросу (в данном примере к свойству ContentControl.ContentProperty):

contentControl1.SetBinding(ContentControl.Content, bind);

В этом примере свойство Text объекта textBox1 связывается со свойством Content объекта contentControl1. То же самое можно было бы выразить на XAML

<Window ... Title=’ExampleBind’>  
 <StackPanel>  
 <TextBox x:Name=’textBox1’ />  
 <ContentControl   
 Margin=’5’ x:Name=’contentControl1’  
 Content=’{**Binding ElementName=textBox1,Path=Text**}’ />  
 </StackPanel>  
</Window>

Когда привязка объявляется в разметке, для задания источника можно использовать свойство ElementName.

Как мы только что увидели, механизм связывания можно применить для привязки свойства Text (строкового) элемента TextBox к свойству Content (имеющему тип object). Так же можно привязать свойство Text к свойству FontFamily:

<Window ... Title=’ExampleBind2’ >  
 <StackPanel>  
 <TextBox x:Name=’textBox1’ />  
 <TextBox x:Name=’textBox2’ />  
 <ContentControl  
 Margin=’5’  
 Content=’{Binding ElementName=textBox1,Path=Text}’  
 **FontFamily=’{Binding ElementName=textBox2,Path=Text}’**/>  
 </StackPanel>  
</Window>

Существует два механизма преобразования: класс **TypeConverter**, существующий в .NET, начиная с версии 1.0, и новый интерфейс **IValueConverter**. В нашем случае с классом FontFamily ассоциирован конвертер типов TypeConverter, поэтому преобразование выполняется автоматически.

Чтобы выполнить нужное преобразование, можно воспользоваться конвертерами значений, ассоциированными с привязкой. Для этого берется источник (строка из свойства Text) и преобразуется в какой-то объект, который понимает получатель (свойство Content).

Начнем с создания простого типа:

public class Human   
{  
 private string \_name;  
 public string Name   
 {  
 get { return \_name; }  
 set { \_name = value; }  
 }  
}

Тип мог бы быть любым: встроенным, библиотечным, разработанным вами. Идея в том, что мы хотим преобразовать свойство Text в объект конкретного типа. Для этого произведем конвертер от интерфейса IValueConverter и реализуем два метода:

public class HumanConverter : IValueConverter   
{  
 public object Convert(  
 object value, Type targetType, object parameter, CultureInfo culture)  
 {  
 Human h = new Human();  
 h.Name = (string)value;  
 return h;  
 }  
 public object ConvertBack(  
 object value, Type targetType, object parameter, CultureInfo culture)   
 {  
 return ((Human)value).Name;  
 }  
}

В более сложных случаях реализовать преобразование в обе стороны может оказаться невозможно. Последний шаг при использовании конвертера – ассоциировать его с привязкой:

<ContentControl  
 Margin=’5’  
 FontFamily=’{Binding ElementName=textBox2,Path=Text}’>  
 <ContentControl.Content>  
 <Binding ElementName=’textBox1’ Path=’Text’>  
 <Binding.Converter>  
 <l:HumanConverter xmlns:l=’clr\_namespace:ExampleBind’/>  
 </Binding.Converter>  
 </Binding>  
 </ContentControl.Content>  
</ContentControl>

Элементы данных и преобразования – две базовые конструкции механизма связывания. Познакомившись с основными ингредиентами данных, мы можем заняться деталями привязки к объектам CLR.

## Привязка к объектам CLR

Данные привязываются к объектам CLR с помощью свойств и списков (списком считается любой тип, реализующий интерфейс IEnumerable). Связывание устанавливает связь между источником и получателем.

Идентификатор имени свойства для связывания с объектом может записываться в двух видах: для простых свойств CLR и для зависимых свойств, производных от класса DependencyProperty. Чтобы понять, в чем разница, начнем с простого примера:

<Window  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <StackPanel>  
 <TextBox Name=’text1’>Hello</TextBox>  
 <TextBox Text=’**{Binding ElementName=text1, Path=Text}**’ />  
 </StackPanel>  
</Window>

Здесь свойство Text объекта TextBox связывается со свойством Text другого объекта. С очень похожим примером мы уже встречались выше. Поскольку Text –зависимое свойство, то этот пример в точности эквивалентен такому:

<Window  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’>  
 <StackPanel>  
 <TextBox Name=’text1’>Hello</TextBox>  
 <TextBox Text=’**{Binding ElementName=text1, Path=(TextBox.Text)}**’ />  
 </StackPanel>  
</Window>

Во втором случае мы воспользовались формой идентификатора свойства, «классифицированной классом». Результат в обоих случаях одинаков, но во втором примере удается избежать применения отражения для разрешения имени «Text» в выражении привязки. Эта оптимизация полезна с двух точек зрения: во-первых, чтобы избежать накладных расходов на отражение, а, во-вторых, чтобы обеспечить привязку к присоединенным свойствам. Например, если бы мы захотели привязать объект TextBox к свойству Grid.Row, то могли бы написать,

<SomeControl SomeProperty=’{Binding ElementName=text1, Path=(Grid.Row)}’ />.

Чтобы лучше понять, как работают пути к свойствам, мы можем взять чуть более сложный объект. Определим класс Person, в котором есть составные свойства Address и Name. В совокупности три класса – Person, Name и Address – образуют небольшую объектную модель, на которой можно продемонстрировать некоторые интересные задачи, возникающие в связи со связыванием. Для начала организуем простое отображение данных о человеке

<!— Window1.xaml —>  
<Window ... Title=’Object Binding’>  
 <StackPanel>  
 <ContentControl Content=’{Binding Path=Name}’ />  
 <TextBlock Text=’{Binding Path=Addresses[0].AddressName}’ />  
 <TextBlock Text=’{Binding Path=Addresses[0].Street1}’ />  
 <TextBlock Text=’{Binding Path=Addresses[0].City}’ />  
 </StackPanel>  
</Window>  
  
// Window1.xaml.cs  
public partial class Window1: Window   
{  
 public Window1()  
 {  
 InitializeComponent();  
 DataContext = new Person(  
 new Name(«Иван», «Иванов»),  
 new Address(“Интуит”,  
 “Интуит.ru”,  
 “Москва”);  
 }  
}

Здесь иллюстрируется привязка к простому свойству (Path=Name) и более сложные пути к свойствам (Path=Addresses[0].AddressName). Квадратные скобки позволяют добраться до отдельных элементов набора. Обратите также внимание, что мы можем составить сколь угодно сложный путь из имен свойств и индексов в списке. Привязка к списку производится точно так же, как к свойству. Путь к свойству должен приводить к объекту, который реализует интерфейс IEnumerable, но в остальном никаких отличий нет. Можно было бы отображать не один адрес, а завести список и привязаться к его свойству ItemsSource (разумеется, тогда мы смогли бы определить шаблон данных для типа адреса):

<ListBox ItemsSource=’{Binding Path=Addresses}’ />

У человека есть единственное имя и нуль или более адресов. До сих пор нас интересовало в основном отображение данных. Если у свойства, к которому мы привязываемся, есть метод установки, то возможна и двусторонняя привязка.

## Редактирование

Чтобы редактировать значения, должен быть какой-то способ узнать, что значение изменилось. Помимо разрешения изменять свойство, существует несколько интерфейсов, которые позволяют объекту или списку рассылать извещения об изменении. Если источник данных уведомляет об изменении, то система связывания сможет отреагировать на модификацию данных. Чтобы наделить наш класс Person способностью извещать об изменениях, у нас есть три возможности:

* реализовать интерфейс **INotifyPropertyChanged**;
* добавить события, с помощью которых мы будем сообщать об изменении;
* создать свойства, производные от класса DependencyProperty.

Использование событий для извещения об изменениях свойств впервые было применено в .NET 1.0 и поддерживается механизмами связывания в Windows Forms и ASP.NET. Интерфейс INotifyPropertyChanged появился в .NET 2.0. Он оптимизирован для привязки к данным, обладает более высокой производительностью и проще как для авторов объектов, так и для самой системы связывания. Но использовать этот интерфейс в обычных сценариях, когда необходимо извещать об изменениях, несколько сложнее.

Применение свойств, производных от DependencyProperty, относительно просто, позволяет объекту воспользоваться преимуществами, которые дает разреженное хранилище, и хорошо сопрягается с другими службами WPF (например, с динамической привязкой к ресурсам и стилизацией). Подробно создание объектов со свойствами, производными от DependencyProperty, обсуждается в лекции о User/Custom control’ах.

Все три способа во время выполнения демонстрируют одинаковое поведение. Вообще говоря, при создании модели данных лучше реализовать интерфейс INotifyPropertyChanged. Для использования свойств, производных от DependencyProperty, требуется, чтобы класс объекта наследовал DependencyObject, а это, в свою очередь, означает, что объект данных должен работать в STA-потоке. Применение событий для извещения об изменениях свойств приводит к разбуханию объектной модели; в общем случае этот механизм лучше оставить для свойств, которые разработчики и так привыкли отслеживать:

public class Name: INotifyPropertyChanged  
{  
 ...  
 public string First  
 {  
 get { return \_first; }  
 set   
 {  
 \_first = value;  
 NotifyChanged(“First”);  
 }  
 }  
...  
 public event PropertyChangedEventHandler PropertyChanged;  
 void NotifyChanged(string property)  
 {  
 if (PropertyChanged != null)  
 {  
 PropertyChanged(this, new PropertyChangedEventArgs(property));  
 }  
 }  
}

Раз мы применяем один и тот же паттерн ко всем трем объектам данным, то можем изменить пользовательский интерфейс и воспользоваться преимуществами новой реализации объектов. Мы заведем два неизменяемых текстовых поля (TextBlock) для отображения текущего имени и два объекта TextBox для редактирования значений. Поскольку класс Name реализует интерфейс INotifyPropertyChanged, то при изменении значений система связывания получит извещение, что приведет к обновлению объектов TextBlock:

<Window ... Text=’Object Binding’>  
 <StackPanel>  
 <TextBlock Text=’{Binding Path=Name.First}’ />  
 <TextBlock Text=’{Binding Path=Name.Last}’ />  
 <Grid>  
 <Grid.RowDefinitions>  
 <RowDefinition />  
 <RowDefinition />  
 </Grid.RowDefinitions>  
 <Grid.ColumnDefinitions>  
 <ColumnDefinition />  
 <ColumnDefinition />  
 </Grid.ColumnDefinitions>  
 <Label Grid.Row=’0’ Grid.Column=’0’>First</Label>  
 <TextBox Grid.Row=’0’ Grid.Column=’1’  
 Text=’{Binding Path=Name.First}’ />  
 <Label Grid.Row=’1’ Grid.Column=’0’>Last</Label>  
 <TextBox Grid.Row=’1’ Grid.Column=’1’  
 Text=’{Binding Path=Name.Last}’ />  
 </Grid>  
 </StackPanel>  
</Window>

Ввод в любое текстовое поле обновляет соответствующую область окна. Отметим, что изменение происходит только при выходе из поля. По умолчанию элемент TextBox обновляет данные только в момент потери фокуса. Чтобы изменить это поведение, нужно указать, что привязка должно обновляться при любом изменении значения свойства. Для этого служит свойство привязки **UpdateSourceTrigger**:

<TextBox Grid.Row=’1’ Grid.Column=’1’  
 Text=’{Binding Path=Name.Last, **UpdateSourceTrigger=PropertyChanged**}’/>

Списки сложнее простого изменения свойств. Для связывания необходимо знать, какие элементы были добавлены или удалены. С этой целью в WPF введен интерфейс **INotifyCollectionChanged**, в котором определено единственное событие CollectionChanged с аргументом такого типа:

public class NotifyCollectionChangedEventArgs: EventArgs  
{  
 public NotifyCollectionChangedAction Action { get; }  
 public IList NewItems { get; }  
 public int NewStartingIndex { get; }  
 public IList OldItems { get; }  
 public int OldStartingIndex { get; }  
}  
  
public enum NotifyCollectionChangedAction  
{  
 Add,  
 Remove,  
 Replace,  
 Move,  
 Reset,  
}

В нашем примере мы можем поддержать динамическое добавление и удаление адресов человека. Проще всего это реализуется в помощью класса **ObservableCollection<T>**, который наследует Collection<T> и дополнительно реализует интерфейс **INotifyCollectionChanged**:

public class Person: INotifyPropertyChanged  
{  
 IList<Address> \_addresses = new ObservableCollection<Address>();  
...  
}

Теперь можно модифицировать наше приложение так, чтобы все адреса отображались в списковом поле, и реализовать пользовательский интерфейс для ввода новых адресов и добавления их в список:

<!— window1.xaml —>  
...  
<StackPanel>  
 <ListBox ItemsSource=’{Binding Path=Addresses}’/>  
 <Button Click=’Add’ Grid.Row=’4’>Add</Button>  
</StackPanel>  
…

// window1.xaml.cs

void Add(object sender, RoutedEventArgs e)  
{  
 Address a = new Address();  
 a.Street1 = \_street.Text;  
 a.City = \_city.Text;  
 ((Person)DataContext).Addresses.Add(a);  
}

В текстовые поля можно вводить новую информацию, а при нажатии кнопки Add список адресов обновляется. Поскольку свойство Addresses реализует интерфейс INotifyCollectionChanged, система связывания получает извещение о новом адресе и корректно обновляет пользовательский интерфейс.

Привязка к объектам CLR – процедура по большей части автоматическая. Реализовав интерфейсы INotifyPropertyChanged и INotifyCollectionChanged, мы наделяем источник данных “интеллектом”.

## Ключевые термины

**Data access object (DAO)** — это объект, который предоставляет абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения. Этот шаблон проектирования применим ко множеству языков программирования, большинству программного обеспечения, нуждающемуся в хранении информации и к большей части баз данных, но традиционно этот шаблон связывают с приложениями на платформе Java Enterprise Edition, взаимодействующими с реляционными базами данных через интерфейс JDBC, потому что он появился в рекомендациях от фирмы Sun Microsystems.

**Remote Data Objects (RDO)** — технология доступа к базам данных компании [Microsoft](http://ru.wikipedia.org/wiki/Microsoft). Представляет собой набор COM-объектов инкапсулирующих [ODBC](http://ru.wikipedia.org/wiki/ODBC)[API](http://ru.wikipedia.org/wiki/API), а также клиентскую курсорную библиотеку. Технология RDO появилась в [1995](http://ru.wikipedia.org/wiki/1995) году одновременно с выходом продукта [Visual Basic 4.0](http://ru.wikipedia.org/wiki/Visual_Basic). RDO позиционировалась как технология более простая чем прямое использование вызовов ODBC и в то же время более эффективная чем технология [DAO](http://ru.wikipedia.org/wiki/Data_Access_Objects). RDO была ориентирована на обработку данных на стороне сервера БД (такого как [MS SQL](http://ru.wikipedia.org/wiki/MS_SQL) Server, [Oracle](http://ru.wikipedia.org/wiki/Oracle_(%D0%A1%D0%A3%D0%91%D0%94)) и т.д.) в отличие от [DAO](http://ru.wikipedia.org/wiki/Data_Access_Objects) ориентированной в основном на обработку данных на стороне [клиента](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82).

**ActiveX Data Objects (ADO)** — [интерфейс программирования приложений](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) для доступа к [данным](http://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5), разработанный компанией [Microsoft](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82) ([MS Access](http://ru.wikipedia.org/wiki/MS_Access), [MS SQL Server](http://ru.wikipedia.org/wiki/MS_SQL_Server)) и основанный на технологии компонентов [ActiveX](http://ru.wikipedia.org/wiki/ActiveX). ADO позволяет представлять данные из разнообразных источников ([реляционных баз данных](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94), текстовых [файлов](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) и т.д.) в [объектно-ориентированном](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) виде.

**ADO.NET** — основная модель доступа к [данным](http://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5) для приложений, основанных на [Microsoft .NET](http://ru.wikipedia.org/wiki/Microsoft_.NET). Не является развитием более ранней технологии [ADO](http://ru.wikipedia.org/wiki/ADO). Скорее представляет собой совершенно самостоятельную технологию. Компоненты ADO.NET входят в поставку оболочки [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework); таким образом, ADO.NET является одной из главных составных частей [.NET](http://ru.wikipedia.org/wiki/Microsoft_.NET).

**Статическое связывание** — связывание цели вызова и вызываемого метода на этапе компиляции, когда с сущностью связывается метод класса, заданного при объявлении сущности.

**Динамическое связывание** — связывание цели вызова и вызываемого метода на этапе выполнения, когда с сущностью связывается метод класса объекта, связанного с сущностью в момент выполнения.

## Краткие итоги

В лекции мы рассмотрели, как в WPF устроена работа с данными приложения. Система привязки к данным глубоко интегрирована в платформу, и при наличии подходящей модели мы можем создавать приложения, целиком управляемые данными.

## Набор для практики

Вопросы:

1. Охарактеризуйте особенности статического и динамического связывания
2. Особенности класса ObservableCollection<T>
3. Интерфейсы INotifyCollectionChanged и INotifyPropertyChanged
4. Использование Converter
5. Способы наделить пользовательский класс способностью извещать об изменениях
6. Основные свойства привязки

## Литература

1. Общие сведения о связывании данных   
   <http://msdn.microsoft.com/ru-ru/library/ms752347.aspx>
2. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 559-600.
3. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008 — стр. 295-321.

# Лекция 7. Пользовательские элементы управления.

**Краткая аннотация лекции:** в лекции вы увидите создавать пользовательские элементы управления и делать их полноценными "гражданами" сообщества классов WPF и Silverliht проектов. Это значит, что вы будете оснащать их свойствами зависимости, чтобы получить поддержку таких важных служб, как привязка данных, стили и анимация. Кроме того, вы узнаете, как создать элемент, лишенный внешнего вида — управляемый шаблонами элемент, который позволяет его потребителю применять различные визуальные представления для большей гибкости.

**Цель лекции:** показать реализацию пользовательского элемента управления двумя путями: через UserControl и CustomControl. Дать представление о том, в какой из вариантов целесообразно использовать в той или иной ситуации. Научить пользоваться свойствами зависимости. Закрепить применение стилей и шаблонов.

## UserControl или CustomControl?

Есть две категории разработчиков элементов управления. Одни создают пользовательские элементы (**UserControl**), другие – нестандартные (**CustomControl**). Названия относятся к двум совершенно разным ситуациям и многие считают эти именования не совсем удачными. Некоторые программисты предпочитают такое определение: ***пользовательские элементы*** *(***UserControl***)* ***– как способ инкапсуляции частей графического интерфейса***, а ***нестандартные (*CustomControl*) – это повторно используемые элементы, которые можно применять и в других приложениях.*** Вопрос о том, где проходит граница между ними, – источник ожесточенных споров.

Из вышенаписанного можно заключить, что UserControl не очень хорошо подходит для повторного использования в рамках *нескольких приложений*. CustomControl по сравнению с UserControl трудоемок для разработки, но предоставляет больше возможностей *тонкой* настройки и поддержки повторного использования.

Так же программист должен помнить о том, что имеется большая вероятность возникновение ситуации, когда потребуется создать наследника от уже созданного UserControl’a, сделать это не возможно, поэтому вам придется переделывать UserControl в CustomControl. Процесс рутинный и трудоёмкий. Именно поэтому при создании пользовательского элемента управления необходимо предварительно рассмотреть все варианты его дальнейшего жизненного цикла. И если возникнут сомнения, что использовать UserControl или CustomControl, оптимальным решением будет выбор в пользу использования CustomControl.

## UserControl

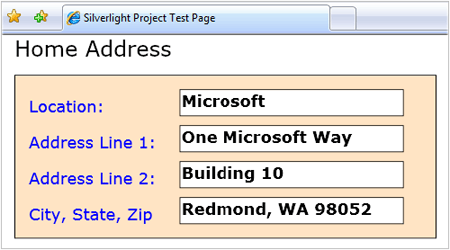
Данный раздел лекции последовательно продемонстрирует три этапа инкапсуляции частей графического интерфейса. Первым шагом будет создание на стартовой странице приложения табличной сетки, где мы разместим основные элементы управления и применим к ним стили. Затем добавим обработчик события KeyDown. В завершающей части этого раздела лекции создадим полноценный UserControl.

Вы можете добавлять пользовательские элементы управления в ваше приложение из-за ряда причин, но наиболее явной будет *инкапсуляция некоторой функциональности*, которую вы собираетесь многократно применять. Наилучшим примером использования будет создание некоторой функциональности и присвоение ее некоторому пользовательскому элементу управления, а затем повторное его использование.

Одним из способов создания пользовательских элементов управления является явное добавление его к вашему приложению. Щелкнув правой кнопкой мыши по проекту и выбрав Add->New Item, вы получите диалоговое окно, которое предложит вам шаблон с пользовательским элементом управления в качестве одной из опций.

### Шаг 1. Создание повторно используемой функциональности

Повторное использование функциональности пригодно в том случае, если данная функциональность стоит того, чтобы ее еще раз использовали. Мы начнем c создания формы (рис. 7.1) как совокупность элементов управления расположенных на странице и закончим выносом повторно используемого кода в отдельный UserControl. Создаваемая форма будет иметь возможность заполнения вручную, а также поддерживать клавишную комбинацию "быстрого вызова", в частности Ctrl-M будет добавлять адрес Microsoft.

  
Рисунок 7.1. Адресная форма, заполненная посредством нажатия Ctrl-M

Рассмотрим XAML разметку адресной формы, которую мы пока разместим на главной странице Silverliht приложения (MainPage). Подразумевается, что первым элементом управления будет StackPanel, поэтому мы можем с легкостью расположить элементы управления один над другим, а в пределах StackPanel у нас Grid в котором располагаются стандартные элементы управления.

<UserControl x:Class="Example.MainPage"  
 xmlns="http://schemas.microsoft.com/client/2007"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Width="500" Height="500">  
   <Border>  
 <StackPanel Background="White">  
      <TextBlock Text="Home Address" FontFamily="Verdana" FontSize="24"  
             HorizontalAlignment="Left" Margin="15,0,0,0"/>  
    <Grid x:Name="AddressGrid" >  
         <!--More Here-->  
     </Grid>  
   </Border>        
  </StackPanel>  
</UserControl>

#### Устанавливаем стили для элементов управления

Табличная сетка (Grid) будет заполнена четырьмя подсказками и четырьмя текстовыми полями ввода, но они все настроены вручную, и пока нас это устраивает. Приведенный ниже код должен располагаться в файле App.xaml в теги Application.Resources. Каждый Style определяет свой TargetType (т.е., Button или TextBlock) и затем устанавливает значение для каждого свойства, стилем которого оно хочет управлять. Мы создаем всего 2 стиля - один для TextBlock, который будет использован в качестве ярлыков, и второй - для получения информации в TextBox. Все TextBlocks будут выровнены по нижнему (Bottom) и левому (Left) краю, и у них будет установлен шрифт Verdana, размер шрифта 18, а текст будет синего цвета. Данные элементы TextBlock также будут иметь отступ в 5 пикселей со всех сторон

    <Application.Resources>  
        <Style TargetType="TextBlock" x:Key="TextBlockPrompt">  
            <Setter Property="VerticalAlignment" Value="Bottom" />  
            <Setter Property="HorizontalAlignment" Value="Left" />  
            <Setter Property="FontFamily" Value="Verdana" />  
            <Setter Property="FontSize" Value="18" />  
            <Setter Property="FontWeight" Value="Medium" />  
            <Setter Property="Foreground" Value="Blue" />  
            <Setter Property="Margin" Value="5" />  
        </Style>

Все элементы TextBoxe будут иметь черный жирный шрифт Verdana размера 18, а также будут выровнены по нижнему левому краю. TextBox будет размером 250 на 30 и также будет иметь отступ в 5 пикселей со всех сторон.

        <Style TargetType="TextBox" x:Key="TextBoxStyle">  
            <Setter Property="FontFamily" Value="Verdana" />  
            <Setter Property="FontSize" Value="18" />  
            <Setter Property="FontWeight" Value="Bold" />  
            <Setter Property="Foreground" Value="Black" />  
            <Setter Property="VerticalAlignment" Value="Bottom" />  
            <Setter Property="HorizontalAlignment" Value="Left" />  
            <Setter Property="Width" Value="250" />  
            <Setter Property="Height" Value="30" />  
            <Setter Property="Margin" Value="5" />  
        </Style>  
    </Application.Resources>

Вместо добавления информации о стиле TextBlock вручную, мы можем привязать информацию об атрибуте Style при помощи "key" для определения желаемого стиля.

<TextBlock Text="Location: "  
     Style="{StaticResource TextBlockPrompt}"  
     Grid.Row="2" Grid.Column="1"  />

Строка Style="{StaticResource TextBlockPrompt}" переходит в App.xaml и находит стиль, ключом которого является TextBlockPrompt и присваивает все значения разом. Точно так же мы можем присвоить все значения стиля TextBox привязанному текстовому полю.

<TextBox x:Name="Location"  
    Style="{StaticResource TextBoxStyle}"  
    Text ="Silverlight Central"  
    Grid.Row="2" Grid.Column="3" />

Оставшиеся элементы в разметке сопоставимы, им так же применяем стиль.

#### Двусторонняя привязка данных

Одним из мощных способов привязки данных, который поможет привязать данные об адресе, например из базы данных, XML-файла или какого-либо другого хранилища данных, является использование **бизнес-объекта** в качестве посредника между хранилищем и пользовательским интерфейсом. Давайте создадим объект для адреса, сопоставимого с данными, которые мы хотим отобразить (несмотря на то, что обычный бизнес-объект, скорее всего, не будет так тесно связан с конкретной страницей пользовательского интерфейса). В таблице 7.1 представлены свойства и переменные класса Address:

| Private Member Variable | Public Property |
| --- | --- |
| Location | Location |
| address1 | Address1 |
| address2 | Address2 |
| City | City |

Таблица 7.1. Структура класса Address.

Структура класса будет похожа на структуру, используемую в лекции о привязке данных.

using System.Collections.Generic;  
using System.ComponentModel;  
  
namespace Example  
{  
    public class Address : INotifyPropertyChanged  
    {

Не забывайте, что каждое свойство кроме своей переменной обладает своим методом для получения данных (Get) и методом установки данных (Set), которые также вызывают событие *notification*.

Заменим все явно заданные значения Text Block привязанными значениями

<TextBox x:Name="Location"  
    Style="{StaticResource TextBoxStyle}"  
    Text ="{**Binding Location, Mode=TwoWay**}"  
    Grid.Row="2" Grid.Column="3" />  
  
…

В оставшихся элементах так же заменяем явно заданные значения.

### Шаг 2. Добавление обработки событий клавиатуры

Для реализации функциональности "быстрых клавиш" нам необходимо осуществить реакцию на события клавиатуры, полученные элементом StackPanel, в частности событие KeyDown. Событие KeyDown является пузырьковым событием.

Мы можем начать с простого и отвечать на любые события клавиатуры путем заполнения формы адресом компании Microsoft. Присвоим классу защищенный объект (private) Address, для которого мы определим память в конструкторе. Также будем обрабатывать стандартное событие Loaded, которое вызывается при загрузке страницы.

public partial class MainPage : UserControl  
{  
    private Address theAddress;  
    public MainPage()  
    {  
        InitializeComponent();  
        address = new Address();  
        Loaded += new RoutedEventHandler(OnLoaded);  
    }

В *OnLoaded* мы хотим создать обработчик события для KeyDown при регистрации любого события клавиши для любого элемента управления в пределах табличной сетки, которую мы назвали AddressGrid в MainPage.xaml следующим образом,

<Grid x:Name="AddressGrid" Background="Bisque" >

При сохранении страницы MainPage.xaml данный определитель мгновенно будет доступен в фоновом коде, и мы можем получить доступ к его свойствам и событиям, включая событие KeyDown.

void OnLoaded(object sender, RoutedEventArgs e)  
{  
    AddressGrid.KeyDown += new KeyEventHandler(AddressGrid\_KeyDown);  
}

Visual Studio 2010 предложит вам создать программную оболочку для обработчика события, что нам как раз и нужно. Нам необходимо заполнить форму адресом компании Microsoft при нажатии Ctrl+M.

void AddressGrid\_KeyDown(object sender, KeyEventArgs e)  
{  
    if (e.Key == Key.M && Keyboard.Modifiers == ModifierKeys.Control)  
    {  
        address.Location = "Microsoft";  
        address.Address1 = "One Microsoft Way";  
        address.Address2 = "Building 10";  
        address.City = "Redmond, WA 98052";  
    }  
    this.DataContext = theAddress;  
}

Обратите внимание на то, что мы завершаем событие установкой **DataContext** страницы в только что заполненный объект Address. Результатом будет объект для привязки к элементам управления.

### Шаг 3. Инкапсуляция разметки и логики в элемент управления UserControl

На самом деле, нам наверняка понадобится наличие домашнего адреса (Home Address), рабочего адреса (Work Address) и адреса выставления счета (Billing Address), и неплохо было бы иметь комбинацию клавиш для каждого из адресов. Это можно реализовать двумя способами:

* продублировать xaml на стартовой странице нашего приложения и его фоновый код для каждого экземпляра (home, work, billing);
* перестроить (инкапсулировать) ранее созданную разметку и код в независимый элемент управления (отдельный UserControl).

Разумный подход - инкапсулировать стандартный xaml и поддерживающий код в элемент управления путем создания пользовательского элемента управления (UserControl).

Добавим в проект Silverlight User Control. Назовем новый элемент управления *AddressUserControl.xaml*.

К проекту будут добавлены два файла

* AddressUserControl.xaml
* AddressUserControl.xaml.cs

AddressUserControl.xaml выглядит очень знакомо при открытии.

<UserControl x:Class="ControlsExample.AddressUserControl"  
    xmlns="http://schemas.microsoft.com/client/2007"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Width="400" Height="300">  
    <Grid x:Name="LayoutRoot" Background="White">  
 …  
    </Grid>  
</UserControl>

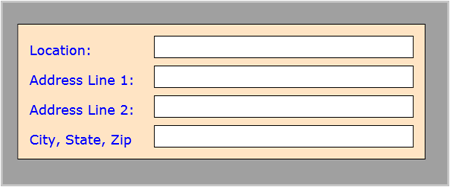
#### Реализация элемента управления

Вернемся к MainPage.xaml и вынесем xaml разметку начиная с Border во вновь созданный UserControl, заменив табличную сетку, созданную там Visual Studio 2010.

Обратите внимание на то, что ваш элемент управления находится в правильном месте, но немного узок. Удалите атрибуты ширины (Width) и высоты (Height) и внешнего пользовательского элемента управления,

<UserControl x:Class="ControlsExample.AddressUserControl"  
    xmlns="http://schemas.microsoft.com/client/2007"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    > **<!--no specific size-->**    <Border BorderBrush="Black"  BorderThickness="1" Margin="15">   
 …

Как только вы удалите их, элемент управления будет выровнен по центру, а также расширен для того, чтобы вместить в себя элементы управления (рис. 7.2).

   
Рисунок 7.2. Автоматическое изменение размера пользовательского элемента управления

#### Добавление кода

Все последующие шаги выполняются в AddressUserControl.xaml.cs: Добавляем переменную экземпляра Address так же, как и в MainPage.xaml.cs. В конструкторе выделяем память для объекта Address и настраиваем обработчик события Loaded. В реализации OnLoaded добавляем новый обработчик для события KeyDown табличной сетки.

public partial class AddressUserControl : UserControl  
{  
    private Address theAddress = null;  
    public AddressUserControl()  
    {  
        InitializeComponent();  
        theAddress = new Address();  
        Loaded += new RoutedEventHandler(OnLoaded);  
    }  
    void OnLoaded(object sender, RoutedEventArgs e)  
    {  
        AddressGrid.KeyDown += new KeyEventHandler(AddressGrid\_KeyDown);  
    }

Реализация AddressGrid.KeyDown была вырезана из MainPage.xaml.cs

void AddressGrid\_KeyDown(object sender, KeyEventArgs e)  
    {  
        if (e.Key == Key.M && Keyboard.Modifiers == ModifierKeys.Control)  
        {  
            theAddress.Location = "Microsoft";  
            theAddress.Address1 = "One Microsoft Way";  
            theAddress.Address2 = "Building 10";  
            theAddress.City = "Redmond, WA 98052";  
        }  
        this.DataContext = theAddress;  
    }  
   }

### Использование пользовательского элемента управления

Мы завершили создание пользовательского элемента управления, но от него нет толку пока вы его не расположите в файле MainPage.xaml. В самой верхней части MainPage.xaml добавьте пространство имен для вашего AddressUserControl’a:

xmlns:controls="clr-namespace:ControlsExample"

Теперь вы можете заменить ваш пользовательский элемент управления для всего содержимого Border в MainPage.xaml. Если вы еще не удалили Border, то удалите и замените своим элементом управления, точно также как вы вставили бы любой другой элемент управления в StackPanel,

<controls:AddressUserControl x:Name="HomeAddress" />

#### Многократное использование

Давайте добавим еще один пользовательский элемент управления к StackPanel и изменим подсказки таким образом, чтобы у нас было следующее

<UserControl x:Class="Example.MainPage"  
    xmlns="http://schemas.microsoft.com/client/2007"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:controls="clr-namespace:ControlsExample"  
    Width="600" Height="500">  
    <StackPanel Background="White">  
        <TextBlock Text="Event Address" FontFamily="Verdana" FontSize="24"  
            HorizontalAlignment="Left" Margin="15,0,0,0"/>  
         <controls:AddressUserControl x:Name="HomeAddress" />  
       <TextBlock Text="Billing Address" FontFamily="Verdana" FontSize="24"  
            HorizontalAlignment="Left" Margin="15,0,0,0"/>  
         <controls:AddressUserControl x:Name="BillingAddress" />             
  </StackPanel>  
</UserControl>

Обратите внимание, что добавление второго экземпляра AddressUserControl требует только наличия двух различных названий. Более того мы смогли вынести весь код из главной страницы приложения (логика которого не относилась к MainPage.xaml.cs)

using System.Windows.Controls;

namespace Example  
{  
    public class MainPage : UserControl  
    {  
        public MainPage()  
        {  
            InitializeComponent();  
        }  
    }  
}

Вся логика была экспортирована и инкапсулирована в пользовательском элементе управления. Вы можете добавить 2 (или 20) пользовательских элемента управления AddressUserControl в ваш интерфейс, не написав и строки кода. При этом, запустив программу, каждый элемент будет работать независимо и каждый поддерживает комбинацию Ctrl + M

## CostomControl

Цель пользовательских элементов управления заключается в предоставлении поверхности визуального проектирования, дополненной шаблоном, что облегчает задачу определения элемента управления в обмен на будущую гибкость. В ситуации, когда функциональность пользовательского элемента управления вполне удовлетворяет, но нужно подстроить его визуальное представление, возникает проблема. Например, предположим, что необходимо использовать ту же самую адресную форму, но предоставить ей другую обложку, которая больше соответствует существующему окну приложения.

Вы будете правы, сказав, что некоторые аспекты пользовательского элемента управления удастся изменить через стили, но часть из них заблокирована внутри, будучи жестко закодированной в разметке.

Решение этой проблемы состоит в создании элемента управления без внешнего вида, т.е. элемента управления на основе одного из базовых классов, не имеющих поверхности визуального проектирования. Такой элемент управления помещает свой код разметки в шаблон по умолчанию, который можно заменить, не затрагивая логики элемента управления.

### Шаг 1. Рефакторинг кода адресной формы

Превратим рассмотренную выше адресную форму в элемент управления без внешнего вида (CustomControl). Первый шаг прост — нужно всего лишь изменить объявление класса, как показано ниже:

public class AdressCustomControl: System.Windows.Controls.Control   
{ ... }

#### Код внутри класса AdressCustomControl

Код точно такой же, как код из пользовательского элемента управления прошлого примера (UserControl) (за исключением того факта, что из конструктора понадобится удалить вызов InitializeComponent()).

Так же необходимо определить **свойства зависимости (DependencyProperty)**, и добавить стандартные оболочки для свойств, которые облегчают доступ к ним и обеспечивают возможность обращения из XAML-разметки. Данный пункт можно было сделать и в версии создаваемого ранее UserControl’a, что собственно лишь еще раз подчеркивает очень тонкую грань между этими двумя понятиями.

#### Определение свойств зависимости

Нужно создать свойства, методы и события, которые будут поступать в наш CustomControl, и на которые будет опираться приложение, использующее его для обработки адреса. В созданном нами CustomControl будут следующие свойства определяющие адрес: Location, Adress1, Adress2, State.

Первый шаг в создании свойства зависимости — это определение статического поля для него с добавленным словом Property в конце его имени:

public static DependencyProperty LocationProperty;

Определение статических полей для свойств — лишь первый шаг. Также понадобится статический конструктор элемента управления, который зарегистрирует свойства, указывая имя свойства, тип данных и класс элемента управления, владеющий данным свойством. Это позволит воспользоваться некоторыми специфическими средствами свойств (вроде наследования значений) за счет передачи объекта FrameworkPropertyMetadata с правильно установленными флагами. Также в этот момент можно присоединить обратные вызовы для проверки достоверности, коррекции значений и уведомлений об изменении значения.

В свойстве Location мы могли бы ввести проверку значения на равенство фразе Microsoft (в случае равенства автоматически изменить значения свойств Adress1, Adress2 и State на нужные нам значения). Ниже приведен код статического конструктора, регистрирующего пять свойств зависимости для нашей адресной формы.

static AdressCustomControl ()   
{   
 LocationProperty = DependencyProperty.Register (   
 "Location", typeof(string), typeof(TextBox),  
 new PropertyChangedCallback (OnLacationChanged));

Adress1Property = DependencyProperty.Register (   
 "Adress1", typeof(string), typeof(TextBox),  
 new PropertyChangedCallback (null));

Adress2Property = DependencyProperty.Register (   
 "Adress2", typeof(string), typeof(TextBox),  
 new PropertyChangedCallback (null));

StateProperty = DependencyProperty.Register (   
 "State", typeof(string), typeof(TextBox),  
 new PropertyChangedCallback (null));

}

Теперь, определив свойства зависимости, можно добавить стандартные оболочки для свойств, которые облегчают доступ к ним и обеспечивают возможность обращения из XAML-разметки:

public string Location  
{  
 get { return (string)GetValue(LocationProperty); }   
 set { SetValue(LocationProperty, value); }   
}

...

Оболочки свойств не должны содержать никакой логики, поскольку свойства могут устанавливаться и извлекаться непосредственно с помощью методов SetValue() и GetValue() базового класса DependencyObject. Например, логика автозаполнения адреса после определения свойства Location в данном примере реализуется с использованием обратного вызова, который инициируются при изменении свойства через его оболочку, либо при прямом вызове SetValue().

private static void OnLocationChanged(DependencyObject sender,  
 DependencyPropertyChangedEventArgs e)   
{   
 string newLocation = (string)e.NewValue;   
 AdressCustomControl adressCustomControl = (AdressCustomControl)sender;   
  
 if(newLocation.Eqals(“Microsoft”)  
 {  
 adressCustomControl.Adress1 = “One Microsoft Way”  
 adressCustomControl.Adress2 = “Building 10”;  
 adressCustomControl.State = “Redmond, WA 98052”;  
 }  
}

Далее предоставим новый стиль для создаваемого CustomControl’a. Этот стиль будет обеспечен новым шаблоном элемента. (Если пропустить этот шаг, будет использован шаблон, определенный в базовом классе.)

Чтобы сообщить о том, что предоставляется новый стиль, следует вызвать метод OverrideMetadata() в статическом конструкторе класса. Этот метод вызывается на свойстве DefaultStyleKeyProperty, которое является свойством зависимости, определяющим стиль по умолчанию для элемента управления. Необходимый код выглядит так:

DefaultStyleKeyProperty.OverrideMetadata(typeof(AdressCustomControl),  
new FrameworkPropertyMetadata(typeof(AdressCustomControl)));

Можно предоставить другой тип, если нужно использовать шаблон другого класса элемента управления, но почти всегда для каждого из собственных пользовательских элементов управления будет создаваться специфический стиль.

### Шаг 2. Рефакторинг кода разметки указателя цвета

После добавления вызова OverrideMetadata понадобится просто подключить правильный стиль. Этот стиль должен быть помещен в словарь ресурсов по имени **generic.xaml**, который следует сохранить в папке Themes проекта. Таким образом, этот стиль будет распознан как стиль по умолчанию для элемента управления. Для добавления файла generic.xaml выполните следующее:

1. Щелкните правой кнопкой мыши на проекте библиотеки классов в окне Solution Explorer и выберите в контекстном меню пункт Add -> New Folder. Назовите новую папку Themes.
2. Щелкните правой кнопкой мыши на папке Themes и выберите в контекстном меню пункт Add -> New Item.
3. В диалоговом окне Add New Item выберите вариант XML file template, введите имя generic.xaml и щелкните на кнопке Add.

Часто библиотека пользовательских элементов управления содержит несколько таких элементов. Чтобы держать их стили отдельно для облегчения редактирования, файл generic.xaml часто использует слияние словарей ресурсов. В следующей разметке показано содержимое файла generic.xaml, который извлекает ресурсы из ресурсного словаря AdressCustomControl.xaml, который необходимо создать в той же папке Themes.

<ResourceDictionary   
 xmlns=[http://schemas.microsoft.com/winfx/200 6/xaml/presentation](http://schemas.microsoft.com/winfx/200%206/xaml/presentation)  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
 <ResourceDictionary.MergedDictionaries>   
 <ResourceDictionary Source="/themes/AdressCustomControl.xaml"/>   
 </ResourceDictionary.MergedDictionaries>   
</ResourceDictionary>

Стиль пользовательского элемента управления должен использовать атрибут TargetType для автоматического присоединения себя к типу. Ниже приведена базовая структура разметки, которая находится в файле AdressCustomControl.xaml:

<ResourceDictionary   
 xmlns=[http://schemas.microsoft.com/winfx/200 6/xaml/presentation](http://schemas.microsoft.com/winfx/200%206/xaml/presentation)  
 xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
 xmlns:local="clr-namespace:Example">  
 <Style TargetType="{x:Type local:AdressCustomControl}">  
 <- Стиль CustomControl ->  
 </Style>   
</ResourceDictionary>

Стиль можно использовать для установки любых свойств в классе элемента управления (независимо от того, наследуются они от базового класса или добавлены вами). Однако наиболее полезная задача, которую выполняет стиль — это применение нового шаблона, определяющего визуальное представление по умолчанию для элемента управления.

Преобразовать обычный код разметки (подобный тому, что используется для нашей адресной формы) в шаблон элемента управления очень легко. Необходимо принять во внимание описанные ниже соображения.

1. При создании выражений привязки, которые связываются со свойствами родительского класса элемента управления, нельзя использовать свойство ElementName. Вместо него нужно применять свойство RelativeSource для указания того, что необходимо привязаться к родительскому элементу управления. Если все, что необходимо — это однонаправленная привязка, обычно можно использовать облегченное расширение разметки TemplateBinding вместо полноценного Binding.
2. Присоединять обработчики событий в шаблоне элемента управления не допускается. Вместо этого потребуется назначить элементам узнаваемые имена и присоединять к ним обработчики событий программно в конструкторе элемента управления.
3. Не именуйте элемент в шаблоне элемента управления, если только не хотите присоединить обработчик событий для программного взаимодействия с ним. При необходимости именования элемента называйте его в стиле PART\_ИмяЭлемента.

Принимая во внимание перечисленные соображения, можно создать следующий шаблон для нашей адресной формы.

<Style TargetType="{x:Type local:AdressCustomControl}">  
 <Setter Property="Template">  
 <Setter.Value>  
 <ControlTemplate TargetType="{x:Type local:AdressCustomControl}">  
 <Border BorderBrush="Black"    
 BorderThickness="1"   
 **Margin="{TemplateBinding Padding}"**>  
 <Grid>  
     <Grid.RowDefinitions>  
         <RowDefinition Height="10" />  
         <RowDefinition Height="auto" />  
         <RowDefinition Height="auto" />  
         <RowDefinition Height="auto" />  
         <RowDefinition Height="auto" />  
         <RowDefinition Height="auto" />  
         <RowDefinition Height="10" />  
     </Grid.RowDefinitions>  
     <Grid.ColumnDefinitions>  
         <ColumnDefinition Width="10" />  
         <ColumnDefinition Width="auto" />  
         <ColumnDefinition Width="10" />  
         <ColumnDefinition Width="auto"/>  
         <ColumnDefinition Width="10" />  
     </Grid.ColumnDefinitions>

<TextBlock Grid.Row="2" Grid.Column="1"  
 Text="Location:"/>   
      <TextBox Grid.Row="2" Grid.Column="3"  
        Text =**"{Binding Path=Location, RelativeSource={  
 RelativeSource TemplatedParent}}"** />  
  
      <TextBlock Grid.Row="3" Grid.Column="1"  
         Text="Address1" />  
      <TextBox Grid.Row="3" Grid.Column="3"  
         Text=**"{Binding Path=Address1, RelativeSource={  
 RelativeSource TemplatedParent}}"** />  
  
      <TextBox Grid.Row="4" Grid.Column="1"  
         Text="Address2"/>  
      <TextBox Grid.Row="4" Grid.Column="3"  
         Text=**"{Binding Path=Address2, RelativeSource={  
 RelativeSource TemplatedParent}}"** />  
  
      <TextBlock Grid.Row="5" Grid.Column="1"  
 Text="City, State, Zip " />  
      <TextBox Grid.Row="5" Grid.Column="3"  
         Text="**{Binding Path=State, RelativeSource={  
 RelativeSource TemplatedParent}}**" />  
 </Grid>  
 </Border>  
 </ControlTemplate>   
 </Setter.Value>   
 </Setter>   
</Style>

Как видите, некоторые выражения привязки заменены расширением **TemplateBinding**. Другие по-прежнему используют расширение Binding, но имеют свойство **RelativeSource**, указывающее на родителя шаблона (пользовательский элемент управления). Хотя и TemplateBinding, и Binding с RelativeSource из TemplatedParent служат одной и той же цели — извлечению данных из свойств пользовательского элемента управления, все же облегченный TemplateBinding более предпочтителен. Однако он не будет работать, если нужна двунаправленная привязка (как в случае с адресом).

### Шаг 3. Оптимизация шаблона элемента управления

В нынешнем виде шаблон элемента управления — адресная форма отвечает всем нуждам, и его можно использовать точно так же, как пользовательский элемент того же назначения. Однако еще остается возможность упрощения этого шаблона за счет удаления некоторых деталей. В данный момент всякий потребитель элемента управления, который пожелает применить специализированный шаблон, будет вынужден возиться с добавлением выражений привязки, чтобы обеспечить продолжение его работы. Это не трудно, но утомительно. Альтернатива состоит в конфигурировании всех выражений привязки в коде инициализации самого элемента управления. В этом случае шаблону не нужно будет указывать эти детали.

Чтобы система заработала, код должен иметь возможность находить нужные ему элементы. Элементы управления (CustomControl) находят требуемые элементы по имени. В результате имена составляющих элементов становятся частью общедоступного интерфейса разрабатываемого элемента управления, а потому должны быть достаточно осмысленными. По существующему соглашению эти имена начинаются с конструкции PART\_, за которой следует собственно имя элемента. В этом имени используются начальные заглавные буквы — как в именах свойств. PART\_AdressTextBox — хороший выбор для именования элемента — элемент для текстового ввода адреса.

Например, ниже показано, как можно модернизировать Template для создаваемого CustomControl:

<TextBlock Grid.Row="2" Grid.Column="1"  
 Text="Location:"/>   
<TextBox Grid.Row="2" Grid.Column="3"  
 x:Name=”PART\_LocationTextBox” />  
  
<TextBlock Grid.Row="3" Grid.Column="1"  
 Text="Address1" />  
<TextBox Grid.Row="3" Grid.Column="3"  
 x:Name=”PART\_Address1TextBox” />  
  
<TextBox Grid.Row="4" Grid.Column="1"  
 Text="Address2"/>  
<TextBox Grid.Row="4" Grid.Column="3"  
 x:Name=”PART\_Address2TextBox” />  
  
<TextBlock Grid.Row="5" Grid.Column="1"  
 Text="City, State, Zip " />  
<TextBox Grid.Row="5" Grid.Column="3"  
 x:Name=”PART\_StateTextBox” />

#### Манипулирование частями шаблона

Подключить выражения привязки можно было бы при инициализации элемента управления, но существует более удачный подход. Существует выделенный метод OnApplyTemplate( ), который должен быть переопределен, если необходимо осуществлять поиск элемента в шаблоне и присоединять обработчики событий или выражения привязки. В этом методе для нахождения нужного элемента применяется метод GetTemplateChild( ) (унаследованный от FrameworkElement). Если элемент для работы не найден, рекомендованный подход заключается в том, чтобы не делать ничего. Дополнительно можете добавить код, проверяющий, что элемент, если он присутствует, относится к корректному типу, и генерирующий исключение, если это не так. (Идея в том, что отсутствие элемента говорит о сознательном исключении определенного средства, в то время как некорректный тип элемента представляет ошибку.)

Ниже показано, как подключить выражение привязки данных к отдельному TextBox в методе OnApplyTemplate():

public override void OnApplyTemplate ()   
{   
 base.OnApplyTemplate ();   
 TextBox location = GetTemplateChild("PART\_LocationTextBox") as TetxBox;  
   
 if (location != null)  
 {  
 Binding binding = new Binding("Location");  
 binding.Source = this;  
 binding.Mode = BindingMode.TwoWay;  
 location.SetBinding(TetxBox. TetxProperty, binding);  
 }  
}

В переопределенной функции OnApplyTemplate() мы имеем возможность подписаться на события (например события из прошлого примера – Ctrl+M). Но не забываем, что предварительно необходимо задать имя элементу управления, в его Template, на событие которого мы хотим подписаться.

#### Документирование частей шаблона

Остался еще последний штрих, который должен быть добавлен к рассматриваемому примеру. В руководствах по проектированию рекомендуется добавлять атрибут TemplatePart к объявлению элемента управления, чтобы документировать используемые в шаблоне имена частей и типы элементов, применяемых для каждой части. Формально этот шаг не является необходимым, но это — фрагмент документации, которая поможет другим разработчикам в использовании вашего класса (и которая может также применяться инструментами визуального проектирования вроде Expression Blend, позволяющими строить специализированные шаблоны элемента управления).

Ниже приведены атрибуты TemplatePart, которые потребуется добавить к классу элемента управления AdressCustomControl:

[TemplatePart(Name="PART\_RedSlider", Type=typeof(RangeBase))]   
[TemplatePart(Name = "PART\_BlueSlider", Type=typeof(RangeBase))]   
[TemplatePart(Name="PART\_GreenSlider", Type=typeof(RangeBase))]   
public class ColorPicker : System.Windows.Controls.Control   
{ ... }

## Ключевые термины

**UserControl** - предоставляет простой способ создания элемента управления.

**CustomControl** – это класс унаследованный от UserControl, который содержит набор полей и методов а также свое представление в виде XAML, которые обеспечивают работу этого контрола. Затем этот контрол может встраиваться в другую страницу (он же самый UserControl).

**DataContext** – получает или задает контекст данных для объекта [FrameworkElement](http://msdn.microsoft.com/ru-ru/library/system.windows.frameworkelement(v=vs.95).aspx), когда он участвует в привязке данных.

**DependencyProperty** – представляет свойство, которое можно задать с использованием методов (например, стили, привязка данных, анимация и наследование).

**Бизнес-объект** (доменные объекты) — это объекты в [объектно-ориентированных компьютерных программах](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), выражающие сущности из модели предметной области, относящейся к программе, и реализующие бизнес-логику программы. Например, программа, управляющая заказами, может содержать такие доменные объекты, как «заказ», «позиция заказа», «счёт-фактура».

## Краткие итоги

Мы рассмотрели разработку пользовательского элемента управления. Вы увидели, как строятся базовые пользовательские элементы управления (UserControl), а также как устроен "золотой стандарт" — основанные на шаблонах элементы, лишенные внешнего вида (CustomControl).

## Набор для практики

**Вопросы:**

1. Назовите основные особенности UserControl
2. Назовите основные особенности CustomControl
3. DependencyProperty и этапы определения
4. Виды привязок в шаблоне пользовательского элемента управления
5. Назначение метода OnApplyTemplate

## Литература

1. Описание класса UserControl

<http://msdn.microsoft.com/ru-ru/library/system.windows.controls.usercontrol.aspx>

1. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 508 – 560.
2. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008 — стр. 123-177.

# Лабораторная работа 3. Пользовательские элементы управления.

**Краткая аннотация лабораторной работы:** в лабораторной работе будут даны задания для самостоятельного выполнения на закрепление пройденной теоретической части лекций 4 - 7, а так же рассмотрен пример, способствующий общему видению решения поставленных в лабораторной работе задач. Для более частных решений опирайтесь на лекции 4 - 7.

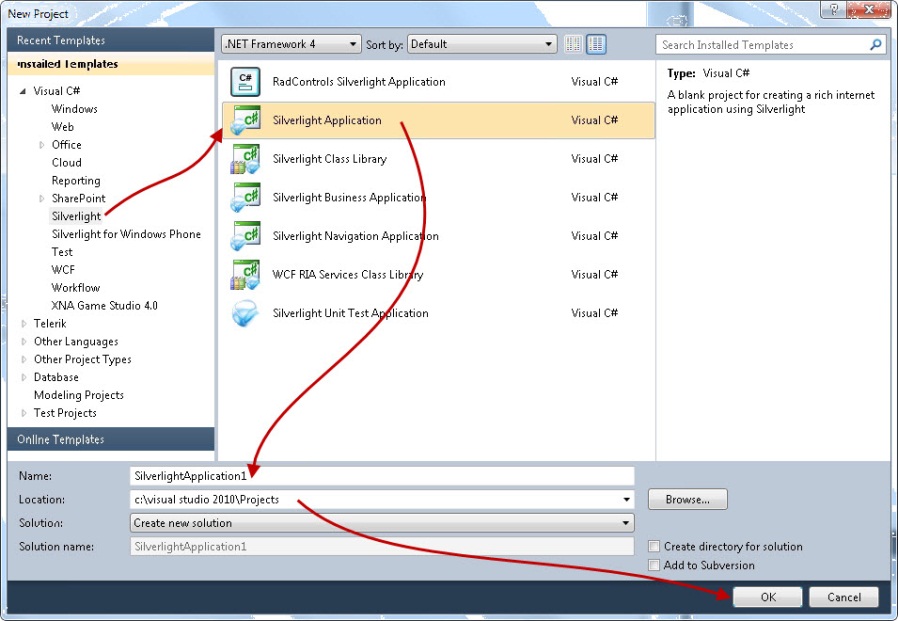
**Цель лабораторной работы:** показать реализацию пользовательского элемента управления двумя путями: средствами дизайнера VisualStudio 2010 и Expression Blend 4. Дать представление о том, какой из вариантов целесообразно использовать в той или иной ситуации. Закрепить знания, полученные в лекциях 4, 5, 6, 7.

## Задания для самостоятельного выполнения

1. Создать пользовательский элемент управления, который позволяет пользователю выбрать временной интервал.
2. Создать пользовательский элемент управления, который позволяет производить конвертацию валют.

## Учебный элемент. Создание пользовательского элемента управления средствами дизайнера VisualStudio

### Шаг 1. Создание Silverlight проекта

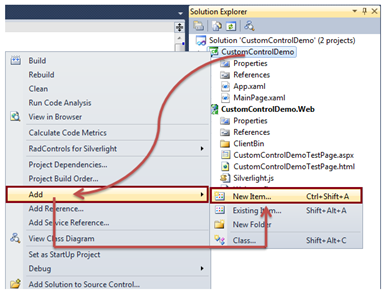
Прежде всего, нам нужно создать Silverlight проект. В левой панели выберите пункт меню Silverlight, а из правой панели "Silverlight Application". Дайте собственное имя для Вашего приложения и решения. Нажмите “ОК” для продолжения (рисунок л. р. 3.1.).  
Рис. л. р. 3.1. Создание нового проекта Silverlight.

Так же на этом этапе (если Вы используете VS2010) имеется возможность выбора целевой версии .NET Framework.

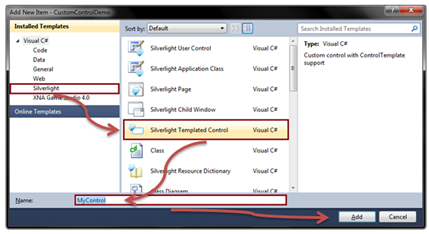
### Шаг 2. Создание пользовательского элемента управления

После того как мы создали проект Silverlight, приступим к созданию пользовательского элемента управления. Обращаем Ваше внимание на то, что в данном учебном элементе, мы создадим самый элементарный пользовательский элемент управления. Предложенное задание для самостоятельного выполнения, немногим сложнее (для его выполнения необходимо опираться на материал предыдущих лекций).

Итак, для создания пользовательского элемента управления щелкните правой кнопкой мыши на ваш проект Silverlight, из контекстного меню выберите пункт "Add" и из контекстного меню второго уровня нажмите кнопку "New item…". Откроется диалоговое окно "Add New Item" (рисунок л. р. 3.2).

  
Рис. л. р. 3.2. Создание нового элемента Silverlight.

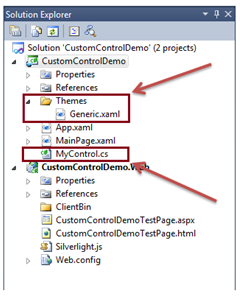
Как показано на скриншоте ниже (рисунок л. р. 3.3), в открывшемся диалоговом окне, необходимо выбрать "Silverlight Templated Control" и задать собственное имя для контрола. Помните, что "Silverlight Templated Control" является шаблоном для пользовательских Silverlight-контролов.

  
Рис. л. р. 3.3. Создание шаблона элемента управления.

Нажмите кнопку "Add", чтобы добавить пользовательский элемент управления в проект. Разверните "Solution Explorer". Там вы найдете следующие вещи:

1. Папку "Themes", содержащую файл с именем "Generic.xaml". Это файл ресурсов по умолчанию для всех ваших стилей элементов управления.
2. Файл "MyControl.cs", который является ничем иным как классом пользовательского элемента управления. Имя класса, по умолчанию, приравнивается имени элемента управления.

Обратите внимание, если Вы создаете несколько элементов управления IDE будет создавать несколько файлов класса контролов, но файл ресурсов будет общим (рисунок л. р. 3.4.).

  
Рис. л. р. 3.4. Solution Explorer после добавления пользовательского элемента управления.

### Шаг 3. Class CustomControlDemo

Рассмотрим созданный класс *CustomControlDemo* более подробно. По умолчанию наш пользовательский элемент управления является наследником от базового класса всех элементов управления - *Control*. Вы можете изменить базовый класс в зависимости от ваших требований. Посмотрите на код, который был создан IDE:

using System.Windows.Controls;  
namespace CustomControlDemo  
{  
 public class MyControl : Control  
 {  
 public MyControl()  
 {  
 this.DefaultStyleKey = typeof(MyControl);  
 }  
 }  
}

Обратите внимание на код внутри конструктора. В конструкторе идет применение стиля из файла ресурсов и установка его в качестве стиля по умолчанию элемента управления.

### Шаг 4. Template CustomControlDemo

Шаблон созданного пользовательского элемента управления Вы можете увидеть в файле *Generic.xaml*. Откройте файл и посмотрите на стиль внутри ResourceDictionary:

<Style TargetType="local:MyControl">  
 <Setter Property="Template">  
 <Setter.Value>  
 <ControlTemplate TargetType="local:MyControl">  
 <Border Background="{TemplateBinding Background}"  
 BorderBrush="{TemplateBinding BorderBrush}"  
 BorderThickness="{TemplateBinding BorderThickness}">  
 </Border>  
 </ControlTemplate>  
 </Setter.Value>  
 </Setter>  
</Style>

Первая строка описывает тип стиля. Вторая строка объявляет шаблон элемента управления. Третья строка устанавливает значение шаблона. На приведенном выше листинге мы видим шаблон, созданный по умолчанию, который мы можем изменить под наши задачи.

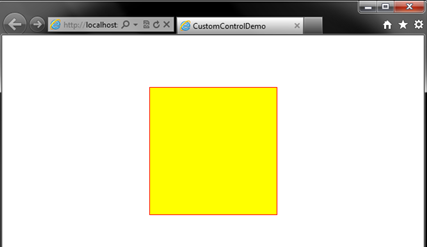
### Шаг 5. Использование Custom Control

Давайте добавим пользовательский элемент управления, который мы создали на нашей главной странице проекта. Чтобы сделать это, Вам нужно добавить XMLNS namespace в XAML. В нашем случае, это имя проекта. Как только Вы объявляете пространства имен, Вы будете иметь доступ к созданному элементу управления.

<UserControl x:Class="CustomControlDemo.MainPage"  
 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>  
 xmlns:CustomControlDemo="clr-namespace:CustomControlDemo"   
 c:Ignorable="d"  
 d:DesignHeight="300" d:DesignWidth="400">  
   
 <Grid x:Name="LayoutRoot" Background="White">  
 <CustomControlDemo:MyControl   
 Width="200"  
 Height="200"  
 Background="Yellow"  
 BorderBrush="Red"  
 BorderThickness="1"/>  
 </Grid>  
</UserControl>

Приведенный выше код говорит сам за себя. Там мы добавили наш пользовательский элемент управления под названием "MyControl" с заданной высотой и шириной. Если Вы не указываете высоту и ширину, то контрол займет весь экран приложения (в приведенном примере).

Как только запустится приложение Вы увидите в браузере прямоугольник -пользовательский элемент управления, который мы создали в этом примере. Меняя шаблон элемента управления, будет меняться и внешний вид соответствующего контрола (рисунок л.р. 3.5).

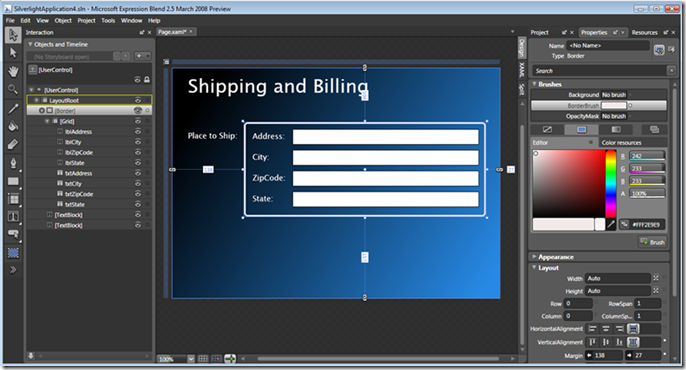
  
Рис. л. р. 3.5. Созданный пользовательский элемент управления.

## Учебный элемент. Создание пользовательского элемента управления средствами дизайнера Expression Blend

В пердыдущем учебном элементе мы рассмотрели, как создавать новый пользовательский элемент управления при помощи диалогового окна VS 2010 “Add New Item“ (добавить новый элемент) с последующим построением UI. Такой способ работы идеален в случае, если вы представляете, как должен выглядеть UI. Также эту задачу можно выполнить и при помощи Expression Blend.

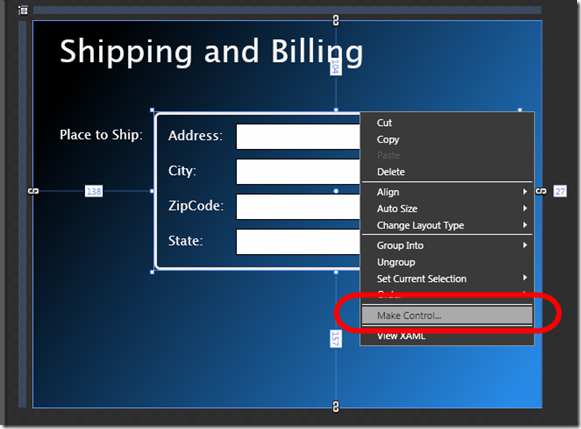
### Шаг 1. Инкапсуляция UI в User Control

В данном примере, мы работаем над формой, который будет позволять прикреплять пользователям информацию об адресах доставки и выставления счета (рисунок л. р. 3.6). Можно начать с создания некоторого UI, который будет принимать информацию об адересе. Что бы это сделать, мы добавим на страницу элементу правления <border> и поместим в него gird layout panel (2 колонки по 4 ряда), а затем поместим labels (метки) и textbox (текстовые поля) там же:

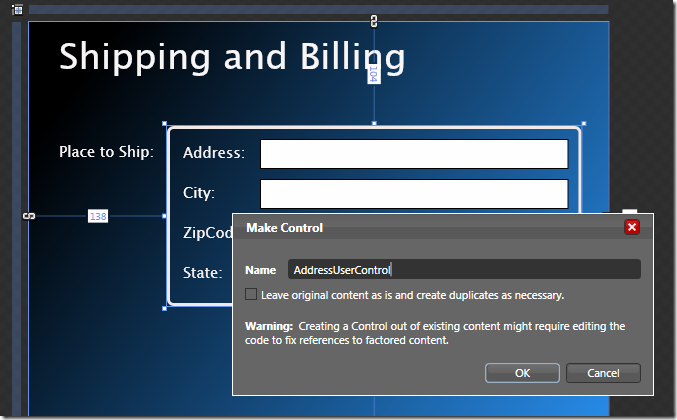
  
Рис. л. р. 3.6. Создаваемая форма в дизайнере Expression Blend.

Можно использовать “Add New Item“ (добавить новый элемент), затем в пустой шаблон контрола при помощи копирования переместить уже собранный UI.

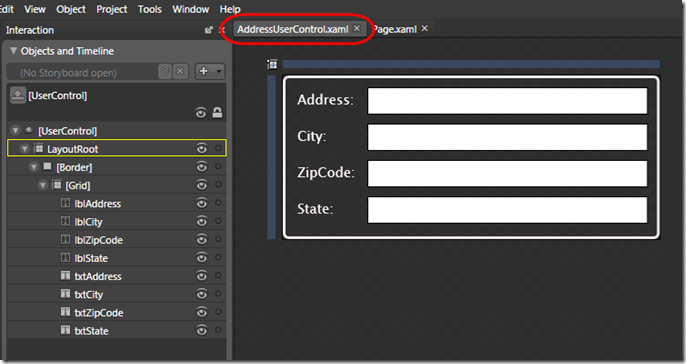
В Blend можно сделать еще проще – выбираем нужные для инкапсуляции элементы урпавления в дизайнере и после щелчка правой кнопкой мыши в появившемся меню щелкаем “Make Control” (сделать элементом управления, рисунок л. р. 3.7):

  
Рис. л. р. 3.7. Процесс инкапсуляции UI

При этом Blend попросит задать имя для нового юзер-контрола (рисунок л. р. 3.8):

  
Рис. л. р. 3.8. Задание имени элементу управления.

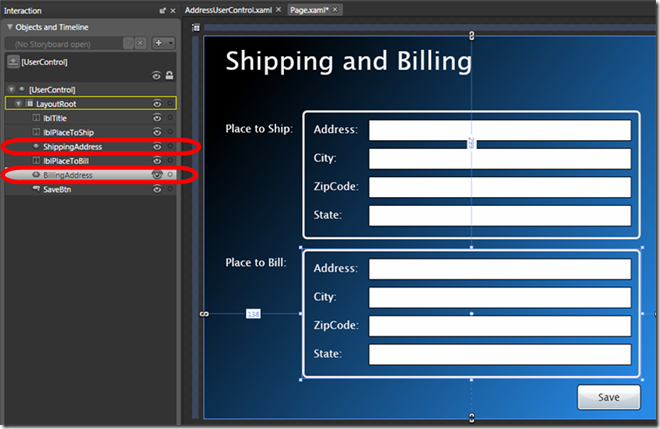
Назовем его “AddressUserControl” и нажимаем ОК. Появится новый юзер-контрол, содержимое которого будет идентично ранее выделенному (рисунок л. р. 3.9):

  
Рис. л. р. 3.9. AddressUserControl.

Когда мы пересоберем весть проект и дойдем до первоначальной страницы, то увидим тот же UI, за исключением того, что пользовательский интерфейс адреса «зашит» в наш AddressUserControl (рисунок л. р. 3.10):

  
Рис. л. р. 3.10. AddressUserControl на первоначальной странице.

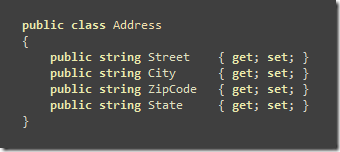
Мы можем переименовать 1 - ый AddressUserControl в “ShippingAddress” (дословно - адрес доставки), а вторую копию превратить в “BillingAddress” (досл. - адрес выставления счета) и использовать их обоих соответственно названиям (рисунок л. р. 3.11).

  
Рис. л. р. 3.11. ShippingAddress и BillingAddress на первоначальной странице.

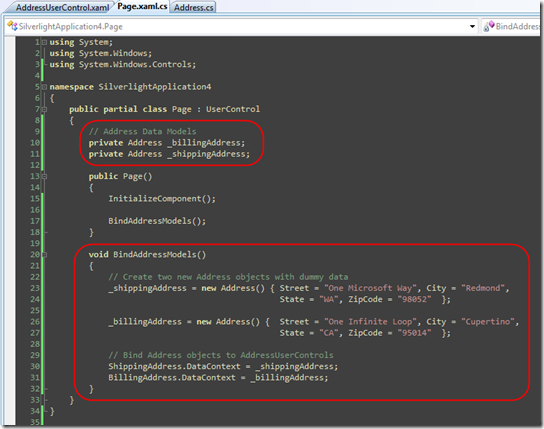
А если мы хотим изменить внешний вид, то изменения будут проводиться единократно и сразу для обоих адреса доставки и выставления счета.

### Шаг 2. Привязка адресов к нашему AddressUserControl – контроллеру.

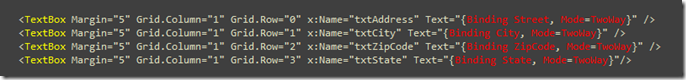
Поскольку с инкапсуляциями покончено, давайте построим класс - модель для вводимой информации (в нашем случае адреса). Зададим класс (рисунок л. р. 3.12) пользуясь удобной функцией «автоматические свойства»:

  
Рис. л. р. 3.12. Класс Address.

В режиме просмотра кода нашего файла Page.xaml мы сразу увидим оба наших объекта (ShippingAddress” и “BillingAddress”), первый будет служить для вставки адресата доставки, а второй – адресата выставления счета (в данном случае мы просто заполним строки «пустой» информацией). Затем привяжем объекты – адреса к нашим элементам управления на странице (рисунок л. р. 3.13). Сделаем мы это, используя свойство “DataContext”, задавая каждому контролу свою модель (1-му- адрес доставки, 2-му – адрес счета):

  
Рис. л. р. 3.13. Установка каждому контролу своей модели через DataContext.

Последним шагом будет добавление выражения {Binding} (привязка) в нашем AddressUserControl.xaml файле, что обеспечит двустороннюю связь между свойствами текста в TextBox и созданной нами моделью информации – адреса, закрепленной нами в качестве контроллера (рисунок л. р. 3.14):

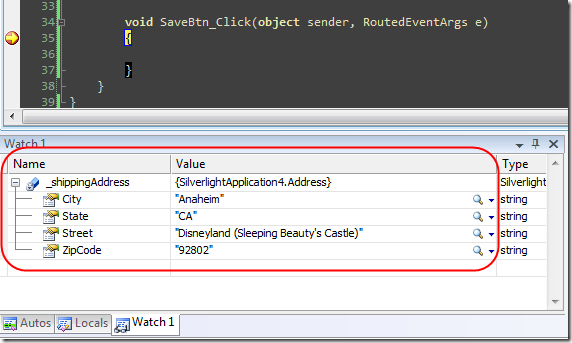
  
Рис. л. р. 3.14. Добавление выражения привязки.

При нажатии F5 для запуска приложения мы увидим, что они связаны (рисунок л. р. 3.15):

  
Рис. л. р. 3.15. Результат созданного проекта с пользовательскими элементами управления.

Поскольку мы задали двустороннюю связь (используя “Mode=TwoWay”), то изменения, вносимые пользователями, будут без специального кода вноситься в модель. Например, мы захотели поменять исходный адрес на адрес Диснейленда:

Устанавливаем breakpoint (рисунок л. р. 3.16) на обработчик “Click” кнопки «Save» (затем на неё нажмём) и видим, как с изменением содержимого TextBox меняется и наша модель “\_shippingAddress”:

  
Рис. л. р. 3.16. Результат работы двусторонней привязки.

Потом можно также реализовать обработчик событий SaveBtn\_Click, чтобы фиксировать модели так, как мы хотим, при этом пальцем не притрагиваясь к UI. Это четкое разделение модели и представления, представленное WPF и поддерживаемоеSilverlight, позволяет и в дальнейшей работе изменять UI адрес – модуля без внесения каких – либо изменений в сам код. Также это упрощает тестовый прогон работоспособности программы.

## Краткие итоги

WPF и Silverlight позволяют легко проводить инкапсуляцию функций UI в элементах управления, и поддерживаемая ими механика пользовательского управления только способствует этому. Привязка вкупе с пользовательским управлением создают границу между моделью/представлением, что позволяет написать рациональный код при работе с информацией.

## Литература

1. Практическое руководство. Создание проекта библиотеки элементов UserControl в WPF  
   <http://msdn.microsoft.com/ru-ru/library/bb514641(v=vs.90).aspx>

# Лекция 8. Использование событий, команд и триггеров в технологиях WPF и Silverlight

**Краткая аннотация лекции:** разбирается способ привязки поведения сущностей к средствам пользовательского интерфейса WPF и Silverlight. Описывается интерфейс ICommand.

**Цель лекции:** команды представляют собой механизм ввода в WPF и Silverlight, обеспечивающий обработку ввода на более семантическом уровне по сравнению с устройствами ввода. Примерами команд являются операции Копировать, Вырезать и Вставить, доступные во многих приложениях. В этом разделе представлены общие сведения о командах, классах, входящих в состав модели команд, а также о порядке использования и создания событий, команд и триггиров в приложениях.

В двух предшествующих лекциях мы рассмотрели, как можно включать в приложение данные и визуализировать их различными способами. До сих пор нас интересовало главным образом, как в приложениях WPF/Silverlight устроен вывод; мы занимались конструированием приложения из элементов управления, которые пользуются визуальными элементами для собственного отображения и менеджерами размещения для позиционирования. Теперь обратим свой взгляд на организацию ввода.

Часто мы хотим, чтобы приложение определенным образом реагировало на движения мышью, нажатия кнопок или клавиш. На платформе Windows Presentation Foundation есть три способа работы с действиями: *события*, *команды* и *триггеры*. В данной лекции мы познакомимся с принципами, применимыми ко всем этим механизмам, а затем углубимся в детали каждого в отдельности.

## Принципиальные основы действий

В том, что касается действий, WPF опирается на три принципа: *композиция элементов*, *слабая связь* и *декларативное описание*. Для того чтобы действия были совместимы со способом построения дерева отображения, в котором одни элементы включают в себя другие, композиция является непременным условием. Тот факт, что элемент управления может радикально изменить свой внешний облик, порождает сложности в случае, когда источник события и код его обработки тесно связаны между собой; поэтому механизм ослабления этой связи также необходим. И, наконец, поскольку декларативная модель программирования пронизывает всю систему, WPF обязана поддерживать и декларативную обработку действий тоже.

### Композиция элементов

Если взглянуть на устройство кнопки, то на самом деле мы увидим, что она составлена из нескольких элементов. Из-за наличия подобной композиции возникает интересная проблема при обработке событий. Памятуя о простоте модели программирования, мы хотим, чтобы код подписки на событие нажатия кнопки был прост и хорошо знаком разработчикам. Мы не должны требовать ничего сверх обычного присоединения обработчика к событию Click:

Button b = new Button();  
b.Content = «Click Me»;  
b.Click += delegate { MessageBox.Show(«You clicked me!»); };

На первый взгляд, выглядит замечательно, но посмотрим, что тут реально происходит. Ведь щелкаем мы не по кнопке, а по какому-то из элементов, из которых кнопка составлена. Чтобы все работало гладко, в WPF введено понятие маршрутизации события, которое позволяет обойти составляющие элементы. Продемонстрировать его мы сможем, добавив в предыдущем примере вторую кнопку, которая будет выступать в качестве содержимого. Щелчок как по первой, так и по второй кнопке приводит к возбуждению события:

Button b = new Button();  
b.Content = new Button();  
b.Click += delegate { MessageBox.Show(«You clicked me!»); };

Получающееся дерево отображение показано на рис. 8.1, из которого также видно, что нам следует озаботиться не только двумя кнопками. Даже в простом случае единственной кнопки композиция событий необходима для того, чтобы маршрутизировать событие мыши до кнопки.

  
Рисунок. 8.1. Дерево отображения (слева) кнопки, в которую вложена другая кнопка (справа)

Композиция элементов отражается на всех аспектах обработки действий, а не только на событиях.

### Слабая связь

Из описания событий в интерфейсе класса Button следует, что он поддерживает не только непосредственные события мыши (MouseUp, MouseDown и т.д.), но и событие Click, которое является абстракцией гораздо более высокого уровня, чем просто событие мыши. Оно возникает и тогда, когда пользователь нажимает пробельную клавишу (при условии, что кнопка владеет фокусом) или клавишу Enter (если это кнопка по умолчанию для окна). Нажатие кнопки – это семантическое событие, тогда как события мыши относятся к физическим.

У написания кода для обработки именно события Click есть два преимущества:

* мы не привязываемся к конкретному жесту пользователя (операции с мышью или клавиатурой);
* мы не ограничиваем себя только кнопкой.

Каждый из элементов CheckBox, RadioButton, Button и Hyperlink поддерживает нажатие. Если обработчик связан с событием Click, то он применим к любому компоненту, который может быть «нажат». Такое отделение кода от действия обеспечивает большую гибкость в реализации обработчиков. Однако самим событиям присуща некая форма зависимости; требуется, чтобы метод-обработчик имел вполне определенную сигнатуру. Например, делегат для обработки события *Button.Click* определен следующим образом:

public delegate void RoutedEventHandler(object sender,RoutedEventArgs e);

Одна из целей WPF – поддержать широкий спектр действий: от тесно связанных физических событий (например, MouseUp) до чисто семантических извещений (например, команды ApplicationCommands.Close, которая сигнализирует о том, что окно должно быть закрыто).

Допуская слабую связь, мы получаем возможность писать шаблоны, которые радикально меняют элемент управления. Например, включив кнопку, ассоциированную с командой Close, мы сможем написать шаблон для окна, который добавляет элемент для закрытия:

<ControlTemplate TargetType=’{x:Type Window}’>  
 <DockPanel>  
 <StatusBar DockPanel.Dock=’Bottom’>  
 <StatusBarItem>  
 <Button **Command=’{x:Static ApplicationCommands.Close}’**>  
 Close  
 </Button>  
 </StatusBarItem>  
 </StatusBar>  
 <ContentPresenter />  
 </DockPanel>  
</ControlTemplate>

Теперь можно заставить окно закрываться, когда любой компонент пошлет команду Close; для этого достаточно добавить в код класса окна соответствующую привязку:

public MyWindow()   
{  
InitializeComponent();  
CommandBindings.Add(  
 new CommandBinding(ApplicationCommands.Close,CloseExecuted));  
}  
void CloseExecuted(object sender, ExecuteRoutedEventArgs e)   
{  
 this.Close();  
}

***Команды – это наименее связанная модель действий в WPF***. Слабая связь обеспечивает полное абстрагирование от источника действия (в данном случае кнопки) и от обработчика действия (в данном случае окна). Мы могли бы изменить стиль окна, воспользовавшись совершенно другим элементом управления, и при этом приложение продолжало бы функцианировать.

### Декларативные действия

Мы видим, что с появлением команд и слабой связи WPF движется в направлении модели, когда программа просто объявляет о своих пожеланиях (например, «я хочу, чтобы окно закрылось, когда вы отдадите эту команду») вместо реализации (например, «вызвать метод Window.Close() при нажатии этой кнопки»).

Одним из столпов WPF является идея декларативного программирования. Помимо визуальных элементов и структуры пользовательского интерфейса, в разметке можно выразить немалую долю программной логики. Декларативная логика особенно полезна, потому что, отталкиваясь от декларативного формата, мы часто можем предоставить более развитые инструменты и, возможно, более содержательные системные службы. Для разных способов обработки действий предусмотрены различные уровни поддержки в декларативной программе. Для событий можно в разметке объявить отвечающую на него функцию, но сам обработчик должен быть реализован в коде. Команды специально задуманы для декларативного использования, поскольку предлагают наилучшее отделение источника действия от его потребителя. У триггеров, пожалуй, самая развитая декларативная поддержка, но им недостает расширяемости, поэтому применять их для решения сложных задач затруднительно.

Все механизмы работы с действиями в той или иной мере поддерживают перечисленные выше принципы. Углубленное изучение действий мы начнем с наиболее известного: событий.

## События

В WPF события ведут себя точно так же, как в любой другой библиотеке классов, входящей в состав .NET. Каждый объект предоставляет набор событий, на которые можно подписаться, определив соответствующий делегат. В WPF имеется дополнительный механизм маршрутизации событий, который позволяет им распространяться вверх по дереву элементов. Существует три вида маршрутизации событий: прямая, всплытие (bubbling) и туннелирование (tunneling). **Прямые события** – это простые события, возникающие от одиночного источника, они почти идентичны стандартным событиям .NET с тем отличием, что регистрируются в системе маршрутизации событий WPF. Некоторые средства платформы (например, триггеры) требуют, чтобы событие было явно зарегистрировано. **Всплывающие и туннельные события** – две стороны одной медали: туннельные события продвигаются от корня дерева к целевому элементу, а всплывающие – в обратном направлении. Обычно эти два вида событий встречаются попарно, причем туннельная версия имеет префикс Preview. Большинство событий ввода (от клавиатуры, от мыши и от пера) имеют как туннельную, так и всплывающую версии, например: MouseRightButtonDown и PreviewMouseRightButtonDown, соответственно.

Написав небольшой пример, в котором создается иерархия элементов и обрабатываются некоторые события, мы сможем увидеть, как соотносятся между собой всплывающие и туннельные события. Ниже создается окно с группирующей рамкой и двумя кнопками:

<Window ...  
 **PreviewMouseRightButtonDown=’WindowPreviewRightButtonDown’  
 MouseRightButtonDown=’WindowRightButtonDown’**>  
 <GroupBox  
 **PreviewMouseRightButtonDown=’GroupBoxPreviewRightButtonDown’  
 MouseRightButtonDown=’GroupBoxRightButtonDown’**>  
 <StackPanel>  
 <Button>One</Button>  
 <Button  
 **PreviewMouseRightButtonDown=’ButtonTwoPreviewRightButtonDown’  
 MouseRightButtonDown=’ButtonTwoRightButtonDown’**>  
 Two  
 </Button>  
 </StackPanel>  
 </GroupBox>  
</Window>

В обработчике каждого события мы можем вывести его имя:

void ButtonTwoPreviewRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«ButtonTwo PreviewRightButtonDown»);  
}  
  
void ButtonTwoRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«ButtonTwo RightButtonDown»);  
}  
  
void GroupBoxPreviewRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«GroupBox PreviewRightButtonDown»);  
}  
  
void GroupBoxRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«GroupBox RightButtonDown»);  
}  
  
void WindowPreviewRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«Window PreviewRightButtonDown»);  
}  
  
void WindowRightButtonDown(object sender, MouseButtonEventArgs e)  
{  
 Debug.WriteLine(«Window RightButtonDown»);  
}

Во время работы этой программы события возникают в следующем порядке:

1. Window PreviewMouseRightButtonDown.
2. Window PreviewMouseRightButtonDown.
3. GroupBox PreviewMouseRightButtonDown.
4. Button PreviewMouseRightButtonDown.
5. Button MouseRightButtonDown.
6. GroupBox MouseRightButtonDown.
7. Window MouseRightButtonDown.

**Туннелирование и всплытие** прекрасно работают для таких встроенных в каждый элемент управления событий, как события мыши. Однако туннелироваться и всплывать может любое событие. Чтобы обеспечить и туннелирование, и всплытие, WPF поддерживает присоединенные события. Подобно тому, как система свойств позволяет присоединять свойства к любому элементу, мы можем присоединить к любому элементу и обработчик событий.

Если мы хотим получать извещения о нажатии любой кнопки в окне, достаточно просто вызвать метод **AddHandler**. У каждого события в WPF имеется свойство типа RoutedEvent, которое во время исполнения ссылается на объект источник. Чтобы присоединить обработчик, мы передаем методу AddHandler объект RoutedEvent и делегат, который необходимо вызвать:

this.AddHandler(Button.ClickEvent, (RoutedEventHandler)delegate   
{  
 MessageBox.Show(«Clicked»);   
});

WPF расширяет базовую модель событий .NET, добавляя систему маршрутизации событий, которая учитывает композицию элементов. Все остальные средства обработки действий построены на базе этой модели маршрутизации.

## Команды

Большинство событий в WPF связаны с деталями реализации конкретных элементов управления: изменился выбор, произошел щелчок, передвинулась мышь и т.д. События хороши, когда нужно выполнить некоторый код в ответ на получение извещения от элемента управления, но часто бывает нужен более абстрактный подход.

Предположим, что нужно реализовать возможность завершения программы. Конечно, необходимо включить соответствующий пункт в меню. Поэтому мы первым делом определим в разметке меню:

<MenuItem Header=’\_File’>  
 <MenuItem Header=’E\_xit’ Click=’ExitClicked’ />  
</MenuItem>

В файле с кодом реализуем обработчик события:

void ExitClicked(object sender, RoutedEventArgs e)  
{  
 Application.Current.Shutdown();  
}

Пока все хорошо, но давайте еще добавим текст, в который входит гиперссылка, позволяющая выйти из программы:

<TextBlock>  
 Вас приветствует моя программа. Если вам надоело, можете  
 <Hyperlink Click=’ExitClicked’>выйти</Hyperlink>.  
</TextBlock>

Вот теперь начинаются неприятности. Мы делаем слишком много предположений о реализации метода ExitClicked, например, что его сигнатура совместима с событием Hyperlink.Click и что он не делает ничего другого, кроме завершения приложения. К тому же, в разметку оказались зашиты произвольно выбранные имена методов из файла с кодом, а дизайнер, который конструирует пользовательский интерфейс, не будет знать, к каким обработчикам событий привязаться.

Для решения этой проблемы и придуманы команды. Они позволяют назначить имя желаемому действию. Чтобы воспользоваться командой, нужно сделать три вещи:

* определить назначение команды;
* написать реализацию команды;
* создать для команды триггер.

Основой всех команд в WPF является довольно простой интерфейс **ICommand**:

public interface ICommand   
{  
 event EventHandler CanExecuteChanged;  
 bool CanExecute(object parameter);  
 void Execute(object parameter);  
}

Метод **CanExecute** позволяет выяснить, находится ли команда в таком состоянии, когда ее можно выполнить. Обычно элементы управления пользуются этим методом, чтобы активировать или деактивировать себя. Иными словами, если ассоциированная с кнопкой команда возвращает false из метода CanExecute, то кнопка деактивируется. Такое обобществление понятия «активен» позволяет нескольким элементам, связанным с одной командой, поддерживать согласованное состояние. Метод Execute основной, его вызов означает выполнение команды. Реализация класса Button (как и любого другого элемента управления, поддерживающего команды) должна включать примерно такой код:

protected virtual void OnClick(RoutedEventArgs e)  
{  
 if (Command != null && Command.CanExecute(CommandParameter))  
 {  
 Command.Execute(CommandParameter);  
 }  
// ... продолжение реализации  
}

Для определения новой команды мы должны реализовать интерфейс ICommand. Поскольку мы хотим, чтобы наша команда закрывала приложение, то можем вызвать метод Shutdown:

public class Exit: **ICommand**  
{  
 public bool CanExecute(object parameter)  
 {  
 return true;  
 }  
  
 public event EventHandler CanExecuteChanged;  
  
 public void Execute(object parameter)  
 {  
 Application.Current.Shutdown();  
 }  
}

Для привязки команды к пункту меню или к ссылке мы указываем в свойстве Command имя команды – Exit:

<MenuItem Header=’\_File’>  
 <MenuItem Header=’E\_xit’>  
 <MenuItem.Command>  
 <l:Exit />  
 </MenuItem.Command>  
 </MenuItem>  
</MenuItem>  
...  
<Hyperlink>  
 <Hyperlink.Command><l:Exit /></Hyperlink.Command>  
 ...  
</Hyperlink>

Так как команда часто вызывается из нескольких мест, принято заводить статическое поле, содержащее экземпляр команды:

public partial class Window1: Window  
{  
 public static readonly ICommand ExitCommand = new Exit();  
 ...  
}

Дополнительный плюс такой реализации заключается в том, что реализацию класса Exit можно скрыть, объявив, что поле имеет тип ICommand. Теперь Exit можно сделать закрытым классом, а в разметке привязаться к статическому полю:

<MenuItem Header=’\_File’>  
 <MenuItem Header=’E\_xit’ Command=’**{x:Static l:Window1.ExitCommand}**’ />  
</MenuItem>

Таким образом, наше окно может раскрывать свою функциональность в виде команд. Однако тут имеется интересная проблема. Сейчас команда Exit реализована на глобальном уровне; мы можем вызвать ее из любого места и тем самым завершить приложение. Предположим, однако, что «exit» должно означать закрытие текущего окна. В таком случае хотелось бы отделить реализацию выхода от определения, как это было с событиями.

Простейший способ добиться такого разделения состоит в том, чтобы прибегнуть к системе событий. Мы можем определить новое событие, связанное с командой, и воспользоваться маршрутизацией для уведомления компонентов:

class Exit : ICommand  
{  
 public static readonly RoutedEvent ExecuteEvent =  
 EventManager.RegisterRoutedEvent(  
 «Execute»,  
 RoutingStrategy.Bubble,  
 typeof(RoutedEventHandler),  
 typeof(Exit));  
...  
}

Поскольку для этого события задана стратегия Bubble, оно будет всплывать от источника. Чтобы возбудить событие, мы изменим реализацию метода Execute, так чтобы он искал текущий элемент (в данном примере мы воспользовались для этой цели методом Keyboard.FocusedElement, но могли бы остановиться на любом механизме обнаружения «текущего»), а затем возбуждал подходящее событие:

public void Execute(object parameter)  
{  
 RoutedEventArgs e =  
 new RoutedEventArgs(Exit.ExecuteEvent, Keyboard.FocusedElement);  
 Keyboard.FocusedElement.RaiseEvent(e);  
}

Это подводит нас к идее привязки команд – возможности отделить реализацию команды от ее назначения. Вернемся к классу Window1 и добавим в него реализацию команды:

public partial class Window1: Window  
{  
 ...  
 public Window1()  
 {  
 InitializeComponent();  
 AddHandler(Exit.ExecuteEvent, ExitExecuted);  
 }  
  
 void ExitExecuted(object sender, RoutedEventArgs e)  
 {  
 this.Close();  
 }  
}

Тут возможно некоторое недопонимание. Напомним, что цель команды – предложить абстракцию того, что должно происходить. В данном случае элемент управления (скажем, MenuItem) вызывает команду в ответ на событие (к примеру, Click). При нашей реализации команда Exit возбудит событие Execute, которое распространится по дереву элементов, а объект Window сможет подписаться на него и выполнить те или иные действия:

1. Пользователь щелкает по пункту меню.
2. MenuItem вызывает метод Execute команды.
3. Реализация команды Exit возбуждает событие Exit.Execute от имени элемента, имеющего фокус (в данном случае MenuItem).
4. Событие всплывает вверх по дереву.
5. Window получает событие Exit.Execute.
6. Window выполняет обработчик события (закрывает окно).

Можно было бы пойти дальше и включить в интерфейс ICommand средства поддержки привязок к вводу (для обработки ввода с клавиатуры, от мыши и пера), параметров и прочего. Однако в каркасе есть встроенный класс RoutedCommand, который большую часть всего этого уже умеет делать. Маршрутизируемые команды позволяют полностью отделить реализацию команды от ее назначения. Паттерн определения новой команды похож на RoutedEvent и DependencyProperty. Команда определяется как статическое свойство; это просто уникальный маркер, обозначающий ее идентичность:

public partial class Window1: Window  
{  
 public static readonly ICommand ExitCommand =  
 new RoutedCommand(«Exit», typeof(Window1));  
 ...  
}

Чтобы при выполнении команды что-то происходило, нужно связать с командой обрабатывающий код (аналогично тому, как мы поступили с событием в предыдущем примере). Маршрутизируемые команды всплывают (как и события), поэтому можно добавить привязку к команде в корневое окно и таким образом увидеть все команды. Для привязки к команде нужно указать вид интересующей команды и код, который будет выполняться при ее поступлении:

public Window1()  
{  
 InitializeComponent();  
 CommandBindings.Add(new CommandBinding(ExitCommand, ExitExecuted));  
}  
  
void ExitExecuted(object sender, ExecutedRoutedEventArgs e)  
{  
 this.Close();  
}

Привязка к команде позволяет решить, следует ли активировать команду, а также с помощью свойства InputBindings отобразить на команды действия по вводу данных:

<Window x:Class=’EssentialWPF.Window1’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’  
 xmlns:l=’clr\_namespace:EssentialWPF’  
 Title=’EssentialWPF’>  
  
<Window.InputBindings>  
 <KeyBinding Key=’A’ Modifiers=’Control’  
 Command=’{x:Static l:Window1.ExitCommand}’ />  
</Window.InputBindings>  
...  
</Window>

Еще одна существенная особенность – это понятие о «безопасных командах». С некоторыми командами, например вырезания, копирования и вставки, сопряжены определенные угрозы. С целью гарантировать, что система выполняет такие операции только по явному запросу пользователя (или если разрешает система управления цифровыми правами), класс RoutedCommand может отслеживать, была ли команда инициирована пользователем.

Использование команд четко отделяет отображение от поведения. Заведя единственное имя, обозначающее семантическое действие, мы обходим многие проблемы тесной связи, которые могут возникнуть при попытке ассоциировать несколько элементов управления с одним обработчиком события. В общем случае логику приложения следует реализовывать в терминах команд, а не обработчиков событий. Во многих типичных ситуациях, когда принято устанавливать прямую ассоциацию с обработчиком события, лучше воспользоваться триггерами. О них мы расскажем чуть ниже.

### Команды и привязка к данным

Одна из наиболее интересных и мощных возможностей команд – это интеграция с привязкой к данным. Поскольку у элементов есть свойства Command и CommandParameter, их можно привязать к данным. А, значит, именно от данных будет зависеть происходящее в программе.

Чтобы понять, как все стыкуется, напишем приложение, которое выводит список всех файлов на диске c:\. Определим простое диалоговое окно со списковым полем и шаблон данных для отображения имени одного файла:

<Window x:Class=’EssentialWPF.DataAndCommands’  
 xmlns=’http://schemas.microsoft.com/winfx/2006/xaml/presentation’  
 xmlns:x=’http://schemas.microsoft.com/winfx/2006/xaml’  
 Title=’Data and Commands’>  
 <ListBox Margin=’2’ Name=’\_files’>  
 <ListBox.ItemTemplate>  
 <DataTemplate>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 </DataTemplate>  
 </ListBox.ItemTemplate>  
 </ListBox>  
</Window>

Затем запишем в свойство ItemsSource список файлов:

public partial class DataAndCommands: Window  
{  
 public DataAndCommands()  
 {  
 InitializeComponent();  
 FileInfo[] fileList = new DirectoryInfo(«c:\\»).GetFiles(«\*.\*»);  
 \_files.ItemsSource = fileList;  
 }  
}

Теперь хотелось бы добавить кнопку для вывода содержимого файла. Но нам не нужно, чтобы запускались произвольные приложения, поэтому следует поставить фильтр на те типы файлов, которые мы согласны загружать. У нас будет две команды: Open и Blocked.

public partial class DataAndCommands: Window  
{  
 public static readonly RoutedCommand OpenCommand =  
 new RoutedCommand(«Open», typeof(DataAndCommands));  
 public static readonly RoutedCommand BlockedCommand =  
 new RoutedCommand(«Blocked», typeof(DataAndCommands));  
 ...  
}

Понадобятся также обработчики этих команд:

public partial class DataAndCommands: Window  
{  
 public DataAndCommands()  
 {  
 InitializeComponent();  
   
 CommandBindings.Add(new CommandBinding(OpenCommand,  
 delegate (object sender, ExecutedRoutedEventArgs e)  
 {Process.Start(«notepad.exe», (string)e.Parameter);}));  
  
 CommandBindings.Add(new CommandBinding(BlockedCommand,  
 delegate (object sender, ExecutedRoutedEventArgs e)  
 {MessageBox.Show((string)e.Parameter, «Blocked»);}));  
 }  
}

Определив обе команды, мы можем модифицировать шаблон данных, включив в него кнопку. Можно воспользоваться привязкой к данным для задания параметра команды (имени файла). Что касается самой команды, то, поскольку мы хотим, чтобы для одних элементов списка вызывалась команда OpenCommand, а для других – BlockedCommand, то реализуем интерфейс IValueConverter для перехода от имени файла к объекту ICommand:

<DataTemplate>  
 <WrapPanel>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 <Button CommandParameter=’{Binding Path=FullName}’>  
 <Button.Command>  
 <Binding>  
 <Binding.Converter>  
 <l:FileToCommandConverter />  
 </Binding.Converter>  
 </Binding>  
 </Button.Command>  
 Show  
 </Button>  
 </WrapPanel>  
</DataTemplate>

Конвертер может проанализировать данные и решить, какую команду выполнять. Например, можно проверить расширение файла:

public class FileToCommandConverter: IValueConverter  
{  
 public object Convert(object value,  
 Type targetType, object parameter, CultureInfo culture)  
 {  
 string ext = ((FileInfo)value).Extension.ToLowerInvariant();  
 if (ext == «.txt»)  
 {  
 return DataAndCommands.OpenCommand;  
 }  
 return DataAndCommands.BlockedCommand;  
 }  
...  
}

Запустив эту программу, мы убедимся, что можно просматривать содержимое только текстовых файлов. Этот пример слишком прост; того же самого можно было бы достичь с помощью метода CanExecute с последующим блокированием выполнения команды для «плохих» файлов. Но важно подчеркнуть, что мы могли бы вернуть любую команду. Таким образом, поведение элемента определяется данными.

Команды обеспечивают слабую связь между пользовательским интерфейсом и поведением. Кроме того, команды открывают возможность подойти к определению поведения приложения на основе данных. Часть поведения относится не к логике самого приложения, а лишь к манипуляциям над визуальным отображением его состояния. Например, когда пользователь проводит мышью над кнопкой, кнопка должна подсвечиваться. Такую логику легко реализовать с помощью команд или обработчиков событий, но, коль скоро это поведение перенесено в код, управлять им с помощью инструментальных средств становится очень трудно, и, следовательно, мы вновь приходим к тесной связи между отображением и поведением. Именно для решения этой проблемы и предназначены триггеры.

## Триггеры

Триггер может сработать по одному из трех условий:

* изменение состояния свойства отображения (Trigger);
* изменение состояние свойства данных (DataTrigger);
* событие (EventTrigger).

Все три типа триггеров при срабатывании запускают некоторую последовательность действий. Существует также два типа триггеров для наборов: MultiTrigger и MultiDataTrigger. Триггерами можно пользоваться только внутри шаблона или стиля. Объекты Trigger и EventTrigger допустимы внутри шаблона элемента управления или стиля, а объекты DataTrigger – только внутри шаблона данных.

### Добавление триггеров к данным

В шаблоне данных есть два способа связать элемент отображения с частью модели данных. Мы уже видели, как это можно сделать с помощью привязки, причем дополнительно можно еще воспользоваться конвертером для преобразования значения из модели данных в нечто понятное элементам отображения.

Класс **DataTrigger** дает декларативный способ задать действия, которые следует выполнить для указанных значений из модели данных. Тем самым DataTrigger – это, по сути дела, простой конвертер значений, определенный в разметке. С помощью привязки он получает значение из модели данных, а когда это значение соответствует заранее заданному, использует последовательность объектов **Setter** и **EventSetter**.

Для демонстрации возьмем последний пример из раздела, посвященного командам, и переделаем конвертер значений в DataTrigger. Вместо того чтобы пользоваться конвертером в привязке к свойству кнопки Command, мы привяжем к нему значение по умолчанию (в данном случае BlockedCommand):

<DataTemplate>  
 <WrapPanel>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 <Button Command=’{x:Static l:DataAndTriggers.BlockedCommand}’  
 CommandParameter=’{Binding Path=FullName}’>  
 Show  
 </Button>  
 </WrapPanel>  
</DataTemplate>

Далее следует описать триггер данных, который будет срабатывать, когда файл имеет расширение «.txt»:

<DataTemplate>  
 <WrapPanel>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 <Button Command=’{x:Static l:DataAndTriggers.BlockedCommand}’  
 CommandParameter=’{Binding Path=FullName}’>  
 Show  
 </Button>  
 </WrapPanel>

<DataTemplate.Triggers>  
 <DataTrigger Binding=’{Binding Path=Extension}’ Value=’.txt’/>  
 </DataTemplate.Triggers>  
</DataTemplate>

Если расширение равно «.txt», то свойству Command должно быть присвоено значение OpenCommand. Для этого нужно включить в триггер объект Setter, который и запишет нужное значение в свойство. В данном случае нужно еще задать объект, которому принадлежит свойство, поскольку мы не хотим изменять свойства в модели данных:

<DataTemplate>  
 <WrapPanel>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 <Button x:Name=’\_showButton’  
 Command=’{x:Static l:DataAndTriggers.BlockedCommand}’  
 CommandParameter=’{Binding Path=FullName}’>  
 Show  
 </Button>  
 </WrapPanel>  
  
 <DataTemplate.Triggers>  
 <DataTrigger Binding=’{Binding Path=Extension}’ Value=’.txt’>  
 <Setter TargetName=’\_showButton’ Property=’Command’  
 Value=’{x:Static l:DataAndTriggers.OpenCommand}’ />  
 </DataTrigger>  
 </DataTemplate.Triggers>  
</DataTemplate>

Поскольку DataTrigger может содержать более одного объекта Setter, то легко организовать выполнение нескольких действий, когда элемент данных примет интересующее нас значение. Добавим для примера еще один объект, который будет показывать подлежащую выполнению команду. Определенный порядок вызова установщиков не гарантируется, хотя в текущей версии они применяются в порядке объявления.

<DataTemplate>  
 <WrapPanel>  
 <TextBlock Text=’{Binding Path=Name}’ />  
 <Button x:Name=’\_showButton’  
 Command=’{x:Static l:DataAndTriggers.BlockedCommand}’  
 CommandParameter=’{Binding Path=FullName}’  
 Content=’Block’/>  
 </WrapPanel>  
 <DataTemplate.Triggers>  
 <DataTrigger Binding=’{Binding Path=Extension}’ Value=’.txt’>  
 <Setter TargetName=’\_showButton’ Property=’Command’  
 Value=’{x:Static l:DataAndTriggers.OpenCommand}’ />  
 <Setter TargetName=’\_showButton’ Property=’Content’  
 Value=’Show’ />  
 </DataTrigger>  
 </DataTemplate.Triggers>  
</DataTemplate>

При работе с классом DataTrigger мы можем только сравнивать значения на равенство. Предыдущий пример отличается от варианта с использованием конвертера значений в одной существенной детали: в первоначальной версии для обработки возможных различий в регистре букв в имени файла вызывался метод ToLowerInvariant. Чтобы сохранить эту возможность, мы можем создать простой конвертер значений для приведения к нижнему регистру:

public class ToLowerInvariantConverter: IValueConverter  
{  
 public object Convert(object value, Type targetType,  
 object parameter, CultureInfo culture)  
 {  
 return ((string)value).ToLowerInvariant();  
 }  
 public object ConvertBack(object value, Type targetType,  
 object parameter, CultureInfo culture)  
 {  
 return value;  
 }  
}

Подключить конвертер к триггеру очень просто:

<DataTrigger Value=’.txt’>  
 <DataTrigger.Binding>  
 <Binding Path=’Extension’>  
 <Binding.Converter>  
 <l:ToLowerInvariantConverter />  
 </Binding.Converter>  
 </Binding>  
 </DataTrigger.Binding>  
 <Setter TargetName=’\_showButton’ Property=’Command’  
 Value=’{x:Static l:DataAndTriggers.OpenCommand}’ />  
 <Setter TargetName=’\_showButton’ Property=’Content’  
 Value=’Show’ />  
</DataTrigger>

С помощью DataTrigger мы можем перенести всю зависящую от пользовательского интерфейса логику (привязку нужной команды) в разметку, оставив в коде лишь самый минимум (преобразование строк). Перенос логики в разметку упрощает создание качественных инструментов для отображения и четко отделяет интерфейс от логики приложения.

### Добавление триггеров к элементам управления

Цель класса **ControlTemplate** – позволить полностью изменять внешний вид элемента, не меняя его логику:

<ControlTemplate TargetType=’{x:Type Button}’>  
 <Border x:Name=’border’>  
 <ContentPresenter />  
 </Border>  
 <ControlTemplate.Triggers>  
 <Trigger Property=’IsPressed’ Value=’true’>  
 <Setter TargetName=’border’ Property=’Background’ Value=’Red’ />  
 </Trigger>  
 </ControlTemplate.Triggers>  
</ControlTemplate>

DataTrigger тут помогает только частично: мы можем декларативно выполнить преобразование данных и привязку. Но для отображения элемента управления нужна большая гибкость. Класс Trigger дополняет функциональность DataTrigger, добавляя свойства EnterActions и ExitActions. Следовательно, в ответ на переход свойства из одного состояния в другое мы можем, например, начинать и заканчивать анимацию. С другой стороны, объект EventTrigger получает извещение от события (к примеру, MouseEnter или Loaded) и позволяет управлять несколькими анимациями.

## Краткие итоги

В этой главе мы рассмотрели, как добавить поведение в ответ на событие, инициированное пользователем или системой, с помощью различных моделей действий: событий, команд и триггеров.

## Набор для практики

Вопросы:

1. Маршрутизируемые события. Их вида и назначения.
2. Команды. Интерфейс ICommand. Реализация команд.
3. Триггеры. Виды триггеров и их особенности.

## Литература

1. Описание класса Trigger

http://msdn.microsoft.com/ru-ru/library/system.windows.trigger.aspx

1. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011., стр. 262 – 287.
2. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008 — стр. 361-366.

# Лабораторная работа 4. Использование команд в технологиях WPF и Silverlight

**Краткая аннотация лабораторной работы:** в лабораторной работе будут даны задания для самостоятельного выполнения на закрепление пройденной теоретической части лекции 8, а так же рассмотрены 3 примера, способствующие общему видению решения поставленной в лабораторной работе задач. Для более частных решений опирайтесь на лекцию 8.

**Цель лабораторной работы:** рассмотреть примеры практического использования стандартных и нестандартных команд. Перевод команд из неактивного состояния в активное и обратно. В качестве примеров команд мы рассмотрим операции Копировать, Вырезать и Вставить, доступные во многих приложениях.

## Задания для самостоятельного выполнения

На основании рассмотренных в лабораторной работе примеров разработать WPF приложение «Блокнот». Позволяющее с помощью меню и горячих клавиш редактировать текст (копировать, вырезать, вставлять, изменять шрифт).

## Учебный элемент. Использование стандартных и нестандартных команд

### Шаг 1. Использование стандартных команд.

**Задача.** Вы хотите, чтобы программа реагировала на команды меню, щелчки по панели инструментов и клавиатурные комбинации.

**Решение.** Технология WPF включает в себя целый ряд стандартных обработчиков команд, реагирующих на события стандартных элементов управления.

Например, если ассоциировать пункт меню Сору (Копировать) со встроенной командой Сору, она автоматически станет активной, как только фокус перейдет на элемент TextBox.

<Window x: Class ="WpfTextEditor.Windowl"   
 ... >   
 <DockPanel Name="dockPanell" VerticalAlignment="Stretch"   
 HorizontalAlignment="Stretch">   
 <Menu DockPanel.Dock="Top" Height="Auto">   
 <MenuItem Header="\_Edit">   
 <MenuItem Header="\_Copy" Cammand="Copy"/>   
 <MenuItem Header="\_Cut" Cammand="Cut"/>   
 <MenuItem Header="\_Paste" Cammand="Paste"/>   
 </MenuItem>   
 </Menu>   
 <ToolBarTray DockPanel.Dock="Top">   
 <ToolBar >   
 <Button Caranand="ApplicationCaiiinands. Copy">   
 <Image Source="Resources\Copy.png" Opacity="l" />   
 </Button>   
 </ToolBar>   
 </ToolBarTray>   
 </DockPanel>   
</Window>

### Шаг 2. Использование нестандартных команд.

**Задача.** Вы хотите создать собственные команды, специфичные для данного приложения.

**Решение.** Обычно команды сгруппированы в статических классах, чтобы к ним удобнее было обращаться. Определим две команды:

public class WpfTextEditorCommands   
{   
 public static RoutedUICommand ExitCommand;   
 public static RoutedUICommand WordWrapCommand;   
   
 static WpfTextEditorCommands ()   
 {   
 InputGestureCollection exitInputs =   
 new InputGestureCollection();   
 exitlnputs.Add(new KeyGesture(Key.F4, ModifierKeys.Alt) ) ;   
 ExitCommand = new RoutedUICommand("Exit application",   
 "ExitApplication",   
 typeof(WpfTextEditorCommands), exitInputs);   
 WordWrapCommand = new RoutedUICommand("Word wrap", "Wordwrap",   
 typeof (WpfTextEditorCommands) ) ;   
 }   
}

Теперь можно связывать эти команды с элементами управления и закреплять их за обработчиками событий:

public partial class Window1:Window  
{  
 public Window1()   
 {  
 InitializeComponent();   
 // Создать обработчики для наших нестандартных команд   
 CommandBinding cmdBindingExit = new   
 CommandBinding (WpfTextEditorCommands.ExitCommand);   
 cmdBindingExit.Executed += new   
 ExecutedRoutedEventHandler(cmdBindingExit\_Executed);   
 CommandBinding cmdBindingWordWrap = new   
 CommandBinding (WpfTextEditorCommands.WordWrapCommand);   
 cmdBindingWordWrap.Executed += new   
 ExecutedRoutedEventHandler(cmdBindingWordWrap\_Executed);   
 this.CommandBindings.Add(cmdBindingExit);   
 this.CommandBindings.Add(cmdBindingWordWrap);   
 }  
  
 void cmdBindingWordWrap\_Executed(object sender, ExecutedRoutedEventArgs e)  
 {  
 textBox.TextWrapping =   
 ((textBox.TextWrapping == TextWrapping.NoWrap) ?  
 TextWrapping.Wrap : TextWrapping.NoWrap);  
 }  
  
 void cmdBindingExit\_Executed(object sender, ExecutedRoutedEventArgs e)  
 {   
 Application.Current.Shutdown();  
 }  
}

Соберем все вместе с помощью XAML-кода:

<Window x:Class="WpfTextEditor.Windowl"   
 ... >   
 <DockPanel Name="dockPanel1"   
 VerticalAlignment="Stretch"   
 HorizontalAlignment="Stretch">   
 <Menu DockPanel.Dock="Top" Height="Auto">   
 <MenuItem Header="\_Fi1e">  
 <MenuItem Header="\_Exit"   
 Conmmand="local:Wpf TextEditor Commands.ExitCammand"/>  
 </MenuItem>   
 <MenuItem Header="\_View">   
 <MenuItem Header="\_Wordwrap" IsCheckable="True"   
 Name="menuItemWordwrap"   
 Command="local:Wpf TextEditorCommands.WordWrapCammand"/>  
 </MenuItem>   
 </Menu>   
 <ToolBarTray DockPanel.Dock="Top">   
 <ToolBar>   
 <CheckBox IsChecked="{Binding Mode=TwoWay,   
 ElementName=menuItemWordWrap, Path=IsChecked}"  
 Command="local:Wpf TextEditorCommands.WordWrapCammand">  
 <Image Source="Resources\WordWrap.png"  
 OpacityMask="White"/>  
 </CheckBox>   
 </ToolBar>   
 </ToolBarTray>   
 </DockPanel>   
</Window>

Приложение почти готово. На разработку мы потратили минимум усилий и при этом познакомились с такими впечатляющими возможностями WPF, как связывание данных и реализация команд.

### Шаг 3. Перевод команд из неактивного состояния в активное и обратно.

**Задача.** Вы хотите избирательно переводить команды из неактивного состояния в активное и обратно, в зависимости от состояния приложения.

**Решение.** Пусть команда Exit (Выход) становится неактивной, когда в текстовое поле введен текст. Внешне это будет проявляться в том, что пункт меню окрасится в серый цвет, а кнопка панели инструментов перестанет действовать.

public partial class Windowl:Window  
{   
 public Windowl()  
 {  
 InitializeComponent();  
 //Создать обработчики для наших нестандартных команд   
 CommandBinding cmdBindingExit =  
 new CommandBinding (WpfTextEditor Commands.ExitCommand);  
 cmdBindingExit.Executed +=  
 new ExecutedRoutedEventHandler(cmdBindingExit\_Executed);   
 cmdBindingExit.CanExecute +=  
 new CanExecuteRoutedEventHandler(cmdBindingElxit\_CanExecute);  
 //. . .   
 }  
  
 void cmdBindingExit\_CanExecute(object sender, CanExecuteRoutedEventArgs e)  
 {  
 e.CanExecute = textBox.Text.Length == 0;  
 }  
}

## Краткие итоги

Мы убедились, что команды представляют собой механизм ввода в WPF и Silverlight, обеспечивающий обработку ввода на более семантическом уровне по сравнению с устройствами ввода. В качестве примеров команд мы рассмотрели операции Копировать, Вырезать и Вставить, доступные во многих приложениях.

## Литература

1. Знакомство с маршрутизированными событиями и командами в WPF  
   <http://msdn.microsoft.com/ru-ru/magazine/cc785480.aspx>

# Лекция 9. Применение паттерна MVVM как оптимального при проектировании WPF и Silverlight приложений

**Краткая аннотация лекции:** лекция дает краткий обзор паттерна MVVM и описывает, как реализовать его фундаментальные характеристики

**Цель лекции:** рассмотреть ключевые классы паттерна MVVM: *вид, модель представления, модель,* А так же их функциональное взаимодействие.Разобрать интерфейсы ответственные за осуществление привязки данных:INotifyPropertyChanged. и INotifyCollectionChanged. Узнать о различных способах реализации команд в паттерне MVVM. Познакомить с интерфейсом IDataErrorInfo. Дать обзор различных возможностей определять зависимости между представлением, моделью представления, и классами модели.

## Паттерн Model-View-ViewModel (MVVM)

**MVVM** поможет вам разделить бизнес-логику и логику представления приложения от его пользовательского **интерфейса (UI)**. Поддержание разделения между логикой приложения и UI помогает решать проблемы разработки и проектирования и может сделать ваше приложение намного более лёгким для тестирования, поддержания, и развития. Это может также значительно улучшить возможности повторного использования кода и позволяет разработчикам и дизайнерам UI более легко сотрудничать при разработке соответствующих частей приложения.

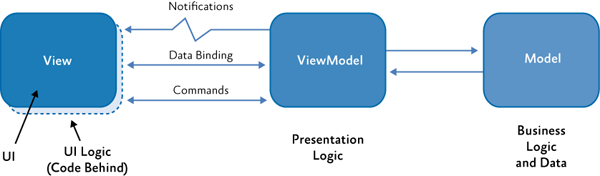
Используя паттерн MVVM, UI приложения, нижележащее представление и бизнес-логика разделяются на три отдельных класса: *вид*, который инкапсулирует UI и его логику; *модель представления*, которая инкапсулирует логику представления и состояния; и *модель*, которая инкапсулирует бизнес-логику приложения и данные.

## Обязанности и характеристики классов.

Паттерн MVVM является близкой разновидностью паттерна Presentation Model, оптимизированного для усиления некоторых базовых возможностей WPF и Silverlight, таких как привязка данных, шаблоны данных, команды, и поведения.

В паттерне MVVM представление инкапсулирует UI и любую логику UI, модель представления инкапсулирует логику представления и состояния, и модель инкапсулирует бизнес-логику и данные. Представление взаимодействует с моделью представления посредством привязки данных, команд и событий уведомления об изменениях. Запросы модели представления, наблюдатели и координаты обновляют модель. Преобразования, валидация и агрегация данных необходимы для отображения данных в представлении.

На рисунке 9.1 показаны три класса MVVM и их взаимодействие.

  
Рисунок 9.1. Взаимодействие ключевых классов MVVM.

## Класс представления (View).

Ответственность представления состоит в том, чтобы определить структуру и появление того, что пользователь видит на экране. В идеале, code-behind (фоновый код) представления содержит только конструктор, который вызывает метод InitializeComponent. В некоторых случаях, code-behind может содержать код логики UI, который реализует визуальное поведение, являющееся трудным или неэффективным для выражения в XAML, такое как сложные анимации, или когда код должен непосредственно управлять визуальными элементами, являющимися частью представления. Недопустимо помещать код логики, нуждающийся в тестировании, в представление. Как правило, код логики в code-behind представления может быть протестирован через автоматизацию UI.

В Silverlight и WPF, выражения привязки данных в представлении вычисляются по отношению к его контексту данных. В MVVM модель представления устанавливается в качестве контекста данных представления. Модель представления реализует свойства и команды, к которым представление может быть привязано и уведомляет представление о любых изменениях состояния через события уведомления об изменениях. Обычно есть непосредственное отношение между представлением и его моделью представления.

Как правило, представления наследуются от классов *Control* или *UserControl*. Однако, в некоторых случаях, представление может быть представлено шаблоном данных, который определяет элементы UI, которые будут использоваться, чтобы визуально представить объект. Используя шаблоны данных, разработчик может легко задать, как модель представления будет представлена, или может изменить её визуальное представление по умолчанию, не изменяя базовый объект или его поведение непосредственно.

Подводя итоги, у представления есть следующие ключевые характеристики:

1. Представление является визуальным элементом, таким как окно, страница, пользовательский элемент управления или шаблон данных. Представление определяет элементы управления, их компоновку и стиль.
2. Представление ссылается на модель представления через свое свойство DataContext. Элементы управления в представлении привязаны к свойствам и командам модели представления.
3. Представление может настроить поведение привязки данных между представлением и моделью представления. Например, представление может использовать преобразователи значений, чтобы отформатировать данные, которые будут показаны в UI, или использовать правила валидации, чтобы предоставить дополнительную проверку вводимых пользователем данных.
4. Представление задаёт и обрабатывает визуальное поведение UI, такое как анимации или переходы, которые могут быть инициированы изменением состояния модели представления или через взаимодействие пользователя с UI.
5. Code-behind представления может определить логику UI, чтобы реализовать визуальное поведение, которое трудно выразить в XAML, или которое требует прямых ссылок на определенные элементы управления UI, определенные в представлении.

## Класс модели представления (View Model).

Модель представления в паттерне MVVM инкапсулирует логику представления и данные для отображения. У него нет никаких прямых ссылок на представление или любое знание о реализации или типе представления. Модель представления реализует свойства и команды, к которым представление может привязать данные и уведомляет представление о любых изменениях состояния через события уведомления. Свойства и команды, которые предоставляет модель представления, определяют функциональность, полагающуюся UI, но представление определяет, как эта функциональность должна будет представлена.

Как правило, модель представления определяет команды или действия, которые могут быть представлены в UI и вызваны пользователем. Типичный примером является то, когда модель представления предоставляет команду Submit, которая позволяет пользователю передать данные веб-сервису или репозитарию данных. Представление может представить эту команду как кнопку так, чтобы пользователь мог нажать её для передачи данных. Как правило, когда команда становится недоступной, ее связанное представление UI становится отключенным. Команды дают способ инкапсулировать пользовательские действия и чисто отделить их от их визуального представления в UI.

В итоге, модель представления имеет следующие ключевые характеристики:

1. Модель представления является неотображаемым классом и не наследуется ни от какого базового класса WPF или Silverlight. Она инкапсулирует логику представления, необходимую для поддержки пользовательских действий в приложении. Модель представления является тестируемой независимо от представления и модели.
2. Модель представления обычно непосредственно не ссылается на представление. Она реализует свойства и команды, к которыми представление может привязать данные. Она уведомляет представление о любых изменениях состояния через события уведомления через интерфейсы INotifyPropertyChanged и INotifyCollectionChanged.
3. Модель представления координирует взаимодействие представления с моделью. Она может преобразовать или управлять данными так, чтобы они могли быть легко использованы представлением, и может реализовать дополнительные свойства, которые, возможно, не присутствуют в модели. Она может также реализовать валидацию данных через интерфейсы IDataErrorInfo или INotifyDataErrorInfo.
4. Модель представления может определить логические состояния, которые представление может визуально представить пользователю.

### Представление или Модель Представления?

Часто определение того, где должна быть реализована определенная функциональность, не очевидно. Общее правило гласит: Что-либо касающееся определенного визуального отображения UI на экране и что может быть модернизировано позже (даже если вы в настоящий момент не планируете модернизировать это), должно войти в представление; что-либо, что важно для логического поведения приложения, должно войти в модель представления. Кроме того, так как у модели представления не должно быть никаких явно заданных знаний об определенных визуальных элементах в представлении, код, чтобы программно управлять визуальными элементами в пределах представления должен находиться в code-behind представления или инкапсулироваться в поведении. Точно так же код для получения или управления элементами данных, которые должны быть показаны в представлении посредством привязки данных, должен находиться в модели представления. Например, цвет выделения выбранного пункта в поле списка должен быть определен в представлении, но список элементов для отображения, и ссылка на выбранный пункт непосредственно, должны быть определены моделью представления.

## Класс модели (Model).

Модель в паттерне MVVM инкапсулирует бизнес-логику и данные. Бизнес-логика определяется как любая логика приложения, которая касается извлечения и управления данными приложения и для того, чтобы удостовериться, что налагаются бизнес-правила, которые гарантируют непротиворечивость и валидность данных. Чтобы максимизировать возможности повторного использования, модели не должны содержать никой специфичной для отображения или использования логики или поведения.

Как правило, модель реализует средства, которые облегчают привязку к представлению. Это обычно означает, что поддерживаются уведомления об изменениях свойств или коллекций через интерфейсы INotifyPropertyChanged и INotifyCollectionChanged. Классы моделей, которые предоставляют наборы объектов обычно наследуются от класса ObservableCollection<T>, который обеспечивает реализацию интерфейса INotifyCollectionChanged. Модель может также поддерживать валидацию данных и сообщение об ошибках через интерфейсы IDataErrorInfo (или INotifyDataErrorInfo). Эти интерфейсы позволяют уведомить WPF и Silverlight, когда значения привязанных данных изменяются для обновления UI. Они также разрешают поддержку валидации данных и сообщения об ошибках в уровне UI.

У модели есть следующие ключевые характеристики:

1. Классы модели являются не визуальными классами, которые инкапсулируют данные приложения и бизнес-логику. Они ответственны за управление данными приложения и за обеспечение их непротиворечивости и валидности, инкапсулируя необходимые бизнес-правила и логику подтверждения правильности данных.
2. Классы модели непосредственно не ссылаются на классы представления или модели представления и не имеют никакой зависимости от того, как эти классы реализуются.
3. Классы модели обычно предоставляют уведомления об изменении свойств или коллекций через интерфейсы INotifyPropertyChanged и INotifyCollectionChanged. Это позволяет им быть легко привязанными к представлению. Классы модели, которые представляют коллекции объектов, обычно наследуются от класса ObservableCollection <T>.
4. Классы модели обычно обеспечивают валидацию данных и сообщение об ошибках через интерфейсыIDataErrorInfo или INotifyDataErrorInfo.
5. Классы модели обычно используются вместе со службой или репозитарием, который инкапсулирует доступ к данным и кэширование.

## Взаимодействие классов.

Паттерн MVVM обеспечивает чистое разделение между пользовательским интерфейсом вашего приложения, его логикой представления, и его бизнес-логикой и данными, разделяя каждый элемент на отдельные классы. Поэтому, когда вы реализуете MVVM, важно разделить код вашего приложения на корректные классы. Хорошо разработанные классы представления, модели представления, и модели будут не только инкапсулировать корректный тип кода и поведения; они также будут разработаны так, чтобы легко взаимодействовать с друг другом через привязку данных, команды, и интерфейсы подтверждения правильности данных.

Взаимодействие между представлением и его моделью представления являются, возможно, самыми важными для рассмотрения, но взаимодействия между классами модели и модели представления также важны. Далее в лекции описываются различные образцы этих взаимодействий и описывают, как их разрабатывать, реализовывая образец MVVM в ваших приложениях.

## Привязка данных.

Привязка данных играет очень важную роль в паттерне MVVM. WPF и Silverlight предоставляют мощные возможности для привязки данных. Ваша модель представления и (в идеале) ваши классы модели должны быть разработаны так, чтобы поддерживать привязку данных. Как правило, это означает, что они должны реализовать корректные интерфейсы.

Привязка данных Silverlight и WPF поддерживает разнообразные режимы. С односторонней привязкой данных, элементы управления UI могут быть связаны с моделью представления так, чтобы они отражали значения базовых данных при отрисовке. Двухсторонняя привязка данных автоматически обновит базовые данные, когда пользователь изменит их в UI.

Чтобы гарантировать, что UI обновляется, когда данные изменяются в модели представления, она должна реализовать соответствующий интерфейс уведомления об изменениях. Если она определяют свойства, к которым могут быть привязаны данные, то она должна реализовать интерфейс **INotifyPropertyChanged**. Если модель представления представляет коллекцию, она должна реализовать **INotifyCollectionChanged** или наследоваться от класса **ObservableCollection <T>**, который реализует этот интерфейс. Оба этих интерфейса определяют событие, которое генерируется всякий раз, когда базовые данные изменяются. Любые связанные элементы управления будут автоматически обновлены, когда эти события будут сгенерированы.

Во многих случаях, модель представления содержит свойства, которые возвращают объекты (которые, в свою очередь, могут содержать свойства, которые возвращают дополнительные объекты). Привязка данных WPF и Silverlight поддерживает привязку к вложенным свойствам через свойство **Path**.

### Реализация INotifyPropertyChanged.

Реализация интерфейса INotifyPropertyChanged в классах модели представления или модели позволяет им уведомлять об изменении значений связанные элементы управления в представлении. Реализация этого интерфейса является явной, как показано в следующем примере.

public class Questionnaire : INotifyPropertyChanged  
 private string favoriteColor;  
 public event PropertyChangedEventHandler PropertyChanged;  
 ...  
 public string FavoriteColor  
 {  
 get { return this.favoriteColor; }  
 set  
 {  
 if (value != this.favoriteColor)  
 {  
 this.favoriteColor = value;  
 if (this.PropertyChanged != null)  
 {  
 this.PropertyChanged(this,  
 new PropertyChangedEventArgs("FavoriteColor"));  
 }  
 }  
 }  
 }  
}

### Реализация INotifyCollectionChanged.

Ваш класс модели представления или модели может представлять коллекцию элементов, или может иметь одно или более свойств, возвращающих коллекцию. В другом случае вы, вероятно, будете хотеть отобразить коллекцию в **ItemsControl**, таком как **ListBox**, или в **DataGrid** в представлении. Эти элементы управления могут быть связанными с моделью представления, которая предоставляет коллекцию или свойство, возвращающее коллекцию через свойство **ItemSource**.

<DataGrid ItemsSource="{Binding Path=LineItems}" />

Чтобы должным образом поддерживать запросы уведомления об изменении, классы модели представления или модели, если его предоставляет коллекция, должны реализовать интерфейс **INotifyCollectionChanged** (в дополнение к интерфейсу INotifyPropertyChanged). Если класс модели представления или модели определяют свойство, которое возвращает ссылку на коллекцию, возвращаемый класс коллекции должен реализовывать интерфейс INotifyCollectionChanged.

Однако, реализация интерфейса INotifyCollectionChanged может быть сложной, потому что она должна посылать уведомления, когда элементы добавляются, удаляются, или изменяются в пределах коллекции. Вместо того, чтобы непосредственно реализовать интерфейс, часто легче использовать или наследоваться от класса коллекции, который уже реализует его. Класс **ObservableCollection<T>** обеспечивает реализацию этого интерфейса и обычно используется или в качестве базового класса, или в свойствах, представляющих коллекцию элементов.

Если вы должны создать коллекцию для привязки данных к представлению, и вам не нужно отслеживать выбор пользователя или поддерживать фильтрацию, сортировку, или группировку элементов в коллекции, можно просто определить свойство в своей модели представления, которое возвращает ссылку на экземпляр ObservableCollection<T>.

public class OrderViewModel: INotifyPropertyChanged  
{  
 public OrderViewModel( IOrderService orderService )  
 {  
 this.LineItems = new ObservableCollection<OrderLineItem>(  
 orderService.GetLineItemList() );  
 }  
   
 public ObservableCollection<OrderLineItem> LineItems { get; private set;}  
}

Если вы получаете ссылку на класс коллекции (например, от другого компонента или службы, которая не реализует INotifyCollectionChanged), можно обернуть эту коллекцию в экземпляр ObservableCollection<T>, используя один из конструкторов, которые принимают в качестве параметра **IEnumerable <T>** или **List<T>**.

### Реализация ICollectionView.

Предыдущий пример кода показывает, как реализовать простое свойство модели представления, которое возвращает коллекцию элементов, которые могут быть показаны через связанные элементы управления в представлении. Поскольку ObservableCollection<T> класс реализует интерфейс INotifyCollectionChanged, элементы управления в представлении будут автоматически обновлены при добавлении или удалении элементов.

Часто вы будете должны более точно управлять тем, как коллекция элементов отображается в представлении, или отслеживать взаимодействие пользователя с отображаемой коллекцией элементов непосредственно внутри модели представления. Например, вы, возможно, должны позволить коллекции элементов фильтроваться или сортироваться согласно логике представления, реализованной в модели представления, или вы, возможно, должны отследить выбранный в настоящий момент пункт в представлении так, чтобы команды, реализованные в модели представления, могли действовать на выбранный элемент.

WPF и Silverlight поддерживают такие сценарии, предоставляя различные классы, которые реализуют интерфейс **ICollectionView**. Этот интерфейс обеспечивает свойства и методы, чтобы позволить коллекции быть отфильтрованной, отсортированной, или сгруппированной, и позволяет отследить или изменить элемент, выбранный в настоящий момент. И Silverlight и WPF предоставляют реализации этого интерфейса, Silverlight даёт класс **PagedCollectionView**, и WPF – класс **ListCollectionView**.

Классы представления коллекции работают, обертывая базовую коллекцию элементов так, чтобы они могли обеспечить автоматическое отслеживание выбранного элемента, сортировку, фильтрацию, и оповещение. Экземпляр этих классов может быть создан программно или декларативно в XAML, используя класс **CollectionViewSource**.

## Команды.

В дополнение к предоставлению доступа к данным, которые будут отображены или отредактированы в представлении, модель представления, вероятно, определит одно или более действий или операции, которые могут быть выполнены пользователем. В WPF и Silverlight, действия или операции, которые пользователь может выполнить через UI, обычно определяются как команды. Команды обеспечивают удобный способ представить действия или операции, которые могут быть легко связаны со средствами управления в UI. Они инкапсулируют код, реализующий необходимые действие, и помогают сохранить его отделённым от визуального представления.

Команды могут быть визуально представлены и вызваны пользователем различными способами. В большинстве случаев, они вызываются в результате щелчка мышью, но они могут также быть вызваны в результате нажатий сочетания клавиш, сенсорных жестов, или любых других событий ввода. Элементы управления в представлении связаны с командами модели представления так, чтобы пользователь мог вызвать их, используя любое входное событие, определённое элементом управления . Взаимодействие между элементами управления UI в представлении и командой может быть двухсторонним. В этом случае, команда может быть вызвана взаимодействием пользователя с UI, и UI может быть автоматически включён или отключён при включении или отключении базовой команды.

Модель представления может реализовать команды как **Command Method**, или как **Command Object** (объект, который реализует интерфейс **ICommand**). В любом случае, взаимодействие представления с командой может быть определено декларативно, не требуя сложного кода обработки событий в code-behind представления. Например, определенные элементы управления в WPF и Silverlight поддерживают команды и предоставляют *свойство Command*, которое может *быть связано с объектом ICommand модели представления*. В других случаях, поведение команды может использоваться для связи элемента управления с методом команды или объектом команды модели представления.

### Реализация Command Objects.

Объект команды является объектом, который реализует интерфейс **ICommand**. Этот интерфейс определяет метод **Execute**, который инкапсулирует необходимую работу, и метод **CanExecute**, который указывает, может ли команда быть вызвана в определённое время. Оба этих метода принимают единственный аргумент в качестве параметра для команды. Инкапсуляция логики реализации необходимой работы в объекте команды означает, что она может легче тестироваться и поддерживаться.

Реализация интерфейса ICommand является понятной, более подробно она рассматривалась в предыдущих лекциях.

## Подтверждение правильности данных и сообщение об ошибках.

Ваша модель представления или модель часто будут обязаны выполнять валидацию данных и сигнализировать о любых ошибки правильности данных представлению так, чтобы пользователь мог их исправить.

Silverlight и WPF оказывают поддержку для управления ошибками подтверждения правильности данных, которые происходят когда изменяются отдельные свойства, связанные с элементами управления в представлении. Для отдельных свойств, которые связываются с элементом управления, модель представления или модель могут сигнализировать ошибку подтверждения правильности данных в пределах метода set свойства, отклоняя входящее плохое значение и выдавая исключение. Если свойство **ValidatesOnExceptions** на привязке данных будет равно *true*, то механизм привязки данных в WPF и Silverlight обработает исключение и покажет пользователю визуальный индикатор, что есть ошибка правильности данных.

Однако, выдачу исключений свойствами таким образом, нужно избегать где только возможно. Альтернативный подход должен реализовать интерфейсы **IDataErrorInfo** или **INotifyDataErrorInfo** (данный интерфейс рассматриваться не будет, в связи с тем, что он поддерживается в настоящий момент только в Silverlight 4 и не доступен в WPF 4 и как следствие его применение в синхронной разработке двух типов приложения является не столь актуальным) в классах модели представления или модели. Эти интерфейсы позволяют модели представления или модели выполнять проверку правильности данных для значений одного или более свойств и возвращать сообщение об ошибке представлению так, чтобы пользователь мог быть уведомлен об ошибке.

### Реализация IDataErrorInfo.

Интерфейс **IDataErrorInfo** предоставляет базовую поддержку валидации данных и сообщения об ошибках. Он определяет два свойства только для чтения: свойство индексатора, с названием свойства в качестве параметр индексатора, и свойство **Error**. Оба свойства возвращают строковое значение.

Свойство индексатора позволяет модели представления или классу модели предоставить сообщение об ошибке, определенное для именованного свойства. Пустая строка или нулевое возвращаемое значение указывают представлению, что измененное значение свойства допустимо. Свойство Error позволяет модели представления или классу модели предоставить сообщение об ошибке для всего объекта. Отметьте, что это свойство в настоящий момент не вызывается механизмом привязки данных Silverlight или WPF.

Свойство индексатора **IDataErrorInfo** вызывают, когда связанное с данными свойство отображается, и всякий раз, когда оно впоследствии изменяется. Поскольку свойство индексатора вызывают для всех свойств, которые изменяются, следует делать все возможное, чтобы гарантировать, что валидация данных настолько быстра и эффективна насколько возможно.

Когда привязывая элементы управления в представлении к свойствам, вы хотите проверить их через интерфейс **IDataErrorInfo**, установите свойство **ValidatesOnDataErrors** в привязке данных в **true**. Это гарантирует, что механизм привязки данных запросит информацию об ошибке из связанного свойства.

<TextBox  
 Text="{Binding Path=CurrentEmployee.Name, Mode=TwoWay,   
 ValidatesOnDataErrors=True, NotifyOnValidationError=True }"  
/>

## Создание и соединение.

Паттерн MVVM помогает вам чисто разделить UI от представления, бизнес-логики и данных, поэтому реализация правильного кода в правильном классе является важным первым шагом в эффективном использовании паттерна MVVM. Управление взаимодействием между классами модели представления и представления посредством привязки данных и команд является также важным аспектом для рассмотрения. Следующий шаг рассмотрит, как представление, модель представления, и классы модели инстанцируются и связываются друг с другом во время выполнения.

Выбор правильной стратегии для этого шага особенно важен, если вы используете контейнер внедрения зависимостей в своем приложении. **Managed Extensibility Framework** (MEF) обеспечивает возможность определять зависимости между представлением, моделью представления, и классами модели и выполнить их контейнером. Более детально применение MEF рассматривается в следующих лекциях.

Как правило, есть непосредственное отношение между представлением и его моделью представления. Модель представления и представление слабо связаны через свойство контекста данных представления. Это позволяет визуальным элементам и поведениям в представлении быть связанными со свойствами, командами, и методами модели представления. Вы должны будете решить, как управлять инстанцированием классов модели представления и представления и их ассоциации через свойство **DataContext** во время выполнения.

Забота должна также быть проявлена при создании и соединении модели представления и представления, чтобы гарантировать, что сохранится слабая связь. Как отмечено в предыдущем разделе, модель представления не должна зависеть ни от какой определенной реализации представления. Точно так же представление не должно зависеть ни от какой определенной реализации модели представления.

Есть множество способов, которыми представление и модель представления могут быть созданы и связаны во время выполнения. Выбор самого подходящего способа для вашего приложения будет в значительной степени зависеть от того, создаете ли вы сначала представление или модель представления, и делаете ли вы это программно или декларативно. Следующие разделы лекции описывают распространенные способы, которыми классы представления и модели представления, могут быть созданы и связаны друг с другом во время выполнения.

### Создание модели представления через XAML.

Возможно, самый простой подход для представления – это декларативно инстанцировать его соответствующую модель представления в XAML. Когда представление будет создано, также будет создан соответствующий объект модели представления. Можно также задать в XAML, что модель представления будет установлена как контекст данных представления.

<UserControl.DataContext>  
 <my:MyViewModel/>  
</UserControl.DataContext>

Этот подход требует, чтобы у вашей модели представления был конструктор по умолчанию (без параметров).

У декларативного создания и присвоения модели представления представлением есть преимущество в том, что он прост и хорошо работает в инструментах времени проектирования, таких как Microsoft Expression Blend или Microsoft Visual Studio. Недостаток этого подхода в том, что у представления есть знание о соответствующей модели представления.

### Создание модели представления программно.

Представление инстанцирует соответствующий экземпляр модели представления программно в конструкторе. Оно может установить его как свой контекст данных, как показано в следующем примере.

public MyView()  
{  
 InitializeComponent();  
 this.DataContext = new MyViewModel();  
}

У программируемого создания и присвоения модели представления в пределах code-behind представления есть преимущество в том, что этот способ прост и хорошо работает в инструментах времени проектирования как Expression Blend или Visual Studio. Недостаток этого подхода в том, что представление должно знать о соответствующем типе модели представления, и что он требует кода в code-behind представления. Используя контейнер внедрения зависимости, такой как MEF, можно помочь обеспечить слабую связь между моделью представления и представлением.

### Создание представления, определенного как шаблон данных.

Представление может быть определено как шаблон данных и связано с типом модели представления. Шаблоны данных могут быть определены как ресурсы, или могут быть определены как встроенные, в пределах элемента управления, который отображает модель представления. «Контент» элемента управления является экземпляром модели представления, и шаблон данных используется для её визуального представления. WPF и Silverlight автоматически инстанцируют шаблон данных и установят его контекст данных в экземпляр модели представления во время выполнения. Этот метод является примером ситуации, в которой в начале инстанцируют модель представления, а уже потом – представление.

Шаблоны данных гибки и легковесны. Разработчик UI может использовать их, чтобы легко определить визуальное представление модели представления, без какого-либо сложного кода. Шаблоны данных ограничиваются представлениям, которые не требуют никакой логики UI (code-behind). Для визуальной разработки и редактирования шаблонов данных, может использоваться Microsoft Expression Blend.

Следующий пример показывает **ItemsControl**, который связан со списком клиентов. Каждый объект потребителя в базовой коллекции является экземпляром модели представления. Представление для клиента определяется встроенным шаблоном данных. В следующем примере представление для каждой потребительской модели представления состоит из **StackPanel** с меткой и текстовым полем, связанным со свойством **Name** модели представления.

<ItemsControl ItemsSource="{Binding Customers}">  
 <ItemsControl.ItemTemplate>  
 <DataTemplate>  
 <StackPanel Orientation="Horizontal">  
 <TextBlock VerticalAlignment="Center" Text="Customer Name: "/>  
 <TextBox Text="{Binding Name}" />  
 </StackPanel>  
 </DataTemplate>  
 </ItemsControl.ItemTemplate>  
</ItemsControl>

Можно также задать шаблон данных как ресурс. Следующий пример показывает шаблон данных, определённый как ресурс и применённый к элементу управления через расширение разметки **StaticResource**.

<UserControl ...>  
 <UserControl.Resources>  
 <DataTemplate x:Key="CustomerViewTemplate">  
 <local:CustomerContactView />  
 </DataTemplate>  
 </UserControl.Resources>  
 <Grid>  
 <ContentControl  
 Content="{Binding Customer}"  
 ContentTemplate="{StaticResource CustomerViewTemplate}">  
 </Grid>  
</Window>

Здесь, шаблон данных обертывает конкретный тип представления. Это позволяет представлению определять поведение code-behind. Таким образом, шаблонный механизм данных может использоваться для того, чтобы поверхностно предоставить связи между представлением и моделью представления. Хотя предыдущий пример показывает шаблон в ресурсах UserControl, его часто помещают в ресурсы приложения для повторного использования.

## Краткие итоги

WPF/Silverlight может предложить разработчикам приложений очень многое, нужно начать мыслить немного иначе, чтобы научиться пользоваться этими возможностями. Шаблон MVVM — простой и эффективный набор рекомендаций для проектирования и реализации приложений WPF/Silverlight. Он позволяет разделять данные, поведение и представление.

## Набор для практики

Вопросы:

1. Паттерн MVVM и его предназначение.
2. Альтернативные решения паттерна MVVM.
3. Роль модели в паттерне MVVM.
4. Роль представдения в паттерне MVVM.
5. Роль модели представления в паттерне MVVM.

## Литература

1. Реализация шаблона Model-View-ViewModel для WPF  
   <http://devlanfear.com/realizaciya-shablona-model-view-viewmodel-dlya-wpf/>
2. Реализация шаблона Model-View-ViewModel средствами MVVM Toolkit  
   <http://megadarja.blogspot.com/2010/04/mvvm-wpf.html>
3. Шаблон представления Model View ViewModel и другие: краткий обзор и сравнение   
   <http://devlanfear.com/shablon-predstavleniya-model-view-viewmodel-i-drugie-kratkij-obzor-i-sravnenie/>

# Лабораторная работа 5. Реализация WPF проекта с помощью MVVM toolkit’а

**Краткая аннотация лабораторной работы:** в лабораторной работе будут даны задания для самостоятельного выполнения на закрепление пройденной теоретической части лекций 8 и 9, а так же рассмотрен пример построения WPF проекта реализованного с использованием «легковесного» MVVM toolkit’а. При решении самостоятельной работы опирайтесь на материалы лекций 8 и 9.

**Цель лабораторной работы:** показать реализацию паттерна Model-View-ViewModel на основе одного из существующего множества MVVM toolkit’ов. Дать представление о том, какой из вариантов реализации паттерна MVVM более уместен в той или иной ситуации. Закрепить знания, полученные в лекциях 8 и 9.

## Задания для самостоятельного выполнения

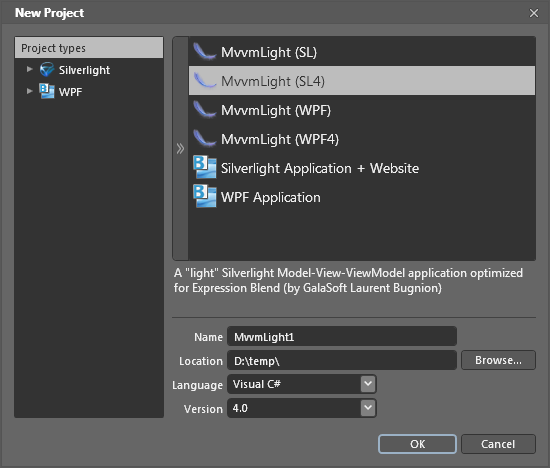
1. Используя созданный в лабораторной работе №1 пользовательский элемент управления, который позволяет задать временной интервал, разработать приложение (реализованное на основе паттерна MVVM) позволяющее планировать распорядок сегодняшнего дня. При разработке приложения допускается использование любого MVVM toolkit’а.
2. Используя созданный в лабораторной работе пользовательский элемент управления, который позволяет производить конвертацию валют, разработать приложение (реализованное на основе паттерна MVVM) позволяющее вести лог денежных операций пользователя. При разработке приложения допускается использование любого MVVM toolkit’а.

## Учебный элемент. Реализация паттерна MVVM средствами Model-View-ViewModel Toolkit 0.1

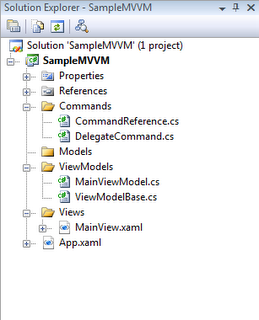
На самом деле MVVM – это просто паттерн, и для его применения не нужны никакие тулкиты. Все тулкиты – это просто хэлперы, которые облегчают Вам жизнь, избавляя от рутины. Согласно Википедии сейчас существует более 14 различных MVVM toolkit’ов.

В лекции №9 вполне детально был рассмотрен паттерн MVVM, поэтому описание использования MVVM toolkit’а в приведенном ниже учебном элементе, будет предельно краткое и схематичное.

Большенство MVVM Toolkit’ов, включают шаблон для Visual Studio и (или) Expression Blend:

  
Рис. л. р. 5.1. Шаблон MVVM toolkit’а для Expression Blend

Если мы реализуем паттерн MVVM средствами Model-View-ViewModel Toolkit 0.1, то данный тулкит сразу создает следующая структура файлов в проекте:

  
Рис. л. р. 5.2. Структура файлов в проекте созданного при помощи MVVM toolkit’а.

### Шаг 1. Постановка задачи

Возьмем какую-нибудь каноничную задачу. Например, отображение списка книг читального зала. У книги есть:

* Название
* Автор
* Доступное количество

Пусть к нам ходят читатели и то и дело берут книги почитать или возвращают их обратно. Требуется в любой момент знать, сколько экземпляров той или иной книги у нас осталось, и можем ли мы ее выдать.

### Шаг 2. Model

В рассматриваемой задаче модель будет состоять из одного простого класса: Book.cs

class Book  
{  
 public string Title { get; set; }  
 public string Author { get; set; }  
 public int Count { get; set; }  
  
 public Book(string title, string author, int count)  
 {  
 this.Title = title;  
 this.Author = author;  
 this.Count = count;  
 }  
}

### Шаг 3. ViewModel

Напишем ViewModel для нашей модели: Book.cs

class BookViewModel: ViewModelBase  
{  
 public Book Book;  
  
 public BookViewModel(Book book)  
 {  
 this.Book = book;  
 }  
  
 public string Title  
 {  
 get { return Book.Title; }  
 set  
 {  
 Book.Title = value;  
 OnPropertyChanged("Title");  
 }  
 }  
  
 public string Author  
 {  
 get { return Book.Author; }  
 set  
 {  
 Book.Author = value;  
 OnPropertyChanged("Author");  
 }  
 }  
  
 public int Count  
 {  
 get { return Book.Count; }  
 set  
 {  
 Book.Count = value;  
 OnPropertyChanged("Count");  
 }  
 }  
}

BookViewModel унаследован от класса ViewModelBase, который предоставляет нам MVVM Toolkit. ViewModelBase в свою очередь, реализует интерфейс INotifyPropertyChanged и содержит функцию OnPropertyChanged. Все это нужно для того, чтобы всегда можно было вызвать событие "изменилось такое-то поле". Как видно в коде, при любом изменении поля мы такое событие вызываем и передаем в качестве параметра его название. Потом на форме биндинг может это событие обработать и, как следствие, интерфейс и ViewModel всегда будут друг с другом синхронизированы.

Помимо BookViewModel у нас есть еще класс MainViewModel, уже сгенерированный и даже связанный с формой. Добавим в него поле:

ObservableCollection<BookViewModel> BooksList { get; set; }

Также слегка изменим конструктор:

public MainViewModel(List<Book> books)  
{  
 BooksList = new ObservableCollection<BookViewModel>  
 (books.Select(b => new BookViewModel(b)));  
}

### Шаг 4. View

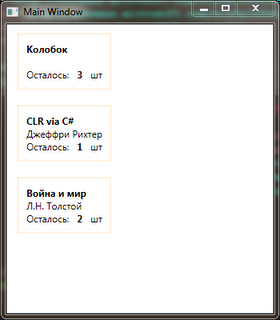
Это и есть наше окно, либо User Control. У любого FrameworkElement-а есть поле DataContext. DataContext может быть любым объектом, иметь какие угодно поля, а его главная задача — являться источником данных для Databinding-а. Форма у нас всего одна, DataContext для нее заполняется в методе OnStartup, что в App.xaml.cs. Немного модифицируем то, что сделал нам MVVM Toolkit, получится следующее: App.xaml.cs

private void OnStartup(object sender, StartupEventArgs e)  
{  
 List<Book> books = new List<Book>()  
 {  
 new Book("Колобок", null, 3),  
 new Book("CLR via C#", "Джеффри Рихтер", 1),  
 new Book("Война и мир", "Л.Н. Толстой", 2)  
 };  
   
 MainView view = new MainView(); // создали View  
 MainViewModel viewModel =   
 new ViewModels.MainViewModel(books); // Создали ViewModel  
 view.DataContext = viewModel; // положили ViewModel во View в качестве  
 // DataContext  
 view.Show();  
}

Осталось написать XAML-код формы. MainView.xaml

<Window x:Class="SampleMVVM.Views.MainView"  
 xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
 xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
 xmlns:c="clr-namespace:SampleMVVM.Commands"  
 Title="Main Window" Height="400" Width="800">  
   
 <ListView ItemsSource="{Binding BooksList}">  
 <ListView.ItemTemplate>  
 <DataTemplate>  
 <Border BorderBrush="Bisque" BorderThickness="1" Margin="10">  
 <StackPanel Margin="10">  
 <TextBlock Text="{Binding Title}" FontWeight="Bold"/>  
 <TextBlock Text="{Binding Author}" />  
 <StackPanel Orientation="Horizontal">  
 <TextBlock Text="Осталось:" />  
 <TextBlock Text="{Binding Count}"   
 FontWeight="Bold" Margin="10,0"/>  
 <TextBlock Text="шт" />  
 </StackPanel>  
 </StackPanel>  
 </Border>  
 </DataTemplate>  
 </ListView.ItemTemplate>  
 </ListView>  
</Window>

Обратите внимание на конструкцию Binding в разметке формы. Именно таким образом можно привязывать поля объекта, находящегося в DataContext-е, к атрибутам контролов. Мы не написали ни строчки кода, но тем не менее при запуске получим следующее:

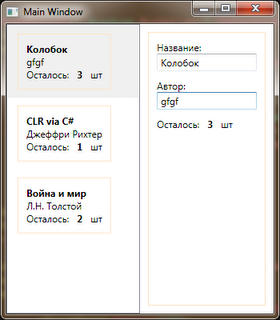
  
Рис. л. р. 5.3. Результат проекта созданного при помощи MVVM toolkit’а.

### Шаг 4. Редактирование

Сделаем так, что для выделенной в списке книги будет открываться редактор. Изменим XAML-разметку формы: MainView.xaml

<Window x:Class="SampleMVVM.Views.MainView"  
 xmlns=<http://schemas.microsoft.com/winfx/2006/xaml/presentation>  
 xmlns:x=<http://schemas.microsoft.com/winfx/2006/xaml>  
 xmlns:c="clr-namespace:SampleMVVM.Commands"  
 Title="Main Window" Height="400" Width="350">  
  
 <Grid>  
 <Grid.ColumnDefinitions>  
 <ColumnDefinition />  
 <ColumnDefinition />  
 </Grid.ColumnDefinitions>  
 <ListView ItemsSource="{Binding BooksList}"   
 IsSynchronizedWithCurrentItem="True">  
 <ListView.ItemTemplate>  
 <DataTemplate>  
 <Border BorderBrush="Bisque" BorderThickness="1"   
 Margin="10">  
 <StackPanel Margin="10">  
 <TextBlock Text="{Binding Title}"   
 FontWeight="Bold"/>  
 <TextBlock Text="{Binding Author}" />  
 <StackPanel Orientation="Horizontal">  
 <TextBlock Text="Осталось:" />  
 <TextBlock Text="{Binding Count}"  
 FontWeight="Bold" Margin="10,0"/>  
 <TextBlock Text="шт" />  
 </StackPanel>  
 </StackPanel>  
 </Border>  
 </DataTemplate>  
 </ListView.ItemTemplate>  
 </ListView>  
   
 <ContentControl Grid.Column="1" Content="{Binding BooksList}">  
 <ContentControl.ContentTemplate>  
 <DataTemplate>  
 <Border BorderBrush="Bisque" BorderThickness="1"   
 Margin="10">  
 <StackPanel Margin="10">  
 <TextBlock Text="Название:"/>  
 <TextBox Text="{Binding Title,   
 UpdateSourceTrigger=PropertyChanged}"  
 Margin="0,0,0,10"/>  
   
 <TextBlock Text="Автор:"/>  
 <TextBox Text="{Binding Author,   
 UpdateSourceTrigger=PropertyChanged}"  
 Margin="0,0,0,10"/>  
 </StackPanel>  
 </Border>  
 </DataTemplate>  
 </ContentControl.ContentTemplate>  
 </ContentControl>  
 </Grid>  
</Window>

Стоит обратить внимание на конструкцию UpdateSourceTrigger=PropertyChanged в строке биндинга. Это значит, что любое изменение, производимое в данном поле, будет немедленно отражаться на источнике. Это легко увидеть:

  
Рис. л. р. 5.4. Результат конструкции UpdateSourceTrigger=PropertyChanged

Если этого не написать, источник будет обновляться только по окончании редактирования (т.е. когда контрол будет терять фокус). Это может привести к следующей ошибке интерфейса: когда жмешь "Сохранить", сохраняется все, кроме только что измененного поля.

### Шаг 5. Команды

Добавим в приложение функциональности. Пусть некие читатели берут книги и возвращают. Соответственно, сделаем две кнопки — «Выдать» и «Забрать», меняющие количество имеющихся в наличии книг. Если книг не осталось (Count = 0), кнопка «Выдать» должна дизаблиться.

В MVVM не пишутся обработчики событий. Функции, которые нужно выполнять контролам, пишутся во ViewModel и биндятся к контролам точно так же, как поля. Только используется механизм команд.

Команда должна представлять из себя экземпляр класса, реализующего интерфейс ICommand. К счастью, MVVM Toolkit снова нам помог и сгенерил целых два таких класса - DelegateCommand для реализации команды без параметров и DelegateCommand<T> - для реализации команды с параметром типа T.

Мы параметры передавать не будем. Код во ViewModel будет таков: BookViewModel.cs

#region Забрать  
  
private DelegateCommand getItemCommand;  
  
public ICommand GetItemCommand  
{  
 get  
 {  
 if (getItemCommand == null)  
 {  
 getItemCommand = new DelegateCommand(GetItem);  
 }  
 return getItemCommand;  
 }  
}  
  
private void GetItem()  
{  
 Count++;  
}  
  
#endregion  
  
#region Выдать  
  
private DelegateCommand giveItemCommand;  
  
public ICommand GiveItemCommand  
{  
 get  
 {  
 if (giveItemCommand == null)  
 {  
 giveItemCommand = new DelegateCommand(GiveItem, CanGiveItem);  
 }  
 return giveItemCommand;  
 }  
}  
  
private void GiveItem()  
{  
 Count--;  
}  
  
private bool CanGiveItem()  
{  
 return Count > 0;  
}  
  
#endregion

Обратите внимание, что этот код добавляется в BookViewModel, а не в MainViewMode. Дело в том, что мы будем добавлять кнопки в ContentControl, DataContext-ом которого является именно BookViewModel.

С первой командой все ясно. Создали команду, и назначили ей в качестве действия метод GetItem, который и будет вызываться при ее активации. Со второй немного интереснее, но тоже просто. Помимо того, что она выполняет некоторое действие, она еще и может проверять с помощью метода CanGiveItem(), может она выполняться или нет.

В XAML-разметку нашей формы добавим следующее

<ContentControl Grid.Column="1" Content="{Binding BooksList}">  
 <ContentControl.ContentTemplate>  
 <DataTemplate>  
 <Border BorderBrush="Bisque" BorderThickness="1" Margin="10">  
 <StackPanel Margin="10">  
 <TextBlock Text="Название:"/>  
 <TextBox Text="{Binding Title,   
 UpdateSourceTrigger=PropertyChanged}"  
 Margin="0,0,0,10"/>  
  
 <TextBlock Text="Автор:"/>  
 <TextBox Text="{Binding Author,   
 UpdateSourceTrigger=PropertyChanged}"  
 Margin="0,0,0,10"/>  
  
 <StackPanel Orientation="Horizontal">  
 <Button Content="Выдать" Command="{Binding   
 GiveItemCommand}" Margin="10,0" />  
 <Button Content="Забрать" Command="{Binding   
 GetItemCommand}" Margin="10,0" />  
 </StackPanel>  
 </StackPanel>  
 </Border>  
 </DataTemplate>  
 </ContentControl.ContentTemplate>  
</ContentControl>

Вот и все. Мы получили требуемую функциональность. Количество экземпляров книги увеличивается и уменьшается, а когда их становится 0, кнопка «Выдать» дизаблится (благодаря упомянутому CanGiveItem).

## Краткие итоги

В приведенном приложении все данные и реализация логики вынесены в отдельное место. В фоновом коде формы мы не добавили ни строчки. XAML понятен. Благодаря паттерну MVVM в коде легко разобраться и его легко сопровождать. Стоит добавить, что рассмотренный MVVM Toolkit содержит весьма полезный набор классов, особенно хорошо подходящий для небольших задач, не требующих модульности или необходимости использовать IoC. Для сложных приложений, можно рекомендовать более «тяжелые» Toolkit’ы, один из них мы будем рассматривать в последующих лекциях.

## Литература

1. MVVM Light Toolkit <http://www.galasoft.ch/mvvm/>
2. WPF Model-View-ViewModel Toolkit <http://wpf.codeplex.com/releases/view/14962>

# Лекция 10. Проектирование приложения с учетом использования единого опыта разработки для настольных и Web-проектов.

**Краткая аннотация лекции:** в лекции рассматривается подходы к разработке Silverlight/WPF приложений с максимальным разделением общего кода и XAML разметки.

**Цель лекции:** показать читателям способы организации проектов, позволяющих использовать разделяемый между Silverlight и WPF код, разобрать особенности архитектуры таких приложений, а также разобрать наиболее часто встречающиеся проблемы и способы их решения.

## Подходы к решению задачи

Как и многие другие вопросы, проблему разделяемого Silverlight/WPF кода можно решить несколькими способами:

* Разрабатывать кроссплатформенный проект с самого начала;
* Сначала разработать продукт для одной платформы, а затем адаптировать его для другой.

### Разработка кроссплатформенного проекта с самого начала

Данное решение оптимально для случая, когда первым приоритетом в разработке стоит качество конечного результата. К сожалению, зачастую в реальном мире приходится жертвовать качеством для быстрого создания готового продаваемого продукта. Сложно спорить с тем, что результата можно добиться раньше, сконцентрировав всё внимание на одной платформе.

### Разработка для одной платформы с последующим портированием на другую

Исходя из сказанного выше, наиболее вероятно, что вам придется следовать этим путем. Здесь перед вами встает выбор: начать с WPF или Silverlight. Безусловно, могут быть определенные стратегические, религиозные или какие-либо другие причины начать с одного или другого, но, пожалуй, в первую очередь стоит обратить внимание на технические аспекты.

Процесс портирования будет проще, если начать разработку с Silverlight и затем перейти на WPF. Следует учесть, что проще не значит лучше. Во многих случаях придется довольствоваться сокращенным функционалом Silverlight, используя сложные конструкции, повторяющие встроенные компоненты WPF.

Если же начать с WPF, то наиболее вероятно, что при портировании на Silverlight придется поплатиться за все удобства WPF и полностью переработать части кода, использующие недостающий в Silverlight функционал.

## Инструментарий

Существует несколько способов организовать синхронное использование кода между Silverlight и WPF версиями приложения так, чтобы при этом было возможно реализовать некоторые участки кода специфично для определенной платформы. Наиболее удобный из них – использование ссылок на файлы для общего функционала и локализация различий при помощи директив препроцессора (в случае если отличается небольшая область кода) и разделяемых классов (для серьезных отличий).

### Создание ссылок на файлы в Visual Studio

Чтобы добавить ссылку на файл в WPF или Silverlight проект в Visual Studio достаточно в меню «Project» вызвать пункт «Add Existing Item…», перейти в папку другого проекта (Silverlight или WPF), выбрать файл и нажать «Add As Link» в меню кнопки «Add».

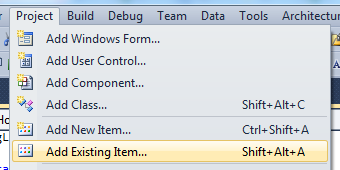


Рисунок 10.1. Меню добавления нового элемента в проект.

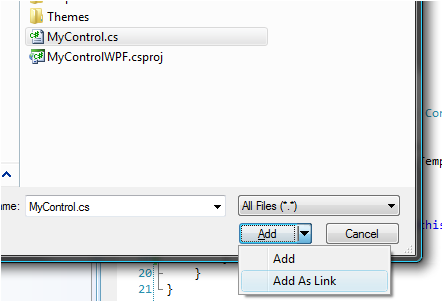


Рисунок 10.2. Выпадающая кнопка для добавления ссылки на файл

### Создание ссылок на XAML файлы

При использовании ссылок на C# или VB файлы компилятор обрабатывает их так же как и при использовании обычных копий файлов. В случае с XAML возникают некоторые проблемы, которые возможно обойти, добавив немного лишнего кода.

Предположим, что существует WPF проект, в котором определен пользовательский элемент управления MyTemplatedControl. Он не содержит никакой логики, всего лишь устанавливает стиль по умолчанию при помощи свойства DefaultStyleKey:

1: public class MyTemplatedControl : Control

2: {

3: public MyTemplatedControl()

4: {

5: this.DefaultStyleKey = typeof(MyTemplatedControl);

6: }

7: }

В файле Generic.xaml, находящемся в папке Themes данного проекта, в стиле по умолчанию объявлен шаблон для данного элемента управления:

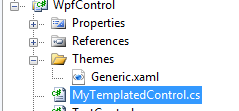


Рисунок 10.3. Пользовательский элемент управления и файл с его стилем по умолчанию.

В соответствии с этим шаблонов элемент управления MyTemplatedControl должен отображать синий прямоугольник – так можно легко проверить, что шаблон успешно найден и применен:

1: <ResourceDictionary

2:xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:local="clr-namespace:TestControl"

5: >

6:

7: <Style TargetType="local:MyTemplatedControl">

8: <Setter Property="Template">

9: <Setter.Value>

10: <ControlTemplate

11: TargetType="local:MyTemplatedControl">

12: <Canvas>

13: <Rectangle Canvas.Left="20"

14: Canvas.Top="20" Width="100"

15: Height="100" Fill="Blue"

16: Stroke="Black" StrokeThickness="3" />

17: </Canvas>

18: </ControlTemplate>

19: </Setter.Value>

20: </Setter>

21: </Style>

22: </ResourceDictionary>

Этот элемент управления добавлен на тестовое окно. Если запустить приложение, то окно будет выглядеть следующим образом:

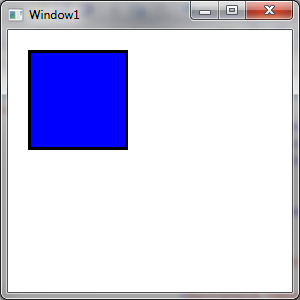


Рисунок 10.4. Тестовое окно.

Предположим, что теперь необходимо создать Silverlight версию данного элемента управления. Для этого создается проект Silverlight, при помощи пункта «Add Existing Item…» и кнопки «Add As Link» добавляются ссылки на файл MyTemplatedControl.cs в корне проекта и файл Generic.xaml в папке Themed. Такой проект удачно скомпилируется, однако если добавить этот элемент управления в Silverlight приложение, на его месте будет показана пустая белая область – очевидно, что шаблон по умолчанию не был применен.

Это происходит так, потому что по какой-то причине (возможно, для этого есть веские основания, или это просто ошибка) XAML компилятор обрабатывает локальные и добавленные по ссылке .xaml файлы по-разному. Если коротко, то для последнего задается упрощенный ресурсный ключ, а значит при поиске стиля по умолчанию файл, добавленный по ссылке, игнорируется средой выполнения, так как он имеет неверный с её точки зрения ресурсный ключ.

Решение данной проблемы – использование объединения словарей при помощи MergedDictionaries: необходимо переместить содержимое файла Generic.xaml в отдельный XAML файл (лучше, если будет создано по отдельному файлу для каждого элемента управления), и включить их при помощи MergedDictionaries в файл Generic.xaml.

MyTemplatedControl.xaml:

1: <ResourceDictionary

2: xmlns=

3: "http://schemas.microsoft.com/winfx/2006/xaml/presentation"

4: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

5: xmlns:local="clr-namespace:TestControl"

6: >

7: <Style TargetType="local:MyTemplatedControl">

8: <Setter Property="Template">

9: <Setter.Value>

10: <ControlTemplate

11: TargetType="local:MyTemplatedControl">

12: <Canvas>

13: <Rectangle Canvas.Left="20"

14: Canvas.Top="20" Width="100" Height="100"

15: Fill="Blue" Stroke="Black" StrokeThickness="3" />

16: </Canvas>

17: </ControlTemplate>

18: </Setter.Value>

19: </Setter>

20: </Style>

21: </ResourceDictionary>

Generic.xaml:

1: <ResourceDictionary

2: xmlns=

3: "http://schemas.microsoft.com/winfx/2006/xaml/presentation"

4: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

5: >

6: <ResourceDictionary.MergedDictionaries>

7: <ResourceDictionary

8: Source="/WpfControl;component/Themes/MyTemplatedControl.xaml" />

9: </ResourceDictionary.MergedDictionaries>

10: </ResourceDictionary>

Затем необходимо добавить файл MyTemplatedControl.xaml в Silverlight проект в качестве ссылки, а файл Generic.xaml – в виде копии, в которую дополнительно вносятся изменения.

Generic.xaml (версия для Silverlight)

1: <ResourceDictionary

2:xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: >

5: <ResourceDictionary.MergedDictionaries>

6: <ResourceDictionary

7: Source="/SlControl;component/MyTemplatedControl.xaml" />

8: </ResourceDictionary.MergedDictionaries>

9: </ResourceDictionary>

Обратите внимание на отличия в файле Generic.xaml для Silverlight:

* Он ссылается на файлы в Silverlight сборке (SlControl);
* Он ссылается на добавленные по ссылке файлы по их «некорректному» ресурсному ключу (обратите внимание на отсутствие ‘Themes’ в пути).

Теперь Silverlight приложение работает правильно:

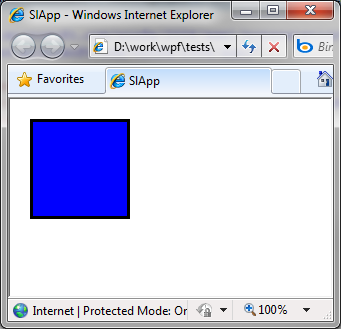


Рисунок 10.5. Корректно работающее тестовое Silverlight приложение.

Таким образом возможно использовать общую XAML разметку в Silverlight и WPF приложениях без необходимости поддерживать 2 различные версии файла. Единственное, что для этого необходимо сделать – создать и поддерживать различные Generic.xaml в каждом проекте.

### Директивы препроцессора

Когда между Silverlight и WPF версиями кода достаточно мало отличий, можно воспользоваться директивами препроцессора, чтобы включить тот или иной блок кода в зависимости от того, в каком проекте компилируется данный файл. Для удобства шаблон проекта Silverlight определяет константу “SILVERLIGHT”. Конечно, возможно определить самостоятельно что-то более короткое и менее сложное в написании, однако надежнее оставить всё как есть – неизвестно, когда и где придется повторно использовать этот код.

Выглядеть это будет следующим образом:

1: private TextBox \_dataTextBox;

2: public override void OnApplyTemplate()

3: {

4: #if SILVERLIGHT

5: \_dataTextBox = this.GetTemplateChild("PART\_DataTextBox")

6: as TextBox;

7: #else

8: \_dataTextBox = this.Template.FindName("PART\_DataTextBox",

9: this) as TextBox;

10: #endif

11: }

### Разделяемые классы

В то время как директивы препроцессора подходят для небольших отличий в коде, по мере увеличения таких блоков код становится достаточно трудночитаем. В таких случаях более подходящими являются разделяемые классы: общий код определяется в части, которая при помощи ссылки разделяется между обоими проектами; методы, реализация которые различна, помещаются в другую часть, специфичную для каждого проекта.

Простейшее приложение, созданное в соответствии с данными принципами, выглядит следующим образом:

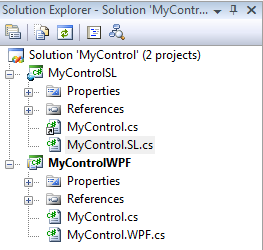


Рисунок 10.6. Простейшее WPF/Silverlight приложение с разделяемыми классами

MyControlWPF – это WPF проект. Он содержит файл MyControl.cs, в котором определен разделяемый класс MyControl. Этот файл состоит из кода, который идентичен и в WPF, и в Silverlight. Он добавлен в MyControlSL (Silverlight проект) в качестве ссылки. Таким образом, достаточно поддерживать одну копию файла, который используется в обоих проектах. MyControl.WPF.cs и MyControl.SL.cs в свою очередь содержат члены класса MyControl, реализация которых различна в WPF и Silverlight.

### Синхронное использование XAML

К сожалению, в случае XAML не существует встроенной поддержки директив препроцессора или схожего приема разделения отличных блоков кода. Конечно, существуют некоторые сторонние решения, однако хоть это и выглядит прогрессивно с точки зрения энтузиаста, кроме случаев действительно больших XAML файлов, сложных в поддержке, обычно более разумно работать с 2 файлами, если различия в разметке не удается обойти. Таким образом, следует либо составлять такую XAML разметку, которая будет корректна как в Silverlight, так и в WPF, и создать ссылку на этот файл в одном из проектов, либо поддерживать 2 параллельные версии файла.

## Решение проблем недостающего функционала

### Отсутствие FrameworkPropertyMetadata

Как известно, в Silverlight отсутствует класс FrameworkPropertyMetadata. Одна из наиболее часто используемых возможностей этого FrameworkPropertyMetadata по сравнению с имеющемся в Silverlight PropertyMetadata – это логические переключатели, которые управляют влиянием свойства на различные аспекты прорисовки объекта, такие как AffectsMeasure, AffectsArrange, и другие. Если один из флагов установлен в истинное значение, соответствующий аспект объявляется недействительным.

К счастью, данное поведение достаточно легко эмулировать при помощи PropertyMetadata и PropertyChangedCallback. Так, если в случае разработки только для WPF можно написать:

1: public static readonly DependencyProperty SomethingProperty

2: = DependencyProperty.Register(

3: "Something", typeof(string), typeof(Window1),

4: new FrameworkPropertyMetadata(string.Empty,

5: FrameworkPropertyMetadataOptions.AffectsMeasure)

6: );

То универсальный WPF/Silverlight код, достигающий такого же эффекта, будет выглядеть следующим образом:

1: public static readonly DependencyProperty SomethingProperty

2: = DependencyProperty.Register(

3: "Something", typeof(string), typeof(Window1),

4: new PropertyMetadata(string.Empty,

5: new PropertyChangedCallback(

6: Window1.SomethingProperty\_Changed)));

7:

8: private static void SomethingProperty\_Changed(

9: DependencyObject d, DependencyPropertyChangedEventArgs e)

10: {

11: ((FrameworkElement)d).InvalidateMeasure();

12: }

Чтобы не допустить копирования кода имеет смысл создать вспомогательный класс, методы которого устанавливают недействительными различные аспекты и их комбинации, и использовать эти методы в обработчиках изменений зависимых свойств, если нет необходимости добавлять в них дополнительную логику.

### Отсутствие приведения значения

WPF имеет достаточно мощный механизм приведения значений зависимых свойств. К сожалению, такого функционала в Silverlight нет. Приведение значений возможно производить при помощи сочетания PropertyChangedCallback и обычных методов класса. В некоторых сложных, но не слишком критичных сценариях возможно имеет смысл использовать приведение значения в WPF версии и не проверять его совсем в Silverlight версии.

### Отсутствие метода OverrideMetadata()

WPF позволяет переопределять метаданные зависимых свойств при помощи метода OverrideMetadata объекта DependencyProperty. В следующем коде переопределяется ширина по умолчанию у элемента управления, наследованного от класса Control:

1: static MyControl()

2: {

3: FrameworkPropertyMetadata newMetadata

4: = new FrameworkPropertyMetadata();

5: newMetadata.DefaultValue = 180.0;

6: Control.WidthProperty.OverrideMetadata(typeof(MyControl),

7: newMetadata);

8: }

Так как в Silverlight такой метод отсутствует, при необходимости изменить значение по умолчанию какого-либо свойства, необходимо просто установить это значение в конструкторе:

1: public MyControl()

2: {

3: this.Width = 180;

4: }

Другая возможность метода OverrideMetadata() – это задание обработчика изменения зависимого свойства базового класса:

1: static MyControl()

2: {

3: #if !SILVERLIGHT

4: FrameworkPropertyMetadata newMetadata

5: = new FrameworkPropertyMetadata();

6: newMetadata.PropertyChangedCallback

7: += MyControl.VisibilityProperty\_Changed;

8: Control.VisibilityProperty.OverrideMetadata(

9: typeof(MyControl), newMetadata);

10: #endif

11: }

После этого в методе VisibilityProperty\_Changed можно обрабатывать все изменения зависимого свойства Visibility.

1: private static void VisibilityProperty\_Changed(

2: DependencyObject d, DependencyPropertyChangedEventArgs e)

3: {

4: // do something here

5: }

К сожалению, т.к. данного метода в Silverlight нет совсем, изящного решения данной проблемы не существует. Однако можно воспользоваться не очень чистым приемом, включающим определение дополнительного зависимого свойства и привязку исходного свойства к нему. Аналог приведенного выше примера для Silverlight приложения в таком случае будет выглядеть следующим образом:

1: #if SILVERLIGHT

2: // объявление дополнительного зависимого свойства, которое

3: // использует такой же PropertyChangedCallback, что и WPF код

4: private static readonly DependencyProperty

5: VisibilityChangedWorkaroundProperty

6: = DependencyProperty.Register(

7: "VisibilityChangedWorkaround", typeof(Visibility),

8: typeof(MyControl),

9: new PropertyMetadata(MyControl.VisibilityProperty\_Changed));

10: #endif

11:

12: public MyControl()

13: {

14: #if SILVERLIGHT

15: // привязка дополнительного зависимого свойства

16: Binding visibilityBnd = new Binding("Visibility");

17: visibilityBnd.Source = this;

18: visibilityBnd.Mode = BindingMode.TwoWay;

19: this.SetBinding(

20: MyControl.VisibilityChangedWorkaroundProperty,

21: visibilityBnd);

22: #endif

23: }

Как видно из примера, определено дополнительное зависимое свойство VisibilityChangedWorkaroundProperty, и в качестве её обработчика изменения указан тот же метод, что и в WPF коде, приведенном выше. Нет необходимости создавать CLR свойство-обертку для дополнительного свойства, и само дополнительное зависимое свойство определено как приватное, так что оно доступно только внутри класса и невидимо для внешних компонент.

Затем в конструкторе дополнительное свойство привязывается к свойству Visibility базового класса. Таким образом, когда свойство базового класса изменится, дополнительное свойство также будет изменено в результате срабатывания привязки, после чего будет вызван обработчик VisibilityProperty\_Changed.

### Отсутствие зависимых свойств только для чтения

Одна из действительно удобных возможностей WPF – определение зависимых свойств только для чтения. Такое свойство полезно, если необходимо отобразить какой-либо статус объекта, определяемый его внутренним состоянием (как, например, свойство IsMouseOver), и разработчик не хочет позволять внешним компонентам изменять данное значение извне. Такого эффекта можно с легкостью достичь при использовании обычных .NET свойств, но что если необходимо создать свойство только для чтения, которое будет обладать всеми возможностями зависимого свойства?

К сожалению, не существует простого решения для Silverlight. Безусловно, можно создать CLR свойство-обертку для зависимого свойства без метода доступа set, однако при этом любой компонент может установить значение свойству при помощи метода SetValue().

Таким образом, в данном случае приходится выбирать: совсем не использовать зависимые свойства только для чтения, или использовать разделение кода (при помощи директив препроцессора или разделяемых классов) между WPF и Silverlight. Следует помнить, что в данном случае придется использовать различный код не только для объявление зависимого свойства, но и в вызове SetValue (т.к. необходимо вызывать SetValue для DependencyPropertyKey в случае зависимого свойства только для чтения). Легко понять, что в таком случае исходный код будет достаточно трудночитаем из-за многочисленных ветвлений препроцессора #if SILVERLIGHT.

### Отсутствие класса Brushes

Класс Brushes в WPF содержит предопределенные сплошные кисти, которые достаточно полезны при использовании кистей в коде.

1: border.Background = Brushes.White;

Данный класс отсутствует в Silverlight. При написании WPF/Silverlight совместимого кода следует использовать класс Colors, который достаточно похож на класс Brushes, только вместо предопределенных кистей содержит предопределенные цвета. Таким образом, код выше может быть переписан на:

1: border.Background = new SolidColorBrush(Colors.White);

Конечно, символов стало больше, однако это лучше, чем дублирование кода из-за настолько небольшого отличия.

При использовании именованных кистей в XAML отличий между Silverlight и WPF нет:

1: <Border Name="border" Background="White">

2: </Border>

### Stroke и Pen

В WPF определен класс Pen, в то время как в Silverlight его нет. Класс Pen достаточно удобно инкапсулирует способ рисования контура фигуры в единственном объекте, который возможно повторно использовать. Так, в WPF он применяется в классе GeometryDrawing (которого, кстати, также нет в Silverlight).

Следует отметить, что во многих случаях даже в WPF его невозможно использовать – вместо него приходится применять свойства StrokeXxx. По-видимому, по этой причине в Silverlight этот класс даже не был реализован.

### Конструкторы по умолчанию для геометрий

Первое, на что придется обратить внимание при работе с геометриями в Silverlight после WPF – отсутствие перегруженных конструкторов. Поэтому, если в WPF было возможно написать:

RectangleGeometry rectangle = new RectangleGeometry(new Rect(10, 10, 50, 40));

то для совместимости WPF/Silverlight данный код придется изменить на:

1: RectangleGeometry rectanble = new RectangleGeometry()

2: {

3: Rect = new Rect(10, 10, 50, 40)

4: };

что занимает не намного больше строк в файле, но зато работает как в Silverlight, так и в WPF.

### Отсутствие свойства No PathSegment.IsStroked

IsStroked позволяет указать, что данный сегмент последовательности линий не должен отображаться. Так, в WPF можно задать следующую последовательность:

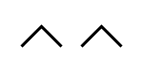


Рисунок 10.7. Простая последовательность линий

при помощи данной XAML разметки:

1: <Path Stroke="Black" StrokeThickness="3">

2: <Path.Data>

3: <PathGeometry>

4: <PathFigure StartPoint="0,20">

5: <LineSegment Point="20,0" />

6: <LineSegment Point="40,20" />

7: <LineSegment Point="60,20" IsStroked="False" />

8: <LineSegment Point="80,0" />

9: <LineSegment Point="100,20" />

10: </PathFigure>

11: </PathGeometry>

12: </Path.Data>

13: </Path>

Несложно догадаться, что в Silverlight данная разметка будет некорректной. Для простых случаев, таких как приведенный выше, возможно обойти это ограничение при помощи простого разделения PathFigure на несколько фигур (по одной на каждый сегмент без разрывов):

1: <Path Stroke="Black" StrokeThickness="3">

2: <Path.Data>

3: <PathGeometry>

4: <PathFigure StartPoint="0,20">

5: <LineSegment Point="20,0" />

6: <LineSegment Point="40,20" />

7: </PathFigure>

8: <PathFigure StartPoint="60,20">

9: <LineSegment Point="80,0" />

10: <LineSegment Point="100,20" />

11: </PathFigure>

12: </PathGeometry>

13: </Path.Data>

14: </Path>

Для эмуляции поведения IsStroked в коде можно воспользоваться методом-расширением, имеющим различные реализации в Silverlight и WPF:

1: public static void AddLineSegment

2: (this PathGeometry pathGeometry, Point point, bool isStroked)

3: {

4: PathFigure figure =

5: pathGeometry.Figures[pathGeometry.Figures.Count - 1];

6:

7: #if SILVERLIGHT

8: if (isStroked)

9: {

10: figure.Segments.Add(new LineSegment { Point = point });

11: }

12: else

13: {

14: // silverlight has no IsStroked properto on LineSegment

15: // so we create a new PathFigure to imitate non-stroked

16: // line segment

17: figure = new PathFigure();

18: figure.StartPoint = point;

19: pathGeometry.Figures.Add(figure);

20: }

21: #else

22: figure.Segments.Add(new LineSegment(point, isStroked));

23: #endif

24: }

Такой метод удобен при создании последовательностей линий в коде:

myPathGeometry.AddLineSegment(myPoint, true);

Этот метод может также быть обобщен на другие типы PathSegment, что можно считать небольшим дополнительным заданием.

Таким приемом можно пользоваться в достаточно простых случаях, однако проблему в следующем примере обойти уже не так тривиально:



Рисунок 10.8. Последовательность линий с заливкой.

В случае WPF такая фигура задается разметкой:

1: <Path HorizontalAlignment="Center" VerticalAlignment="Center"

2: Stroke="Black" Fill="LightGreen" StrokeThickness="3">

3: <Path.Data>

4: <PathGeometry>

5: <PathFigure StartPoint="0,40">

6: <LineSegment Point="0,20" IsStroked="False" />

7: <LineSegment Point="20,0" />

8: <LineSegment Point="40,20" />

9: <LineSegment Point="60,20" IsStroked="False" />

10: <LineSegment Point="80,0" />

11: <LineSegment Point="100,20" />

12: <LineSegment Point="100,40" IsStroked="False" />

13: </PathFigure>

14: </PathGeometry>

15: </Path.Data>

16: </Path>

К сожалению, для достижения такого же эффекта в Silverlight приходится использовать 2 независимые пересекающиеся последовательности – одну для заливки, и другую для собственно отображения линий:

1: <Path Fill="LightGreen" StrokeThickness="3">

2: <Path.Data>

3: <PathGeometry>

4: <PathFigure StartPoint="0,40">

5: <LineSegment Point="0,20"/>

6: <LineSegment Point="20,0" />

7: <LineSegment Point="40,20" />

8: <LineSegment Point="60,20"/>

9: <LineSegment Point="80,0" />

10: <LineSegment Point="100,20" />

11: <LineSegment Point="100,40" />

12: </PathFigure>

13: </PathGeometry>

14: </Path.Data>

15: </Path>

16:

17: <Path Stroke="Black" StrokeThickness="3">

18: <Path.Data>

19: <PathGeometry>

20: <PathFigure StartPoint="0,20">

21: <LineSegment Point="20,0" />

22: <LineSegment Point="40,20" />

23: </PathFigure>

24: <PathFigure StartPoint="60,20">

25: <LineSegment Point="80,0" />

26: <LineSegment Point="100,20" />

27: </PathFigure>

28: </PathGeometry>

29: </Path.Data>

30: </Path>

Очевидно, что для написания совместимой WPF/Silverlight разметки во всех подобных случаях придется использовать Silverlight вариант.

## Краткие итоги

В данной теме были рассмотрены рекомендации к организации проектов, позволяющих разделять Silverlight и WPF код, разобраны особенности архитектуры таких приложений, а также описаны наиболее часто встречающиеся проблемы и способы их решения. Написание кроссплатформенного WPF/Silverlight приложения, безусловно, требует больших усилий, чем раздельная разработка, однако отсутствие необходимости поддерживать 2 различные версии приложения позволяет существенно упростить дальнейшую доработку и сопровождение.

## Набор для практики

Вопросы:

1. Подходы к решению вопроса о разработке Silverlight/WPF приложений с максимальным разделением общего кода. Плюсы и минусы каждого.
2. Создание ссылок в Visual Studio
3. Директивы препроцессора
4. Способы решения проблем недостающего функционала

## Литература

1. Sharing Code Between Silverlight and WPF  
   <http://msdn.microsoft.com/en-us/library/ff921109(v=pandp.40).aspx>
2. Notes from DevReach <http://devblog.ailon.org/devblog/>

# Лекция 11. Реализация паттерна MVVM с использованием IoC-контейнера, как метод избавления от зависимости между компонентами системы.

**Краткая аннотация лекции:** в лекции рассматривается реализация MVVM шаблона с применением инверсии зависимостей и аспектно-ориентированного программирования для максимальной инкапсуляции слоев приложения.

**Цель лекции:** показать читателям на примере фрагментов кода организацию многослойного кроссплатформенного Silverlight/WPF MVVM приложения при помощи IoC контейнера, а также разобрать пример использования Managed Extensibility Framework для разрешения зависимостей между компонентами системы.

## Принцип инверсии зависимостей

Инверсия зависимости – это особый вид инверсии контроля, который применяется в Объектно-ориентированном подходе для удаления зависимостей между классами. Зависимости между классами превращаются в ассоциации между объектами. Ассоциации между объектами могут устанавливаться и меняться во время выполнения приложения. Это позволяет сделать модули менее связанными между собой.

Можно выделить 2 основных принципа инверсии зависимостей:

* Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций;
* Абстракция не должна зависеть от реализации. Реализация должна зависеть от абстракции.

Рассмотрим пример программы, которая копирует в файл данные, введенные с клавиатуры.

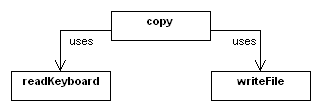


Рисунок 11.1. Схема программы копирования данных.

Здесь используются 3 класса: один класс (иногда его называют сервис) отвечает за чтение с клавиатуры, второй – за вывод в файл, а третий высокоуровневый класс объединяет два низкоуровневых класса с целью организации их работы.

Класс *Сopy* может выглядеть примерно следующим образом:

1: public static class CopyManager

2: {

3: public static void Copy()

4: {

5: var keyboard = new Keyboard();

6: var file = new File();

7: byte[] buffer;

8:

9: while ((buffer = keyboard.Read()) != null)

10: {

11: file.Write(buffer);

12: }

13: }

14: }

Низкоуровневые классы Keyboard и File обладают высокой гибкостью. Можно легко использовать их в контексте, отличном от класса CopyManager. Однако сам класс CopyManager не может быть повторно использована в другом контексте. Например, для отправки данных из файла системному обработчику логов.

Используя принцип инверсии зависимостей, можно сделать класс CopyManager независимым от объектов источника и назначения данных. Для этого необходимо выработать абстракции для этих объектов, и сделать модули зависимыми от этих абстракций, а не друг от друга.

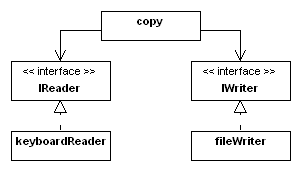


Рисунок 11.2. Схема программы с введенными абстракциями.

1: public interface IReader

2: {

3: public byte[] Read();

4: }

5:

6: public interface IWriter

7: {

8: public void Write(byte[] arg);

9: }

Класс CopyManager должен полагаться только на выработанные абстракции и не делать никаких предположений по поводу индивидуальных особенностей объектов ввода/вывода:

1: public static class CopyManager

2: {

3: public static void Copy(IReader reader, IWriter writer)

4: {

5: byte[] buffer;

6:

7: while ((buffer = reader.Read()) != null)

8: {

9: writer.Write(buffer);

10: }

11: }

12: }

Примерно следующим образом выглядит использование данного класса:

1: CopyManager.Copy(new Keyboard(), new File());

Теперь класс Copy можно использовать в различных контекстах копирования. Изменение его поведения достигается путем ассоциации его с объектами других классов (но которые зависят от тех же абстракций).

Несмотря на простоту выполненных действий, был получен очень важный результат. Теперь код обладает следующими качествами:

* класс может быть использован для копирования данных в контексте отличном от данного;
* возможно добавлять новые устройства ввода/вывода не меняя при этом класс Copy.

Таким образом, снизилась хрупкость кода, повысилась его мобильность и гибкость.

## Формы инверсии зависимостей

Существует две формы инверсии зависимостей: активная и пассивная. Различие между ними состоит в том, как объект узнает о своих зависимостях во время выполнения.

При использовании пассивной формы зависимые объекты внедряются в зависимый. Зависимому объекту не надо прилагать никаких усилий, все нужные сервисы он получает через свой интерфейс.

Активная форма, в отличие от пассивной, предполагает, что зависящий объект будет сам получать свои зависимости при помощи вспомогательных объектов.

Каждая из форм инверсии зависимостей имеет подтипы, которые характеризуют детали связывания объектов между собой.

Пассивная инверсия зависимостей (Dependency Injection):

* *внедрение через конструктор*: для ассоциирования объекта с конкретными реализациями абстракций используется конструктор. При использовании этого типа инверсии зависимостей необходимые объекты передаются в конструктор в качестве аргументов.
* *внедрение через устанавливаемое свойство*: требуется определение отдельного свойства, имеющего set-метод, для каждого из инъецируемых объектов. От предыдущего типа инъекции она отличается местом инъекции. Следует отметить, что внедрение через конструктор и внедрение через устанавливаемое свойство не исключают друг друга.
* *внедрение через интерфейс*: задаются интерфейсы, которые определяют методы для связывания, один интерфейс на каждую зависимость. Зависимый объект должен реализовывать все эти интерфейсы. Определяется также единый интерфейс для всех сервисов. Каждый сервис реализует этот интерфейс таким образом, чтобы внедрить себя в зависящий объект. Таким образом, сервисы сами внедряют себя в зависимый объект посредством установленного интерфейса.
* *внедрение через поле*: в .NET и Java существует возможность получить доступ к private/protected полям объекта. Эта техника может быть использована для внедрения сервисов в зависящий объект напрямую, без использования set-методов и конструкторов.

Активная инверсия зависимостей (Dependency Lookup):

* pull-подход: предполагается наличие в системе общедоступного объекта, который знает обо всех используемых сервисах. В качестве такого объекта может выступать объект, реализующий паттерн Service Locator. Локатор реализует паттерн синглетона, благодаря чему доступ к нему можно получить из любого места приложения.
* push-подход: данная методика отличается от pull-подхода тем, как объект узнает об объекте-локаторе. При использовании pull-подхода класс сам получал локатор посредством класса-синглетона. Push-подход характеризуется тем, что объект-локатор (или как его иногда называют контекст) передается в класс извне (обычно через конструктор).

## IoC контейнер

Очень важное понятие, связанное с инверсией зависимостей – это IoC контейнеры. IoC контейнер – это специальный объект-сборщик, который на основании схемы зависимостей между классами и абстракциями может создать граф объектов. Любой IoC контейнер реализует принцип инверсии зависимостей.

Одной из реализаций IoC контейнера является MEF. Это очень мощный проект, который затрагивает очень много аспектов конструирования ПО. Одним из таких аспектов является конструирование объектов на основании его связей. Причем связи между объектами могут задаваться в виде атрибутов и анализироваться в процессе выполнения приложения.

## MEF

Библиотека MEF появилась относительно недавно, но быстро завоевала популярность у .NET разработчиков за простоту использования и эффективность. Она позволяет строить модульные приложения с минимальным уровнем связности частей (parts) приложения. Эта библиотека включает в себя не только Dependency Injection контейнер, но большой объём инфраструктуры: множество механизмов поиска элементов композиции в сборках, удалённых XAP файлах, механизм пометки элементов композиции с помощью .Net атрибутов и т.д. Также существует версия MEF для Silverlight, которая имеет незначительные отличия от настольной версии.

IoC контейнер в MEF инкапсулируется классом CompositionContainer. Он содержит каталог типов, доступных для инъектирования. Наиболее удобным каталогом является DirectoryCatalog, включающий в себя типы из всех сборок, найденных в папке с приложением:

1: CompositionContainer container =

2: new CompositionContainer(new DirectoryCatalog("."));

Для удовлетворения зависимостей определенного класса используется метод ComposeParts:

1: container.ComposeParts(this);

Стоит иметь в виду, что как и другие IoC контейнеры, перед передачей компоненты в качестве импорта MEF удовлетворяет все её зависимости.

Компоненты в MEF не зависят друг от друга напрямую, вместо этого они зависят от контракта, который указывается строковым идентификатором. Каждый компонент объявляет контракты, которые он реализует, а каждая зависимость указывает необходимый ей контракт.

Реализация какого-либо контракта в терминологии MEF называется экспортом. Экспорты задаются при помощи атрибута [System.ComponentModel.Composition.ExportAttribute]. Данным атрибутом помечается класс сервиса, реализующего контракт. В качестве аргумента в конструктор атрибута передается имя реализуемого контракта. Следующий пример определяет контракт ILogger и объект Logger, его реализующий:

1: public interface ILogger

2: {

3: void Log(string message);

4: }

5:

6: [Export(typeof(ILogger))]

7: public class Logger : ILogger

8: {

9: public void Log(string message)

10: {

11: Console.WriteLine(message);

12: }

13: }

Чтобы различать несколько реализаций одного контракта, в экспорте возможно указать его имя:

1: [Export("GUI", typeof(ILogger))]

2: public class Logger : ILogger

3: {

4: public void Log(string message)

5: {

6: MessageBox.Show(message);

7: }

8: }

В таком случае возможно как получить все реализации контракта ILogger, так и найти конкретную спецификацию, работающую с графическим интерфейсом, по ключевому слову GUI.

По умолчанию MEF работает по принципу пассивной инверсии зависимостей. Зависимости от компонент в терминологии MEF называются импортами. Импорты с внедрением через устанавливаемое свойство или поле задаются при помощи атрибута [System.ComponentModel.Composition.ImportAttribute].

1: public class Processor

2: {

3: [Import]

4: public ILogger Logger { private get; set; }

5:

6: public void Process()

7: {

8: Logger.Log("Hello world");

9: }

10: }

При внедрении через конструктор, этот конструктор помечается атрибутом [System.ComponentModel.Composition.ImportingConstructorAttribute].

1: public class Processor

2: {

3: [ImportingConstructor]

4: public Processor(ILogger logger)

5: {

6: \_logger = logger;

7: }

8:

9: // Private fields

10: private ILogger \_logger;

11:

12: public void Process()

13: {

14: \_logger.Log("Hello world");

15: }

16: }

Обычно MEF определяет имя импортируемого контракта исходя из CLR типа свойства, поля или параметра конструктора (в зависимости от выбранного способа внедрения). Помимо этого существует возможность указать имя контракта вручную, через аргумент конструктора атрибута:

1: public class Processor

2: {

3: [Import("GUI")]

4: public ILogger Logger { private get; set; }

5:

6: public void Process()

7: {

8: Logger.Log("Hello world");

9: }

10: }

Помимо пассивной инверсии зависимостей при помощи MEF возможно реализовать активную инверсию зависимостей с использованием библиотеки CommonServiceLocator. Данная библиотека содержит обобщенный интерфейс для разрешения зависимостей, абстрагированный от конкретного IoC контейнера:

1: var locator = new MefServiceLocator(CompositionContainer);

2: ServiceLocator.SetLocatorProvider(() => locator);

## MVVM и IoC

Легко заметить, что при помощи инверсии зависимостей возможно производить не только разрешение зависимостей между моделями представления и различными сервисами, но и сопоставлять представления и модели представления.

Здесь необходимо принять архитектурное решение, так как при разрешении зависимостей необходимо иметь в виду порядок создания объектов:

* Конструировать модель представления первой. В таком случае представление будет иметь зависимость (прямую, или на уровне конвенции наименований) от модели представления.
* Конструировать представление первым. Тогда модель представления будет требовать в качестве зависимости представление.

В большинстве случаев первый вариант является оптимальным выбором, т.к., с одной стороны, для покрытия кода тестами модель представления должна быть полностью работоспособной, даже если с ней не ассоциировано ни одного представления, и с другой стороны, работа представления без модели представления бессмысленна – без логики и привязки данных это будет всего лишь пустой неинтерактивный макет.

### Определение моделей представления

Для возможности разрешения моделей представления через IoC контейнер, безусловно, в первую очередь необходимо зарегистрировать их в качестве экспорта в MEF:

1: [Export]

2: [PartCreationPolicy(CreationPolicy.NonShared)]

3: public class MainViewModel

4: {

5: //

6: }

Обратите внимание на атрибут PartCreationPolicy: он указывает на то, что данный экспорт необходимо создавать заново при удовлетворении каждой новое зависимости, т.е. он не является синглетоном.

Для разрешения модели в контексте текущего IoC контейнера представления достаточно воспользоваться определенным выше интерфейсом ServiceLocator: так, для получения модели представления MainViewModel необходимо выполнить код:

1: var mainViewModel =

2: ServiceLocator.Current.GetInstance<MainViewModel>();

В данном случае можно возразить, что с точки зрения языка корректно создать экземпляр модели представления посредством оператора new, не привлекая к работе IoC контейнер. Однако, хоть такая операция и не произведет ошибок компиляции, код будет не совсем верен: у созданной через конструктор по умолчанию модели представления не будут удовлетворены зависимости (импорты), так как IoC контейнер о ней будет попросту не знать. Также использование IoC контейнера для создания моделей представления открывает возможность использования элементов аспектно-ориентированного программирования, о чем будет сказано несколько позже.

### Определение представлений

Следующим этапом является регистрация представлений и декларативная привязка их к соответствующим моделям представления.

Для абстрагирования от конкретной реализации представления в соответствии с принципами инверсии управления необходимо определить общий интерфейс:

1: public interface IView

2: {

3: object DataContext { set; }

4: }

Представления будут реализовывать данный интерфейс, в качестве ключа экспорта указывая свое имя:

1: [Export("MainView", typeof(IView))]

2: [PartCreationPolicy(CreationPolicy.NonShared)]

3: public partial class MainView : IView

4: {

5: public MainView()

6: {

7: InitializeComponent();

8: }

9: }

### Сопоставление модели представления и представления

Теперь, когда MEF имеет информацию как о представлении, так и о модели представления, следует реализовать логику сопоставления, которая на основании соглашении об именовании определяет представление, соответствующее переданной модели представления.

Обычной практикой является добавление суффикса ViewModel моделям представления и суффикса View представлениям. Таким образом, имея имя типа модели представления, достаточно заменить в нем ViewModel на View, чтобы получить имя представления.

1: var view = ServiceLocator.Current.GetInstance<IView>(

2: viewModel.GetType().Name.Replace("ViewModel", "View"));

На практике часто возникает ситуация, в которой целое семейство моделей представления, объединенное в одну иерархию, соответствует единственному представлению. Например, модели представления по редактированию доменных объектов UserEditViewModel, RoleEditViewModel будут иметь общего предка EditViewModel, и обеим моделям представления должно быть сопоставлено одно представление EditView, содержащее элемент управления DataForm, предоставляющий обобщенный интерфейс редактирования объекта на основе отражения типов.

Простым решением будет создание 2 наследников для представления EditView, UserEditView и RoleEditView, не содержащих никакой логики и служащих для соблюдения соглашения о наименовании. Однако более разумно включить в логику поиска соответствия обход по всему дереву наследования, до тех пор, пока не будет найдено соответствие:

1: IView view;

2: Type viewModelType = viewModel.GetType();

3:

4: do

5: {

6: view = ServiceLocator.Current.GetInstance<IView>(

7: viewModelType.Name.Replace("ViewModel", "View"));

8:

9: if (view != null)

10: {

11: break;

12: }

13:

14: viewModelType = viewModelType.BaseType;

15: } while (viewModelType != null);

Для возможности повторного использования данная логика добавляется в наследника класса ContentControl, который по переданной через зависимое свойство ViewModel модели представления находит соответствующее представление и устанавливает её своим содержимым:

1: /// <summary>

2: /// Presents a View corresponding to the View Model

3: /// </summary>

4: public class ViewModelPresenter : ContentControl

5: {

6: /// <summary>

7: /// Initializes a new instance

8: /// </summary>

9: public ViewModelPresenter()

10: {

11: HorizontalContentAlignment = HorizontalAlignment.Stretch;

12: VerticalContentAlignment = VerticalAlignment.Stretch;

13: }

14:

15: // Dependency properties

16: public static readonly DependencyProperty ViewModelProperty

17: = DependencyProperty.Register(

18: "ViewModel", typeof(object), typeof(ViewModelPresenter),

19: new PropertyMetadata(null, OnViewModelPropertyChanged));

20:

21: // Private fields

22: private Type \_oldViewType;

23:

24: /// <summary>

25: /// View Model to be presented

26: /// </summary>

27: public object ViewModel

28: {

29: get { return GetValue(ViewModelProperty); }

30: set { SetValue(ViewModelProperty, value); }

31: }

32:

33: private static void OnViewModelPropertyChanged(

34: DependencyObject changedObject,

35: DependencyPropertyChangedEventArgs args)

36: {

37: var contentControl = (ViewModelPresenter)changedObject;

38: contentControl.RefreshContentPresenter();

39: }

40:

41: private void RefreshContentPresenter()

42: {

43: if (ViewModel == null)

44: {

45: Content = null;

46: \_oldViewType = null;

47:

48: return;

49: }

50:

51: IView view;

52: Type viewModelType = ViewModel.GetType();

53:

54: do

55: {

56: view = ServiceLocator.Current.GetInstance<IView>(

57: viewModelType.Name.Replace("ViewModel", "View"));

58:

59: if (view != null)

60: {

61: break;

62: }

63:

64: viewModelType = viewModelType.BaseType;

65: } while (viewModelType != null);

66:

67: if (view != null)

68: {

69: Type viewType = view.GetType();

70:

71: if (viewType == \_oldViewType && Content is IView)

72: {

73: ((IView)Content).DataContext = ViewModel;

73: }

74: else

75: {

76: view.DataContext = ViewModel;

77: Content = view;

78: \_oldViewType = viewType;

79: }

80: }

81: else

82: {

83: Content = "View hasn't been found";

84: \_oldViewType = null;

85: }

86: }

87: }

В данном элементе управления также реализовано простейшее кеширование: если представление не изменилось, а изменилась только модель представления, то создания нового объекта не произойдет, и новая модель представления будет назначена контекстом представления взамен старой.

Далее, в если в XAML разметке необходимо отобразить представление, соответствующее определенной модели представления, достаточно вставить элемент управления ViewModelPresenter и передать ему в зависимое свойство данную модель представления. Внутри ViewModelPresenter будет отображено представление, найденное по описанному выше алгоритму.

### Реализация INotifyPropertyChanged средствами аспектно-ориентированного программирования

Как было сказано ранее, так как создание экземпляров моделей представления происходит посредством IoC контейнера, возможно расширить логику создания объекта и дополнять его произвольными аспектами.

Данная функциональность недоступна в базовой версии MEF, и требует использования дополнительных библиотек, MefContrib и DynamicProxy (которые помимо перехвата создания объекта реализует некоторые другие полезные функции, выходящие за рамки лекции).

В целях лучшей расширяемости приложения следует объявить перечисление со всеми доступными аспектами (на текущий момент с 1 аспектом):

1: [Flags]

2: public enum Aspects

3: {

4: NotifyPropertyChanged = 1,

5: }

А также необходимо написать небольшой метод-расширение, добавляющий к контейнеру MEF логику перехвата создания объектов. Если в метаданных создаваемого экземпляра найден атрибут AspectMetadata и указан аспект NotifyPropertyChanged, он будет обернут прокси-объектом, реализующим логику INotifyPropertyChanged:

1: public static class AopExtensions

2: {

3: public const string AspectMetadata = "AspectMetadata";

4:

5: public static InterceptionConfiguration AddAopInterception(

6: this InterceptionConfiguration interceptionConfiguration)

7: {

8: return interceptionConfiguration

9: .AddInterceptionCriteria(

10: new PredicateInterceptionCriteria(

11: new PropertyChangedDynamicProxyExportInterceptor(),

12: def => def.ExportDefinitions.Any(export =>

13: export.Metadata.ContainsKey(AspectMetadata) &&

14: ((Aspects)export.Metadata[AspectMetadata] &

15: Aspects.NotifyPropertyChanged) != 0)));

16: }

17: }

1: internal class PropertyChangedDynamicProxyExportInterceptor :

2: IExportedValueInterceptor

3: {

4: private static readonly ProxyGenerator Generator =

5: new ProxyGenerator();

6: private readonly Type[] \_additionalInterfaces = new[]

7: { typeof(INotifyPropertyChanged) };

8:

9: public object Intercept(object value)

10: {

11: object proxy = Generator.CreateClassProxy(

12: value.GetType(), \_additionalInterfaces,

13: new[] { new PropertyChangedInterceptor() });

14:

15: Type currentType = value.GetType();

16:

17: do

18: {

19: FieldInfo[] fields =

20: currentType.GetFields(BindingFlags.Instance |

21: BindingFlags.NonPublic | BindingFlags.Public)

22: .Where(field =>

23: (field.Attributes & FieldAttributes.InitOnly) == 0)

24: .ToArray();

25:

26: fields.Select(field => new { Field = field,

27: Value = field.GetValue(value) })

28: .ForEach(desc =>

29: desc.Field.SetValue(proxy, desc.Value));

30:

31: currentType = currentType.BaseType;

32: } while (currentType != null);

33:

34: return proxy;

35: }

36: }

1: internal class PropertyChangedInterceptor : IInterceptor

2: {

3: private event PropertyChangedEventHandler PropertyChanged;

4:

5: public void Intercept(IInvocation invocation)

6: {

7: string methodName = invocation.Method.Name;

8:

9: if (invocation.Method.IsSpecialName)

10: {

11: if (invocation.Method.DeclaringType ==

12: typeof(INotifyPropertyChanged))

13: {

14: if (methodName.StartsWith("add\_"))

15: {

16: PropertyChanged+=(PropertyChangedEventHandler)

17: invocation.Arguments[0];

18: }

19: else

20: {

21: PropertyChanged-=(PropertyChangedEventHandler)

22: invocation.Arguments[0];

23: }

24:

25: return;

26: }

27:

28: if (methodName.StartsWith("set\_"))

29: {

30: PropertyInfo propertyInfo = invocation.Proxy

31: .GetType().GetProperty(methodName.Substring(4));

32: object oldValue = propertyInfo.GetValue(

33: invocation.Proxy,

34: invocation.Arguments.SkipLast(1).ToArray());

35:

36: invocation.Proceed();

37:

38: if (oldValue == invocation.Arguments

39: [invocation.Arguments.Length - 1])

40: {

41: return;

42: }

43:

44: OnPropertyChanged(invocation.Proxy,

45: new PropertyChangedEventArgs(propertyInfo.Name));

46:

47: return;

48: }

49: }

50:

51: invocation.Proceed();

52: }

53:

54: private void OnPropertyChanged(object sender,

55: PropertyChangedEventArgs e)

56: {

57: PropertyChangedEventHandler eventHandler=PropertyChanged;

58:

59: if (eventHandler != null)

60: {

61: eventHandler(sender, e);

62: }

63: }

64: }

Последним шагом является вызов метода-расширения для MEF контекста приложения:

1: var catalog = new InterceptingCatalog(new DirectoryCatalog("."),

2: new InterceptionConfiguration().AddAopInterception());

Теперь любой класс, помеченный атрибутом [ExportMetadata( AopExtensions.AspectMetadata, Aspects.NotifyPropertyChanged)], при разрешении через IoC контейнер, автоматически получит реализацию INotifyPropertyChanged для своих виртуальных свойств.

## Краткие итоги

В данной теме были рассмотрена архитектура композитного MVVM приложения, в котором соответствие между моделями представления и представлениями осуществляется при помощи IoC контейнера MEF. Также описано расширение контейнера, позволяющее автоматически реализовывать интерфейс INotifyPropertyChanged на всех моделях представления, помеченных атрибутом. Может показаться, что используемые приемы достаточно перегружены для реализации небольшого проекта, но необходимо осознавать, что при расширении проекта, выстроенная архитектура принесет свои плоды и позволит сопровождать проект с минимальными усилиями.

## Набор для практики

Вопросы:

1. Инверсия зависимости. Активная и пассивная.
2. Назначение IoC контейнера
3. Библиотека MEF
4. Построение композитного MVVM приложения с применением IoC контейнера MEF

## Литература

1. WPF Apps With The Model-View-ViewModel Design Pattern  
   <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>
2. Using MEF to Expose Interfaces in Your Silverlight MVVM Apps  
   <http://msdn.microsoft.com/en-us/magazine/gg535672.aspx>
3. Managed Extensibility Framework  
   <http://mef.codeplex.com/>

# Лекция 12. Особенности отображения диалоговых окон в WPF и Silverlight версиях приложения.

**Краткая аннотация лекции:** в лекции рассматривается реализация логики отображения и управления окнами в соответствии с шаблоном MVVM для максимального повторного использования кода между Silverlight и WPF.

**Цель лекции:** показать читателям на примере фрагментов кода механизм отображения диалоговых окон в многослойном кроссплатформенном Silverlight/WPF MVVM приложении.

## Понятие ICloseableViewModel и IChildViewModel

В предыдущей лекции было введено понятие базовой модели представления, и определяющего её интерфейса, IViewModel. Получившееся простейшее приложение состояло из главной модели представления, MainViewModel, и соответствующего ей представления, MainView.

Сложно представить себе программу, обходящуюся одним главным окном. В любом MVVM приложении необходимо не только взаимодействовать с сервисами на уровне главной модели представления, но и отображать дополнительную информацию в диалоговых окнах. В соответствии с шаблоном MVVM вся логика приложения должна находиться на уровне модели представления, а значит должна быть возможность из произвольной модели представления инициировать открытие дочерней модели представления.

Логично ввести понятие *закрываемой модели представления*, которая может быть отображена вызовом метода Show() и закрыта вызовом метода Close():

1: /// <summary>

2: /// Interface of a View Model that can be closed

3: /// </summary>

4: public interface ICloseableViewModel : IViewModel

5: {

6: /// <summary>

7: /// Shows whether View Model has been closed

8: /// </summary>

9: bool IsClosed { get; }

10:

11: /// <summary>

12: /// Event raised when the View Model is closed

13: /// </summary>

14: event EventHandler Closed;

15:

16: /// <summary>

17: /// Closes View Model

18: /// </summary>

19: void Close();

20:

21: /// <summary>

22: /// Registers View Model to be shown

23: /// </summary>

24: void Show();

25: }

Флаг IsClosed сообщает о том, была ли модель представления уже закрыта, а событие Close инициируется как раз в момент закрытия.

Для дальнейшего расширения иерархии интерфейсов моделей представления необходимо ввести *интерфейс модели представления с заголовком*:

1: /// <summary>

2: /// Interface of a View Model that has a title

3: /// </summary>

4: public interface IEntitledViewModel : IViewModel

5: {

6: /// <summary>

7: /// View Model's title

8: /// </summary>

9: string Title { get; }

10: }

Спецификацией понятия закрываемой модели представления будет *модель представления дочернего окна*, имеющая заголовок:

1: /// <summary>

2: /// Interface of Child View's View Model

3: /// </summary>

4: public interface IChildViewModel

5: : ICloseableViewModel, IEntitledViewModel

6: {

7: //

8: }

## Понятие IChildViewModelManager

Очевидно, что модели представления дочерних окон не могут работать сами по себе, и необходим механизм, отображающий их в настоящие дочерние окна целевой платформы (Silverlight или WPF).

Для этих целей необходимо ввести *менеджер дочерних моделей представления*: он ведет учет видимых в настоящий момент моделей представления и сообщает о них всем внешним слушателям, подписанным на изменения коллекции ViewModels:

1: /// <summary>

2: /// Interface of class managing closeable View Models of

3: /// the specified type

4: /// </summary>

5: public interface IChildViewModelManager

6: : ICloseableViewModelPresenter<IChildViewModel>

7: {

8: /// <summary>

9: /// Collection of managed View Models

10: /// </summary>

11: ReadOnlyObservableCollection<IChildViewModel>

12: ViewModels { get; }

13: }

Здесь используется понятие *презентера закрываемой модели представления*, который принимает к регистрации видимые модели представления:

1: /// <summary>

2: /// Interface of closeable View Model presenter

3: /// </summary>

4: [InheritedExport]

5: public interface ICloseableViewModelPresenter<in TViewModelBase>

6: where TViewModelBase : ICloseableViewModel

7: {

8: /// <summary>

9: /// Shows <paramref name="viewModel" />

10: /// </summary>

11: void ShowViewModel(TViewModelBase viewModel);

12: }

Реализация менеджера дочерних моделей представления принимает к регистрации видимые модели представления дочерних окон и добавляет их во внутреннюю коллекцию. Одновременно с регистрацией происходит привязка к событию Closed модели представления, по которому она снимается с регистрации в менеджере и удаляется из коллекции:

1: /// <summary>

2: /// Manages closeable View Models of the specified type

3: /// </summary>

4: internal class ChildViewModelManager

5: : IChildViewModelManager

6: {

7: // Private fields

8: private readonly ObservableCollection<IChildViewModel>

9: \_viewModelsInternal =

10: new DispatchObservableCollection<IChildViewModel>();

11: private ReadOnlyObservableCollection<IChildViewModel>

12: \_viewModels;

13:

14: /// <summary>

15: /// Collection of managed View Models

16: /// </summary>

17: public ReadOnlyObservableCollection<IChildViewModel>

18: ViewModels

19: {

20: get

21: {

22: return \_viewModels ?? (\_viewModels =

23: new ReadOnlyObservableCollection<IChildViewModel>(

24: \_viewModelsInternal));

25: }

26: }

27:

28: #region ICloseableViewModelPresenter<TViewModelBase> Members

29:

30: void ICloseableViewModelPresenter<IChildViewModel>

31: .ShowViewModel(IChildViewModel viewModel)

32: {

33: ShowViewModelCore(viewModel);

34: }

35:

36: #endregion

37:

38: /// <summary>

39: /// Closes <paramref name="viewModel" />

40: /// </summary>

41: protected virtual void CloseViewModelCore

42: (IChildViewModel viewModel)

43: {

44: viewModel.Closed -= OnViewModelClosed;

45:

46: Debug.Assert(\_viewModelsInternal.Contains(viewModel));

47: \_viewModelsInternal.Remove(viewModel);

48: }

49:

50: /// <summary>

51: /// Shows <paramref name="viewModel" />,

52: /// adding it to collection

53: /// </summary>

54: protected virtual void ShowViewModelCore

55: (IChildViewModel viewModel)

56: {

57: Debug.Assert(!viewModel.IsClosed);

58: viewModel.Closed += OnViewModelClosed;

59:

60: Debug.Assert(!\_viewModelsInternal.Contains(viewModel));

61: \_viewModelsInternal.Add(viewModel);

62: }

63:

64: private void OnViewModelClosed(object sender, EventArgs e)

65: {

66: Debug.Assert(sender is IChildViewModel);

67: CloseViewModelCore((IChildViewModel)sender);

68: }

69: }

## ChildViewManager

Теперь, когда IChildViewModelManager ведет учет моделей представления дочерних окон, можно перейти к следующему понятию – *менеджеру дочерних представлений*. Он прослушивает коллекцию ViewModel на предмет изменения по событию CollectionChanged интерфейса INotifyCollectionChanged: если в коллекцию добавлена новая модель представления, то необходимо создать для неё дочернее окно и сохранить соответствие в словаре; затем, при удалении модели представления из коллекции достаточно найти дочернее окно по словарю, закрыть его и удалить запись о соответствии.

Следует заметить, что так как объекты пользовательского интерфейса как правило не позволяют взаимодействовать с собой из потока, отличного от GUI потока, а вызовы слоя моделей представления могут происходить в любом контексте, в том числе в потоке из пула потоков (наиболее часто в нем выполняются обратные методы вызова асинхронных операций Windows Communication Foundation сервисов), то необходимо явно отправлять события изменений коллекции ViewModels на Dispatcher GUI потока посредством DispatcherSynchronizationContext.

1: #if SILVERLIGHT

2: using ChildViewType = System.Windows.Controls.ChildWindow;

3: #else

4: using ChildViewType = System.Windows.Window;

5: #endif

6:

7: public class ChildViewManager

8: {

9: [ImportingConstructor]

10: public ChildViewManager

11: (IEnumerable<IChildViewModel> viewModelCollection)

12: {

13: OnViewModelCollectionChanged(viewModelCollection,

14: new NotifyCollectionChangedEventArgs

15: (NotifyCollectionChangedAction.Reset));

16:

17: var notifiable = viewModelCollection

18: as INotifyCollectionChanged;

19:

20: if (notifiable != null)

21: {

22: notifiable.CollectionChanged += (sender, e)

23: => DispatcherSynchronizationContext.Post(arg =>

24: OnViewModelCollectionChanged(sender, e), null);

25: }

26: }

27:

28: // Private readonly fields

29: protected static readonly DispatcherSynchronizationContext

30: DispatcherSynchronizationContext =

31: new DispatcherSynchronizationContext(

32: #if SILVERLIGHT

33: Deployment.Current.Dispatcher

34: #else

35: Application.Current.Dispatcher

36: #endif

37: );

38:

39: // Private fields

40: private readonly IDictionary<IChildViewModel, ChildViewType>

41: \_childViews =

42: new Dictionary<IChildViewModel, ChildViewType>();

43:

44: /// <summary>

45: /// Closes all managed <see cref="ChildViewPresenter" />

46: /// </summary>

47: protected virtual void CloseAllViews()

48: {

49: foreach (KeyValuePair<IChildViewModel, ChildViewType>

50: pair in \_childViews)

51: {

52: CloseView(pair.Key);

53: }

54: }

55:

56: /// <summary>

57: /// Closes specified <see cref="ChildViewPresenter" />

58: /// </summary>

59: protected virtual void CloseView

60: (IChildViewModel childViewModel)

61: {

62: Debug.Assert(\_childViews.ContainsKey(childViewModel));

63:

64: ChildViewType childView = \_childViews[childViewModel];

65: \_childViews.Remove(childViewModel);

66: childView.Close();

67: }

68:

69: /// <summary>

70: /// Shows specified <see cref="ChildViewPresenter" />

71: /// </summary>

72: protected virtual void ShowView

73: (IChildViewModel childViewModel)

74: {

75: ChildViewType childWindow = new ChildViewPresenter

76: { DataContext = childViewModel };

77: \_childViews.Add(childViewModel, childWindow);

78: childWindow.Show();

79: }

80:

81: private void OnViewModelCollectionChanged(object sender,

82: NotifyCollectionChangedEventArgs e)

83: {

84: switch (e.Action)

85: {

86: case NotifyCollectionChangedAction.Add:

87: foreach (IChildViewModel viewModel in e.NewItems)

88: {

89: ShowView(viewModel);

90: }

91: break;

92:

93: case NotifyCollectionChangedAction.Remove:

94: foreach (IChildViewModel viewModel in e.OldItems)

95: {

96: CloseView(viewModel);

97: }

98: break;

99:

100: case NotifyCollectionChangedAction.Reset:

101: CloseAllViews();

102:

103: foreach (IChildViewModel viewModel

104: in (IEnumerable)sender)

105: {

106: ShowView(viewModel);

107: }

108: break;

109:

110: default:

111: throw new ArgumentOutOfRangeException

112: ("e.Action is out of range", (Exception)null);

113: }

114: }

115: }

ChildViewPresenter – это окно Window в случае WPF и ChildWindow в случае Silverlight. Данные элементы управления содержат в себе ViewModelPresenter, который уже в свою очередь определяет, какое представление необходимо отобразить в дочернем окне.

Дополнительной обработки требует случай закрытия дочернего окна через нажатие на иконку в верхнем правом углу. Реализованная логика менеджеров поддерживает корректную инициацию закрытия дочернего окна исключительно со слоя представления, поэтому закрытие одного только ChildViewPresenter не приведет к срабатыванию логики закрытия модели представления.

Для правильной работы приложения при попытке пользователя закрыть дочернее окно системными средствами необходимо отменить это закрытие в событии OnClosing и инициировать корректное закрытие модели представления, которое в свою очередь затем приведет к закрытию окна. С точки зрения пользователя поведение приложения будет корректным, и при этом архитектура приложения не будет загрязнена дополнительной логикой обработки специального случая.

Реализация ChildViewPresenter для WPF:

1: <Window x:Class="TestProject.ChildViewPresenter"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:viewModelMapping="clr-namespace:TestProject"

5: SizeToContent="WidthAndHeight"

6: Title="{Binding Title}"

7: WindowStartupLocation="CenterScreen">

8:

9: <viewModelMapping:ViewModelPresenter

10: x:Name="ViewModelPresenter" ViewModel="{Binding}" />

11: </Window>

Для Silverlight:

1: <controls:ChildWindow x:Class="TestProject.ChildViewPresenter"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:controls=

5: "http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"

6: xmlns:ViewModelMapping="clr-namespace:TestProject"

7: Title="{Binding Title}">

8:

9: <ViewModelMapping:ViewModelPresenter ViewModel="{Binding}" />

10: </controls:ChildWindow>

Код данных элементов управления одинаковый как в Silverlight, так и в WPF:

1: public partial class ChildViewPresenter

2: {

3: /// <summary>

4: /// Initializes a new instance

5: /// </summary>

6: public ChildViewPresenter()

7: {

8: InitializeComponent();

9: }

10:

11: /// <summary>

12: /// Underlying View Model

13: /// </summary>

14: private ICloseableViewModel ViewModel

15: {

16: get

17: {

18: Debug.Assert(DataContext == null

19: || DataContext is ICloseableViewModel);

20: return (ICloseableViewModel)DataContext;

21: }

22: }

23:

24: /// <summary>

25: /// Processes window closing

26: /// </summary>

27: protected override void OnClosing(CancelEventArgs e)

28: {

29: base.OnClosing(e);

30:

31: Debug.Assert(ViewModel != null);

32:

33: if (!ViewModel.IsClosed)

34: {

35: e.Cancel = true;

36: ViewModel.Close();

37: }

38: }

39: }

## Реализация ChildViewModelBase

После реализации всех необходимых менеджеров можно перейти непосредственно к ChildViewModelBase. Этот класс будет предоставлять своим наследникам базовый функционал, позволяющий им работать как моделям представления дочернего окна.

В класс также добавлены некоторые члены, которые пригодятся при его использовании:

* Команда CloseCommand позволяет быстро привязывать элементы управления, закрывающие окно (такие, как кнопка «Отмена» или «Закрыть»);
* Метод Show сделан виртуальным, чтобы спецификации дочерних окон могли при необходимости переопределить его и произвести необходимую инициализацию при показе модели представления.
* Защищенный виртуальный метод OnClosing позволяет наследнику при помощи аргумента CancelEventArgs отменить закрытие окна. Это полезно, например, при необходимости отобразить диалог подтверждения закрытия, и не закрывать модель представления, если пользователь передумал.
* При помощи защищенного виртуального метода OnClosed классы, находящиеся ниже в иерархии наследования, могут обработать событие закрытия модели представления, например, очистив выделенные при создании или в методе Show ресурсы.

1: /// <summary>

2: /// Base class for View Model supporting closing

3: /// </summary>

4: public abstract class ChildViewModelBase : IChildViewModel

5: {

6: /// <summary>

7: /// Shows whether a View Model has been closed

8: /// </summary>

9: public virtual bool IsClosed { get; private set; }

10:

11: /// <summary>

12: /// View Model's title

13: /// </summary>

14: public virtual string Title

15: {

16: get { return "Безымянный"; }

17: }

18:

19: #region Commands

20:

21: private ICommand \_closeCommand;

22:

23: /// <summary>

24: /// Command closing the tab

25: /// </summary>

26: public ICommand CloseCommand

27: {

28: get

29: {

30: return \_closeCommand ??

31: (\_closeCommand = new ActionCommand(Close));

32: }

33: }

34:

35: #endregion

36:

37: #region Injected properties

38:

39: [Import]

40: public virtual ICloseableViewModelPresenter<IChildViewModel>

41: ViewModelPresenter { get; set; }

42:

43: #endregion

44:

45: /// <summary>

46: /// Event raised when the View Model is closed

47: /// </summary>

48: public event EventHandler Closed;

49:

50: /// <summary>

51: /// Closes View Model

52: /// </summary>

53: public void Close()

54: {

55: if (IsClosed)

56: {

57: return;

58: }

59:

60: var cancelEventArgs = new CancelEventArgs();

61: OnClosing(cancelEventArgs);

62:

63: if (cancelEventArgs.Cancel)

64: {

65: return;

66: }

67:

68: OnClosed();

69: }

70:

71: /// <summary>

72: /// Registers View Model to be shown

73: /// </summary>

74: public virtual void Show()

75: {

76: ViewModelPresenter.ShowViewModel(this);

77: }

78:

79: /// <summary>

80: /// Processes View Model closing attempt

81: /// </summary>

82: protected virtual void OnClosing(CancelEventArgs e)

83: {

84: //

85: }

86:

87: /// <summary>

88: /// Processes View Model close

89: /// </summary>

90: protected virtual void OnClosed()

91: {

92: IsClosed = true;

93:

94: if (Closed != null)

95: {

96: Closed(this, EventArgs.Empty);

97: }

98: }

99: }

## Реализация ModalChildViewModelBase

Полезным расширением реализованной модели представления ChildViewModelBase является модель представления модального диалога ModalChildViewModelBase. Модальный диалог позволяет в унифицированной форме сообщить результат выполнения вызвавшему его объекту при помощи свойства ModalResult.

Как и во всех стандартных оконных средах выполнения, установка значения ModalResult автоматически закрывает модель представления.

1: /// <summary>

2: /// Base class of Child View's View Model providing its modal

3: /// result

4: /// </summary>

5: public abstract class ModalChildViewModelBase :

6: ChildViewModelBase, IModalChildViewModel

7: {

8: // Private fields

9: private bool? \_modalResult;

10:

11: /// <summary>

12: /// Gets or sets modal result

13: /// </summary>

14: /// <remarks>

15: /// Setting <see cref="ModalResult" /> causes View Model

16: /// to close

17: /// </remarks>

18: public virtual bool? ModalResult

19: {

20: get { return \_modalResult; }

21: protected set

22: {

23: if (value == null)

24: {

25: throw new InvalidOperationException(

26: "Unable to set ModalResult to null");

27: }

28:

29: \_modalResult = value;

30: Close();

31: }

32: }

33: }

## Реализация MessageViewModel

Развивая идею ModalChildViewModelBase, целесообразно реализовать одну из наиболее часто используемых моделей представления – модель представления сообщения MessageViewModel.

Данная модель представления отображает окно с сообщением, заданное набором параметров: тип сообщения (информационное, предупреждение, ошибка), доступные для нажатия кнопки (ОK, ОК/Отмена, Повторить/Отмена, или любой заданный пользователем набор), заголовок окна, и непосредственно само сообщение.

Команда ButtonCommand является здесь универсальной для любой заданной пользователем кнопки: в качестве аргумента ей передается индекс кнопки, который затем записывается в MessageResult; также при этом устанавливается свойство ModalResult, что приводит к закрытию окна.

1: /// <summary>

2: /// View model of modal message

3: /// </summary>

4: [Export(typeof(MessageViewModel))]

5: [PartCreationPolicy(CreationPolicy.NonShared)]

6: [ExportMetadata(AopExtensions.AspectMetadata,

7: Aspects.NotifyPropertyChanged)]

8: public class MessageViewModel : ModalChildViewModelBase

9: {

10: /// <summary>

11: /// Initializes a new instance

12: /// </summary>

13: protected MessageViewModel()

14: {

15: //

16: }

17:

18: // Predefined button sets

19: public static readonly IList<string> OkButtons

20: = new[] { "OК" };

21: public static readonly IList<string> OkCancelButtons

22: = new[] { "OК", "Отмена" };

23: public static readonly IList<string> RetryCancelButtons

24: = new[] { "Повторить", "Отмена" };

25:

26: // Private fields

27: private IList<string> \_buttons = OkButtons;

28:

29: /// <summary>

30: /// List of available buttons

31: /// </summary>

32: public virtual IList<string> Buttons

33: {

34: get { return \_buttons; }

35: set { \_buttons = value; }

36: }

37:

38: /// <summary>

39: /// Message execution result

40: /// </summary>

41: public virtual int MessageResult { get; protected set; }

42:

43: /// <summary>

44: /// Body of the message

45: /// </summary>

46:

47: public virtual string Message { get; set; }

48:

49: /// <summary>

50: /// Type of the message

51: /// </summary>

52: public virtual MessageType MessageType { get; set; }

53:

54: /// <summary>

55: /// Message's caption

56: /// </summary>

57: [Localizable(true)]

58: public virtual string Caption { get; set; }

59:

60: /// <summary>

61: /// View model's title

62: /// </summary>

63: public override string Title

64: {

65: get { return Caption; }

66: }

67:

68: #region Commands

69:

70: private ICommand \_buttonCommand;

71:

72: public ICommand ButtonCommand

73: {

74: get

75: {

76: return \_buttonCommand ??

77: (\_buttonCommand = new ActionCommand(

78: param => SetMessageResult(param.ToString())));

79: }

80: }

81:

82: #endregion

83:

84: /// <summary>

85: /// Processes View Model closing attempt

86: /// </summary>

87: protected override void OnClosing(CancelEventArgs e)

88: {

89: base.OnClosing(e);

90: e.Cancel = ModalResult == null;

91: }

92:

93: /// <summary>

94: /// Sets specified button's index as message result

95: /// </summary>

96: private void SetMessageResult(string selectedButton)

97: {

98: if (IsClosed)

99: {

100: return;

101: }

102:

103: Debug.Assert(Buttons.Contains(selectedButton));

104: MessageResult = Buttons.IndexOf(selectedButton);

105: ModalResult = true;

106: }

107: }

Как видно из кода, особенностью представленной модели представления является то, что переопределяемый метод OnClosing не позволяет закрыть окно, не установив MessageResult – таким образом вызывающему коду гарантируется детерминированный результат отображения сообщения.

Соответствующее представление в соответствии с шаблоном MVVM не содержит сложной логики и лишь визуализирует состояние, представленное в модели представления:

1: <UserControl x:Class="TestProject.MessageView"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:controlsHelpers

5: ="clr-namespace:PassportOffice.View.Controls">

6:

7: <Grid>

8: <Grid.Resources>

9: <controlsHelpers:CommandReference

10: x:Key="ButtonCommandReference"

11: Command="{Binding ButtonCommand}" />

12: <controlsHelpers:MessageTypeConverter

13: x:Key="MessageTypeConverter" />

14: </Grid.Resources>

15:

16: <Grid.ColumnDefinitions>

17: <ColumnDefinition Width="Auto" />

18: <ColumnDefinition Width="Auto" />

19: </Grid.ColumnDefinitions>

20: <Grid.RowDefinitions>

21: <RowDefinition Height="\*" />

22: <RowDefinition Height="Auto" />

23: </Grid.RowDefinitions>

24:

25: <Image x:Name="ImageMessage" Height="48"

26: UriSource="{Binding MessageType,

27: Converter={StaticResource MessageTypeConverter}}"

28: Grid.Column="0" Grid.Row="0" />

29:

30: <TextBlock Margin="3" MinWidth="300" MaxWidth="600"

31: Text="{Binding Message}" VerticalAlignment="Center"

32: TextWrapping="Wrap" Grid.Column="1" Grid.Row="0" />

33:

34: <ItemsControl x:Name="ButtonsControl"

35: HorizontalAlignment="Center"

36: ItemsSource="{Binding Buttons}" Margin="0,5,0,0"

37: Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1">

38: <ItemsControl.ItemsPanel>

39: <ItemsPanelTemplate>

40: <StackPanel Orientation="Horizontal" />

41: </ItemsPanelTemplate>

42: </ItemsControl.ItemsPanel>

43: <ItemsControl.ItemTemplate>

44: <DataTemplate>

45: <Button Content="{Binding}"

46: Command="{StaticResource ButtonCommandReference}"

47: CommandParameter="{Binding}" />

48: </DataTemplate>

49: </ItemsControl.ItemTemplate>

50: </ItemsControl>

51: </Grid>

52: </UserControl>

## Краткие итоги

В рамках данной лекции была реализована логика отображения диалоговых окон на уровне моделей представления. Для этого добавлены набор слушателей слоя представления, которые, не нарушая принципы MVVM, обеспечивают отображение регистрируемых моделей представления диалоговых окон. Также разработанные базовые классы в качестве примера были расширены наследниками, специализирующимися в отображении сообщений различного характера.

## Литература

# Лабораторная работа 6. Построение кроссплатформенного Silverlight/WPF приложения.

**Краткая аннотация лабораторной работы:** в лабораторной работе рассматривается дополнение реализованного в предыдущих частях каркаса до полноценного примера, иллюстрирующего наиболее часто используемые приемы разработки с синхронным использованием кода.

**Цель лабораторной работы:** показать читателям на примере фрагментов кода полную реализацию многослойного кроссплатформенного Silverlight/WPF MVVM приложения. Данная лабораторная работа является обобщающим занятием по курсу.

**Для более детального представления о можно скачать исходный код** [**итогового**](SampleSolution.zip) **проекта**

## Создание проектов

В первую очередь необходимо создать проекты будущего приложения. В простейшем случае это будут 3 сборки: WPF Application, Silverlight Application, а также Web Application для хостинга Silverlight приложения.

Окно создание проекта вызывается нажатием пункта меню File – New – Project:

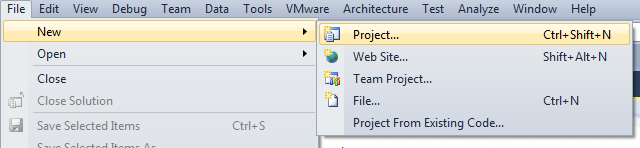


Рисунок л.р. 6.1. Вызов окна создания проекта.

В появившемся окне необходимо перейти в раздел Other Project Types – Visual Studio Solutions и выбрать Blank Solution:

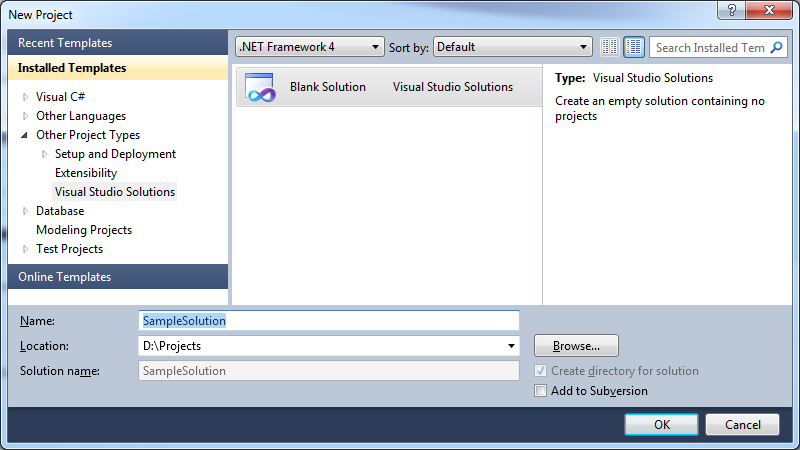


Рисунок л.р. 6.2. Создание пустого решения.

В созданное решение следует добавить Silverlight и WPF проекты, используя контекстное меню обозревателя решений:

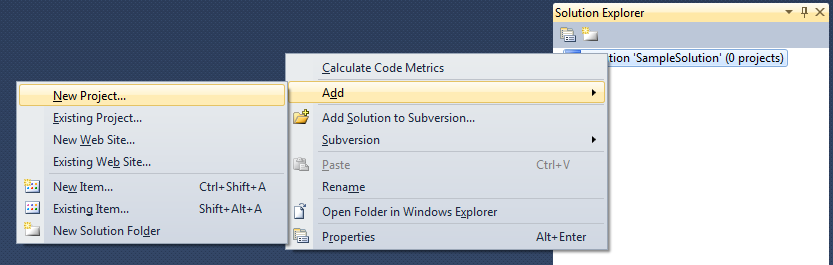


Рисунок л.р. 6.3. Добавление проекта в решение.

Для добавления Silverlight приложения необходимо выбрать пункт Silverlight Application в разделе Visual C# – Silverlight:

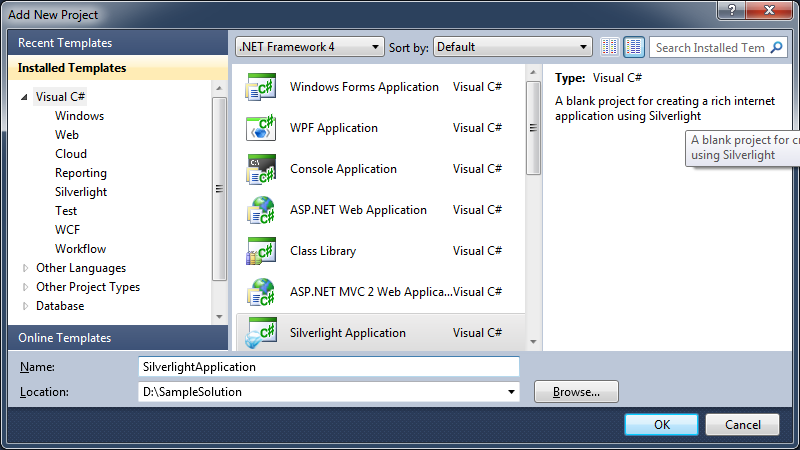


Рисунок л.р. 6.4. Добавление Silverlight приложения.

В появившемся диалоговом окне следует подтвердить создание Web Application для размещения Silverlight приложения и выбрать версию Silverlight:

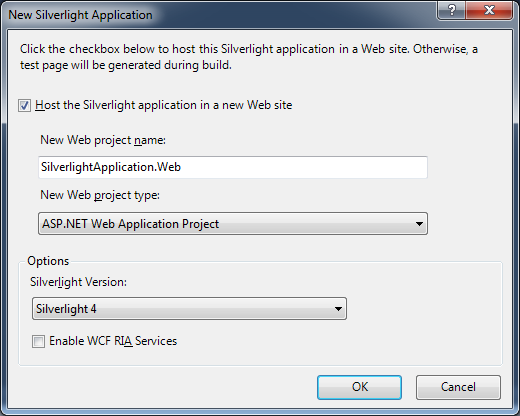


Рисунок л.р. 6.5. Задание параметров Silverlight приложения.

Аналогично необходимо добавить WPF приложение:

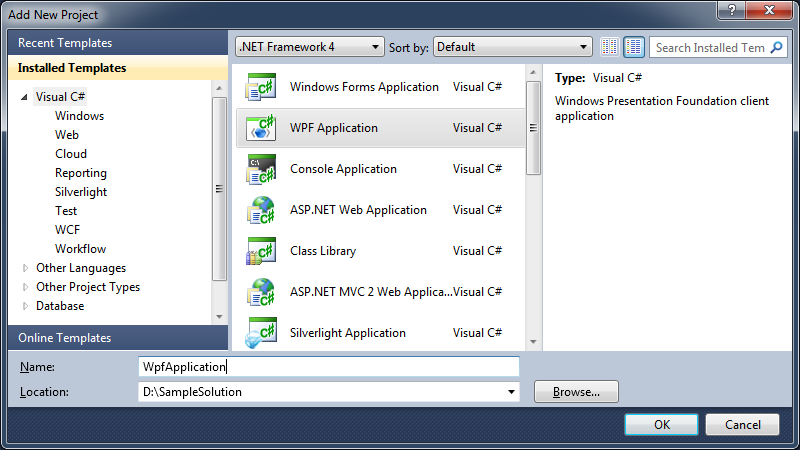


Рисунок л.р. 6.6. Добавление WPF приложения.

Вместе с проектами были автоматически созданы главные окна, MainWindow.xaml и MainPage.xml, для WPF и Silverlight соответственно. Так как эти сущности различны в двух версиях (в случае WPF это окно, в Silverlight – UserControl), их невозможно повторно использовать. Однако в них можно поместить приведенный в лекции #11 элемент управления ViewModelPresenter, который будет делегировать создание внутренней разметки представлению, соответствующему находящейся в DataContext модели представления.

MainWindow.xaml:

1: <Window x:Class="CrossPlatformApplication.MainWindow"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:view="clr-namespace:CrossPlatformApplication"

5: Title="{Binding Title}" SizeToContent="WidthAndHeight">

6:

7: <view:ViewModelPresenter ViewModel="{Binding}" />

8: </Window>

MainPage.xaml:

1: <UserControl x:Class="SilverlightApplication.MainPage"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:view="clr-namespace:CrossPlatformApplication">

5:

6: <view:ViewModelPresenter ViewModel="{Binding}" />

7: </UserControl>

Согласно принципам MVVM главному представлению приложения должна соответствовать модель представления. Для начала достаточно пустого класса, общего для WPF и Silverlight:

1: [Export]

2: [PartCreationPolicy(CreationPolicy.NonShared)]

3: [ExportMetadata(AopExtensions.AspectMetadata,

4: Aspects.NotifyPropertyChanged)]

5: public class MainViewModel : IEntitledViewModel

6: {

7: public string Title

8: {

9: get { return "Test Application"; }

10: }

11: }

Затем для корректного запуска приложения необходимо добавить логику инициализации IoC контейнера MEF и MVVM окружения, а также модифицировать логику отображения главных окон. Как в WPF, так и в Silverlight приложении модифицируется код App.xaml и App.xaml.cs.

App.xaml в WPF:

1: <Application x:Class="WpfApplication.App"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

4: <Application.Resources>

5:

6: </Application.Resources>

7: </Application>

App.xaml.cs в WPF:

1: public partial class App

2: {

3: #region Injected properties

4:

5: [Import]

6: public ChildViewManager ChildViewManager

7: { private get; set; }

8:

9: #endregion

10:

11: protected override void OnStartup(StartupEventArgs e)

12: {

13: base.OnStartup(e);

14:

15: // Create interception configuration

16: var cfg = new InterceptionConfiguration()

17: .AddAopInterception();

18:

19: var container = new CompositionContainer(

20: new InterceptingCatalog(new AggregateCatalog(

21: new DirectoryCatalog(".", "\*.exe"),

22: new DirectoryCatalog(".", "\*.dll")), cfg));

23:

24: var locator = new MefServiceLocator(container);

25: ServiceLocator.SetLocatorProvider(() => locator);

26: container.ComposeExportedValue<IServiceLocator>(locator);

27: container.ComposeParts(this);

28:

29: new MainWindow { DataContext =

30: locator.GetInstance<MainViewModel>() }.Show();

31: }

32: }

App.xaml в Silverlight:

1: <Application x:Class="SilverlightApplication.App"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

4: <Application.Resources>

5:

6: </Application.Resources>

7: </Application>

App.xaml.cs в Silverlight:

1: public partial class App

2: {

3: public App()

4: {

5: Startup += Application\_Startup;

6: Exit += Application\_Exit;

7: UnhandledException += Application\_UnhandledException;

8:

9: InitializeComponent();

10: }

11:

12: #region Injected properties

13:

14: [Import]

15: public ChildViewManager ChildViewManager

16: { private get; set; }

17:

18: #endregion

19:

20: private void Application\_Startup

21: (object sender, StartupEventArgs e)

22: {

23: // Create interception configuration

24: var cfg = new InterceptionConfiguration()

25: .AddAopInterception();

26:

27: var container = new CompositionContainer(

28: new InterceptingCatalog(

29: new DeploymentCatalog(), cfg));

30: var locator = new MefServiceLocator(container);

31: ServiceLocator.SetLocatorProvider(() => locator);

32: container.ComposeExportedValue<IServiceLocator>(locator);

33: container.ComposeParts(this);

34:

35: RootVisual = new MainPage

36: { DataContext = locator.GetInstance<MainViewModel>() };

37: }

38:

39: private void Application\_Exit(object sender, EventArgs e)

40: {

41:

42: }

43:

44: private void Application\_UnhandledException

46: (object sender, ApplicationUnhandledExceptionEventArgs e)

47: {

48: if (!System.Diagnostics.Debugger.IsAttached)

49: {

50: e.Handled = true;

51: Deployment.Current.Dispatcher.BeginInvoke(

52: delegate { ReportErrorToDOM(e); });

53: }

54: }

55:

56: private void ReportErrorToDOM

57: (ApplicationUnhandledExceptionEventArgs e)

44: {

45: try

46: {

47: string errorMsg = e.ExceptionObject.Message

48: + e.ExceptionObject.StackTrace;

49: errorMsg = errorMsg.Replace('"',

50: '\'').Replace("\r\n", @"\n");

51:

52: System.Windows.Browser.HtmlPage.Window.Eval(

53: "throw new Error(\"Unhandled Error in Silverlight Application "

54: + errorMsg + "\");");

55: }

56: catch (Exception)

57: {

58: }

59: }

60: }

Как видно из кода, главные окна приложения будут отображать представление MainView, соответствующее модели представления MainViewModel. Пусть это представление будет содержать простой текст «Hello world»:

1: <UserControl x:Class="CrossPlatformApplication.MainView"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

5: xmlns:mc

6: ="http://schemas.openxmlformats.org/markup-compatibility/2006"

7: mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

8:

9: <Grid x:Name="LayoutRoot" Background="White" MinHeight="150"

10: MinWidth="200">

11: <TextBlock Text="Hello world"

12: HorizontalAlignment="Center"

13: VerticalAlignment="Center" />

14: </Grid>

15: </UserControl>

Естественно, как говорилось в лекции #12, необходимо создать представление в Silverlight проекте и добавить ссылку на него в WPF проект. Теперь оба приложения готовы к запуску и отображают на своем главном окне сообщение «Hello world».

Следующим этапом следует некоторое усложнение примера – размещенная на главном представлении кнопка будет отображать модальное дочернее окно, считывающее строку у пользователя. Введенные данные модифицируют исходный текст на главном представлении.

В первую очередь необходимо создать модель представления диалога, которая наследуется от введенного в лекции #14 класса ModalViewModelBase:

1: [Export]

2: [PartCreationPolicy(CreationPolicy.NonShared)]

3: [ExportMetadata(AopExtensions.AspectMetadata,

4: Aspects.NotifyPropertyChanged)]

5: public class TextInputModalChildViewModel :

6: ModalChildViewModelBase

7: {

8: public virtual string Text { get; set; }

9: }

Далее создается представление, содержащее поле ввода, привязанное к свойству Text:

1: <UserControl

2: x:Class="CrossPlatformApplication.TextInputModalChildView"

3: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

4: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

5: xmlns:i=

6: "http://schemas.microsoft.com/expression/2010/interactivity"

7: xmlns:ic=

8: "http://schemas.microsoft.com/expression/2010/interactions"

9: xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

10: xmlns:mc=

11: "http://schemas.openxmlformats.org/markup-compatibility/2006"

12: mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

13:

14: <StackPanel x:Name="LayoutRoot" Background="White">

15: <TextBlock Text="Введите новый текст" />

16: <TextBox Text="{Binding Text, Mode=TwoWay}" />

17: <StackPanel Orientation="Horizontal"

18: HorizontalAlignment="Right">

19: <Button Content="OK" Command="{Binding CloseCommand}"

20: Width="80" Height="30" Margin="6">

21: <i:Interaction.Triggers>

22: <i:EventTrigger EventName="Click">

23: <ic:ChangePropertyAction

24: PropertyName="ModalResult" Value="True"

25: TargetObject="{Binding}" />

26: </i:EventTrigger>

27: </i:Interaction.Triggers>

28: </Button>

29: <Button Content="Cancel"

30: Command="{Binding CloseCommand}" Width="80"

31: Height="30" Margin="6" />

32: </StackPanel>

33: </StackPanel>

34: </UserControl>

Здесь используется библиотека Expression Interactions, входящая в состав Microsoft Expression Blend 4, которая позволяет использовать кроссплатформенные Silverlight/WPF триггеры, схожие по функциональности с WPF триггерами, а также добавляют понятие поведения элемента управления, аналога которому в стандартной поставке WPF нет.

Далее, для работы с асинхронными моделями представления модальных дочерних окон удобно использовать библиотеку Reactive Extensions (Rx Framework):

1: public static class ObservableHelper

2: {

3: public static IObservable<EventPattern<EventArgs>>

4: ObserveClosed(this ICloseableViewModel childViewModel)

5: {

6: return Observable.FromEventPattern(

7: handler => childViewModel.Closed += handler,

8: handler => childViewModel.Closed -= handler);

9: }

10:

11: public static IObservable<T> SelectSender<T>

12: (this IObservable<EventPattern<EventArgs>> observable)

13: {

14: return observable.Select(ev => (T)ev.Sender);

15: }

16:

17: public static IObservable<T> WhereSucceeded<T>

18: (this IObservable<T> observable)

19: where T : IModalChildViewModel

20: {

21: return observable.Where(vm => vm.ModalResult.HasValue

22: && vm.ModalResult.Value);

23: }

24: }

1: public static class ViewModelExtension

2: {

3: /// <summary>

4: /// Resolves and shows Closeable View Model of

5: /// type <typeparamref name="T" />

6: /// </summary>

7: public static IObservable<T> ResolveAndShow<T>

8: (this IServiceLocator serviceLocator,

9: Action<T> prepareAction = null)

10: where T : ICloseableViewModel

11: {

12: var viewModel = serviceLocator.GetInstance<T>();

13:

14: if (prepareAction != null)

15: {

16: prepareAction(viewModel);

17: }

18:

19: IObservable<T> result = Observable.FromEventPattern

20: (handler => viewModel.Closed += handler,

21: handler => viewModel.Closed -= handler)

22: .SelectSender<T>();

23:

24: viewModel.Show();

25:

26: return result;

27: }

28: }

Класс ObservableHelper позволяет, с одной стороны конструировать IObservable последовательность из события закрытия окна, и с другой стороны предоставляет удобные LINQ-подобные методы для взаимодействия с данной последовательностью. Так, метод WhereSucceeded налагает на последовательность ограничение, которое позволяет ей выполниться лишь в случае завершения диалога с положительным ModalResult (что, как правило, происходит при подтверждении какой-либо операции).

Класс ViewModelExtension представляет метод, который одним вызовом разрешает закрываемую модель представления из IoC контейнера, показывает её и возвращает IObservable последовательность её закрытия.

Используя введенные сущности, в модели представления главного окна добавляется логика отображения диалога. Для видимого эффекта использования диалога введенный в окне текст заменяет приветственный текст по умолчанию:

1: [Export]

2: [PartCreationPolicy(CreationPolicy.NonShared)]

3: [ExportMetadata(AopExtensions.AspectMetadata,

4: Aspects.NotifyPropertyChanged)]

5: public class MainViewModel : EntitledViewModelBase

6: {

7: // Private fields

8: private string \_text = "Hello world";

9:

10: public virtual string Text

11: {

12: get { return \_text; }

13: protected set { \_text = value; }

14: }

15:

16: #region Commands

17:

18: private ICommand \_showTextInputCommand;

19:

20: public ICommand ShowTextInputCommand

21: {

22: get

23: {

24: return \_showTextInputCommand ??

25: (\_showTextInputCommand =

26: new ActionCommand(ShowTextInput));

27: }

28: }

29:

30: #endregion

31:

32: #region Injected properties

33:

34: [Import]

35: public IServiceLocator ServiceLocator { private get; set; }

36:

37: #endregion

38:

39: public override string Title

40: {

41: get { return "Test Application"; }

42: }

43:

44: private void ShowTextInput()

45: {

46: ServiceLocator

47: .ResolveAndShow<TextInputModalChildViewModel>()

48: .WhereSucceeded()

49: .Subscribe(vm => Text = vm.Text);

50: }

51: }

Соответственно, также изменяется разметка представления главного окна:

1: <UserControl x:Class="CrossPlatformApplication.MainView"

2: xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

3: xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

4: xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

5: xmlns:mc=

6: "http://schemas.openxmlformats.org/markup-compatibility/2006"

7: mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

8:

9: <Grid x:Name="LayoutRoot" Background="White"

10: MinHeight="150" MinWidth="200">

11: <TextBlock Text="{Binding Text}"

12: HorizontalAlignment="Center"

13: VerticalAlignment="Center" />

14: <Button Content="Изменить"

15: Command="{Binding ShowTextInputCommand}"

16: Width="80" Height="30"

17: HorizontalAlignment="Right"

18: VerticalAlignment="Bottom" Margin="6" />

19: </Grid>

20: </UserControl>

## Краткие итоги

В рамках данной лекции было написано несложное кроссплатформенное Silverlight/WPF приложение, агрегирующее все наработки предыдущих лекций. Выполненное в соответствии с шаблоном MVVM, оно стремится максимально повторно использовать как код, так и разметку, использует MEF для гибкого разрешения зависимостей между сущностями, в том числе между моделями представления и представлениями, применяет аспект INotifyPropertyChanged на модели представления, избавляя от необходимости ручного программирования логики нотификации об изменениях в каждом свойстве. При этом объем содержательного кода минимален и приложение крайне просто расширяется любым функционалом. Приведенный пример можно использовать как каркас для Silverlight/WPF приложений с богатым пользовательских интерфейсом.

# Список литературы

1. Raffaele Garofalo, Applied WPF 4 in Context, Apress, May 2011
2. Microsoft Application Architecture Guide, Microsoft Press, October 2009
3. К. Андерсон Основы Windows Presentation Foundation. Пер. с англ. А. Слинкина — М.: ДМК Пресс, 2008.
4. Мэтью Мак-Дональд, Pro Silverlight 4 in C#, Apress, November 2010
5. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4 с примерами на С# 2010 Пер. с англ. –М.: ООО “И.Д. Вильямс”, 2011.
6. Нейгел, К. C# 4.0 и платформа .NET 4 для профессионалов. / К. Нейгел, Б. Ивьен, Дж. Глинн, К. Уотсон.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2011.

1. Это аналог «свойства по умолчанию» в Visual Basic. [↑](#footnote-ref-1)
2. Для этого следует добавить атрибут System.Windows.Markup.ContentPropertyAttribute в определение типа. [↑](#footnote-ref-2)