Technologie Windows Presentation Foundation

The Technology of Windows Presentation Foundation

Bakalářská práce

Vítězslav Kuliš

Vedoucí bakalářské práce: Ing. Václav Novák, CSc.

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

2010

Anotace

Tato bakalářská práce se zaměřuje na vývoj uživatelského rozhraní pomocí technologie Windows Presenatation Foundation (dále WPF). Po uvedení základních rysů a architektury porovnává vybrané oblasti s alternativní technologií Windows Forms. Dále se práce zabývá nasazením WPF v oblasti podnikových aplikací s využitím návrhového vzoru Model-View-ViewModel a popisuje s tím související problémy.

Abstract

This bachelor thesis is focused on developing of a user interface using the Windows Presentation Foundation (WPF). After an introduction of the main outlines and architecture of the WPF the thesis compares particular areas of the WPF with the alternative technology Windows Forms. Furthermore, the work also deals with the implementation of the WPF to the enterprise applications area using the Model-View-ViewModel design pattern and describes related issues.

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s §47b zákona č. 111/1998 Sb. V platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 5.1.2010

…..……

Vítězslav Kuliš

OBSAH

1 Úvod

Již od roku 1985 je možné vyvíjet programy pro operační systém Windows. S postupem času a uvedením nových verzí systému Windows přicházely i nové přístupy, jak psát programy pro tento systém. Od prvního přístupu, který využíval rozhraní Windows API a jazyk C, se dospělo až k vydání platformy NET. Framework. Tato platforma přinesla několik nových technologií a změn v podobě řízeného kódu, nového způsobu pro přístup k databázím nebo nový způsob psaní webových aplikací. Jednou z technologií byla i knihovna Windows Forms, která nám umožňuje vytvářet uživatelské rozhraní pro systém Windows.

Přestože Windows Forms je bohatě vybavené rozhraní, které již řadu let plní potřeby vývojářů po celém světě, během svého působení nedosáhlo žádných významnějších změn a navíc trpí několika nedostatky především týkající se přizpůsobení vzhledu. Výraznou změnou v tomto směru, ale i v dalších, přináší verze .NET Framework 3.0 a následující .NET Framework 3.5, kde jedním z hlavních přínosů je zcela nová technologie pro psaní uživatelského rozhraní, a to Windows Presentation Foundation (dále jen WPF).

Cílem této práce je popsat základní model grafického subsystému WPF uvnitř NET. Framework 3.5. Dále porovnat vybrané funkcionality WPF s dosavadním frameworkem Windows Forms a aplikovat je na konkrétních případech. Dalším cílem je výběr oblasti, kde jako uživatelské rozhraní dominují Windows Forms, a navrhnout pro tuto oblast řešení za pomoci vhodné metody ve WPF.

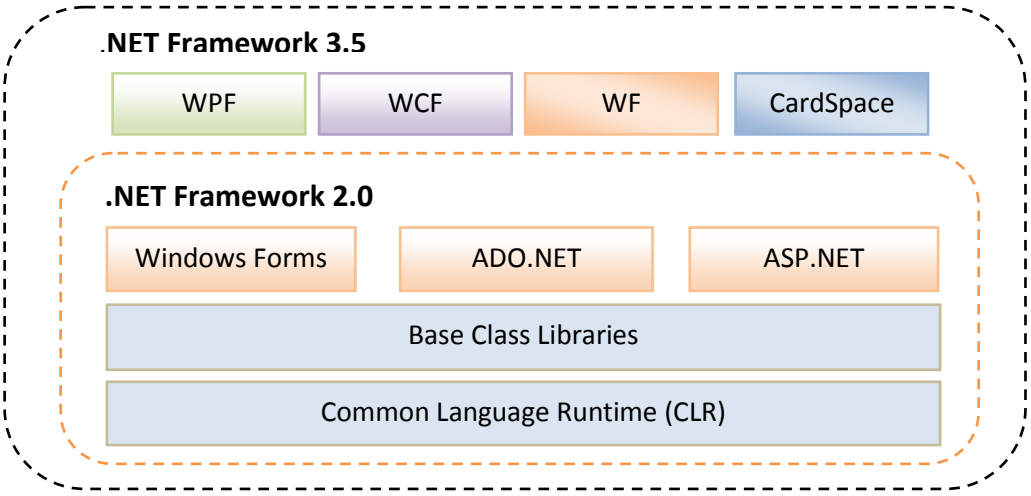
2 Windows Presentation Foundation

2.1 Představení WPF uvnitř .NET Framework 3.5

Uvolnění .NET Framework 3.0 následné verzi 3.5 bylo dalším krokem ve vývoji software na platformě Windows. Tento framework přináší několik zcela nových technologií, které významně ovlivnily pohled na řešení některých současných problematik. Je postaven na jádru předchozí verze, kterou rozšiřuje o následující technologie:

- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- Windows CardSpace

Obrázek 1:Model .NET 3.5



Jednou z problematik v oblasti vývoje software, které je v současné době věnována velká pozornost, je uživatelské rozhraní (UI). Požadavky na UI jsou kladeny jak ze strany uživatele, tak ze strany vývojáře či designéra. Uživatel vyžaduje přehledné a přívětivé rozhraní s pokročilými možnostmi vizualizace dat a interaktivními prvky. Naproti tomu programátor vyžaduje efektivní a pohodlný způsob, jak toho docílit. Z uvedených technologií .NET

Frameworku 3.5 je to právě technologie WPF, která se snaží vyhovět těmto požadavkům.

2.2 Hlavní rysy WPF

2.2.1 Hardwarová akcelerace

Koncept vykreslování uživatelského rozhraní ve WPF již není záležitostí GDI/GDI+, jak tomu bylo dříve, ale je postaveno nad rozhraním DirectX. Veškeré grafické prvky WPF jsou převáděny na 3D primitiva, textury a další objekty DirectX a následně vykreslovány na grafické kartě.[1] CPU je tedy odlehčeno od zpracování složitých a časově náročných grafických operací. Grafický hardware však není nezbytný a v případě jeho absence je zodpovědnost vykreslování přenesena na CPU.

2.2.2 Nezávislost na rozlišení

Dosud bylo zvykem, že jednotlivé rozměry prvků a jejich podřízených vlastností byly určovány v pixelech. Není tomu však ve WPF, kde jsou tyto hodnoty určovány v tzv. *device-independent units*[2]. Jednotkou je $^1/_{96}$ palce. Tento fakt má za následek, že při změně nastavení DPI zobrazovacího zařízení se nezmění velikost rozměrů okna, ale aplikace se plynule přizpůsobí novému nastavení. WPF aplikuje následující rovnici pro výpočet skutečné velikosti v pixelech.

$$jendotky\ v\ pixelech = jednotky\ nezávislé\ na\ rozlišení\ \times systémové\ DPI$$

Jestliže naše systémové nastavení ve Windows je 96 DPI, pak uvedená rovnice vrátí výsledek 1 pixel a jednotky si v tomto případě odpovídají. Pokud se rozhodneme rozlišení změnit např. na hodnotu 120, z důvodu monitoru pracujícího ve vysokém rozlišení, vzorec vrátí hodnotu 1.25 pixelů. V reálném případě si můžeme představit grafický objekt, jehož šířka je 96 v jednotkách nezávislých na rozlišení, což v systému s nastavením 96 DPI představuje

[1] NATHAN, Adam, LEHENBAUER, Daniel. Windows Presentation Foundation Unleashed. 1st edition. Indianapolis : Sams Publishing, 2007. 638 s. ISBN 0-672-32891-7.

[2] MATHEW, M.: Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second edition. Apress U.S.A. 2008. ISBN 978-1590599556

1 palec. Při změně nastavení na 120 DPI a aplikování vzorce bude skutečná šířka objektu 120 pixelů, ale stále 1 palec. Kdyby WPF uvádělo rozměry v pixelech, došlo by ke změně velikosti objektu, což není v tomto případě žádoucí.

2.2.3 Integrace

Dnešní doba je zahlcena informacemi v různých formách, které je nutné prezentovat. Programování aplikací ve Windows dosud vyžadovalo pro zpracování různých druhů informací různé technologie. WPF tento přístup mění a kromě jednotného přístupu k UI je do modelu WPF integrována podpora pro práci s 2D a 3D grafikou, multimédii (video, zvuk) a dokumenty.

2.2.4 Jazyk XAML

XAML (XML for Application Markup language) ve volném překladu znamená značkovací jazyk XML pro aplikace. Tato novinka nám umožňuje ve WPF deklarativně definovat uživatelské rozhraní ve stylu zápisu XML jazyka. Návrh aplikace s pomocí XAML je velmi podobný návrhu webových formulářů v ASP.NET. Primárně se zde rovněž vychází z modelu založeného na událostech, ale k dispozici máme i další možnosti. Nicméně, uživatelské rozhraní je stále možné psát v procedurálním kódu jednoho z rodiny .NET jazyků.

Jedna z předních výhod XAML je, že umožňuje takovou separaci vzhledu aplikace od souvisejícího aplikačního kódu, než tomu bylo dosud. Z toho plyne i zdokonalení spolupráce mezi grafickým návrhářem a programátorem. Kromě toho je výhodou již samotný deklarativní kód, který je mnohem kratší a přehlednější. [3]

Deklarace elementů v XAML kódu je ekvivalentní vytváření odpovídajících .NET objektů. Nastavení XAML atributů je ekvivalentní nastavení vlastností objektu stejného názvu či připojení událostí stejného

[3] Xu, J. Ph.D.: Practical WPF Graphics Programming. UniCAD Publishing U. S. A 2007. ISBN 978-0-9793725-1-3

názvu.[4] Následující ukázka znázorňuje deklaraci stejného prvku v XAML a procedurálním kódu.

Ukázka 1: Deklarativní vs. procedurální kód

```
<TextBlock Background="Black"></TextBlock>

TextBlock textBlock = new TextBlock();
textBlock.Background = Brushes.Black;
```

2.2.5 Bohatá kompozice a přizpůsobivost

WPF nabízí velice pokročilou kompozici ovládacích prvků. Znamená to, že prvky lze vzájemně zanořovat, a to až do takových případů jako vložení videa do položek menu nebo naplnění prvku `ComboBox` položkami s animovanou grafikou. Především je důležité, že tato kompozice nás stojí minimální úsilí a vše lze zajistit v deklarativním kódu.

Další významnou vlastností WPF prvků je přizpůsobivost vzhledu. Prvky je možné stylovat podobně jako CSS v HTML. Pomocí šablon lze kompletně změnit vzhled daného prvku. Vzhledové vlastnosti se na prvky napojují staticky nebo dynamicky. V případě druhé možnosti je možné vzhled měnit za běhu aplikace.

2.2.6 Vlastnosti závislostí

Základem všech .NET objektů jsou vlastnosti a události. WPF přichází s novým typem vlastností, které se nazývají *vlastnosti závislostí*. Na rozdíl od původních vlastností se liší tím, že jsou závislé na dalších vlivech a přidávají funkcionality navíc. Ve WPF se staly základem pro datové vazby, animace nebo styly a další. Nicméně, práce s nimi se neliší od běžných vlastností. Jejich velkým přínosem je zabudovaný notifikační systém, který dokáže automaticky reagovat na změny vlastnosti, a možnost dědění skrze strom elementů.

Základem pro vlastnosti závislostí je třída `DependencyObject`. Samotná definice vlastnosti závislostí se liší od běžných vlastností a spočívá v deklaraci statického pole typu `DependencyProperty`. Abychom mohli

[4] MATHEW, M.: Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second edition. Apress U.S.A. 2008. ISBN 978-1590599556

vlastnost v systému závislostí využívat, je nutné provést registraci prostřednictvím metody `Register` třídy `DependencyProperty`. Velmi významnou roli přináší parametr typu `PropertyMetadata`, který je nepovinnou součástí této metody. Pomocí tohoto parametru je možné určit defaultní hodnotu objektu, událost pro validaci a úpravu hodnoty vlastnosti, událost vyvolanou při změně hodnoty, implicitní nastavení datových vazeb a řadu dalších voleb. Pro získání hodnoty vlastnosti se využívá metod ze třídy `DependecyObject`, a to `GetValue` a `SetValue`, které se zapouzdří do běžné nestatické vlastnosti, takže i přesto, že vlastnost závislosti je definována jako statická, se jí můžeme dotazovat instančně.

Ukázka 2: Definice vlastnosti závislostí

```
public static DependencyProperty WorkSourceItemProperty;

WorkSourceItemProperty =
      DependencyProperty.Register("WorkSourceItem",
      typeof(EmployeeOnProject),typeof(WorkSourceChartItem2D),
            new PropertyMetadata(null,
      OnWorkSourceItemPropertyChanged));

public EmployeeOnProject WorkSourceItem
{
      get
      {
            return
(EmployeeOnProject)GetValue(WorkSourceItemProperty);
      }
      set
      {
            SetValue(WorkSourceItemProperty, value);
      }
}
```

Dalším typem vlastností závislostí jsou takzvané *přidružené vlastnosti* (*atached properties*)[5]. Některé prvky definují takové vlastnosti závislostí, které sami nevyužívají, ale distribuují je pro libovolné jiné prvky. Jedná se právě o přidružené vlastnosti. Především takové vlastnosti definují prvky kontejnerového typu. Podřízené prvky těchto vlastností využívají a díky tomu jsou odlehčeny od implementace navíc, která by jinak byla nezbytná například pro pozicování v rodičovském prvku. Deklarace přidružených vlastností je analogická prvnímu případu s výjimkou metody pro registraci. Místo metody

[5] NAGEL, Christian , et al. C# 2008 : Programujeme profesionálně. 1. vyd. Brno : Computer Press, 2009. 2 sv. (1128, 776 s.). ISBN 978-80-251-2401-7.

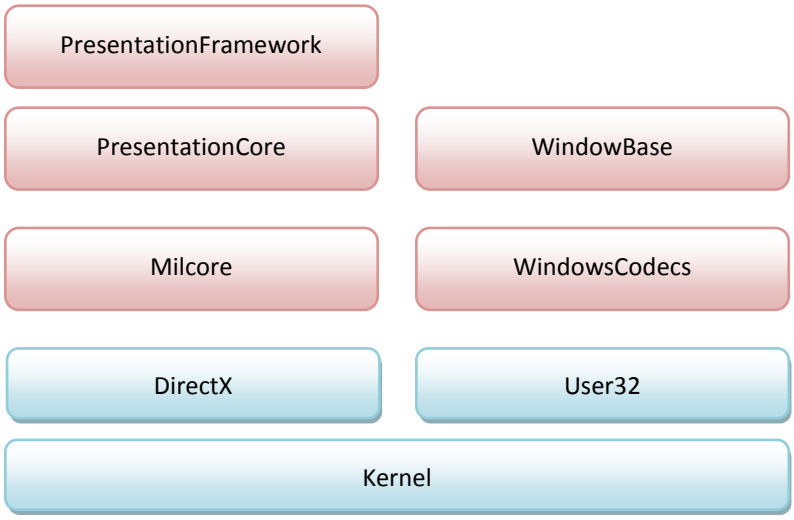
`Register` se využívá metoda `RegisterAttached` a získání hodnot vlastnosti se děje výhradně statickou formou.

2.3 Architektura WPF

WPF je postaveno na vícevrstvé architektuře, kde hlavní programovací model je napsán v řízeném kódu s výjimkou knihoven Milcore a WindowCodecs. Milcore je napsána v neřízeném kódu především z důvodu její těsné integrace s knihovnou DirectX. Právě prostřednictvím této knihovny dochází k přeměně .NET objektů do grafických primitiv a textur v DirectX.[6]

Na obrázku níže je model architektury WPF, kde fialovou barvou jsou znázorněny komponenty WPF a modrou nativní knihovny systému Windows.

Obrázek 2: Model architektury WPF



PresentationFramework – zahrnuje především typy, které reprezentují ovládací prvky jako `Window`, `Panel`, `TextBox` atd.

PresentationCore – tato knihovna je zdrojem základních typů ve WPF, od kterých je většina ovládacích prvků či tvarů odvozena. Příkladem jsou `Visual`, `UIElement` nebo `UIElement3D`.

[6] MSDN Library : Windows Presentation Foundation [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

WindowsBase – kromě základní typů WPF jako `DispatcherObject` a `DependencyObject` poskytuje typy, které je možné využít i mimo rozsah WPF, např. `ObservableCollection`.

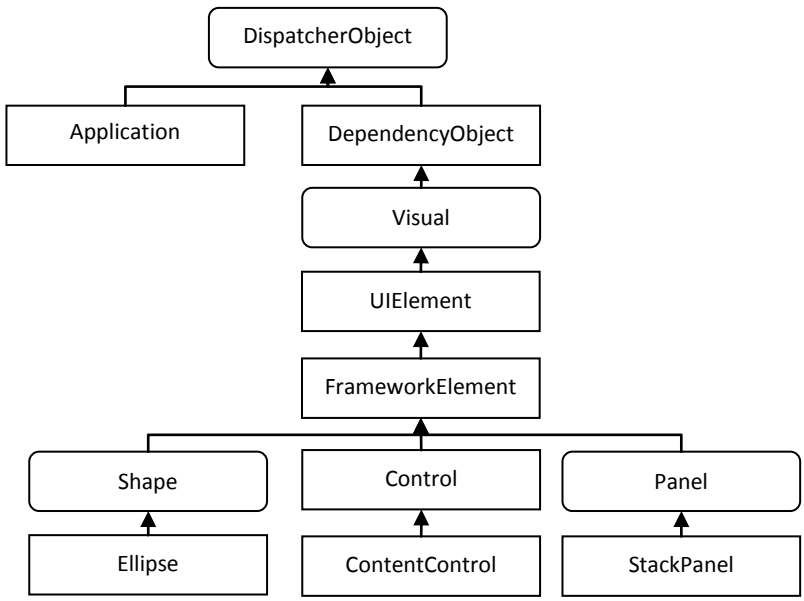
WindowsCodecs – API pro práci s bitmapovou grafikou.

Milcore – jádro vykreslovacího systému WPF. Poskytuje převod .NET objektů do neřízeného kódu DirectX.

2.3.1 Hierarchie tříd

Každý objektově orientovaný jazyk má svůj hierarchický model tříd. Nemůže tomu být jinak u WPF. Na rozdíl od Windows Forms se hierarchie tříd WPF značně liší. Následující obrázek zachycuje alespoň vztahy bázových tříd.

Obrázek 3: Hierarchie bázových tříd WPF



Kořenem stromu je abstraktní třída `DispatcherObject`. Stejně jako v případě Windows Forms, i WPF zakazuje volání metod či vlastností mimo podproces, ve kterém byly vytvořeny. Objekt třídy odvozené od `DispatcherObject`, tedy každý WPF objekt, má k dispozici vlastnost `Dispatcher`, jež udržuje objekt, pomocí kterého je možné provést přepnutí podprocesu.

Jedna ze dvou tříd, která je přímo odvozená od `DispatcherObject`, nese název `Application`. Aplikace WPF může vytvořit pouze jednu instanci objektu `Application`, která není viditelná a slouží jako centrální bod pro zbývající části programu. Druhou třídou ve WPF odvozenou od `DispatcherObject` je `DependencyObject`. Všechny třídy využívající vlastnosti závislostí jsou odvozeny právě od `DependencyProperty`. Abstraktní třída `Visual` je základní třídou pro všechny vizuální prvky. Třída `UIElement` poskytuje základní definici událostí pro práci se vstupním zařízením, metody pro vykreslování a pro práci s rozložením. Následující třída v hierarchii `FrameworkElement` již implementuje některé funkcionality navržené ve třídě `UIElement`. Navíc přidává podporu pro datové vazby a styly.[7]

Na rozdíl od Windows Forms, kde všechny vizuální elementy jsou odvozeny od třídy `Control` a označovány jako ovládací prvky, WPF na tuto problematiku pohlíží jiným pohledem. Uvnitř WPF je ovládacím prvkem pouze takový objekt, který je interaktivní. Nejvíce takových prvků je odvozeno právě od třídy `Control` (tato třída nemá nic společného se stejnojmennou třídou ve Windows Forms). Na stejné úrovni hierarchie se nacházejí také abstraktní třídy `Shape` a `Panel`. Třída `Shape` je základem pro elementy určující tvar jako `Rectangle`, `Ellipse` atd. Od třídy `Panel` se odvozují všechny prvky, které slouží jako kontejner pro prvky podřízené. Panel zároveň definuje virtuální metody pro správu rozmístění těchto prvků.[8]

Zde popsané bázové třídy jsou jen zlomkem z celé hierarchie tříd. Jsou však těmi nejdůležitějšími a je užitečné je znát.

[7] PETZOLD, Charles. Mistrovství ve Windows Presention Foundation. 1. vyd. Brno : Computer Press, a.s. , 2008. 928 s. ISBN 978-80-251-2141-2.

[8] NAGEL, Christian , et al. C# 2008 : Programujeme profesionálně. 1. vyd. Brno : Computer Press, 2009. 2 sv. (1128, 776 s.). ISBN 978-80-251-2401-7.

2.3.2 Strom elementů

Jedním z nových konceptů ve WPF jsou takzvané stromy elementů. Jejich podstatou je uspořádání množiny elementů do hierarchické struktury. Stromy jsou dále rozlišeny na logické a vizuální.

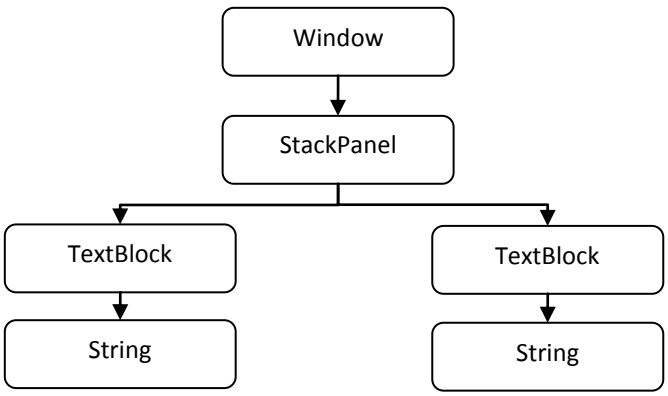
2.3.2.1 Logický strom

Hierarchická struktura logického stromu odpovídá přesně struktuře elementů definovaných v XAML nebo v procedurálním kódu. Logické stromy mají odpovědnost za dědičnost vlastností závislostí mezi elementy, předávání směrovaných událostí, hledání zdrojů a vyhledávání názvů elementů pro datové vazby.[9] V procedurálním kódu lze využít třídu `LogicalTreeHelper` pro práci s tímto typem stromu. Logický strom z ukázky níže znázorňuje obrázek 4.

Ukázka 3: Ukázka XAML kódu

```
<Window
xmlns=
"http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Strom elementů">
  <StackPanel Margin="4">
        <TextBlock>TextBlock1</TextBlock>
        <TextBlock>TextBlock2</TextBlock>
  </StackPanel>
</Window>
```

Obrázek 4: Logický strom

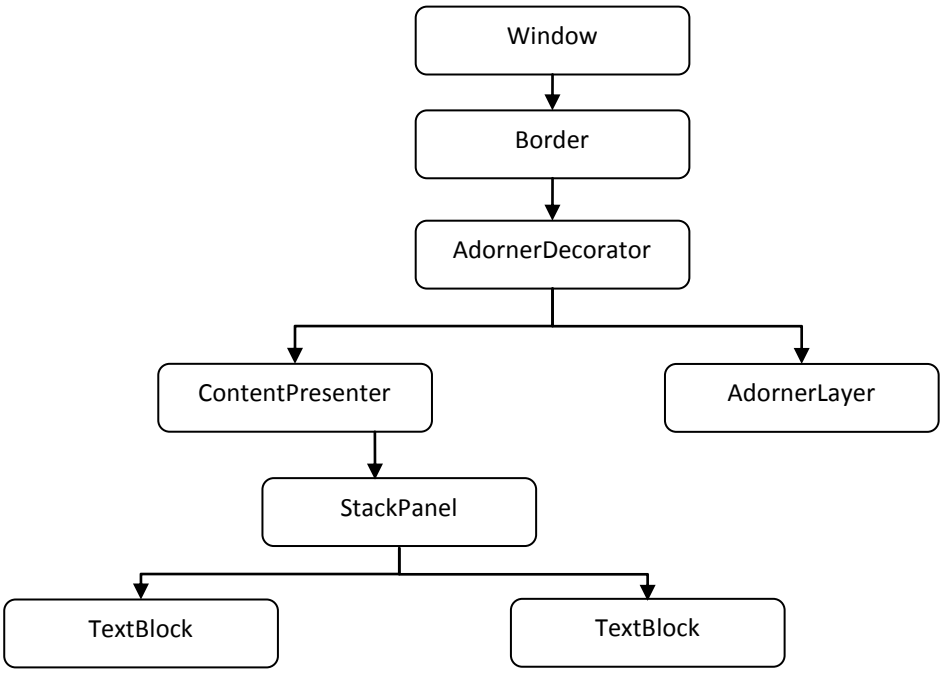


[9] MOSERS, Christian. WPF Tutorial.net : Logical- and Visual Tree [online]. [2008] [cit. 2009-12-10]. Dostupný z WWW: <http://www.wpftutorial.net/LogicalAndVisualTree.html>.

2.3.2.2 Vizuální strom

Vizuální strom zahrnuje všechny elementy, které jsou ve finální fázi vykresleny na grafickém hardware. Na rozdíl od logického stromu dekomponuje elementy na menší části, které pak společně tvoří výsledný vzhled grafického prvku. V podstatě rozšiřuje logický strom s tím rozdílem, že nezahrnuje elementy, které nejsou odvozeny od tříd Visual nebo Visual3D. Pro procházení stromu je možné využít statickou třídu `VisualTreeHelper`. Vizuální strom z ukázky 3 je na následujícím obrázku.

Obrázek 5: Vizuální strom



3 Porovnání WPF a Windows Forms

S příchodem WPF čelí vývojáři otázce výběru grafického frameworku na platformě .NET pro nové aplikace nebo migraci z předchozího frameworku Windows Forms. V souvislosti s WPF je především vyzdvihována stránka týkající se nastavitelnosti a přizpůsobitelnosti vzhledu, ale WPF dosáhlo výrazných inovací i v oblastech pro práci s daty, událostmi či vytváření vlastních ovládacích prvků. Právě těmto aspektům budou věnovány následující kapitoly.

3.1 Model událostí

Grafické aplikace na platformě .NET využívají tzv. *Event-driven model.* V podstatě se zde jedná o zachycení a následné zpracování události generované ovládacím prvkem. Tato událost může být např. kliknutí myši, stisknutí klávesy apod. Základní princip je v obou zúčastněných technologiích podobný.

3.1.1 Směrované události ve WPF

Hlavní změnou ve zpracování událostí ve WPF oproti Windows Forms jsou tzv. *routed event*[10]. Tento pojem můžeme do češtiny přeložit jako směrování. Ve Windows Forms zpracovává událost pouze element, který je jejím vlastníkem. Pokud například stiskneme tlačítko nad prvkem typu `Button`, zpracovávat událost bude právě tento prvek a žádný jiný. Naproti tomu ve WPF je stejná situace o mnoho sofistikovanější, jelikož `Button` i ostatní ovládací prvky mohou obsahovat kompozici prvků dalších. Otázkou je, který z nich bude danou událost zachycovat. Odpovědí jsou právě směrované události.

[10] *MSDN Library : Windows Presentation Foundation* [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

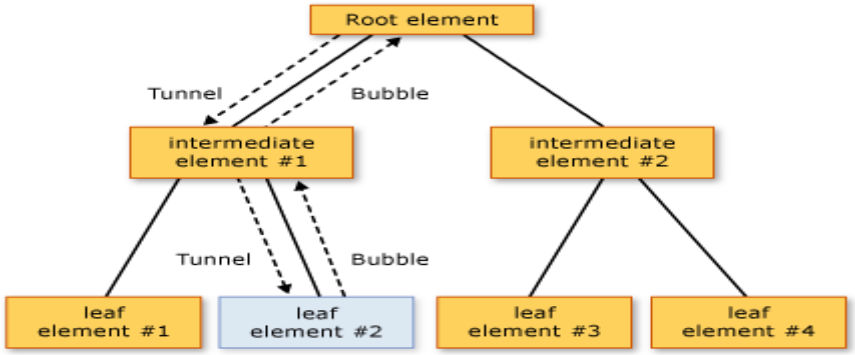
3.1.1.1 Strategie předávání událostí

Směrované události se od běžných liší tím, že se předávají skrze strom elementů. Zmíněné předávání je možné řídit a lze vybrat z několika strategií, které determinují jeho směr.

Strategie jsou následující:

- Bubbling
- Direct
- Tunneling

Obrázek 6: Směrování událostí ve WPF (převzato[11])



Tunneling strategie určuje předávání události od kořene stromu elementů ke zdrojovému elementu, který událost vyvolal.

Bubbling určuje předávání události od elementu, který ji zachytil, ke kořeni stromu elementů.

Direct pracuje stejně jako běžné události. Událost vznikne pouze na zdrojovém elementu.[12]

Systém WPF používá konvenci pro rozlišení strategie událostí na základě jejich názvů. Tunelovací události jsou na rozdíl od probublávajících uvedeny s prefixem *Preview*.

---

[11] *MSDN Library : Windows Presentation Foundation* [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

[12] NATHAN, Adam, LEHENBAUER, Daniel. Windows Presentation Foundation Unleashed. 1st edition. Indianapolis : Sams Publishing, 2007. 638 s. ISBN 0-672-32891-7.

Směrování událostí zajišťuje, že stejná událost může být zavolána na více posluchačích v rámci stromu závislostí.[13]

Pro názorný příklad může posloužit obrázek 6, kde by *root element* a *leaf element #2* obsluhovaly stejnou událost. Pokud prvek *leaf element #2* vyvolá událost, tak v případě, že se jedná o tunelovací strategii, je zpracována nejprve v prvku *root element* a až poté ve zdrojovém prvku *leaf element #2*. V probublávající strategii by byl postup opačný. V obou případech by skrze prvek *intermediate element 1* byla událost pouze předána dál, protože nedefinuje její zpracování.

3.1.1.2 RoutedEventArgs

Běžná signatura události ve WPF a Windows je téměř shodná. Důležitým aspektem u WPF směrované události je parametr typu `RoutedEventArgs`. Právě prostřednictvím tohoto parametru si směrované události sdílejí data. Typ `RoutedEventArgs` vystavuje několik vlastností, z nichž pro směrování jsou nejdůležitější `Source` a `Handled`. `Source` obsahuje instanci zdrojového elementu, na kterém byla událost vyvolána. Vlastnost `Handled` může nabývat logické hodnoty `true` nebo `false`. Implicitně je nastavena na hodnotu `false`, pokud se ale hodnota změní na `true`, událost se již dále nepředává ve stromu elementů.

3.1.1.3 Připojené události

Jednou z výhod spojenou se směrovanými událostmi jsou takzvané připojené události, které nám dávají možnost zpracovat danou událost i na prvku, který ji nedefinuje.[14] Například tak můžeme určit, aby po kliknutí na tlačítko byl zdrojem události jiný prvek. Zajistit to lze způsobem prezentovaným na následující ukázce.

---

[13] *MSDN Library : Windows Presentation Foundation* [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

[14] NATHAN, Adam, LEHENBAUER, Daniel. Windows Presentation Foundation Unleashed. 1st edition. Indianapolis : Sams Publishing, 2007. 638 s. ISBN 0-672-32891-7.

Ukázka 4: Připojená událost ve WPF

```
<StackPanel Grid.Row="2" ButtonBase.Click="Button_Click">
   <Button>Zpracuj</Button>
</StackPanel>
```

3.2 Datové vazby (Data Binding)

Data Binding (volně přeloženo *datové vazby*) je obecně proces, který umožňuje provázání dat mezi grafickým rozhraním a datovým modelem aplikace. V minulosti byl programátor nucen synchronizaci dat provádět pomocí vlastních procedurálních technik. Nyní v dobách moderního programování je tento proces doslova automatizovaný.[15] Provedení změn na jedné straně reflektuje změny na straně druhé a naopak. Implementace tohoto procesu dosáhla ve WPF oproti Windows Forms značných změn a rozšíření. V následujících kapitolách budou nastíněny problematiky a srovnání případů data bindingu z pohledu obou technologií.

3.2.1 Datové vazby uvnitř Windows Forms

Data binding uvnitř Windows Forms umožňuje svazovat data z různých datových struktur, jako jsou pole, kolekce nebo tabulky relační databáze. Mimo to lze vázat data na jednotlivé vlastnosti ovládacích prvků. Tyto vlastnosti jsou např. `Text`, `Backgound`, `Size` atd.

3.2.1.1 Typy datových vazeb

Data binding ve Windows Forms můžeme rozdělit na dva typy[16]:

- Jednoduhý data binding (*Simple data binding*)
- Komplexní data binding (*Complex data binding*)

V prvním případě se jedná o využití data bindingu s prvky, které zobrazují pouze jednu hodnotu jako například komponenta `TextBox`. Ve druhém

[15] PETZOLD, Charles. *Mistrovství ve Windows Presention Foundation*. 1. vyd. Brno : Computer Press, a.s. , 2008. 928 s. ISBN 978-80-251-2141-2.

[16] *MSDN Library : Windows Forms* [online]. c2009 [cit. 2009-11-01]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ef2xyb33.aspx>.

případě je ovládací prvek svázán s více než jednou hodnotou, typicky `DataGridView`, `ComboBox` nebo `ListBox`.

Ukázka 5: Jednoduchý data binding na prvku `ComboBox`

```
orderStatesCmb.DataBindings.Add("SelectedValue", orders,
"Status");
```

Ukázka 6: Komplexní data binding ve Windows Forms

```
IList<Order> orders = Orders.GetOrders();
this.ordersGridView.DataSource = orders;
```

3.2.1.2 BindingContext

Základem data bindingu ve Windows Forms jsou třídy `BindingContext` a `CurencyManager`. Třída `BindingContext` je součástí každého formuláře a udržuje kolekci hodnot typu `BindingMangerBase`. Tato kolekce je prázdná, dokud na formuláři nepřidáme datovou vazbu. Jelikož je třída `BindingMangerBase` abstraktní, konkrétní instance budou typu `CurrencyManger` nebo `PropertyManger` v závislosti na použitém zdroji dat. Pokud je zdrojem dat přidaného data bindingu seznam hodnot, je vytvořen objekt typu `CurrencyMannger`, v opačném případě `PropertyManager`. Obě odvozené třídy jsou zodpovědné za synchronizaci asociovaných datových vazeb. Navíc, objekt `CurrencyManager` udržuje aktuální pozici záznamu ve zdroji dat a koordinuje změny s navázanými prvky při změně záznamu. Každý ovládací prvek formuláře může být přidružen k takovému správci datových vazeb. V případě, že k jednomu datovému zdroji dat budou vázány dva ovládací prvky, do kolekce objektu `BindingContext` bude přidán pouze jeden objekt typu `CurrencyManager`.[17]

[17] NAGEL, Christian , et al. C# 2008 : Programujeme profesionálně. 1. vyd. Brno : Computer Press, 2009. 2 sv. (1128, 776 s.). ISBN 978-80-251-2401-7.

3.2.1.3 Nástroje datových vazeb

Vazba dat ve Windows Forms s sebou přináší nepříjemnost v podobě psaní nemalého množství kódu. Pokud bychom museli psát ručně všechny vazby dat na formuláři, práce na aplikaci by byla značně neefektivní a rutinní. V tomto případě nám vychází vstříc produkt Visual Studio. Prostřednictvím vizuálního návrháře Visual Studia je možné vytvářet zdroje dat a přidávat vazby na ovládací prvky. Potřebný kód je generován automaticky. Kromě datových vazeb je možné určit typ a formát vázaných dat.

3.2.2 Implementace datových vazeb ve WPF

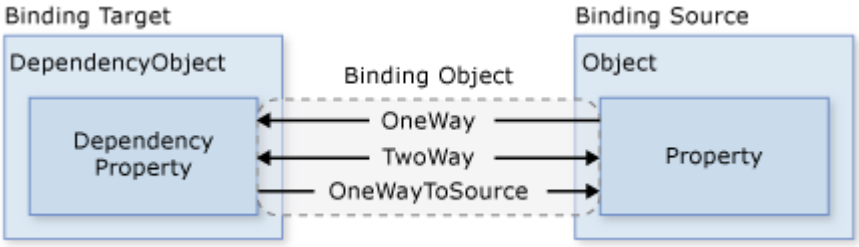
Jak již bylo naznačeno v úvodu kapitoly, data binding ve WPF jednoznačně přesahuje možnosti datové vazby ve Windows Forms. Jedná se především o následující záležitosti:

- Flexibilita
- Deklarativní zápis
- Vliv na separaci aplikační logiky od prezenční vrstvy
- Datové šablony
- Řízení procesu data bindingu
- Pokročilé sdílení datového zdroje

Ve WPF existuje více způsobů, jak zavést data binding, ale nemusí se vždy jednat pouze o vazby mezi aplikační logikou aplikace a elementem grafického rozhraní jako v případě Windows Forms, ale i o vazby mezi jednotlivými UI elementy při splnění určitých podmínek.

3.2.2.1 Datový tok

Obrázek 7: Interakce mezi objekty při procesu data bindingu (převzato [18])



Z výše uvedeného obrázku je patrné, že data binding je zprostředkován mezi dvěma objekty, z nichž jeden nazýváme *cílový* a druhý *zdrojový*. Aby data binding mohl plnit svůj účel, musí být splněny tyto podmínky:

- Cílový objekt musí být `Dependency Property` (viz kapitola 2.2.6)
- Implementovat rozhraní `INotifyPropertyChanged` (pouze pro případy *OneWay* a *TwoWay*)

Jednou z předností data bindingu ve WPF je určení směru toku dat mezi zdrojem dat a cílovým prvkem. Obrázek 6 znázorňuje tři ze čtyř scénářů směru toku dat, které WPF podporuje.

OneWay scénář propaguje změny pouze od datového zdroje k cílovému prvku. Pokud upravíme hodnotu cílového prvku, změna se v datovém zdroji neprojeví. Především je tento scénář vhodný pro vázání vlastností, které jsou jen pro čtení (*read-only*).

TwoWay scénář podporuje obousměrnou synchronizaci datového zdroje a cílového prvku.

OneWayToSource je reverzním případem scénáře OneWay. Propaguje změny pouze směrem od cílového prvku do zdroje dat.

OneTime scénář propaguje změny jako v případě OneWay, ale pouze jednou. Scénář je vhodný pro obsah, který se nebude měnit.

[18] *MSDN Library : Windows Presentation Foundation* [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

3.2.2.2 Deklarativní datové vazby

Jedním z důležitých rozdílů oproti Window Forms je již samotná forma zápisu data bindingu. Zatímco ve Windows Forms je tento zápis výhradně imperativní formou, ve WPF je možné použít i zápis deklarativní. Tato skutečnost má velký dopad při vývoji vícevrstvých aplikací a je jednou z důležitých podmínek pro oddělení prezence od aplikační logiky. Následující příklad ukazuje oba možné případy zápisu data bindingu ve WPF.

Ukázka 7: Deklarativní data binding vs. procedurální

```
<TextBox Text="{Binding Name}" Name="NameTxb"></TextBox>

NameTxb.SetBinding(TextBox.TextProperty,new Binding("Name"));
```

3.2.2.3 DataContext

Ve Window Forms je nutné ke každé datové vazbě určit zdroj dat zvlášť. Oproti tomu ve WPF každý `FrameworkElement` a `FrameworkContentElement` objekt definuje vlastnost závislostí s názvem `DataContext`, která se dědí ve stromu elementů.[19] To znamená, jestliže nastavíme vlastnost `DataContext` nejvyššímu elementu v hierarchii stromu elementů, mohou tento zdroj využívat i podřízené prvky. Při hledání zdroje dat se postupuje od elementu s datovou vazbou směrem ke kořeni dokumentu.

Výhoda této funkcionality je, že prvky s datovou vazbou nevyžadují znalost svého zdroje dat a pouze definují jeho vlastnost. `DataContext` je možné nastavit v procedurálním i deklarativním kódu.

3.2.2.4 Objektový datový zdroj

Další novinkou oproti technologii Windows Forms je možnost použití objektového datového zdroje přímo v deklarativním kódu. Podobná možnost je zahrnuta i v ASP.NET, což dokazuje, že vývoj ve WPF je blízký programování webových aplikací na platformě Windows.

[19] PETZOLD, Charles. Mistrovství ve Windows Presention Foundation. 1. vyd. Brno : Computer Press, a.s. , 2008. 928 s. ISBN 978-80-251-2141-2.

Objektový datový zdroj je reprezentován třídou `ObjectDataProvider`. Funkcionalita třídy zajišťuje vytvoření instance předaného objektu, který bude použit jako datový zdroj. Zároveň `ObjectDataProvider` vystavuje vlastnost `Method`, která určuje název metody, jež je volána na instanci vytvořeného objektu. Na rozdíl od deklarativních zdrojů v ASP.NET je možné použít pouze metodu, která vrací kolekci objektů. Možnost definice metody pro mazání či aktualizace není ve WPF datovém zdroji implementována.

Další užitečnou vlastností objektového datového zdroje je možnost určení, zda se datový objekt vytvoří ve vlastním vlákně.[20] Realizace je v podobě vlastnosti logického typu s názvem `IsAsynchronous`. Tuto možnost oceníme především v případě, že vytvoření datového objektu trvá delší čas.

Ukázka 8: Deklarace objektového datového zdroje v XAML

```
<Window.Resources>
      <ObjectDataProvider
            IsAsynchronous="True"
            MethodName="GetOrders"
            ObjectType="{x:Type data:Orders}"
            x:Key="OrdersDataProvider">
      </ObjectDataProvider>
</Window.Resources>

<Grid DataContext="{Binding Source={StaticResource
OrdersDataProviders}}">
```

Předchozí ukázka představuje definici deklarativního objektového datového zdroje, který vytvoří instanci třídy `Orders` a zavolá její metodu `GetOrders`. Vše je provedeno asynchronně z důvodu uvedení vlastnosti `IsAsynchronous`. Objektový datový zdroj ovládacího prvku `Grid` je svázán s vlastností `DataContext`, kde se již může definovat vlastní prezentace poskytovaných dat.

3.2.2.5 Datové šablony

Pokud ve Windows Forms chceme svázat objekt neprimitivního typu s prvky grafického rozhraní, je nutné navázat každou vlastnost objektu zvlášť.

[20] MATHEW, M.: Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second edition. Apress U.S.A. 2008. ISBN 978-1590599556

V případě, že stejná prezentace objektu se nachází na více místech v aplikaci, musíme tak stále opakovat stejné deklarace datových vazeb.

WPF přináší vylepšení s názvem `DataTemplate`. Datové šablony řeší problém nastíněný v odstavci výše a především nám dávají příležitost graficky znázornit libovolný objekt. V praxi to znamená, že pro zvolený typ objektu vytvoříme datovou šablonu, ve které v rámci možností WPF definujeme grafické prvky a datové vazby. Jestliže v nějakém prvku, který podporuje správu obsahu, naváženeme objekt s definovanou šablonou, WPF nastaví obsah prvku na hodnotu obsahu šablony.

Ukázka 9: Datová šablona v XAML

```
<Window.Resources>
      <DataTemplate DataType="{x:Type data:Order}" >
            <Grid>
                  <Grid.RowDefinitions>
                        <RowDefinition></RowDefinition>
                  </Grid.RowDefinitions>
                  <Border Background="AliceBlue" >
                        <TextBlock>
                              <Label>Detail objednávky</Label>
                              <Label Content="{Binding
                        Name}"></Label>
                        </TextBlock>
                 </Border>
            </Grid>
       </DataTemplate>
</Window.Resources>
<Grid >
   <ContentControl Content="{Binding}"></ContentControl>
</Grid>
```

V ukázce je jedna datová šablona reprezentující objekt `Order`, který je nastaven jako datový kontext okna. V prvku `ContenControl` je nastavena datová vazba bez explicitního určení vázané vlastnosti, tudíž bude použita instance z datového kontextu a zobrazen obsah datové šablony.

3.2.2.6 Hierarchické datové vazby

Hned v úvodu se zaměříme na zpracování prezentace hierarchických dat ve Window Forms, kde se k těmto účelům primárně využívá ovládací prvek `TreeView`. V následující ukázce vytvoříme položky prvku `TreeView`, jehož

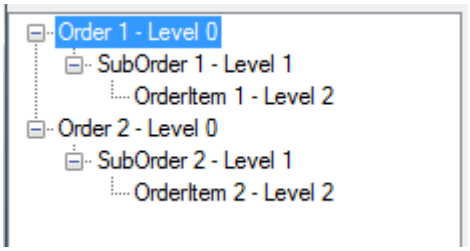
zdrojem je kolekce objektů s hierarchickými vazbami. Objekty v jednotlivých úrovních struktury jsou nehomogenního typu.

Ukázka 10: Vazba dat prvku `TreeView` ve Windows Forms

```
foreach (HierarchialOrder hItem in orders)
{
      int i = 0;
      TreeNode[] orderNodes =
          new TreeNode[hItem.SubOrders.Count()];
      foreach (SubOrder sItem in hItem.SubOrders)
      {
            int j = 0;
            TreeNode[] orderItemNodes =
                  new TreeNode[sItem.OrderItems.Count()];
            foreach (OrderItem oItem in sItem.OrderItems)
            {
                  TreeNode oNode = new TreeNode(oItem.Name);
                  orderItemNodes[j++] = oNode;
            }
            TreeNode sNode =
                new TreeNode(sItem.Name, orderItemNodes);
            orderNodes[i++] = sNode;
      }
      TreeNode topLevelNode =
           new TreeNode(hItem.Name,orderNodes);
      treeView1.Nodes.Add(topLevelNode);
}
```

Výsledek spuštění ukázky po expandování položek bude vypadat následovně.

Ukázka 11: Výstup ovládacího prvku `TreeView` ve Windows Forms



Samožřejmě se jedná o jeden z mnoha způsobů, jak naplnit ovládací prvek `TreeView` ve Windows Forms, ale všechny způsoby spojuje neefektivní psaní kódu na pozadí. Prvek nenabízí automatizovaný data binding a naplnění musí zajistit programátor sám.

Ve WPF je využíván také ovládací prvek `TreeView`. Ten je možné plnit daty jako v předchozím případě. Nabízí se však i mnohem pohodlnější způsob plnění, a to pomocí hierarchických datových šablon ve spojení s data

bindingem. Hierarchické šablony jsou rozšířením datových šablon a přidávají navíc vlastnosti pro práci s podřízenou kolekcí dat.

Ukázka 12:Vazba dat prvku `TreeView` ve WPF

```
<TreeView
      ItemsSource="{Binding Source={StaticResource
      OrdersDataProvider}}">
   <TreeView.Resources>
      <HierarchicalDataTemplate
                  DataType="{x:Type data:HierarchicalOrder}"
                  ItemsSource="{Binding SubOrders}">
            <TextBlock Text="{Binding Name}"></TextBlock>
      </HierarchicalDataTemplate>
      <HierarchicalDataTemplate
                   DataType="{x:Type data:SubOrder}"
                  ItemsSource="{Binding OrderItems}"  >
            <TextBlock Text="{Binding Name}"></TextBlock>
      </HierarchicalDataTemplate>
      <DataTemplate DataType="{x:Type data:OrderItem}" >
            <TextBlock Text="{Binding Name}"></TextBlock>
      </DataTemplate>
   </TreeView.Resources>
</TreeView>
```

Na předchozí ukázce je řešení plnění `TreeView` za pomoci WPF. Výsledek bude stejný jako v případě Windows Forms, ale postup, který k němu vedl, je mnohem více přímočarý. Například rozšíření ukázky o další úroveň by vyřešilo pouhé přidání další hierarchické datové šablony, na rozdíl od ukázky ve Windows Forms, kde bychom museli provádět složité modifikace procedurálního kódu. Vysoký podíl má právě přirozenější forma deklarativního zápisu.

3.2.2.7 Konvertory

V určitých případech datových vazeb požadujeme, aby vázaná hodnota byla jiného datového typu. Ve Windows Forms není jednoduché takový mechanismus zajistit a především u ovládacích prvků, které pracují se seznamem hodnot, to znamená mnoho kódu na pozadí. Nabízí se i vhodné použití vlastních konvertorů, které je založeno na odvození třídy `TypeConverter` a překrytí metod zajišťující konverzi. K danému typu či vlastnosti pak lze přidružit atribut, který určuje konvertor. Jakmile je ovládací prvek svázán s takovým typem, vytvoří instanci našeho konvertoru a zavolá obslužné metody.

Ukázka 13: Využití třídy `TypeConverter` ve WindowsForms

```
[TypeConverter(typeof(OrderStateConverter))]
public enum OrderState
```

Nevýhodou takového přístupu je, že konverzi svazujeme přímo s daným typem. Konverze by ale měla být záležitostí prezence než aplikační logiky.

Z takové myšlenky vychází i WPF a přichází s novým typem konvertorů. Jejich srdcem je rozhraní `IValueConverter` (případně `IMultiValueConverter` pro vícenásobnou konverzi), které deklaruje dvě metody pro zajištění obousměrné konverze. Jakoukoliv třídu, která implementuje toto rozhraní, lze nastavit jako hodnotu vlastnosti `Converter` deklarativní či procedurální datové vazby, čímž zajistíme požadovanou konverzi.

Vezmeme-li například v úvahu případ objednávky se stavy, které jsou realizovány pomocí výčtového typu, tak pro zobrazení aktuálního stavu bude vyžadován převod na typ textový. Pro řešení situace se nabízí použití konvertorů.

Ukázka 14: WPF konvertor

```
public class OrderStateConverter : IValueConverter
{
      #region IValueConverter Members
      public object Convert(object value, Type targetType,
            object parameter, System.Globalization.CultureInfo
                culture)
      {
            if (value is Order.WPFOrderState)
            {
                  Order.WPFOrderState state =
                          (Order.WPFOrderState)value;
                  switch (state)
                  {
                    case Order.WPFOrderState.NEW:
                           return "Nová";
```

Implementace takového konvertoru je demonstrována v ukázce 14. Abychom mohli konvertor z ukázky využít v XAML kódu, nejprve musíme zaregistrovat datový typ konvertoru. Následně ho lze zahrnout do všech prvků

podporujících data binding, a to prostřednictvím vlastnosti data bindingu `Converter`.

3.2.2.8 Obsluha chyb a validace dat

Při zavedení datové vazby je nutné se vypořádat i s možností vzniklých chyb, které mohou být způsobeny například neplatným uživatelským vstupem či nekonzistencí dat aplikace. Již v samotné obsluze chyb je systém WPF a Windows Forms odlišný. Hlavním rozdílem je, že WPF zachycené chyby ignoruje.[21] Dokonce v případě, že navážeme prvek na neexistující vlastnost, nevznikne chyba, která by zabránila kompilaci nebo běhu programu.

Ve Windows Forms je možné řešit chyby pomocí obsluhy událostí, jež některé prvky vystavují po vzniklé chybě, jako např. `DatagridView`. Valná většina případů chyb je způsobena neshodou datového typu. Data binding Windows Forms nám dává možnost explicitního určení typu a formátu vázané hodnoty (ne pro všechny ovládací prvky), což zabrání vstupu jiného typu. Pro validaci se jako vhodné řešení nabízí využití rozhraní `IDataErrorInfo` a komponenty `ErrorProvider`, která je zodpovědná za zobrazení chybové zprávy uživateli. V případě, že však chceme správu chyb a validaci implementovat podle vlastních požadavků, není model Windows Forms příliš vhodný.

WPF poskytuje mnohem flexibilnější model řízení chyb a validace. Jak bylo v úvodu naznačeno, implicitně chyby datových vazeb ignoruje. Je až na vývojáři, jak s tím naloží. V podstatě máme zde několik možnosti, které jsou těsně spjaté s data bindingem a závisí na způsobu, jakým je řešena validace a obsluha chyb dat.

Jeden z případů je, že naše objekty při neplatné hodnotě vyhazují výjimku. Nastavení vlastnosti `ValidatesOnExceptions` u datové vazby na hodnotu `true` zajistí, že WPF bude obsluhovat výjimky. Pokud výjimka vznikne, mechanismus data bindingu vytvoří novou instanci objektu `ValidationError` a přidá ji do kolekce `Errors` přidružené vlastnosti

[21] MATHEW, M.: Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second edition. Apress U.S.A. 2008. ISBN 978-1590599556

`Validation`.[22] S touto kolekcí můžeme pracovat, jestliže požadujeme zobrazení chybových zpráv.

Další volbou je využití `IDataErrorInfo`. Princip je velmi podobný předcházejícímu s rozdílem, že místo vlastnosti `ValidatesOnExceptions` použijeme vlastnost `ValidationOnDataErrors` a validace probíhá prostřednictvím členů implementovaných z rozhraní `IDataErrorInfo`.

Ukázka 15: Vlastní validační pravidlo ve WPF

```
public class RequiredFieldValidator : ValidationRule
{
   public override ValidationResult Validate(object value,
            System.Globalization.CultureInfo cultureInfo)
   {
      ValidationResult validationResult = null;
      if (value != null)
      {
          bool isValid = (value.ToString() != string.Empty);
          validationResult =
                  new ValidationResult(isValid,
                           ErrorMessage);
      }
      return validationResult;
      }
}
```

Poslední uvedenou možností je využití vlastních validačních pravidel. Tato volba je především vhodná, jestliže se nám v aplikaci opakují stejná validační pravidla. Pro její aplikaci je třeba vytvořit validační pravidlo, které je reprezentované třídou, jež je odvozená od abstraktní třídy `ValidationRule` a překrývá metodu `Validate`. Pravidlo, které vyžaduje vyplněné pole, demonstruje ukázka 15. Takto vytvořené pravidlo lze deklarativním způsobem použít v XAML kódu libovolného prvku podporujícího data binding (viz ukázka 16).

[22] MSDN Library : Windows Presentation Foundation [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

Ukázka 16: Použití vlastního validačního pravidla ve WPF

```
<TextBox  Height="25" Name="NameTxb">
      <TextBox.Text>
            <Binding Path="Name" Mode="TwoWay">
                  <Binding.ValidationRules>
                        <local:RequiredFieldValidator
                              ErrorMessage="Pole nesmí být
                        prázdné" />
                  </Binding.ValidationRules>
            </Binding>
      </TextBox.Text>
</TextBox>
```

Z uvedených příkladů je vidět, že WPF nabízí více možností pro obsluhu a validaci dat než Windows Forms. Kromě toho je tento proces ve WPF mnohem více transparentní, což umožňuje vývojáři mít nad ním daleko větší kontrolu.

3.3 Příkazy (Commands)

Jednou z mnoha novinek, které nám WPF přináší, jsou takzvané *commands* (volně přeloženo *příkazy*). Z obecného hlediska se nejedná o nic nového, jelikož se vychází z návrhového vzoru *Command*. Tento vzor nám zajišťuje odstínění objektu od zpracování jeho úloh.[23] V souvislosti s WPF jsou těmito úlohami myšleny akce při zpracování události. Ve WPF není zahrnuta pouze čistá implementace vzoru, ale je dále rozšířen o další funkcionality.

3.3.1 Rozhraní ICommand

Základem pro všechny WPF příkazy je rozhraní `ICommand`, které je součástí jmenného prostoru `System.Windows.Input.ICommand` a vystavuje tři následují členy.[24]

Execute – metoda, která může přímo či nepřímo vykonávat část aplikační logiky nebo předat odpovědnost na příslušné události.

[23] *Command Pattern* [online]. 2005 [cit. 2009-11-19]. Dostupný z WWW: <http://objekty.vse.cz/Objekty/Vzory-Command>.

[24] NATHAN, Adam, LEHENBAUER, Daniel. *Windows Presentation Foundation Unleashed*. 1st edition. Indianapolis : Sams Publishing, 2007. 638 s. ISBN 0-672-32891-7.

CanExecute – metoda indikující stav příkazu. Vrací `true`, pokud je možné příkaz vykonat.

CanExecutedChanged – událost, která vznikne, když se změní stav příkazu.

3.3.2 RoutedCommand

Jeden z vestavěných příkazů ve WPF je typu `RoutedCommand`. Tento typ spojuje dohromady funkcionalitu směrovaných událostí a příkazů. Jeho hlavním přínosem je nezávislost provedené akce uživatelského rozhraní na aplikační logice, automatická správa stavu ovládacích prvků a snížení množství kódu dosažené konsolidací UI událostí.

3.3.2.1 Model RoutedCommand

Model RoutedCommand tvoří následující čtyři koncepty[25] :

- Commands
- Command source
- Command target
- Command binding

Commands zastupuje jednotlivé implementace rozhraní `ICommand`. `RoutedCommand` toto rozhraní implementuje.

Command source je objekt, který je zdrojem události pro vykonání příkazu. Takovým objektem je například ovládací prvek `Button` nebo `MenuItem`. Zdrojový ovládací prvek musí implementovat rozhraní `ICommandSource`, které definuje vlastnosti `Command`, `CommandTarget` a `CommandParameter`.

Command target určuje prvek, který spouští směrované události `CanExecute` a `Execute`. Implicitně se jedná o prvek, který má zaměřen vstup. Ovládací prvky implementující rozhraní `ICommandSource` však mohou toto nastavení změnit a explicitně command target určit. Změna má vliv

[25] MSDN Library : Windows Presentation Foundation [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

pouze na příkazy typu `RoutedCommand`. U jiných typů nebude na command target brán ohled.

Command binding svazuje jednotlivé `RoutedCommand` s obsluhou jejich událostí. Výhodou tohoto návrhu je, že konkrétní `RoutedCommad` se definuje pouze jednou a následně ho lze využít na více místech aplikace.

Ukázka 17: Command binding ve WPF

```
<Window.CommandBindings>
    <CommandBinding
        Command="ApplicationCommands.Open"
        CanExecute="CommandBinding_CanExecute"
        Executed="CommandBinding_Executed"
        PreviewCanExecute="CommandBinding_CanExecute"
        PreviewExecuted="CommandBinding Executed" />
</Window.CommandBindings>
```

Na předchozím obrázku je možné vidět způsob deklarace command bindingu v XAML kódu. Pro command binding musíme určit konkrétní příkaz a obsluhu událostí, kde `CanExecute` a `PreviewCanExecuted` určují, zda může být proveden. `Executed` a PreviewExecuted obsahují aplikační logiku dané události. Události s prefixem *Preview*, jako v případě směrovaných událostí, určují strategii směrování od kořene stromu elementů do cílového objektu (command target). Události bez prefixu určují strategii opačnou.

3.3.2.2 Zpracování a vykonaní RoutedCommand

Začlenění `RoutedCommand` bude vyžadovat ovládací prvek s podporou příkazů, který zajistí mapování událostí pro zpracování a implementaci dané úlohy.

Následující příklad bude demonstrovat příkaz, jehož úkolem bude ukončení aplikace. Zdrojem příkazu je ovládací prvek `MenuItem` a mapování zahrnuje obsluhu událostí `CanExecute` a `Executed`.

Ukázka 18: Použití příkazu s prvkem `MenuItem`

```
<MenuItem Command="ApplicationCommands.Close" ></MenuItem>

<CommandBinding Command="ApplicationCommands.Close"
                CanExecute="CloseCommand_CanExecute"
                Executed="CloseCommand_Executed" />
```

Položku mapovaní `CommanBinding` je možné přidat i v procedurálním kódu (viz následující ukázka).

Ukázka 19: Definice command bindingu v C#

```
CommandBinding closeBinding =
   new CommandBinding(ApplicationCommands.Close);
closeBinding.CanExecute +=
   new CanExecuteRoutedEventHandler(CloseCommand_CanExecute);
closeBinding.Executed +=
      new ExecutedRoutedEventHandler(CloseCommand_Executed);

CommandBindings.Add(closeBinding);
```

Metoda `CloseCommand_CanExecute` určuje pomocí hodnoty vlastnosti `CanExecute` parametru `CanExecuteRoutedEventArgs`, zda může být příkaz proveden. V této metodě můžeme tedy uvést kód, který zabrání nebo povolí uzavření okna. Volání metody je prováděno automaticky a v případě, že vlastnost směrovaného parametru `CanExecute` je nastavena na logickou hodnotu `false`, je i vlastnost `IsEnabled` zdrojového ovládacího prvku nastavena na `false`. Až v opačném případě je možné provést uživatelskou vstupní akci, jejímž následkem je vykonání aplikační logiky příkazu, v tomto případě metody `CloseCommand_Executed`.

3.3.2.3 RoutedUICommand

Součástí WPF je i třída `RoutedUICommand`, která je odvozena od `RoutedCommand` a přidává navíc pouze jedinou vlastnost `Text`. `RoutedUICommand` je určen pro příkazy, které jsou nejčastěji součástí ovládacích prvků typu `Menu` a je nutné prezentovat název jejich prováděné úlohy. Za tímto účelem je právě definována vlastnost `Text`, jež udržuje název této úlohy. Výhodou je, že v návrhu aplikace již není nutné explicitně uvádět tento název ani zajišťovat lokalizaci.

3.3.2.4 Standardní příkazy

Při návrhu UI aplikací se často dostáváme do situace, kdy zjistíme, že některé operace se stále opakují a stávají se pro nás rutinní záležitostí. Příkladem jsou aplikace, které pracují se soubory. Všechny mají společné akce jako *Otevřít*, *Založit* nebo *Tisknout* soubor. Jednotlivé akce je nutné, v případě

požadavku, zahrnout do hlavního menu aplikace, kontextového menu, nástrojové lišty atd. včetně lokalizace a navěšení událostí pro zpracování. Takto vypadala práce ve Windows Forms, ale WPF nás pomocí příkazů od této práce osvobozuje a navíc disponuje již množinou předpřipravených takzvaných *standardních příkazů*. Celá množina těchto příkazů se v systému WPF dělí na další podmnožiny dle klasifikace použití. Každý konkrétní příkaz z tabulky uvedené níže je statickou vlastností statické třídy reprezentující danou skupinu.

Název statické třídy	Použití
ApplicationCommands	Běžné aplikační příkazy (Undo, Redo), příkazy pro práci se schránkou (Copy, Paste, Cut) a pro práci se soubory (Open, Save, Print).
EditingCommands	Příkazy pro práci s textem jako formátování, přesunování či označování.
ComponentCommands	Pohyb či označovaní v UI ovládacích prvcích.
NavigationCommands	Poskytuje navigaci pro aplikace typu prohlížeč nebo pracující s dokumenty.
MediaCommands	Typické akce související s médii (Play, Stop, DecreaseVolume, atd.)

3.3.2.5 Uživatelské příkazy

Ve WPF nejsme omezeni pouze na *standardní příkazy* (viz předchozí kapitola), ale lze vytvářet i vlastní. V nejjednodušším případě vytvoříme statickou třídu, která bude zastupovat skupinu vlastních příkazů a konkrétní příkaz bude vlastností této třídy.

Ukázka 20: Vlastní `RoutedCommand`

```
public static class CustomCommands
{
      private static RoutedUICommand _minimalize;
      public static RoutedUICommand Minimalize
      {
            get
            {
              if (_minimalize == null)
              {
                  _minimalize = new RoutedUICommand();
                  _minimalize.Text = Resources.Minimalize;
               }
               return _minimalize;
            }
      }
}
```

Příkaz na předchozí ukázce zastupuje úlohu pro minimalizaci aplikace. Můžeme vidět, že vytváříme instanci `RoutedUICommand` a přiřazujeme název úlohy ze souboru zdrojů. Použití takto vytvořeného příkazu je podobné jako u *standardních příkazů*. Nejprve se přidá do přislušné kolekce `CommandBindings` a sváže s událostmi, které v tomto případě zajistí minimalizaci aplikace. Následně tento příkaz můžeme přiřadit do podporujícího ovládacího prvku.

3.4 Vlastní ovládací prvky

Pří vývoji grafických aplikací se občas dostaneme do situace, že našim požadavkům nevyhovuje žádný hotový ovládací prvek z palety, kterou poskytuje daná technologie. Řešením je pak vytvoření vlastního ovládacího prvku. Důvody pro takový krok bývají zpravidla dva.

1. Požadujeme netradiční vzhled ovládacího prvku
2. Požadujme rozšířit prvky o novou funkcionalitu

Tvorba ovládacích prvků ve WPF se od Windows Forms liší v mnoha ohledech. Formuláře a ovládací prvky ve Windows Forms jsou již od základu jednotného vzhledu a případné stylování vyžaduje tvorbu vlastního prvku. Naproti tomu ovládací prvky WPF jsou takřka bez vzhledu a jejich stylování je záležitostí programátora či designéra. Z důvodu, že grafický návrh WPF komponent je realizován pomocí XAML a máme možnosti tento návrh měnit

například pomocí stylů či šablon, nemusíme v tomto směru vytvářet prvek vlastní. Názorným příkladem jsou následující ukázky, kde požadujeme panel s gradientním pozadím.

Ukázka 21: Gradientní přechod ve Windows Forms

```
public partial class GradientPanel : Panel
{
      public GradientPanel()
      {
            InitializeComponent();
      }
      protected override void OnPaint(PaintEventArgs pe)
      {
            base.OnPaint(pe);

            LinearGradientBrush gradientBrush =
                new LinearGradientBrush(new Point(0, 0),
                    new Point(this.Width, 0),
                    Color.FromArgb(50,52, 89,69),
                    Color.FromArgb(128, 69, 100, 90));
            pe.Graphics.FillRectangle(gradientBrush,
                  this.ClientRectangle);
```

Ve Windows Forms v případě takového zdánlivě jednoduchého požadavku jsme nuceni vytvořit nový prvek a vykreslit pozadí pomocí grafických funkcí knihovny GDI+ (viz ukázka 21). Naproti tomu ve WPF stačí pouhá deklarativní definice pozadí.

Ukázka 22: Gradientní přechod ve WPF

```
<StackPanel Height="51" VerticalAlignment="Top">
      <StackPanel.Background>
            <LinearGradientBrush StartPoint="0,0"
EndPoint="1,0" >
                  <GradientStop Color="#32345345" Offset="0" />
                  <GradientStop Color="#80456456" Offset="1">
            </LinearGradientBrush>
      </StackPanel.Background>
</StackPanel>
```

Pro případy, kde od ovládacího prvku vyžadujeme funkcionalitu navíc, i ve WPF budeme vytvářet prvek vlastní. Pro vytvoření budeme volit mezi takzvaným `UserControl` nebo `CustomControl`.

3.4.1 UserControl

Významem použití je `UserControl` podobný jako stejnojmenný prvek ve Windows Forms. Jeho použití se nabízí ve chvíli, kdy máme více prvků, které vzájemně tvoří logické uskupení a lze je znovu využít jako celek. Stejně jako ve Windows Forms je tento prvek složen ze dvou souborů, ale s tím rozdílem, že soubor grafického návrhu je tvořen v XAML kódu. Na rozdíl od `CustomControl` na něj nemůžeme aplikovat styl, ale podporují takzvaný *design-time*[26], což umožňuje tvoření vzhledu i v externích grafických editorech.

Tvorba `UserControl` spočívá v kompozici grafických prvků, přidávání vlastností a logiky pro obsluhu událostí. `UserControl` je odvozen od třídy `ContentControl`, což reflektuje závislost na vlastnostech bázové třídy. Právě tento fakt bývá důvodem rozhodnutí pro vytvoření prvku `CustomControl` místo `UserControl`. `UserControl` mění oproti bázové třídě některé vlastnosti jako zaměření, zarovnání elementů či nastavení pro tabulátor a zdroj směrovaných událostí. Pokud na nějakém prvku uvnitř `UserControl` nastane událost, jejím zdrojem není tento prvek, nýbrž celý `UserControl`.

`UserControl` lze zahrnout do stromu elementů deklarativní definicí v XAML kódu a poté k němu přistupovat jako k běžnému prvku.

Obrázek 8 představuje `UserControl`, který obsahuje funkcionalitu pro transformaci bitmapové grafiky. Je zde definována vlastnost závislosti, která přijímá cestu k bitmapě a asociuje události pro sledování její změny. Logika transformací je řešena v kódu na pozadí.

Tento případ je vhodným příkladem pro vytvoření `UserControl`, jelikož využívá a sdružuje pouze vestavěné prvky a nevyžaduje explicitní stylování. Ukázka je i součástí příkladů přiložených k této práci.

[26] Grafická prezentace prvku v době návrhu

Obrázek 8: Ovládací prvek UserControl

3.4.2 CustomControl

Metoda tvorby `CustomControl` je založena na odvozování bázových tříd WPF a změně stávající funkcionality či přidávání nové.

Jejich tvorba se již od Windows Forms výrazně liší. Hlavním důvodem je oddělení chování a vzhledu prvků ve WPF. Pokud ve WPF vytvoříme nový `CustomControl`, automaticky se vytváří základní šablona stylu elementu a vkládá se do obecného slovníku stylů. V závislosti na odvozené komponentě a našich požadavcích pak v kódu prvku určíme, zda překryjeme původní styl a aplikujeme náš či nikoliv. Pokud se ho rozhodneme překrýt, v automaticky vytvořeném stylu pro tento prvek můžeme upravit jeho základní šablonu. Tímto dostáváme prvek, který je nazýván termínem *lookless*. Teprve po přidání požadovaných funkcionalit ve formě vlastností, událostí nebo příkazů máme možnost obohatit prvek o patřičný vzhled.

Ukázka 23: Definice pro překrytí stylu WPF prvku

```
DefaultStyleKeyProperty.OverrideMetadata(
    typeof(ProgressIndicator),
    new FrameworkPropertyMetadata(typeof(ProgressIndicator)));
```

Klíčovým bodem pro vyvtvoření `CustomControl` ve WPF je výběr bázové třídy. Obecným doporučením objektového programování je odvozovat od nejnižší třídy hierarchie, která poskytuje potřebné vlastnosti.[27] Na rozdíl od Windows Forms, jak již bylo řečeno, u WPF není každý prvek ovládací, takže

[27] PETZOLD, Charles. Mistrovství ve Windows Presention Foundation. 1. vyd. Brno : Computer Press, a.s. , 2008. 928 s. ISBN 978-80-251-2141-2.

již tento důvod je základem výběru. Samotný výběr ale již závisí na problému, který daný prvek řeší a jestliže při něm chceme být úspěšní, jedním z kroků je alespoň základní znalost rozdílů bázových tříd.

S ohledem na návržený model prvků ve WPF vyplývá, že s požadavkem na inovaci vzhledu se vyhneme tvorbě vlastního prvku, i přesto existují situace, kdy tomu tak není. WPF se neomezuje pouze na případy běžných ovládacích prvků a dokonce ani na dvourozměrný prostor. Právě v případě návrhu složitější 3D visualizace jsme nuceni vytvářet `CustomControl` z důvodu vzhledu prvku, ale také dalších souvislostí zajišťujících například interakci s prvkem.

3.5 Interoperabilita

Jednou ze společných vlastností obou zúčastněných technologií je jejich vzájemná interoperabilita, která umožňuje integraci jedné technologie do druhé a naopak. Existují dva nejčastější scénáře, kdy se k takovému kroku uchýlit. Prvním z nich je případ, kdy máme již hotovou Windows Forms aplikaci, kterou požadujeme rozšířit o nový prvek, jenž by byl ve stávající technologii jen těžko realizovatelný. Proto vytvoříme prvek raději ve WPF a integrujeme ho do aplikace ve Windows Forms. Druhým případem je reverzní situace, kdy WPF integruje takový Windows Forms prvek, který není obsažen v základní paletě vestavěných prvků.

3.5.1 Integrace WPF do Windows Forms

Pro integraci WPF prvků do Windows Forms můžeme volit mezi několika možnostmi. Jednou z nich je zahrnout WPF prvky do separátního okna, na které se budeme v aplikaci odkazovat. Okno je samozřejmě potomkem WPF třídy `Window`. Druhou možností je hostovaní WPF prvku přímo ve formuláři Windows Forms, a to pomocí ovládacího prvku `ElementHost`. `ElementHost` se chová jako každý jiný prvek ve Windows Forms a jeho zodpovědností je vykreslení právě jednoho WPF prvku. Pokud bychom chtěli hostovat prvků více, nejlepším řešením se nabízí vytvoření `UserControl` ve WPF, které lze v prvku `ElementHost` rovněž využít.

Je důležité připomenout, že ani ve Visual Studiu 2008 se `ElementHost` nenachází ve standardní nabídce prvků, což pro něj eliminuje podporu *design-time*. Naštěstí máme možnost ho do nabídky prvků přidat ručně a této podpory poté využít. Jakmile prvek přetáhneme na náš formulář, automaticky se do projektu zahrnou patřičné reference a můžeme prostřednictvím průvodce (*wizardu*) určit WPF element.

Následující ukázka integruje `UserControl` (zastoupený třídou `ImageControl`) prezentovaný v kapitole 3.4.1, který umožňoval jednoduché transformace obrázku. Výsledek prvku bude stejný jako v předešlém případě s tím, že hostujícím prvkem je formulář Windows Forms.

Ukázka 24: Prvek `ElementHost` ve Windows Forms

```
ImageControl imageControl = new ImageControl();
imageControl.ImageName = "netLogo.png";

_elementHost = new ElementHost();
_elementHost.Child = imageControl;
_elementHost.Dock = DockStyle.Fill;

this.Controls.Add(_elementHost);
```

3.5.2 Integrace Windows Forms do WPF

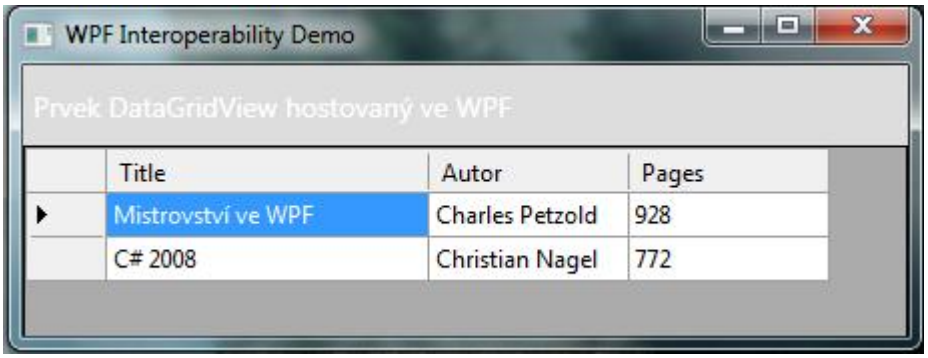
Situace pro integraci Windows Forms do WPF je velmi analogická předchozí. Opět lze využít metodu separátního okna. Tentokrát ale ve formě Windows Forms formuláře, který budeme volat z WPF aplikace. V případě přímé integrace Windows Forms ovládacího do WPF využijeme prvku `WindowsFormsHost`. Tento WPF ovládací prvek je principiálně podobný prvku `ElementHost` z Windows Forms. Jeho obsahem může být libovolný Windows Forms ovládací prvek včetně typu `UserControl`.

`ElemenHost` je již součástí WPF standardní palety prvků, takže ho lze bez dalších náležitostí začlenit do XAML kódu (samozřejmě může být vytvořen i imperativním způsobem jako ostatní WPF prvky) a hostovat Windows Forms ovládací prvek. Samotné přidání konkrétního prvku bude v XAML kódu vyžadovat přiřazení aliasu na sestavení, kde se prvek nachází, nejčastěji `Windows.Forms.Control`.

Ukázka 25: Prvek `WindowsFormHost` ve WPF

```
<my:WindowsFormsHost  Name="windowsFormsHost1" >
   <wf:DataGridView Name="WFGrid"></wf:DataGridView>
</my:WindowsFormsHost>
```

Obrázek 9: Náhled Windows Forms prvku integrovaného do WPF



Ačkoliv se může zdát, že jsme dosáhli dobrého řešení, jak nahradit absenci WPF prvků, tento přístup přináší i své nedostatky. S hostovaným prvkem nelze zacházet jako s běžným WPF prvkem. Především má jednotný design, na který nelze použít WPF stylování a pozicování. Rovněž nelze na prvek aplikovat datové vazby či animace. Před přistoupením k hostování Windows Forms prvku je dobré zvážit, zda se nevyplatí vytvořit vlastní WPF prvek s ekvivalentními vlastnostmi Windows Forms prvku, ale výhodami WPF modelu.

3.6 Shrnutí výsledků porovnání

V předchozích částech této kapitoly byla nastíněna problematika a porovnání některých částí z frameworku WPF s technologií Windows Forms. Byly vybrány především takové oblasti, které nelze hodnotit z hlediska vizuálního efektu, ale funkcionality. Vybrané oblasti mají vliv na kontinuální vývoj aplikace, oddělení týmových rolí a znovupoužitelnosti. Z pěti vybraných problematik se ze strany WPF ukázalo, že tři jsou inovací technologie stávající, jedna je zcela nová a poslední byla přidána pro vzájemnou interoperabilitu obou technologií.

Největší část byla věnována problematice datových vazeb, které oproti technologii Windows Forms dosáhly značných změn a vylepšení. Jejich

největší předností se ukázala flexibilita použití a zároveň jednoduchost začlenění do aplikace. Díky možnosti deklarativního stylu zápisu se rovněž podílí na oddělení logiky a chování při prezentaci dat. Deklarativní styl má rovněž velký význam v případě změn v datovém modelu aplikace, které zahrnují úpravy datových vazeb. Takové úpravy přináší ve WPF jen nepatrné úsilí na rozdíl od podobné situace ve Windows Forms. S datovými vazbami ve WPF je spojena celá řada dalších funkcionalit, které je obohacují jako například konvertory, validátory či objektové nebo XML datové zdroje. Zpracování datových vazeb ve WPF jde zcela jinou cestou než Windows Forms a poskytuje pokročilejší způsoby použití, které se odrazí jako pozitivní hledisko ve výběru technologie.

V porovnání modelu událostí WPF vychází ze stejného základu jako Windows Forms, ale přidává jednu zcela novou vlastnost, a tou je směrování. Směrování ve WPF zajišťuje předávání události podle určené strategie skrze vizuální strom a dává tak šanci na ní reagovat i ostatním elementům. Na rozdíl od Windows Forms, kde vlastník události byl vždy ten prvek, který ji vykonal, ve WPF lze pomocí přidružených událostí asociovat zdroje události k jinému prvku. Směrování ve WPF je velmi pokročilou technikou a lze ji aplikovat na celou řadu případů při zpracování prezenční logiky.

Jednou z nově přidaných funkcionalit ve WPF jsou příkazy, které poskytují odstínění zpracování úloh od vrstvy UI. Jednou z konkrétních implementací je takzvaný `RoutedCommand`. Kromě zmíněné obecné výhody plynoucí z příkazů, využití `RoutedCommand` redukuje počet volání událostí tím, že stejné úlohy ujednotí v jeden požadavek. Prostřednictvím RoutedCommand také získávají výhodu grafičtí designéři, kteří pouze pro danou úlohu definují příkaz a o jeho zpracování už se nemusí zajímat.

V případě porovnání tvorby vlastních prvků je zásadním rozdílem, že z hlediska požadavku na vzhled ve WPF nový prvek vytvářet nebudeme, ale místo toho upravíme pouze styl a šablonu tohoto prvku. Pokud vyžadujeme nový prvek, tak z důvodu nových vlastností a funkcionality. V případě `UserControl` se může jednat i o seskupení logicky souvisejících prvků, které lze v aplikaci znovu využít.

Poslední část byla věnována problematice integrace Windows Forms do WPF a naopak. Ukázalo se, že je možné obě technologie za určitých podmínek spojit a vyplnit tak nedostatky jedné či druhé strany. Především z hlediska integrace Windows Forms do WPF je však nutné tento krok pečlivě zvážit, jelikož jsou s ním spojené i další komplikace.

4 Využití WPF v podnikových aplikacích

V současnosti nevznikají na poli softwarových produktů pouze uživatelské či jednoúčelové aplikace, nýbrž robustní systémy, které se skládají z nespočetně modulů a komponent a zároveň vyžadují jiný přístup vývoje. Právě takové systémy lze nalézt často v podnikové sféře.

4.1 Obecné přístupy a doporučení

V případě podnikových aplikací je především kladen důraz na práci s daty. Úkolem takové aplikace je uchovávat, zpracovávat a zobrazovat data z domény cíleného oboru. Právě zobrazení či vizualizace je nedílnou součástí a její problematiku je třeba řešit za pomocí vhodných prostředků.

Z důvodu, že vývoj podnikových aplikací není otázkou dnů či měsíců a na vývoji se obvykle podílí celý tým lidí, je zapotřebí do vývoje zavést řadu metodik, které udrží konzistentní průběh. Zejména je třeba zajistit udržitelnost aplikace a případný kontinuální rozvoj již hotových či nových součástí. Abychom mohli tyto podmínky dodržet, je již v samém počátku důležitý výběr technologie, kterou bude daná aplikace využívat.

V dnešní době je vývoj podnikových aplikací postaven na vícevrstvé architektuře, kde každá vrstva má svojí odpovědnost a je mezi nimi snaha o co možné nejvolnější vazby. Následující kapitoly jsou zaměřeny na integraci technologie WPF do těchto aplikací, kde bude figurovat v roli prezenční vrstvy.

4.2 Podnikové aplikace z pohledu Windows Forms a WPF

Před příchodem technologie WPF využívaly podnikové aplikace na platformě Microsoft pro prezenční vrstvu především grafický framework Windows Forms nebo ASP.NET. Nyní v době již uvolněného WPF stále převládá vývoj s využitím Windows Forms. Nabízí se tedy otázka, proč se nekoná masová inovace na technologii novou. Z následujících aspektů je možné posoudit důvod:

- Jednotný vzhled grafických komponent Windows Forms bez potřeby další modifikace
- Vestavěné funkcionality Windows Forms komponent
- Podpora Visual Studia v *design-time* a *wizards*
- Nevyzrálost WPF

Na druhou stranu je ale i mnoho problémových úskalí, která se mohou projevit až v průběhu vývoje. Výhody WPF nespočívají pouze v možnostech stylování animací komponent. Jak je patrné z kapitoly 3.2, WPF přichází se zcela novým modelem pro práci s daty a nabízí nové způsoby zápisu kódu. Právě tyto novinky nám dávají následující výhody:

- Oddělení vzhledu a logiky
- Pokročilá vazba dat
- Snadná tvorba a přizpůsobivost netypických komponent
- Prostředky pro psaní udržovatelného a čistého kódu

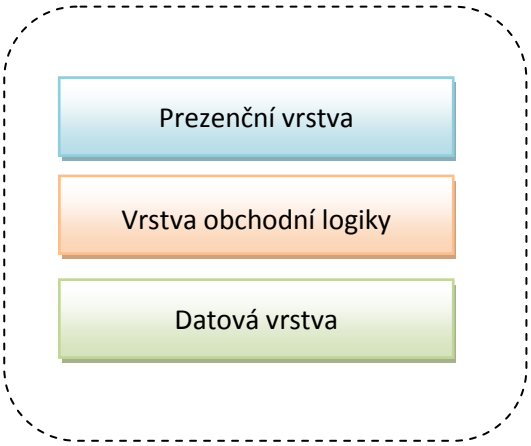
Nevýhodou oproti alternativní technologii Windows Forms se může zdát především absence některých komponent či jejich počáteční takzvaný *lookless* stav. Jednou z možností, jak předejít této skutečnosti, je zakoupení komponent třetích stran, kde však nastávají další problémy, které jsou zapříčiněny závislostí na cizím a mnohdy netransparentním kódu. Jiným řešením je návrh vlastní knihovny, čímž získáme větší možnosti přizpůsobení komponent vlastním potřebám a zároveň usnadnění s jejich udržovatelností.

4.3 Návrhový vzor MVVM (Model-View-ViewModel)

4.3.1 Vícevrstvá architektura

Jak již bylo dříve zmíněno, podnikové aplikace jsou zpravidla postaveny na vícevrstvé architektuře. Mezi nejznámější patří třívrstvá architektura, kde jednotlivé vrstvy znázorňuje následující obrázek.

Obrázek 10: Model vícevrstvé architektury



Cílem tohoto uspořádání je seskupení logicky souvisejících komponent aplikace do separátních vrstev, které mezi sebou komunikují. Směr komunikace je jednosměrný, a to od nejvyšší vrstvy po nejnižší. Především je důležité odstínění jednotlivých vrstev, které má zásadní vliv na případné změny v implementaci

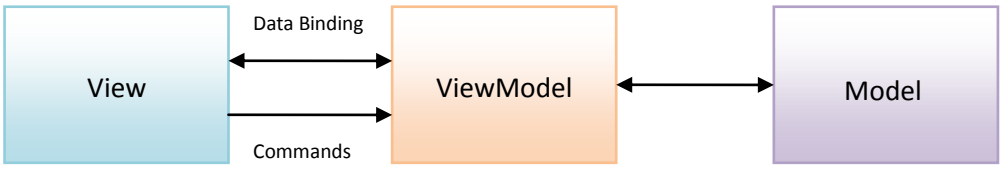
4.3.2 Základy MVVM ve WPF

Z hlediska faktu, že WPF již od základu odděluje vzhled a chování, bychom mohli usoudit, že plně zapadá do modelu vícevrstvé architektury. Tento předpoklad však není pro případ vícevrstvé architektury zcela správný. Chováním je v tomto případě myšlena spíše prezenční logika namísto obchodní. S tím souvisí především úloha událostí, které se běžně navazují na ovládací prvky a zpracovávají obchodní logiku.

V případě WPF je možné aplikovat návrhový vzor, který nám poskytuje způsob, jak dosáhnout požadovaného odstínění prezenční vrstvy a vrstvy obchodní logiky, a to velice snadným způsobem. Navíc přináší i další výhody,

kterým bude věnována pozornost v dalších částech práce. Tento vzor nese název Model-View-ViewModel (dále jen MVVM).

Obrázek 11: Schéma vzoru Model-View-ViewModel

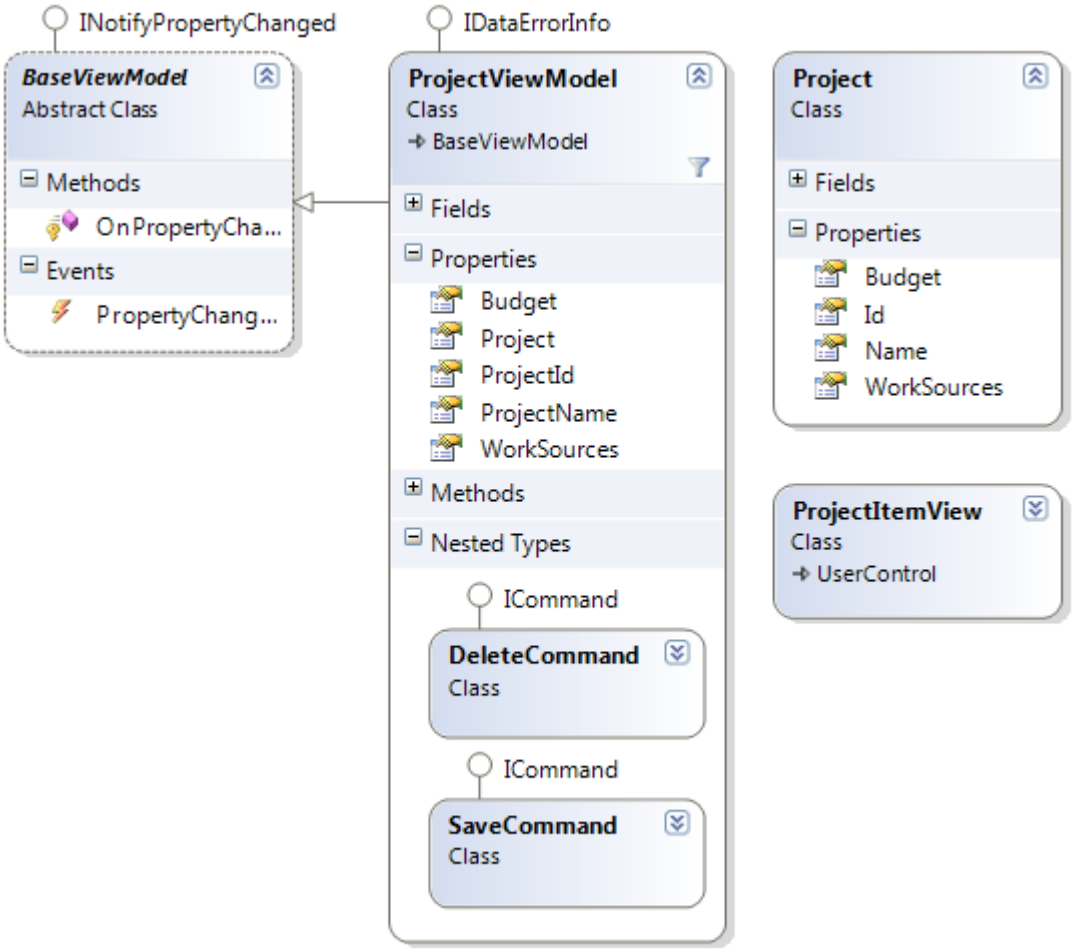


View je část pro prezentaci dat. Z pohledu WPF se jedná o XAML kód prvku `Window` nebo uživatelského ovládacího prvku (`UserControl`). View prezentuje a synchronizuje data pomocí data bindingu z předaného ViewModelu, který získá například prostřednictvím vlastnosti `DataContext`. Jednotlivé akce od uživatele jako kliknutí uživatele na tlačítko jsou obsluhovány pomocí příkazů obsažených rovněž ve ViewModelu. Příkazy zajišťují předání a zpracování události. View neudržuje žádné reference na Model.

ViewModel je jakousi abstrakcí View, ale zároveň obaluje Model a poskytuje tak data, která bude View prezentovat. Kromě toho obsahuje jednotlivé příkazy pro zpracování událostí uživatelského vstupu a může udržovat stav View. ViewModel nemá na View referenci a v žádném případě by se v něm neměl vyskytnout odkaz na grafický prvek.

Model zahrnuje data nebo obchodní logiku. Je zcela nezávislý na použité vrstvě uživatelského rozhraní. Udržuje stav a zajišťuje zpracování problémové domény.

4.3.3 Implementace MVVM ve WPF



Obrázek 12: Model tříd s využitím vzoru MVVM

Pro implementace vzoru nám poslouží příklad, jehož návrh můžeme vidět na předchozím obrázku. Schéma je odvozením obecného modelu vzoru na konkrétní případ, kde Model je zastoupen entitou `Project` a reprezentuje data pro tento případ nespecifikovaného projektu.

4.3.3.1 ViewModel

ViewModel představuje třída `ProjectViewModel`, která zapouzdřuje Model a přidává nové vlastnosti, které souvisí s prezenční logikou. Model je předán prostřednictvím konstruktoru. Jelikož data z ViewModel budou prostřednictvím datových vazeb synchronizována s View, musí být zajištěna implementace rozhraní `INotifyPropertyChanged`. V případě více ViewModelů je z hlediska objektového programování vhodné extrahovat toto

rozhraní do vlastní abstraktní třídy, kterou lze následně použít jakou bázovou pro všechny ViewModely (viz následující ukázka).

Ukázka 26: Bázový ViewModel

```
public abstract class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string
         propertyName)
    {
            if (PropertyChanged != null)
            {
                  PropertyChanged(this,
                        new
                  PropertyChangedEventArgs(propertyName));
            }
    }
}
```

4.3.3.2 View

View je realizováno pomocí prvku `UserControl` s názvem `ProjectItemView`, kde se nachází veškerá jeho grafická prezentace. Inicializace datového kontextu lze provést v konstruktoru uživatelského prvku.

Ukázka 27: Inicializace prvku DataContext ve View

```
public ProjectViewItem()
{
      InitializeComponent();
      this.DataContext = new ProjectViewModel(new Project());
}
```

Samotné View je deklarativně zahrnuto v XAML kódu části rodičovského okna bez explicitního předání datového kontextu. Datový kontext se dědí skrze strom elementů, takže bude dostupný i pro prvek typu `UserControl`. U každého prvku View, který je synchronizován s ViewModelem, je určen deklarativní data binding.

Ukázka 28: Začlenění View do rodičovského okna

```
<Grid Margin="5">
   <view:ProjectItemView></view:ProjectItemView>
</Grid>
```

4.3.3.3 Commands

Kromě prezentace dat Modelu je nutné rovněž zajistit zpracování událostí vyvolaných uživatelem. Pro tyto účely lze využít WPF příkazy, které byly prezentovány v kapitole 3.3. O jejich instancování a uchování je zodpovědný ViewModel. V uvedeném příkladě jsou využity dva příkazy s názvem `SaveCommand` a `DeleteCommand`, které zajišťují provedení uživatelské akce pro uložení a smazání projektu. Jejich deklarace je provedena prostřednictvím interních tříd ve ViewModelu a stejně jako data z Modelu jsou navázány na View prostřednictvím datových vazeb (viz následující ukázka).

Ukázka 29: Definice příkazů ve View

```
<StackPanel Grid.Row="4" Orientation="Horizontal"
Grid.ColumnSpan="2">
      <Button     Content="Uložit" Margin="5"
                  Command="{Binding ProjectSaveCommand}" />
      <Button     Content="Smazat" Margin="5"
                  Command="{Binding ProjectDeleteCommand}" />
</StackPanel>
```

Pokud uživatel klikne na jedno z tlačítek, je automaticky vykonána metoda `Execute` přiřazeného příkazu a zpracování akce pro změnu Modelu. Rovněž lze využít i metodu `CanExecute`, která určuje, zda může být příkaz vykonán.

Ukázka 30: Implementace rozhraní `ICommand`

```
internal class SaveCommand : ICommand
{
      private ProjectViewModel _viewModel;
      internal SaveCommand(ProjectViewModel viewModel)
      {
            _viewModel = viewModel;
      }
      public bool CanExecute(object parameter)
      {
            return true;
      }
      public event EventHandler CanExecuteChanged;
      public void Execute(object parameter)
      {

      _viewModel._repository.SaveEntity(_viewModel._project);
      }
}
```

Využití metody `CanExecute` je vhodné především v případě validace dat, jak bude ukázáno v kapitole 4.3.5. Jak můžeme vidět na ukázce 30, příkaz žadnou vlastní logiku neobsahuje a vše zpracovává prostřednictvím ViewModelu, na který udržuje referenci.

4.3.4 Oddělení rolí grafického designéra a programátora

Při vývoji aplikací se zpravidla projektový tým dělí na menší sub-týmy, které nesou odpovědnost za určitou část projektu. Jednou z takových separací je role grafického designéra a programátora. Pokud bychom vycházeli z premisy, že programátor by zastával obě tyto role, je zřejmé, že vývoj by ztrácel na efektivitě a nároky na jednoho vývojáře by se enormně zvyšovaly. Aby však bylo možné toto rozdělení realizovat, musí použitá technologie splňovat alespoň následující kritéria:

1. Oddělení části definující grafický vzhled a programové zpracování
2. Dostupné nástroje pro návrh a realizaci uživatelského rozhraní

Z prvního kritéria vyplývá, že by daná technologie měla poskytovat takový model, který umožní tvořit grafické rozhraní nezávisle na implementaci. Druhé kritérium stanoví, že by k technologii měly existovat vhodné grafické nástroje, za jejichž pomocí může designér tvořit grafické rozhraní s odstíněním od implementačních detailů. V podstatě je zde snaha, aby závislost grafického návrháře na programátorovi byla co nejmenší.

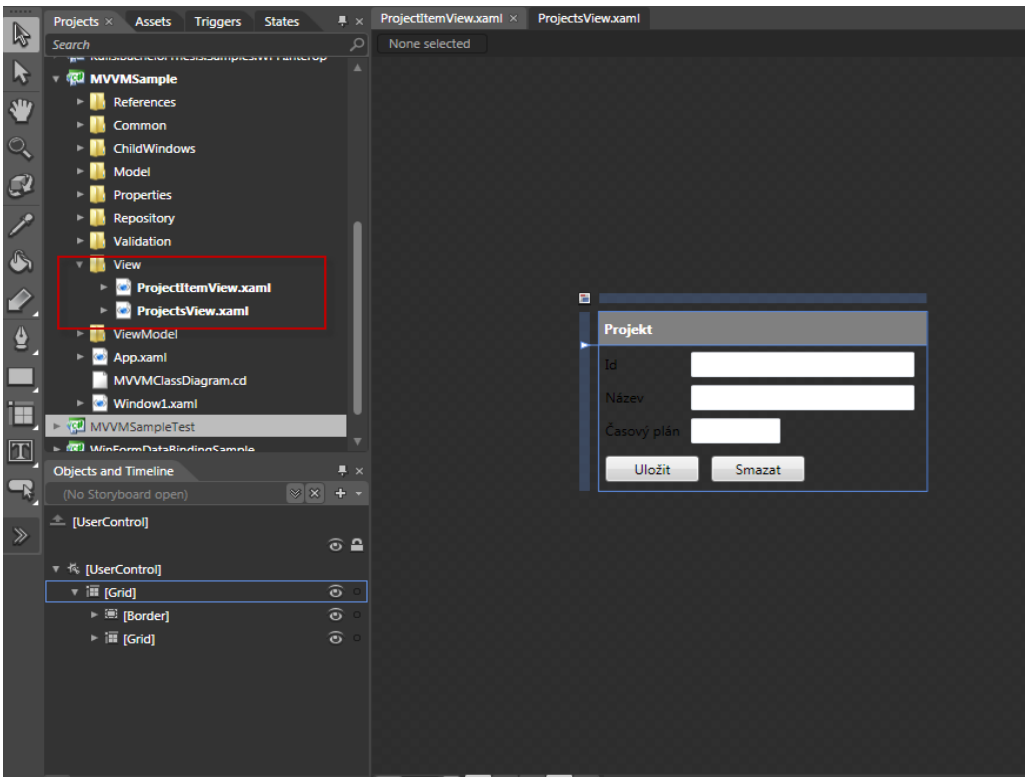
WPF nám v tomto směru nabízí řadu možností a umožňuje splnit stanovená kriteria. Jak již bylo v předcházejících kapitolách zmíněno, WPF podporuje oddělení grafické reprezentace a aplikační logiky. Především možnost deklarativního zápisu pomocí jazyka XAML je přínosem pro grafického designéra. Navíc, právě spojení s návrhovým vzorem MVVM je cestou ke striktnímu oddělení těchto vrstev. Grafický designér tak bude zproštěn znalosti jiného jazyka mimo XAML a tvorby kódu na pozadí (*code-behind*). Naopak ve Windows Forms, i v případě využití některého z návrhových vzorů pro oddělení prezenční a aplikační vrstvy, se grafický designér nevyhne psaní programového kódu, a to především z důvodu událostmi řízeného modelu a imperativního způsobu zápisu grafických komponent.

Značným omezením ze strany Windows Forms je v tomto ohledu i dostupnost nástrojů pro tvorbu grafického rozhraní. V podstatě máme nástroj jediný, který je integrován ve vývojovém prostředí Visual Studio. Jak programátor, tak grafický designér využívají stejný nástroj. Naproti tomu u WPF je situace jiná. Existuje hned několik nástrojů, které lze použít pro tvorbu grafického rozhraní nezávisle na prostředí Visual Studio. Jedním takovým, a patrně nejpokročilejším, je nástroj Microsoft Expression Blend. Mimo jeho rozsáhlé možnosti grafického návrhu je jednou z jeho předností i interoperabilita s prostředím Visual Studio, popřípadě Team Foundation Server. Microsoft Expression Blend umožňuje otevření či založení projektu Visual Studia a tedy i práci nad tímto projektem. Programátor a grafický designér tak mohou současně pracovat souběžně na stejném projektu. V praxi bude samozřejmě pro správu projektu použit např. Team Foundation Server nebo některý z verzovacích systémů.

V případě aplikace s využitím vzoru MVVM se celý tento proces ještě více zjednodušuje. Jelikož bylo dosaženo striktnějšího oddělení, je grafický designér téměř (v závislosti na typu aplikace a rozsáhlosti grafického rozhraní) odstíněn od psaní procedurálního kódu a může se zaměřit pouze na psaní deklarativního XAML. V příkladu, na kterém použití MVVM demonstruji, se bude grafik v programu Expression Blend zabývat pouze tvorbou View a příslušných souborů zdrojů (viz Obrázek 13).

Kromě samotného grafického designu aplikace je však také nutné zajistit prezentaci dat na UI a propagaci akcí směřovaných od uživatele do vrstvy aplikační logiky. Jak již bylo popsáno v předchozí kapitole, vzor MVVM řeší tyto požadavky pomocí datových vazeb a příkazů. Aby mohl designér správně zakomponovat datové vazby, musí mít s vývojářem dohodnutý jakýsi kontrakt, v písemné či jiné formě, který určí, pro jaké grafické komponenty bude použita daná vazba. Tímto kontraktem by mohl být pro každé View jeho ViewModel, ze kterého by designér mohl odvodit potřebné informace. Lepším řešením je však určení vazeb mezi View a ViewModelem již v samotné specifikaci UI rozhraní.

Obrázek 13: Návrh View v nástroji Expression Blend



4.3.5 Validace dat

Validace dat je jedním z dalších problémových úskalí, které je nutné řešit při implementaci podnikových aplikací. Takovéto aplikace především vyžadují konzistentní stav a musí být tedy zajištěno, aby vstupní data, především od uživatele, byla náležitě ověřena a případné nekonzistence eliminovány ještě před jejich zpracováním v aplikační logice. Dle kapitoly 0 umožňuje WPF dva způsoby použití validace dat a jejich výběr závisí na typu cílové aplikace. V případě vícevrstvé architektury je nutné si uvědomit, že validační pravidla jsou součástí modelu aplikace, tedy aplikační logiky, a ve většině případů jsou vázány přímo s doménovými objekty. Z těchto důvodů vyhovuje více validace s využitím rozhraní `IDataErrorInfo`. Kdybychom použili druhý způsob, který WPF nabízí a umožňuje definici vlastních validačních pravidel, již z důvodu, že musíme pracovat se jmenným prostorem `System.Windows.Controls`, validaci svazujeme s prezenční vrstvou. V případě záměny této vrstvy bychom pak museli vytvořit validační pravidla nová. Je však možné oba způsoby validace kombinovat a vhodným způsobem začlenit do návrhu aplikace.

Při návrhu validace ve spojení a návrhovým vzorem MVVM v demonstračním příkladě se vychází z následujících premis:

1. Validační pravidla jsou součástí doménových objektů.
2. Implementace rozhraní `IDataErrorInfo` je záležitostí jednotlivých ViewModelů.
3. Za vyhodnocení validačního pravidla není přímo zodpovědný ViewModel, ale na něm nezávislá třída.

ViewModel z obrázku 12 rozšíříme o vlastnost `IsValid`, jež udržuje logickou hodnotu validity Modelu, a implementaci rozhraní `IDataErrorInfo`. Jelikož je ViewModel v podstatě obálkou doménového objektu, hodnota vlastnosti `IsValid` značí, zda jsou splněny či nesplněny všechny validační pravidla platná pro tento objekt. Klíčovým bodem pro validaci je implementace indexeru z rozhraní `IDataErrorInfo`. Právě v okamžiku změny hodnoty v prvku s datovou vazbou se WPF dotazuje na indexer, který vrací hodnotu `null` nebo textový popis chyby. Rozhodnutí, jestli vrátí `null` nebo popis chyby, je závislé na vnitřní implementaci indexeru. V našem případě je implementace znázorněna na Ukázka 31.

Ukázka 31: Vlastnost určující validitu Modelu

```
public string this[string columnName]
{
    get
    {
        string errors = MVVMSample.Validation.Validator.
            Validate<Project>(_project,
                    PropertyMappings[columnName]);

        return errors;
    }
}
```

Statická třída `Validator` z ukázky obsahuje generickou metodu `Validate`, která ověřuje validační pravidla pro jednu vlastnost doménového objektu. Prostřednictvím parametrů je metodě předán objekt i název validované vlastnosti. Validační pravidla jsou reprezentována pomocí atributů vztahujících se k jednotlivým vlastnostem objektu a implementace konkrétního atributu zahrnuje validační logiku. V okamžiku volání statické metody `Validate`

třídy `Validator` se pomocí reflexe získají všechny validační atributy předané vlastnosti a provede validace, která vrátí popis chyby nebo hodnotu `null`.

Ukázka 32: Vlastnost objektu Modelu s definovaným validačním pravidlem

```
[RequiredFieldRule]
public string Name
{
      get { return _name; }
      set { _name = value; }
}
```

Ve View je nastavení validace součástí deklarace datové vazby na konkrétním prvku. To znamená, že u každého validovaného prvku musíme rozšířit deklaraci datové vazby o vlastnost `ValidatesOnDataErrors` s hodnotou `true`. Tímto způsobem dosáhneme situace, kdy na neplatné hodnoty od uživatele bude WPF reagovat a zobrazí implicitní nebo námi specifikované chybové upozornění. Kromě zobrazení informace validační chyby je ale také nutné zamezit provedení související funkcionality jako například uložení objektu. Toho docílíme pomocí příkazů, které zajistí, že ovládací prvek (například `Button`), jež volá akci závislou na validaci, nebude aktivní, dokud nebude validní vstup. Jelikož interakce mezi View a ViewModel příkazů využívá, musíme pouze implementovat logiku pro metodu `CanExecute` u každého příkazu. Volání této metody se provádí automaticky prostřednictvím událostí a její obsah je znázorněn na následující ukázce.

Ukázka 33: Implementace metody `CanExecute` z rozhraní `ICommand`

```
public bool CanExecute(object parameter)
{
      return _viewModel.IsValid;
}
```

Je tedy vidět, že je vrácena pouze hodnota vlastnosti `IsValid`, která určuje, zda je model validní. Všechny grafické komponenty, které jsou s tímto příkazem svázány, jsou na základě vrácené logické hodnoty z metody `CanExecute` vyhodnoceny jako aktivní nebo neaktivní. Při samotném

provedení metody `Execute`, která je rovněž z rozhraní `ICommand`, není již nutné provádět další validační operace a data mohou být bezpečně zpracována.

4.3.6 Testování WPF aplikace

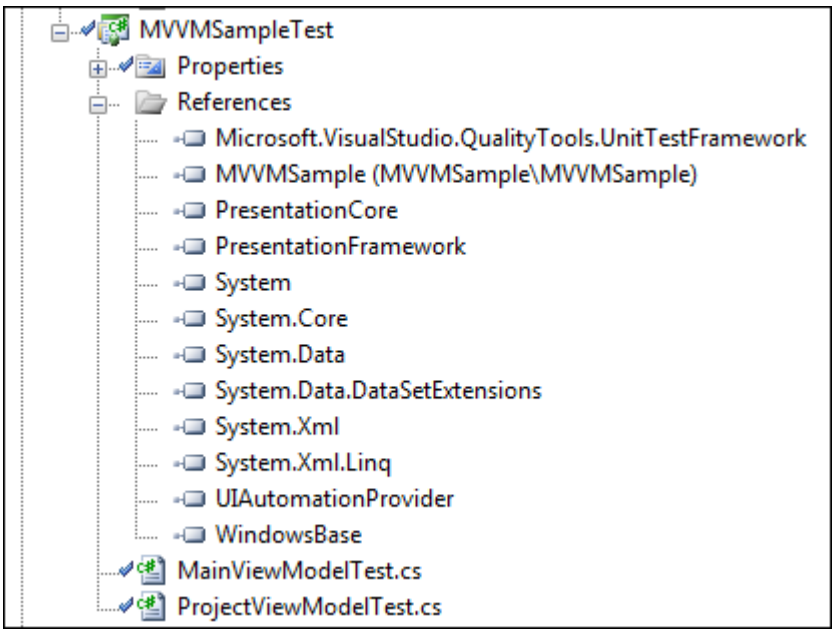
V poslední době se rozvijí metodiky vývoje aplikací, které staví především na testování. Proto, abychom je mohli aplikovat v praxi v plném rozsahu, je nutné, aby architektura aplikace splňovala určitá kriteria, kde jedním z nich je vhodná technologie pro tvorbu uživatelského rozhraní.

Způsobů testování je celá řada jako například jednotkové (unit), manuální, integrační, funkcionální, regresivní atd. V tomto případě se budu zabývat pouze testováním vrstvy UI. Ve většině případů se akce vykonané z UI testují manuálně, tedy kliknutím a ověřením výstupu, nebo pomocí specializovaných nástrojů, které tyto akce automatizují.

Technologie WPF nám žádné nástroje pro testování neposkytuje. Je však vhodné testovat operace, které se budou volat v závislosti na vykonané akci z UI bez nutnosti spuštění aplikace. Dosažením této metody získáme mnoho výhod jako např. snížení režie manuálního testování, možnost automatizovaného a regresivního testování. Cílem je tedy nahradit UI za vhodný objekt a simulovat jednotlivé uživatelské akce. Lze tak dosáhnout opět pomocí vzoru MVVM. Vzhledem ke striktnímu oddělení UI a aplikační logiky můžeme simulovat uživatelské akce prostřednictvím programového kódu. Jako vhodné pro tento kód se nabízí tzv. *Unit testy*, které jsou využity i v následujícím řešení.

Pro jednotkové testy je použit testovací framework ze jmenného prostoru `Microsoft.VisualStudio.TestTools.UnitTesting`, který je součástí Visual Studia 2008. Pro samotné testování je třeba vytvořit nový projekt podle šablony `Test Project` a následně přidat, buď za pomoci průvodce nebo ručně, jednotkové testy pro každý ViewModel. V případě použití průvodce je vygenerován v jednotkovém testu kód navíc (vztahující se k danému ViewModelu), který ale není pro běžné účely žádoucí. Projekt pro uvedený příklad je uveden na následujícím obrázku.

Obrázek 14: Testovací projekt ve Visual Studiu 2008



Z obrázku je patrná reference na projekt s umístěnými ViewModely, bez které by nebylo možné projekt přeložit.

Samotné testování bude postaveno především na jednotlivých ViewModelech s tím, že pro každý je vytvořen vlastní test. Každý test neboli testovací třída obsahuje sadu testovacích metod, kde každá metoda testuje právě jednu operaci či případ. Z ukázkového příkladu tedy vezmeme View, které obsahuje grafické komponenty pro zadání nového či editaci projektu a akce pro uložení změn a odstranění. Datovým kontextem tohoto View je `ProjectViewModel`. V projektu byl tedy vytvořen jednotkový test `ProjectViewModel` obsahující dvě testovací metody. V následující ukázce je vidět první z nich.

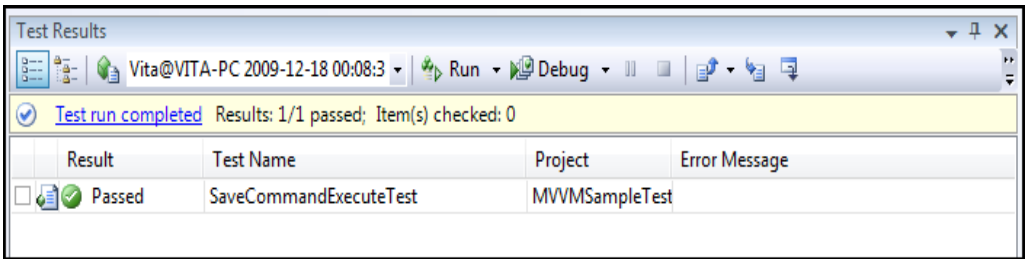
Ukázka 34: Jednotkový test testující příkaz

```
[TestMethod]
public void SaveCommandExecuteTest()
{
      Project invalidProject = new Project();
      ProjectViewModel viewModel =
                  new ProjectViewModel(invalidProject);
      Assert.IsFalse(viewModel.ProjectSaveCommand.CanExecute(
                        null));

}
```

Tento krátký test vyhodnocuje, zda je možné provést operaci uložení objektu `Project`. Je zde zastoupena simulace uživatele při stisknutí tlačítka *Uložit* na UI s nevyplněnými hodnotami položek. V testu postupujeme vytvořením instance `ProjectViewModel`, kde prostřednictvím konstruktoru je předán objekt `Project`. V konečné fázi je provedena aserce logické hodnoty vrácené metodou `CanExecute`, která je implementací rozhraní `ICommand` realizovaném ve třídě `ProjectSaveCommand`. Implementace zahrnuje ověření podmínek validity ukládaného projektu. Pro tento případ, jelikož žádná z vlastností objektu `Project` nebyla nastavena, metoda vrací logickou hodnotu `false` a celá aserce je vyhodnocena pozitivně. Výsledek po spuštění testu ve Visual Studiu vypadá následovně.

Obrázek 15: Výstup provedeného testu ve Visual Studiu 2008



Docílili jsme toho, že akce, které obyčejně provádí uživatel z grafického rozhraní aplikace, je možné testovat i bez jeho účasti.

Takto navržené testy nám také ušetří režii vynaloženou při manuálním testovaní, automatizují proces testování a ve výsledku vylepšují celkovou kvalitu aplikace. Výhodou je i údržba testů, které jsou tímto odstíněny od grafického rozhraní a změnou některé jeho části není nutné test upravovat. Úprava je nezbytná až v případě změny některého ViewModelu.

4.3.7 Další východiska

Z předchozích kapitol se ukázalo, že ve spojení se vzorem MVVM je spjata řada dalších souvislostí, pro které je nutné najít vhodné řešení. Již samotná implementace vzoru MVVM je záležitostí, kterou musíme řešit v každém novém projektu. Na pomoc nám však přichází několik frameworků, které zahrnují implementaci MVVM včetně těchto souvislostí. Jejich výhoda je, že nám dávají možnost soustředit se na náš vlastní kód a zprostit se tak rutinních

záležitostí spojených se vzorem MVVM. Některé z nich jako například MVVM Toolkit[28] podporují dokonce šablony pro Visual Studio. Jeden z nejrozsáhlejších frameworků v tomto směru je Prism[29]. Tento framework je navržen pro vytváření složitých aplikací, které se skládají z modulů, jež jsou na sobě nezávislé. Právě modularita má velký vliv na další členění v týmu a zajištění efektivnějšího vývoje. Každý rozsáhlejší framework má však i svoje stinné stránky a těmi jsou úskalí spojená s jeho studiem. Proto při výběru konkrétního frameworku nebo vlastního řešení je nutný vzít v úvahu míru složitosti aplikace.

[28] http://wpf.codeplex.com/wikipage?title=WPF%20Model-View-ViewModel%20Toolkit&referringTitle=Home

[29] http://msdn.microsoft.com/en-us/library/cc707819.aspx

5 Závěr

V úvodu práce jsem stručně představil základy technologie WPF. Zaměřil jsem se především na hlavní rysy a architekturu modelu WPF. Jedním z cílů práce bylo porovnání technologie Windows Forms a WPF. Vzhledem k tomu, že technologie WPF zahrnuje mnoho nových funkcionalit a vylepšení, pro komparaci s Windows Forms jsem zvolil pouze několik oblastí, které by bylo možno podrobněji zkoumat. Především se jedná o takové oblasti, které nejsou s technologií uváděny na prvním místě, ale mají velký vliv při vývoji aplikací. U každé z nich jsem podrobně popsal teoretické základy pro pochopení dané problematiky a zaměřil se na představení výhod technologie WPF vůči Windows Forms. Velkou pozornost jsem věnoval především rozboru datových vazeb, které se ukázaly jako klíčové při řešení druhého cíle práce.

Dalším cílem bylo nalezení oblasti, kde dominuje jako uživatelské rozhraní Windows Forms a poté do této oblasti nasadit WPF za použití vhodné metody. Cílovou oblastí se staly podnikové aplikace, kde stále na platformě Windows převládá vývoj s využitím Windows Forms. K výběru mě vedl i fakt, že WPF aplikace jsou často zařazovány do oboru malých či klientských aplikací vyžadujících bohatý vzhled. V práci jsem představil základní model architektury aplikací ve vybrané oblasti a zaměřil jsem se na integraci WPF do tohoto modelu. Rovněž jsem zmapoval požadavky na grafické rozhraní podnikových aplikací a podle toho jsem vybral vhodnou metodu, na jejímž základě tyto požadavky lze realizovat ve WPF. Touto metodou se stal návrhový vzor Model-View-ViewModel (dále MVVM). V práci popisuji implementaci tohoto vzoru ve WPF a současně s tím řeším problematiku zmíněných požadavků. Předmětem řešení za použití vzoru MVVM je oddělení rolí grafického návrháře a programátora, řešení validace dat a metoda testování uživatelského rozhraní. Závěrem se zmiňuji o dalších východiscích řešení.

Práce zahrnuje dva hlavní přínosy. Prvním je podání detailního pohledu do pěti oblastí WPF a jejich porovnání s alternativní technologií Windows Forms. Jednotlivé komparace nám přiblíží obsažené změny či nové funkcionality a pomohou tak při rozhodování přechodu na technologii WPF.

Druhým přínosem práce je představení implementace vzoru Model-View-ViewModel, který řeší způsob, jakým lze integrovat technologii WPF do vícevrstvé architektury.

Součástí práce je také sada projektů ve Visual Studiu 2008, které zahrnují příklady vztahující se k probíraným oblastem a jejichž části kódu byly v práci využity při vysvětlování konkrétní problematiky.

Literatura

[1] ANDRADE, Chris, et al. Professional WPF Programming : .NET Development with the Windows® Presentation Foundation. [s.l.] : Wrox, 2007. 480 s. ISBN 978-0-470-04180-2.

[2] *Command Pattern* [online]. 2005 [cit. 2009-11-19]. Dostupný z WWW: <http://objekty.vse.cz/Objekty/Vzory-Command>.

[3] MACDONALD, Matthew. *Pro .NET 2.0 Windows Forms and Custom Controls in C#*. 1st edition. New York : Apress, Inc., 2006. 1037 s. ISBN 1-59059-439-8

[4] MATHEW, M.*: Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5*, Second edition. Apress U.S.A. 2008. ISBN 978-1590599556

[5] *MSDN Library : Windows Forms* [online]. c2009 [cit. 2009-11-01]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ef2xyb33.aspx>.

[6] *MSDN Library : Windows Presentation Foundation* [online]. c2009 [cit. 2009-11-02]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms754130.aspx>.

[7] NAGEL, Christian , et al. C# 2008 : Programujeme profesionálně. 1. vyd. Brno : Computer Press, 2009. 2 sv. (1128, 776 s.). ISBN 978-80-251-2401-7.

[8] NATHAN, Adam, LEHENBAUER, Daniel. *Windows Presentation Foundation Unleashed*. 1st edition. Indianapolis : Sams Publishing, 2007. 638 s. ISBN 0-672-32891-7.

[9] MOSERS, Christian. WPF Tutorial.net : Logical- and Visual Tree [online]. [2008] [cit. 2009-12-10]. Dostupný z WWW: <http://www.wpftutorial.net/LogicalAndVisualTree.html>.

[10] PETZOLD, Charles. *Mistrovství ve Windows Presention Foundation*. 1. vyd. Brno : Computer Press, a.s. , 2008. 928 s. ISBN 978-80-251-2141-2.

[11] Xu, J. Ph.D.: *Practical WPF Graphics Programming.* UniCAD Publishing U. S. A 2007. ISBN 978-0-9793725-1-3

Seznam obrázků

Seznam ukázek

Přílohy

A Obsah přiloženého CD

- Elektronická verze této práce
- Projekty Visual Studia 2008 s ukázkovými příklady
    - Porovnání Windows Forms a WPF
        - Datové vazby
        - Model událostí
        - Vlastní ovládací prvky
        - Interoperabilita
    - Model-View-ViewModel příklad
    - Demo aplikace