Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj WPF aplikace s implementací MVVM návrhového vzoru

Karel Pospíchal

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Karel Pospíchal

Informatika

Název práce

Vývoj WPF aplikace s implementací MVVM návrhového vzoru

Název anglicky

Developing WPF application with the implementation of MVVM design pattern

---

Cíle práce

Diplomová práce je tematicky zaměřena na problematiku vývoje WPF aplikace v jazyce C# pomocí MVVM návrhového vzoru s využití Entity Frameworku.
Hlavním cílem práce je naprogramovat aplikaci, která využívá výše zmíněných technologií a postupů.
Dílčí cíle diplomové práce jsou:
-popsat postupy vývoje aplikace,
-popsat použité technologie.

Metodika

Metodika řešené problematiky diplomové práce je založena na studiu odborné literatury a dalších informačních zdrojů. Vlastní řešení je realizováno vývojem WPF aplikace v jazyce C# s implementací MVVM návrhového vzoru a s využití Entity Frameworku. Při zpracování praktické části bude využito vybraných standardních nástrojů softwarového inženýrství. Na základě syntézy teoretických poznatků a znalostí získaných při vývoji aplikace budou formulovány závěry.

Doporučený rozsah práce

60-80 stran

Klíčová slova

C#, WPF, MVVM, Entity Framework, LINQ, XAML, Visual Studio, .Net Framework

---

Doporučené zdroje informací

JOHN PAUL MUELLER. Microsoft ADO.NET entity framework step by step. Sebastopol (Calif.): O'Reilly Media, 2013. ISBN 0735664161.

LERMAN, Julia. Programming Entity framework. 2nd ed. Sebastopol: O´Reilly, c2010. ISBN 978-0-596-80726-9.

PECINOVSKÝ, R. *Návrhové vzory : [33 vzorových postupů pro objektové programování].* Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

PETZOLD, Charles. Mistrovství ve Windows Presentation Foundation: [aplikace = kód + markup]. Vyd. 1. Brno: Computer Press, 2008. Mistrovství. ISBN 978-80-251-2141-2.

PIALORSI, P. – RUSSO, M. *Microsoft LINQ : kompletní průvodce programátora.* Brno: Computer Press, 2009. ISBN 978-80-251-2735-3.

VICE, Ryan a Muhammad Shujaat SIDDIQI. MVVM survival guide for enterprise architectures in Silverlight and WPF. Mumbai: Packt Publishing, 2012. ISBN 9781849683425.

Předběžný termín obhajoby

2016/17 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 02. 2017

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj WPF aplikace s implementací MVVM návrhového vzoru" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 17.3.2017 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu diplomové práce Ing. Jiřímu Brožkovi Ph.D. za jeho rady, připomínky a odbornou pomoc. Děkuji také mému příteli Ing. Thomasu Katakalidisovi, MA za jeho věcné připomínky ke zpracování této diplomové práce.

Vývoj WPF aplikace s implementací MVVM návrhového vzoru

Souhrn

Diplomová práce je tematicky zaměřena na problematiku vývoje WPF aplikace s implementací návrhového vzoru MVVM a s využitím technologie objektově-relačního mapování Entity Framework. V rámci práce je vytvořena aplikace pro Český statistický úřad, které má za úkol nahradit a inovovat stávající systém zpracování a tvorby mikro-datových souborů z domácnostních šetření. Tato aplikace je založena na výše zmíněných technologiích. Technologie použité při vývoji této aplikace jsou popsány v teoretické části této práce. Nejdříve je zde popsána platforma .NET a také programovací jazyk C#, který byl použitý při vývoji aplikace. Dále se již teoretická část zaměřuje na samotný návrhový vzor MVVM a s ním spojené technologie jako jsou WPF, jeho značkovací jazyk XAML a také Entity Framework. Praktická část následně shrnuje postupy vývoje aplikace vytvořené v rámci práce a především se zaměřuje na tvorbu a provázání jednotlivých vrstev návrhového vzoru MVVM. Veškeré tyto postupy jsou zde demonstrovány na ukázkách kódu z vytvořené aplikace. Na závěr jsou v práci zhodnoceny použité postupy a případný další rozvoj vytvořené aplikace.

Klíčová slova: C#, WPF, MVVM, Entity Framework, LINQ, XAML, Visual Studio, .NET

Developing WPF application with the implementation of MVVM design pattern

Summary

The thesis is focused on development of WPF application with the MVVM design pattern implementations using the technology of object-relational mapping Entity Framework. The application created within the thesis is will be used for the purpose of the Czech Statistical Office. The application would contribute to replacement and innovation of existing system of processing and creation micro-data files from household surveys. The development is based on the aforementioned technologies which are further described in the Theoretical part of the thesis. First of all, there is a description of the .NET Framework and also C# programming language used in the development. Consequently, the Theoretical part is dedicated to the MVVM design pattern and associated technologies such as Microsoft WPF, its XAML markup language and also Entity Framework. The Practical part of the thesis looks into the steps in development of the application and is mainly focused on the creation and binding of MVVM design pattern layers. All of these techniques are shown on code examples of the application. The evaluation of applied techniques and further development of the developed application are discussed in the conclusion of the thesis.

Keywords: C#, WPF, MVVM, Entity Framework, LINQ, XAML, Visual Studio, .NET

Obsah

1 Úvod

Hlavní přínosem této práce je vytvoření aplikace založené na technologiích Windows Presentation Foundation (WPF), Entity Frameworku a návrhového vzoru MVVM (Model-View-ViewModel), která je zpracována pro Český statistický úřad. Tato aplikace je vyvíjena jako součást nově vznikajícího systému pro sběr dat z domácnostních šetření a důvodem jejího vzniku je přechod na modernější technologie sloužící k ukládání dat a celkové zobecnění systému pro zpracování dat z různých domácností šetření. Těmito šetřeními jsou například IŠD (Výběrové integrované šetření v domácnostech), SILC (Výběrové šetření Životní podmínky), AES (Šetření o vzdělávání dospělých). Navrhovaná aplikace bude především sloužit ke správě a výpočtu odvozených ukazatelů z výběrových domácnostních šetření.

Windows Presentation Foundation tedy zkráceně WPF je nový framework pro vývojáře formulářových aplikací, se kterým přichází Microsoft od verze .NET Framework 3.0, aby postupně nahradil starší Windows Forms, které není možné využít pro tvorbu aplikací na zařízeních, jako jsou tablety nebo mobilní telefony.

S rozvojem mobilních zařízení bylo žádoucí vymyslet způsob, kterým lze provozovat stejnou aplikaci na velikostně i typově různých zobrazovacích zařízení tak, aby byla aplikace stále pro uživatele pohodlně ovladatelná. Z tohoto důvodu přichází Microsoft na trh s WPF, který zavádí jako jednotku délky tzv. DIP (Device Independent Pixel) a čistě vektorovou grafiku, aby výsledné grafické rozhraní vypadalo vždy na každém zařízení stejně. U WPF je k vykreslování využívána technologie Direct3D, což je rozhraní pro akcelerovanou grafiku. Díky tomu aplikace vytvořené ve WPF méně zatěžují procesor a jsou tedy responsivnější a mohou být graficky bohatší. K vytváření grafický prototypů slouží ve WPF nástroj Microsoft Blend, který je určen zejména pro grafiky a designéry.

S WPF také přichází nový návrhový vzor MVVM (Model-View-ViewModel). MVVM odděluje data, stav aplikace a uživatelské rozhraní. V praxi to může například znamenat tvorbu aplikace určené na standardní desktopové počítače i na mobilní telefony. Pro takový případ je vhodné využití právě návrhového vzoru MVVM, který dovoluje oddělit programovou a datovou logiku od prezentační části aplikace. Pro každé zařízení

zvlášť tedy stačí vytvořit dvě různá „View“ neboli uživatelská rozhraní a celou zbylou logiku aplikace ponechat pro obě zařízení stejnou.

WPF je přímo uzpůsobeno, aby se používalo s návrhovým vzorem MVVM a toho je docíleno podporou tzv. „bindingu“ a „commandů“, které slouží jako náhrada za prostředí řízené událostmi jednotlivých prvků formuláře.

Tato diplomová práce se ve své teoretické části věnuje nejen popisu WPF a MVVM, ale i technologiím s nimi souvisejícím a v neposlední řadě Entity Frameworku, který je následně využitý v praktické části práce. Samotná praktická část je tvořena ukázkami programového kódu v jazyce C# z autorem vytvořené aplikace, která implementuje všechny popsané technologie z teoretické části.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na problematiku vývoje WPF aplikace v jazyce C# pomocí MVVM návrhového vzoru s využití Entity Frameworku. Hlavním cílem práce je návrh a implementace aplikace pro Český statistický úřad, která využívá výše zmíněných technologií a postupů. Aplikace bude sloužit k tvorbě mikro-datových souborů a správě odvozených ukazatelů z domácnostních šetření.

Výsledná aplikace bude implementovat návrhový vzor MVVM a musí využívat pro ukládání meta-datových popisů centrální databázi Českého statistického úřadu. Objektově-relační mapování na struktury vytvořené v databázi bude zajištěno pomocí technologie Entity Framework.

Dílčí cíle diplomové práce jsou:

- popsat použité technologie
- popsat důležité postupy implementace návrhového vzoru MVVM na vyvíjené aplikaci

2.2 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu odborné literatury a dalších informačních zdrojů. Zpracování teoretických východisek je zaměřeno na popis použitých technologií k vývoji vlastní aplikace.

Vlastní řešení je realizováno vývojem WPF aplikace v jazyce C# s implementací MVVM návrhového vzoru a s využití Entity Frameworku. Při zpracování praktické části bude využito vybraných standardních nástrojů softwarového inženýrství. Na základě syntézy teoretických poznatků a výsledků testování aplikace budou formulovány závěry.

V práci jsou použity termíny z anglického jazyka, kterým chybí ustálený český ekvivalent (např. *data binding* nebo *command*) – tyto termíny jsou v práci skloňovány podle českých gramatických vzorů.

3 Teoretická východiska

Teoretická část diplomové práce popisuje vývojové prostředí Microsoft .NET se zaměřením především na programovací jazyk C#, Windows Presentation Foundation zkráceně WPF a značkovací jazyk XAML, který se při práci ve WPF používá. Následně je teoreticky zpracován návrhový vzor MVVM, který s WPF přímo souvisí a Entity Framework, který slouží k vytvoření ORM (Object/Relational Mapping) modelu. Veškeré teoretické poznatky jsou dále využity v praktické části práce, která je zaměřena na vytvoření aplikace s využitím všech výše zmíněných technologií.

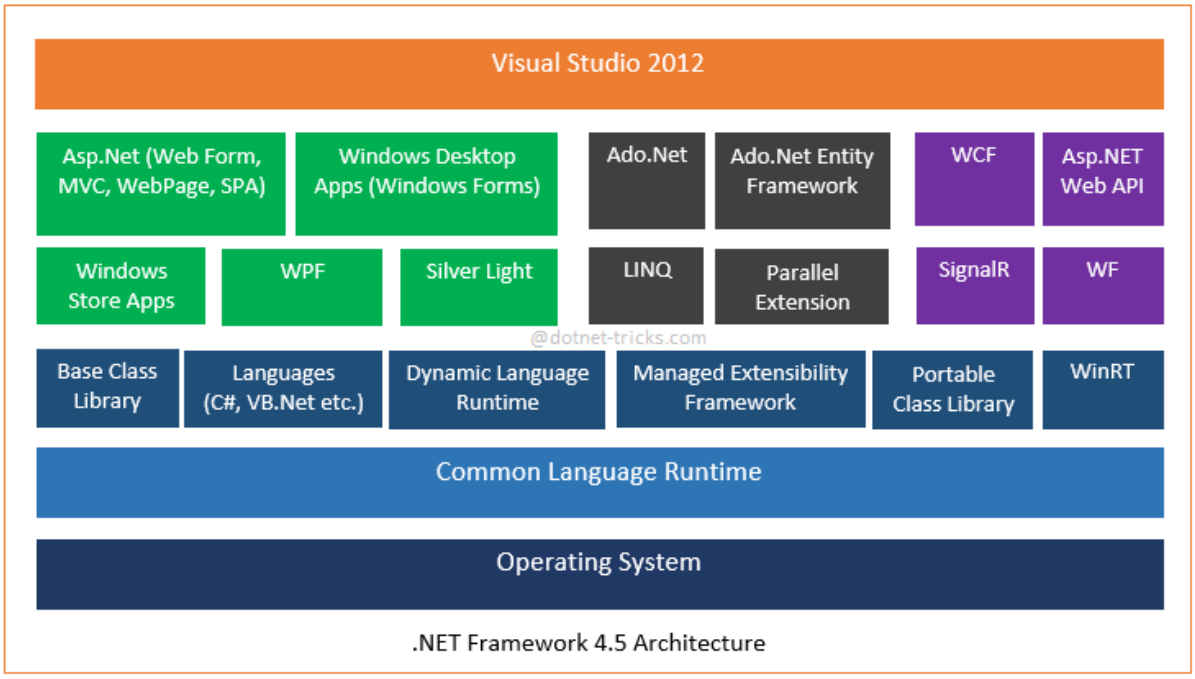
3.1 Platforma Microsoft .NET

Platforma Microsoft .NET je tvořena celou řadou produktů od firmy Microsoft, jejímž základem je .NET Framework, který spolu s Visual Studiem tvoří komplexní vývojové prostředí. Celá platforma nabízí vývoj jak pro desktopové aplikace, tak pro ty mobilní a webové. Aplikace vytvářené v rámci platformy je možné psát v různých programovacích jazycích (např.: C#, Visual Basic, C++, JScript, F#). Dále jsou součástí platformy také různá serverová prostředí, které umožňují provoz škálovatelných aplikací, které se využívají například k řízení obchodních procesů. (KOCAN, 2001)

3.1.1 .NET Framework

.NET Framework se skládá z tzv. „common language runtime“ zkráceně CLR a knihoven, které jsou jeho největší výhodou. V podstatě je dodáván s připravenou řadou struktur a komponent pro práci např.: s konzolí, databázemi, grafikou a mnoha dalšími. Common Language Runtime je základem .NET Framework, který je možné také nazývat virtuální stroj, protože napsaný kód, ať už v kterémkoli z podporovaných jazyků, je nejdříve zkompilován do tzv. Common Intermediate Language (CIL) a tento mezikód je díky jednoduchosti relativně rychle interpretovatelný za běhu aplikace. O tento překlad za běhu se stará virtuální stroj, který Microsoft nazývá CLR - Common Language Runtime. Poslední verze operačních systémů Windows již nyní obsahují ve své základní instalaci nějakou verzi .NET Frameworku, která se standardně aktualizuje zároveň s automatickými systémovými aktualizacemi. Pro spuštění aplikací vyvinutých v .NET je nutné, aby byla nainstalovaná minimálně stejná nebo vyšší verze .NET Frameworku pro

který byla aplikace napsána. Na následujícím obrázku je zobrazena struktura .NET Frameworku 4.5. (ČÁPKA, 2016)



Obrázek 1 - Struktura .NET Framework 4.5 (Understanding.Net Framework 4.5 Architecture, 2013).

3.1.2 Common Language Runtime (CLR)

CLR vychází ze specifikace Common Language Infrastructure (CLI) neboli standardu ECMA-335, na jehož vývoji se podílelo mnoho významných firem (např.: Fujitsu, Intel Corporation, Novell Corporation, Microsoft, Hewlett-Packard). Tato specifikace definuje virtuální stroj, který dovoluje použití více vysokoúrovňových jazyků na různých výpočetních architekturách, aniž by bylo nutné upravovat kompilátory tak, aby vyhovovaly danému prostředí. CLR je už tedy přímou implementací specifikace CLI ze strany Microsoftu.

Následující funkční popis CLR vychází primárně ze zdroje od jeho výrobce Microsoft (Common Language Runtime (CLR), 2016). Spravovaný kód neboli Common Intermediate Language (CIL) je napsaný kód převedený kompilátorem do jazyka, který běží v CLR v modulu runtime. Výhody tohoto spravovaného kódu jsou podpora integrace více programovacích jazyků, odchytávání

stejných výjimek v různých jazycích, vylepšené zabezpečení, snadnější verzování a nasazení do provozu, zjednodušený model pro interakci komponent, jednodušší úpravy a analýza.

Aby mohl být v modulu runtime spouštěn spravovaný kód, musí kompilátor vytvářet meta data, která popisují typy proměnných a odkazy na jiné objekty v kódu. Tyto meta data jsou ukládána společně se spravovaným kódem a každý spustitelný CLR soubor je musí obsahovat. Runtime používá meta data k vyhledání a načtení tříd, rozmístění instancí v paměti, vyvolávání metod, generování nativního kódu, vyššímu zabezpečení a vymezení kontextu.

Runtime automaticky spravuje rozložení objektů v paměti, jejich reference mezi nimi a uvolnění z paměti, pokud již nadále nejsou potřeba. Objekty, jejichž životní cyklus je spravován tímto způsobem se nazývají spravovaná data (managed data). Garbage collection neboli uvolňování nepoužívaných objektů z paměti, eliminuje nedostatek paměti, jakož i běžné programovací chyby. Pokud je použit spravovaný kód, je možné v aplikacích vytvořených v .NET Frameworku použít spravovaná i nespravovaná data. Vzhledem k tomu, že kompilátory poskytují vlastní datové typy, není vždy potřeba znát, zda jsou data spravována či nespravovaná.

Common language runtime usnadňuje návrh komponent a aplikací, jejichž objekty dokáží vzájemně interagovat napříč různými programovacími jazyky. Lze například definovat třídu a následně použít jiný jazyk k odvození této třídy nebo zavolání metody v původní třídě. Je možné také předat instanci třídy metodě třídy napsané v jiném jazyce. Tato integrace mezi jazyky je možná, protože kompilátory a nástroje, které cílí na runtime, používají obecný systém typů definovaný v CLR.

Jako součást svých meta dat všechny spravované komponenty přenáší informace o zdrojích, se kterými byly vytvořeny. Runtime používá tyto informace k zajištění, že naprogramovaná komponenta nebo aplikace má specifikovány všechny verze veškerých zdrojů, jež potřebuje ke svému běhu, což dělá kód méně náchylný k chybě z důvodu nespecifikované závislosti. Registrační informace a stavová data objektů nejsou ukládány do registrů, kde se obtížně spravují. Místo toho jsou informace o definovaných typech (a jejich závislostech) uloženy s kódem jako meta data, což dělá replikaci nebo odebírání komponent méně složitým.

Při použití Common Language Runtime jako virtuálního stroje je dosaženo následujícího souhrnu výhody:

- Vylepšení výkonu aplikace.
- Schopnost snadno používat komponenty vyvinuté v jiných programovacích jazycích.
- Rozšiřitelné datové typy poskytované knihovnou tříd.
- Jazykové funkce jako jsou například: dědění, implementace rozhraní a přetížení metod.
- Podpora více vláknových a škálovatelných aplikací.
- Podpora strukturovaného zpracování výjimek.
- Podpora vlastních atributů.
- Automatickou správu objektů v paměti (Garbage collection).
- Používání delegátů namísto ukazatelů (pointrů) na funkce pro zvýšení typové bezpečnosti.

3.1.3 **C#**

C# (čteno [sí-šarp]) je objektově orientovaný programovací jazyk s hierarchickým přístupem k dědění. Objekt na nejvyšší úrovni je System.Object, ze které všechny ostatní objekty dědí a tím dokáží sdílet podobné funkce (SKEET, 2014).

„Jazyk C# byl navržen a implementován firmou Microsoft pro platformu .NET. Ze všech vyšších programovacích jazyků, v nichž lze vyvíjet programy pro tuto platformu, poskytuje největší možnosti, neboť byl vyvíjen společně s ní. Hovoříme-li o C#, musíme mluvit i o platformě .NET a hlavně o ní – jinak to prostě nejde. Jazyk C# totiž neobsahuje žádné vlastní knihovny; opírá se výhradně o knihovny tohoto prostředí“ (VIRIUS, 2012, s. 19).

3.1.3.1 **Vznik**

V lednu 1999 Anders Hejlsberg vytvořil tým za účelem vybudovat nový programovací jazyk s názvem Cool tedy „C-like Object Oriented Language“. Microsoft původně chtěl zachovat název Cool jako konečný název, ale kvůli problémům s ochrannou známkou tak nakonec neučinil. V době, kdy byl .NET projekt zveřejněn na konferenci

Professional Developers Conference v červenci v roce 2000, byl jazyk přejmenován na C# (HASAN, 2012).

Hlavním návrhářem a architektem společnosti Microsoft byl Anders Hejlsberg, který dříve pracoval také na vývoji Turbo Pascalu, Embarcadero Delphi (CodeGear Delphi a Borland Delphi) a Visual J++. V rozhovorech a článcích uvedl, že nedostatky většiny hlavních programovacích jazyků (např. C++, Java, Delphi a Smalltalk) jsou již podchyceny v základech CLR, podle kterého se řídí design jazyka C# (OSBORN, 2000); (HAMILTON, 2008).

James Gosling, který vytvořil programovací jazyk Java v roce 1994, a Bill Joy, spoluzakladatel společnosti Sun Microsystems, nazvali C# za imitaci Javy. Klaus Kreft a Angelika Langer (autoři knihy C++ streams) na svém blogu uvedli, že *„Java a C# jsou téměř totožné programovací jazyky, které se nudně opakují a postrádající inovaci“*, „*Málokdo bude tvrdit, že Java nebo C# jsou revoluční programovací jazyky, které změnil způsob, jakým se v budoucnu psát programy*“, nebo „*C# si vypůjčil hodně z Javy a naopak. C# nyní podporuje boxing a unboxing, v budoucnu budeme mít podobnou funkci i v Javě“* (KREFT et. al., 2003). Anders Hejlsberg argumentoval, že C# není klon Javy, ale že má mnohem blíže k C ++.

Od vydání C# 2.0 v listopadu 2005 se vývoj C# a Java ubírá stále více po odlišných trajektoriích a stávají se poněkud méně podobné. Jeden z prvních velkých rozdílů přišel s přidáním generik do obou jazyků, které měli zcela odlišnou implementaci. C# využívá „první třídu“ k zhmotnění generických objektů, které mohou být také použity jako kterákoli jiná třída. Naproti tomu generika v Java je v podstatě funkce syntaxe jazyka a nemají tak vliv na generovaný byte kód, protože kompilátor provádí výmaz informací z rodového typu, poté co ověří jejich správnost (HASAN, 2012).

Kromě toho, C# přidal několik významných funkcí z důvodu přizpůsobení funkčnímu stylu programování, který vyvrcholil přidáním sady rozšíření LINQ uvolněných s C# 3.0, který podporuje lambda výrazy, rozšíření metod (extension methods) a anonymní třídy. Všechna tato rozšíření pomáhají vývojářům snížit množství často používaného kódu, který je obsažen v běžných úkolech, jako jsou dotazování do databáze, parsování XML souboru, nebo vyhledávání přes datové struktury. Přesouvá se tak důraz do skutečné logiky programu a zlepšuje tak čitelnost a udržovatelnost (HASAN, 2012).

Název C# vznikl podobně jako název C++, kde „++“ znamená, že proměnná by měla být inkrementována. Autoři chtěli v názvu zachovat to, že byl jazyk odvozen z jazyka C a symbol mřížky vznikl svázáním čtyř symbolů „+“ (tedy C++++) a figuruje zde i hudební aspekt (vysokého C). To vše je jakým si naznačením, že jazyk je dalším pokračovatelem C++ (HAMILTON, 2008).

3.1.3.2 **Přehled**

Syntaxe C# je velmi expresivní, ale je také jednoduchá a snadno zapamatovatelná. Syntaxi využívající složenou závorku pozná v jazyce C# okamžitě každý, kdo zná jazyk C, C++ nebo Java. Vývojáři, kteří znají některý z těchto jazyků, jsou obvykle schopni začít produktivně pracovat v jazyce C# v relativně velmi krátké době. Syntaxe C# zjednodušuje mnoho složitostí jazyka C++ a obsahuje další funkce, jako jsou typy shodnotou null, enumerátory, delegáty, lambda výrazy a přímý přístup do paměti, což například v Javě možné není. Jazyk C# podporuje generické metody, typy a iterátory, které umožňují vývojářům implementovat vlastní chování iterací (Introduction to the C# Language and the.NET Framework, 2016).

C# jako objektově orientovaný jazyk podporuje koncepty zapouzdření, dědičnosti a polymorfismu. Všechny proměnné a metody, včetně metody Main, vstupního bodu aplikace, jsou zapouzdřeny v rámci definice třídy. Třída může dědit pouze z jedné nadřazené třídy, ale může implementovat libovolný počet rozhraní. Metody, které jsou nadřazeny virtuálním metodám v nadřazené třídě, vyžadují pro zabránění náhodnému předefinování klíčové slovo override. Struktura je v jazyce C# v podstatě „odlehčená třída“; je to typ na zásobníku, který umí implementovat rozhraní, ale nepodporuje dědění (Introduction to the C# Language and the.NET Framework, 2016).

Dále jsou v jazyce C# podporovány následující jazykové konstrukce, které usnadňují vývoj softwarových komponent. Tento souhrn byl zpracován podle zdroje Virius (VIRIUS, 2011):

- Delegáty (delegates) – což jsou zapouzdřené ukazatele na metodu. Využívají se ředevším k předávání metod jako parametrů jiným metodám.
- Vlastnosti (properties) – jsou přístupové metody, které se v kódu používají jako proměnné.

- Atributy (attributes) – jsou doplňující informace, které mohou ovlivnit chování programu za běhu.
- XML dokumentace
- Language-Integrated Query (LINQ), který umožňuje dotazování nad různými datovými zdroji.

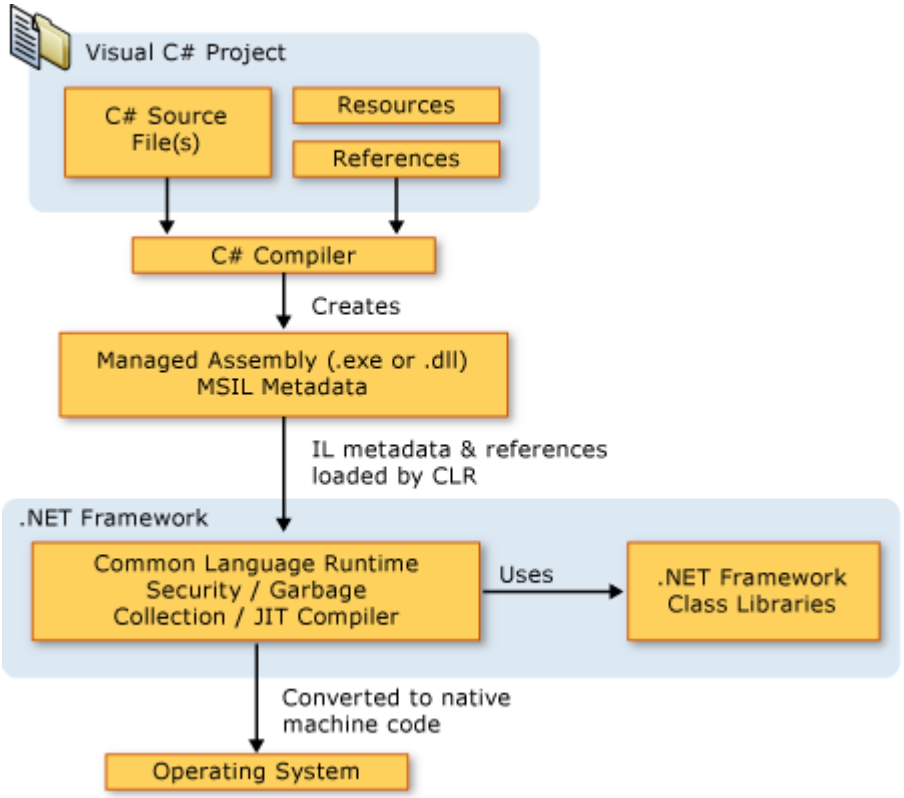
Pracovat s jiným softwarem, jako jsou například objekty typu COM nebo nativní knihovny DLL Win32, lze prostřednictvím procesu nazývaného „*Interoperabilita*“. Interoperabilita umožňuje programům napsaných C# provádět téměř veškeré akce, které umožňují nativní aplikace C++. C# podporuje i ukazatele (pointers) a tzv. nebezpečný kód pro případ, kdy je nezbytně nutný přímý přístup do paměti (Introduction to the C# Language and the.NET Framework, 2016).

3.1.3.3 **Vztah jazyka C# a .NET Frameworku**

Programovací jazyk C# je speciálně navržen a určen k použití s .NET Frameworkem, aby mohl být program spuštěn v běhovém prostředí .NET Frameworku - konkrétně v běhovém modulu CLR vyžaduje překlad, který probíhá ve dvou krocích.

V prvním kroku je zdrojový kód v jazyce C# zkompilován do zprostředkujícího jazyka IL (Intermediate Language). Tento IL kód a zdroje, jako jsou například obrázky a text, jsou uloženy na disku v tzv. sestavení (assembly) ve formátu PE (Portable Executable). V tomto souboru je i tzv. manifest, který obsahuje dodatečné informace o sestavení (např.: o jazykových verzích nebo požadavcích na zabezpečení), v zásadě jde o meta data programu a seznam datových typů (Introduction to the C# Language and the.NET Framework, 2016).

V druhém kroku je spuštěn soubor PE a tím je načten do CLR, kde jsou nejdříve provedeny různé akce na základě informací v manifestu sestavení. Potom, pokud jsou splněny požadavky na zabezpečení, modul CLR provádí kompilaci kódu IL na nativní strojové instrukce za běhu programu (JIT – just in time). CLR zároveň také poskytuje jiné služby související s automatickým uvolňováním paměti, hromadným zpracováním výjimek a správou prostředků. Následující diagram znázorňuje vztahy při kompilaci a chování za běhu: souborů se zdrojovým kódem v jazyce C#, knihoven .NET Frameworku, sestavení a modul CLR (Introduction to the C# Language and the.NET Framework, 2016).



Obrázek 2 - Vztahy při kompilaci jazyka C# (Introduction to the C# Language and the.NET Framework, 2016).

3.1.3.4 Syntax

Jak již bylo výše zmíněno jazyk C# je svojí základní syntaxí velice podobný s jazyky C++ a Java. Většina příkazů je ukončena středníkem (;), příkazy tedy mohou přesahovat na více řádků, přičemž není nutné uvádět žádný znak pro pokračování (například podtržítko (_) ve Visual Basicu). Lze vytvářet bloky příkazů pomocí složených závorek ({}). Komentáře na jeden řádek začínají dvěma lomítky (//), více řádkové lomítkem a hvězdičkou (/*) a končí hvězdičkou a lomítkem (*/). Nejčastější chybou pří kompilaci, které se dopouštějí programátoři, kteří nejsou zvyklí na jazyk ve stylu C, je vynechání středníku. Zároveň se základní syntaxí C# je dobré zmínit, že se v ní rozlišují velká a malá písmena. Například proměnné pojmenované *Prom* a *prom*, jsou dvě odlišné proměnné, které mohou nabývat různých hodnot.

3.2 Windows Presentation Foundation (WPF)

S kontinuálním nárůstem cross-platform aplikací založených na HTML a JavaScript, Windows naléhavě potřeboval novou technologii s rámovou konstrukcí, jako

mají Windows Forms a vyvinul Windows Presentation Foundation (WPF), který umožňuje softwarovým vývojářům a grafikům vytvářet moderní uživatelské rozhraní, bez nutné znalosti dalších složitých technologií. Stručně řečeno, WPF si klade za cíl spojit nejlepší vlastnosti systémů, jako jsou DirectX (3D a hardwarovou akceleraci), Windows Forms, Adobe Flash (silnou podporu animace) a HTML.
Mezi nejvýznamnější technologické funkce WPF patří dle (PRAJAPATI, 2015):

- **Využívání DirectX** – v případě, že je nezbytné použít akcelerované grafické vykreslování GPU, WPF vyžívá DirectX namísto zastaralého subsystému GDI. DirectX umožňuje provádět složité grafické úkoly na GPU a tím snížit zatížení CPU a dovoluje tak zároveň CPU dělat jinou práci a tím zvýšit výkon aplikace.
- **Deklarace uživatelského rozhraní pomocí XAML** – Extensible Application Markup Language (XAML) je založený na značkovacím jazyce XML. XAML je jazyk pro vytváření vizuální prezentace ve WPF aplikacích.
- **Dependency Properties** – WPF zavádí nový typ vlastnosti (property) nazvané DependencyProperty. DependencyProperty je závislá na více zdrojích, které určují její hodnotu v jakémkoli časovém bodě. Hodnota DependencyProperty je dynamicky vypočtena. Jejími hlavními výhodami jsou snížené nároky na paměť, dědičnost hodnoty a notifikace při její změně.
- **Data binding** – je proces, který vytváří spojení mezi uživatelským rozhraním aplikace a její logikou. S pomocí data bindingu je možné vzít téměř jakoukoli vlastnost jakéhokoli objektu a „nabindovat“ (tedy navázat) ji na příslušnou DependencyProperty vizuálního objektu.
- **Šablony a styly** – umožňují provádět složité změny a redefinice na vzhledu ovládacího prvku, aniž by bylo nutné vytvořit nový. Lze také změnit strukturu a vzhled ovládacího prvku změnou jeho ControlTemplate.

3.2.1 Vznik

První verze WPF vyšla v listopadu 2006, byla nazvána WPF 3.0, protože byla dodávána jako součást rozhraní .NET Frameworku 3.0. Druhé vydání WPF 3.5, přišlo téměř o rok později a v srpnu 2008 již vyšla třetí verze, opět skoro o rok později. Tato verze byla součástí Service Packu 1 pro .NET 3.5, ale tento service pack, v případě WPF,

neopravoval pouze chyby, ale obsahoval mnoho nových funkcí a vylepšení (NATHAN, 2010).

Kromě těchto hlavních verzí představil Microsoft v srpnu 2008 „WPF Toolkit“ na http://wpf.codeplex.com, který se aktualizuje několikrát do roka. WPF Toolkit je používán jako způsob, jak rychleji dodávat nové funkce a ovládací prvky v experimentální formě (často i se zdrojovým kódem). Funkce a ovládací prvky z WPF Toolkitu jsou později implantovány do další nové verze WPF a to na základě zpětné vazby od uživatelů o jejich vhodnosti a připravenosti (NATHAN, 2010).

Když byla vydána první verze WPF prakticky neexistovali žádné podpůrné nástroje nebo ovládací prvky, ale bezprostředně na to vyšlo primitivní WPF rozšíření pro Visual Studio 2005 a také první verze Expression Blend. Nyní již Visual Studio má nejen zabudovanou silnou podporu WPF, ale bylo i předěláno, aby se stalo WPF aplikací samo o sobě. Expression Blend, aplikace postavená také na WPF, složí k navrhování a vytváření prototypů uživatelského rozhraní.

V posledních několika letech bylo vydáno od nejrůznějších společností mnoho aplikací založených na WPF, mezi nejvýznamnější patří například Autodesk, SAP, Disney, Blockbuster, Roxio, AMD, Hewlett-Packard, Lenovo. Microsoft má přirozeně sám dlouhý seznam softwaru vytvořeného ve WPF (Visual Studio, Expression, LifeCam, Games for Windows LIVE Marketplace, Office Communicator Attendant, Active Directory Administrative Center, PowerShell ISE).

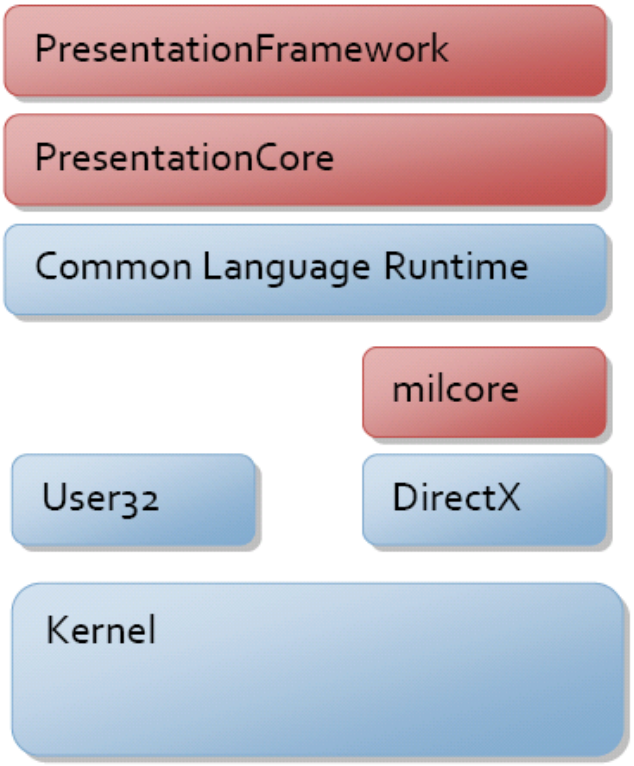
Verze WPF	Datum vydání	.NET verze	Verze Visual Studia	Hlavní charakteristiky
3.0	2006-11	3.0	-	První verze. WPF lze vyvíjet ve Visual Studiu 2005.
3.5	2007-11	3.5	VS 2008	Změny a vylepšení: Aplikační model, data binding, ovládací prvky, dokumenty, poznámky, 3D prvky uživatelského rozhraní.
3.5 SP1	2008-08	3.5 SP1	-	Nativní podpora úvodní obrazovky, Nový ovládací prvek WebBrowser, podpora DirectX pixel shader.
4.0	2010-04	4.0	VS 2010	Nové ovládací prvky: kalendář, DataGrid a DatePicker. Vícedotykové ovládání
4.5	2012-08	4.5	VS 2012	Nový ovládací prvek Ribbon, rozhraní INotifyDataErrorInfo.
4.5.1	2013-10	4.5.1	VS 2013	Žádná významná změna.
4.5.2	2014-05	4.5.2	-	Žádná významná změna.
4.6	2015-07	4.6	VS 2015	Podpora vysokého rozlišení a dotyková vylepšení.

Tabulka 1 - Verze WPF (PRAJAPATI, 2015).

3.2.2 Architektura

Přehled architektury WPF byl zpracován podle zdroje od výrobce Microsoft (WPF Architecture, 2016). Popis architektury je tvořen hierarchií tříd, které pokrývají většinu subsystémů WPF, tyto subsystémy jsou znázorněné na schématu níže (Obrázek 3). Červené části jsou hlavní součásti WPF, přičemž část nazvaná Milcore je napsaná

v nespravovaném kódu, tedy pod úrovní CLR a to z důvodu lepší integrace WPF s DircetX.



Obrázek 3 - WPF architektura (WPF Architecture, 2016)

Většina objektů ve WPF je odvozena ze třídy **System.Threading.DispatcherObject**, která zajišťuje řízení jednotlivých vláken objektů, aby nedošlo k jejich souběhu. Odvozením objektů z třídy DispatcherObject se vytvoří CLR objekty, které se budou chovat jako STA (Single-Threaded Apartment) objekty a tím je dán disptacheru ukazatel na objekt v čase jeho vzniku. Mezi objekty patřící do STA vláken je potřeba komunikovat a ověřovat, že tato komunikace probíhá ve správném vlákně – v tom spočívá role dispatchera, který řídí přístupy z různých vláken k objektům.

Třída **System.Windows.DependencyObject** umožňuje jeden z hlavních cílů architektury WPF, kterým je preference vlastností objektů před jejich metodami nebo událostmi. Vlastnosti jsou deklarativní a dovolují více specifikovat záměr. Hodnoty DependencyObject jsou automaticky aktualizovány, pokud se změní závislá vlastnost a tato hodnota je poděděna. Díky tomuto systému je umožněn tzv. „data binding“.

Třída **System.Windows.Media.Visual** zajišťuje uspořádání stromu z vizuálních objektů do kterého jsou uživatelská rozhraní ve WPF sestavována. Každý takový vizuální objekt v sobě obsahuje instrukce a meta data, jak objekt vykreslovat. Třída Visual je spojujícím bodem mezi dvěma subsystémy, spravovaným API a nespravovaným Milcore.

System.Windows.UIElement určuje hlavní subsystém, který se stará o rozvržení visuální prvků a jejich událostí. Na úrovni UIElement, je zajištěno dynamické a flexibilní rozvržení ovládací prvků, které je definované spíše vlastnostmi než imperativní logikou.

System.Windows.FrameworkElement zprostředkovává mnoho funkcí z jádra WPF, jako je například přímý přístup k animacím pomocí metody BeginStoryboard. Dvě nejdůležitější funkce, které třída zajišťuje, jsou data binding a styly.

System.Windows.Controls.Control je třída, ze které už se dále odvozují jednotlivé ovládací prvky. Její nejvýznamnější charakteristikou je využívání systému šablon, které dovoluje jednoduše parametrizovat vykreslování prvku. Dále poskytuje sadu vlastností, které se buď přizpůsobují modelu interakcí (commandy a události) anebo zobrazení (které je určeno šablonou).

3.2.3 XAML

XAML (čteno [zaml]), je zkratkou Extensible Application Markup Language, je to značkovací jazyk, který se využívá ke konstruování a inicializování .NET objektů. Ačkoli se dá XAML využít v mnoha jiných oblastech je hlavně využíván k vytváření uživatelských rozhraní ve WPF. Jinými slovy se v XAMLu definují uspořádání ovládacích prvků, jako jsou například různé panely, tlačítka, textboxy atd., které vytváří okno WPF aplikace.

Zvětší části, při vytváření nějakého uživatelského rozhraní, je XAML kód generován automaticky. K tomu se používá program Expression Blend, který slouží hlavně grafikům a je využívá ke složitějším grafickým konstrukcím. Samotní vývojáři aplikací pak využívají Visual Studio, kde lze vytvářet základní uživatelská rozhraní. Díky generovanému XAMLu je pak dosaženo jednoduší spolupráce mezi programátory a grafiky.

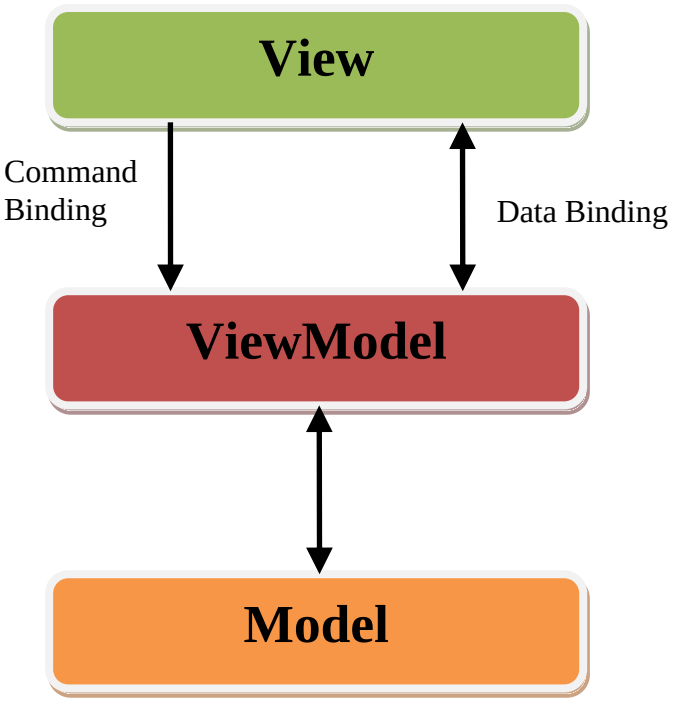
XAML, jako úplný jazyk je myšlen hlavně v souvislosti .NET, kde pomocí XML syntaxe definuje strom .NET objektů, ale XAML jako takový by mohl být využíván i jinými platformami. XAML je v .NET využit jako podmnožina i pro jiné technologie než je pouze WPF (MACDONALD, 2010):

- XPS XAML – je součást WPF XAMLu, která definuje elektronické dokumenty representované v XML.
- Silverlight XAML – je podmnožinou WPF zaměřenou na webové aplikace projektu Silverlight.
- WF XAML – zahrnuje elementy, které popisují obsah Windows Workflow Foundation (WF).

3.3 Model-View-ViewModel (MVVM)

V roce 2005 vývojář John Gossman, pracující tehdy na vývoji WPF, publikoval na svém blogu první představu o návrhovém vzoru MVVM. Ve své příspěvku popsal variaci na již známý návrhový vzor Model/View/Controler (MVC), kde za „View“ byli více odpovědní grafičtí designeři než klasičtí programátoři aplikací. Dále uvedl, že jeho hlavní motivací bylo vylepšit pracovní postupy mezi grafiky uživatelského rozhraní a programátory aplikací (LIKNESS, 2012).

Tento návrhový vzor se tedy snaží hlavně oddělit logiku aplikace od uživatelského rozhraní a to rozdělením aplikace do tří vrstev: Model, View a ViewModel. Kde **View** je definováno deklarativním způsobem (v případě XAMLu) a je zodpovědné za vstupy od uživatele, klávesové zkratky, zobrazované prvky apod. **Model** popisuje data, se kterými aplikace pracuje. **ViewModel** se stará hlavně o specifické úkoly, které se nedají zobecnit v Modelu (komplexní UI operace), drží stav View a stará se o zobrazení Modelu ve View a tím spojuje Model a View.



Obrázek 4 - MVVM

Hlavními přednosti návrhového vzoru MVVM, tak jak je definuje zdroj Likness (LIKNESS, 2012), mimo to že odděluje design od logiky, jsou:

- **Oddělená architektura** – MVVM následuje osvědčené postupy návrhu software.
- **Designer/developer workflow** – MVVM dovoluje paralelní vývoj více členných vývojových týmů pracujících na stejném projektu.
- **Unit testing** – automatické testování a ověřování fungování a korektnosti implementace systému.
- **Použití data-bindingu** – MVVM má přímé využití data-bindingu mezi View a ViewModelem.
- **Vylepšuje znovu-použitelnost kódu –** kdy různé kostry ViewModelu nebo různé pomocné třídy k vylepšení View mohou být využity v jiných firemních projektech.
- **Modularita** – MVVM podporuje modulární design, což umožňuje lépe spravovat jednotlivé části aplikace.
- **Rozšiřitelnost** – MVVM aplikaci lze jednodušeji rozšiřovat například o další moduly a obrazovky uživatelského rozhraní.

3.3.1 **Model**

MVVM Model bývá často zaměňován za datový model, ale je to spíše konkrétní implementace doménového modelu. Model zahrnuje vše, co je nutné k vyřešení zadaného business problému. Jednoduchým příkladem by mohl bankovní systém, který obsahuje zákazníky a účty. Objektová representace účtů a zákazníků je obsažena v modelu a popisuje, jak spolu objekty souvisejí (zákazník může mít víc jak jeden účet). Také popisuje jejich stav (účet je otevřen nebo uzavřen) a chování (účtu narůstá hodnota) (LIKNESS, 2012).

Model by měl odhalovat pouze ty části, které jsou nezbytné pro další vrstvy aplikace. Například presentační vrstva by neměla nic vědět o tom, jak jsou data uchovávaná (v databázi či XML souboru), nebo jak jsou data získávána (přes síť jako binární objekty nebo lokálně ze souboru). Modelu, který by byl příliš otevřený, tedy odhaloval by například způsob uchování dat, by mohl v aplikaci zapříčinit vznik nadbytečné závislostí a zbytečně komplikovaný kód (LIKNESS, 2012).

3.3.2 **View**

View je v zásadě uživatelské rozhraní, které interaguje s uživatelem. Uživatelské rozhraní bývá založeno na značkovacím jazyce XAML, pokud je implantován návrhový vzor MVVM. XAML spolupracuje se systémem DependecyProperties a díky tomu View dokáže okamžitě presentovat uživateli informace podle jeho vstupů. Tabulka níže ukazuje běžné součásti View.

Komponenta	Popis
XAML	Zajišťuje rozvržení ovládacích komponent v uživatelském rozhraní.
Value convertors	Speciální třídy umožňující transformaci data do podoby v uživatelském rozhraní a nazpět.
Data templates	Šablony mapující části dat na ovládací prvky.
Visual state groups	Pojmenované stavy, které ovlivňují vlastnosti různých grafických elementů a dodávající tak visuální změny na základě vnitřní logiky.
Storyboards	Animace.
Behaviors	Algoritmus chování, který je použitelný zároveň pro různé typy ovládacích prvků.
Triggers	Algoritmus, který se spouští s danou událostí na ovládacím prvku.
Code-behind	Zde jsou možná kódová rozšíření XAMLu.

Tabulka 2 - View komponenty (LIKNESS, 2012)

3.3.3 **ViewModel**

Vrstva ViewModel je to co dělá MVVM jedinečným. Představuje ji jednoduchá třída, které je odpovědná za součinnost interakcí mezi View a Modelem. ViewModel by měl být těžištěm prezentační logiky aplikace a ke komunikace s View využívat zejména těchto tří metod (LIKNESS, 2012):

- Data-bindingu
- Visual states
- Commandů nebo metod

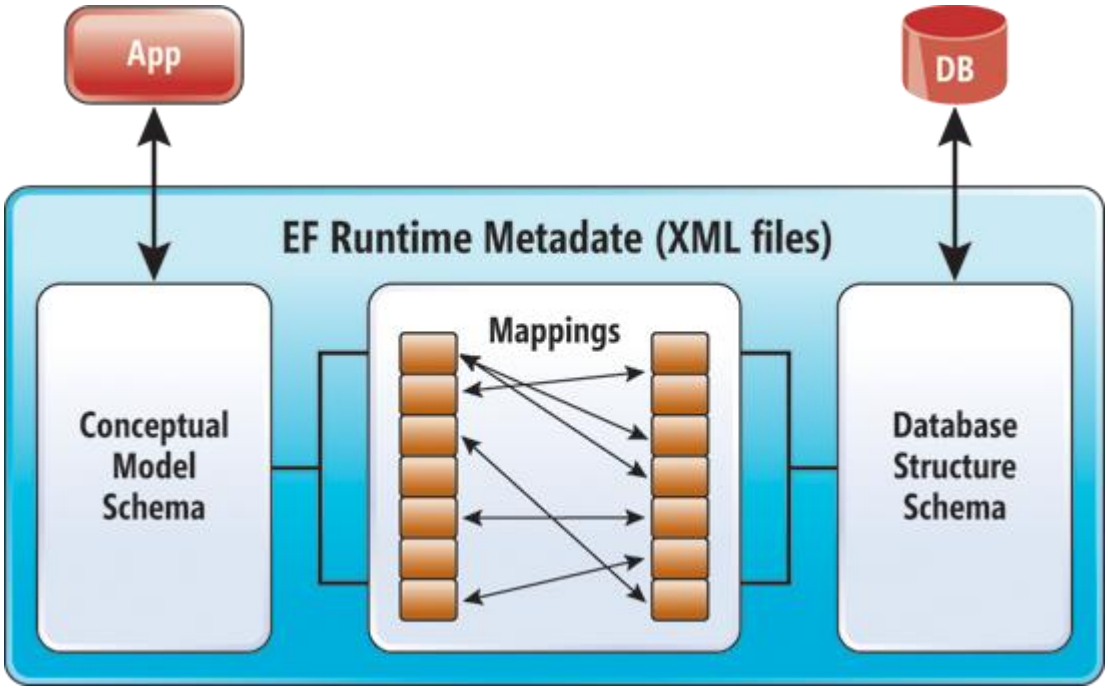
ViewModel má nejčastěji zabudovaný nějaký druh property-changed rozhraní a validačního rozhraní. Pomocí těchto rozhraní jsou změny hodnot vlastností objektů okamžitě promítány do View anebo naopak do ViewModelu a data zadávána uživatelem okamžitě validována. Je možné také sdílet stejný ViewModel pro více různých View, například pokud jsou tyto View jen jinou vizuální reprezentací stejných dat, nebo lze dynamicky měnit různá View pro stejný ViewModel, na základě daných podmínek.

3.4 **Entity Framework**

Entity Framework je technologie od firmy Microsoft. V mnoha různých projekt je možné se setkat s tzv. „*databázovou vrstvou*“, která odděluje objektově orientovaný kód od databázového úložiště. Tato vrstva se stará o transformaci objektů na sloupce a řádky,

které ukládá do tabulek. Entity Framework společně s Language-Integrated Query (LINQ) jsou technologie vyvinuté k automatickému vytvoření takovéto databázové vrstvy, které překlenuje nesoulad mezi objektový a relačním pojetím dat. S pomocí Entity Frameworku lze v grafické návrhářské prostředí Visual Studia designovat jednotlivé entity a vazby mezi nimi. Tato technologie tedy hlavně slouží k vývoji aplikací, které pracují s daty. Společně s Visual Studiem tvoří úplný ekosystém k vývoji datově orientovaných aplikací. Tento ekosystém lze použít k vývoji desktopových aplikací, server-side aplikací, internetových aplikací používajících ASP.NET nebo Silverlight a Windows Comunication Foundation (WCF) aplikací.

Entity Framework Designer je nástroj, který slouží k vytvoření konceptuálního modelu, databázového modelu a mapovací vrstvy mezi nimi. Tento nástroj umožňuje vývoj modelu obousměrně, jak vytvořením nového objektového modelu, tak automatickým generování objektů z importované dříve vytvořené databáze. Dovoluje tedy vytvoření databáze podle modelu anebo naopak aktualizace modelu podle změn v databázi.



Obrázek 5 - Entity Framework schéma (LERMAN, 2010).

3.4.1 **Modely**

Modely vytvořené Entity Frameworkem jsou popsány Entity Data Modelem (EDM). EDM je formální struktura, která definuje data používaná ve vytvářené aplikaci.

EDM definuje datové typy, typy vazeb mezi entitami a mapování na schéma databáze. Struktura EDM je tvořená třemi vrstvami: konceptuální vrstvou, databázovou vrstvou a mapovací vrstvou. Syntaxe každé z těchto vrstev je založena na XML a jejich schéma je definované v EDM. V projektu vytvářené aplikace bývají všechny tři vrstvy zabaleny do jednoho souboru s koncovkou .edmx. Každá z těchto tří vrstev je definována jiným jazykem (TENNY et. al., 2010):

- **Conceptual Schema Definition Language (CSDL)** – definuje syntax konceptuální vrstvy, ve které vývojář vytvoří entity, vazby mezi nimi a případně hierarchii dědičnosti.
- **Store Shema Definition Language (SSDL)** – definuje syntax databázové vrstvy, které obsahuje tabulky, sloupce a jejich datové typy, které EntityClient mapuje na podkladovou relační databázi.
- **Mapping Specification Language (MSL)** – definuje syntax mapovací vrstvy, mezi jinými věcmi pro představu určuje, jak jsou vlastnosti entit svázány se sloupci v relační tabulce.

3.4.2 **Terminologie**

V této kapitole je popsáno několik nejzákladnějších termínů používaných v Entity Frameworku. Prvním pojmem, od kterého se následně odvíjejí další, je *EntityType*, který definuje nový typ, v podstatě jako třída. Instance EntityType pak odkazuje na konkrétní entitu. EntityType stejně jako třída má jednu nebo více vlastností. Tyto vlastnosti mohou být definovaný jako základní datové typy (integer, string), *ComplexTypes* (používaný k seskupení souvisejících vlastností) nebo kolekce. *Navigační* vlastnosti odkazují na jinou entitu podle nastavené vazby mezi EntityType. Typy takových to vazeb jsou například: 1-1, 1-n (nebo 0..1-n) a n-n. Každý EntityType musí mít také *EntityKey*, který je složen z jedné, nebo více vlastností a slouží k jednoznačné identifikaci instance dané EntityType. EntityKey je obvykle vyžadován podkladovou relační databází, kde reprezentuje primární klíč. Veškeré instance EntityType, nebo jejich odvozené typy, jsou obsaženy v *EntitySet* (TENNY et. al., 2010).

4 Vlastní práce

Cílem v praktické části diplomové práce bylo vytvořit aplikaci v prostředí .NET založené na technologii Windows Presentation Foundation (WPF), která využívá návrhový vzor MVVM a technologii Entity Framework. Tato část práce se zaměřuje na praktickou demonstraci užití návrhového vzoru MVVM ve vytvářené aplikaci. Popisuje části aplikace, které souvisí, nebo jsou součástí MVVM a zobrazuje postupy použití na ukázkách programového kódu.

V úvodu práce byl stanoven účel využívání aplikace - ta je vyvíjena pro Český statistický úřad a to konkrétně pro potřeby zpracování mikro dat z domácnostních šetření. Aplikace vzniká jako součást nového systému, který by měl nahradit a sjednotit staré postupy při zpracování mikro dat z domácnostních šetření, které jsou například IŠD (Integrované šetření v domácnostech), SILC (Výběrové šetření Životní podmínky), AES (Šetření o vzdělávání dospělých) atd. Na úvod navazují teoretická východiska, kde je hlavním zjištěním metodika užití technologií použitých k vypracování vlastní práce.

Vlastní práce je rozdělena na 7 hlavních kapitol, jež jsou zejména zaměřeny na popis jednotlivých vrstev návrhového vzoru MVVM. Vlastní práce začíná kapitolou Popis aktuálního stavu, kde jsou uvedeny postupy a technologie, které má vytvářená aplikace nahradit. Následující kapitola Zadání je základní definicí metodiků a dalších odborných pracovníků ČSÚ. Definice obsahuje, co má aplikace dělat a jakým způsobem se má chovat. Následující kapitola Databáze je již zaměřena zejména na tvorbu vrstvy modelu a propojení aplikace s úložištěm. Kapitola View navazuje s popisem důležitých částí a poznatků při tvorbě vrstvy grafického rozhraní. Poslední vrstva návrhového vzoru, ViewModel, je obsahem kapitoly MVVM. Součástí této kapitoly jsou také další nezbytné části a doplňky použité k propojení vrstev aplikace. Kapitola Nasazení se již přesouvá k tomu jakým způsobem je aplikace instalována a jak probíhají automatické aktualizace. Poslední kapitola Vlastní práce, Další rozvoj, představuje vhodná vylepšení návrhového vzoru MVVM v aplikaci.

4.1 Popis aktuálního stavu

Český statistický úřad pro zpracování mikro dat z domácnostních šetření využíval do současnosti různé postupy pro každé jednotlivé šetření. První krok, předání dat z terénu, však probíhal u většiny šetření stejně. Data se v domácnostních šetřeních sbírají za pomoci softwaru Blaise, který data získaná tazateli ukládá do své síťové databáze. Tato databáze, pak byla převáděna do jednotlivých tabulek a ve formátu csv předána k dalšímu zpracování a vytvoření finálních mikro dat. Výjimkou pro tento krok bylo výběrové šetření informačních technologií, které patří mezi menší úlohy a šetří se v rámci IŠD – u tohoto šetření se provádělo veškeré nutné zpracování pro vytvoření výsledných mikro dat při převodu z Blaise databáze do výstupních csv souborů.

V rámci přechodu na nový systém byl již tento první krok nahrazen a veškerá data se nyní automaticky zpracovávají do databáze Oracle. Tabulky, které vznikají, si metodici jednotlivých úloh definují v programu BDBConverter. Centrální databáze Oracle tedy nyní obsahuje jak samotná data z terénu, tak i veškeré popisy těchto dat (meta data). Dostupné jsou tedy i informace o způsobu uložení dat až do úrovně sloupců. Tato data jsou dle nastavení příslušných práv přístupna pouze určeným metodikům a případně dalším osobám, které data zpracovávají.

Další zpracování dat z terénu do podoby finálních mikro dat probíhá u každé úlohy různým způsobem, většinou je to však pomocí softwaru Visual FoxPro, od jehož užívání se v dnešní době spíše ustupuje. Formát dbf, do kterého se data z úloh ukládají, je dnes již zastaralý a není ani příliš podporován novými statistickými softwary. Musí se tedy opět převádět do formátu csv, aby jej mohli dále používat jiné statistické výpočetní systémy, jako jsou například SPSS (Statistical Package for the Social Sciences) nebo SAS (Statistical Analysis System). Další nevýhodou je skladování dat v souborech, kdy je nutné tato data uchovávat po dobu několika let – zde opět chybí integrace moderního databázového systému, kde by byla všechna data integrovaná, dostupná a zálohovaná po určenou dobu.

Cílem je tedy vytvořit aplikaci pro všechny metodiky různých úloh, která by sjednotila a zjednodušila zpracování mikro dat. Nahradila by již staré nepodporované technologie a také zajistila vyšší ochranu ztráty dat a jejich zabezpečení před zneužitím.

Zásadní je také vytvářet meta data, tedy popis o tom jak data vznikla, který bude dále využit při budování navazující části systému pro zpracování dat. Výsledkem by měla být další aplikace sloužící k vytváření výsledných agregovaných dat.

4.2 Zadání

Nově vznikající aplikace s názvem **OUdot**, jejíž vývoj je předmětem této diplomové práce, by měla umožňovat zpracování dat do podoby definované uživatelem aplikace. Uživatelé aplikace by měli být převážně metodici jednotlivých šetření a případně jejich další spolupracovníci. Z toho vyplývá, že by aplikace měla být maximálně uživatelsky přívětivá, přehledná a měla by obsahovat integrované funkce pro transformaci dat.

Aplikace by měla obsahovat možnost vlastního nadefinování výstupních tabulek, které budou obsahovat odvozené ukazatele (sloupečky tabulky nebo také proměnné). Tabulky by měli být primárně definovány:

- názvem
- primárním klíčem, který bude přebíraný ze zdrojových tabulek
- definicí řádků, které tabulka bude obsahovat (jejich počet)
- pořadí v jakém bude tabulka vypočítána

Odvozené ukazatele, jak již bylo výše zmíněno, by měly být vytvářeny k jednotlivým tabulkám a budou především určeny:

- názvem
- popisem
- metodikou, jejich výpočtu
- typem (číslo, text, kategorie atd.)
- zdroji z kterých je vypočten/odvozeny
- typem funkce sloužící k jeho výpočtu
- PL/SQL kódem, sloužícímu k výpočtu
- Generací a pořadím v generaci, ve kterém je odvozený ukazatel vypočten

Veškerá tato meta data budou dále použita k výpočtu finálních mikro dat. Po dokončení definic, uživatel spustí výpočet a program provede výpočty jednotlivých odvozených ukazatelů v pořadí podle definované generace (od nejnižší) a následně podle:

1. pořadí tabulek (od nejnižšího)
2. pořadí výpočtu odvozeného ukazatele v dané generaci (od nejnižšího)

Zdroji dat, jež by měly sloužit k výpočtu odvozených ukazatelů, budou zejména vstupy sbírané v rámci terénních výzkumů – ty jsou obsaženy i s jejich popisem v centrální databázi Oracle. Dále externí data, která si uživatelé za pomoci OUdotu budou moci nahrát do databáze a následně je využít k výpočtu. Také by mělo být možno využít jako zdroj jiný odvozený ukazatel, který je v nižší generaci (případně pokud jsou ve stejné generaci, jeho pořadí výpočtu bude nižší).

Veškeré vytvořené definice pro jednotlivé úlohy by mělo také být možné převést do dalších období, kde se budou již pouze upravovat a doplňovat.

Program je vyvíjen na šetření SILC, které z hlediska tvorby mikro dat patří k nejsložitějším. Mikro data vzniknou jak za pomoci OUdotu, tak i starým způsobem zpracování pomocí Visual FoxPro a následně budou data porovnána. Díky tomu bude aplikace doplněna o další nezbytné funkce a lépe připravena k nasazení na zbylá šetření.

4.2.1 Shrnutí zadání

1. Definice tabulek musí obsahovat:
    a. název
    b. popis
    c. primární klíč, který bude přebíraný ze zdrojových tabulek
    d. definice získání/výpočtu řádků, které tabulka bude obsahovat
    e. pořadí v jakém bude tabulka vypočítána

2. Definice odvozených ukazatelů (sloupců tabulky) musí obsahovat:
    a. názve
    b. popis
    c. metodika výpočtu
    d. typ (číslo, text, kategorie atd.)
    e. zdroje, z kterých je vypočten/odvozen
        i. data z terénu
        ii. jiný odvozený ukazatel
        iii. externí zdroj
    f. typ funkce sloužící k výpočtu (automaticky generuje PL/SQL kód)
    g. PL/SQL kód, sloužící k výpočtu
    h. Generací a pořadím v generaci, ve kterém je odvozený ukazatel vypočten
3. Výpočet je proveden podle:
    a. generace
    b. pořadí tabulky
    c. pořadí v dané generaci
4. Veškeré výstupy ukládány přímo do centrální databáze Oracle
5. Veškeré definice možnost převést do dalších období

4.3 Databáze

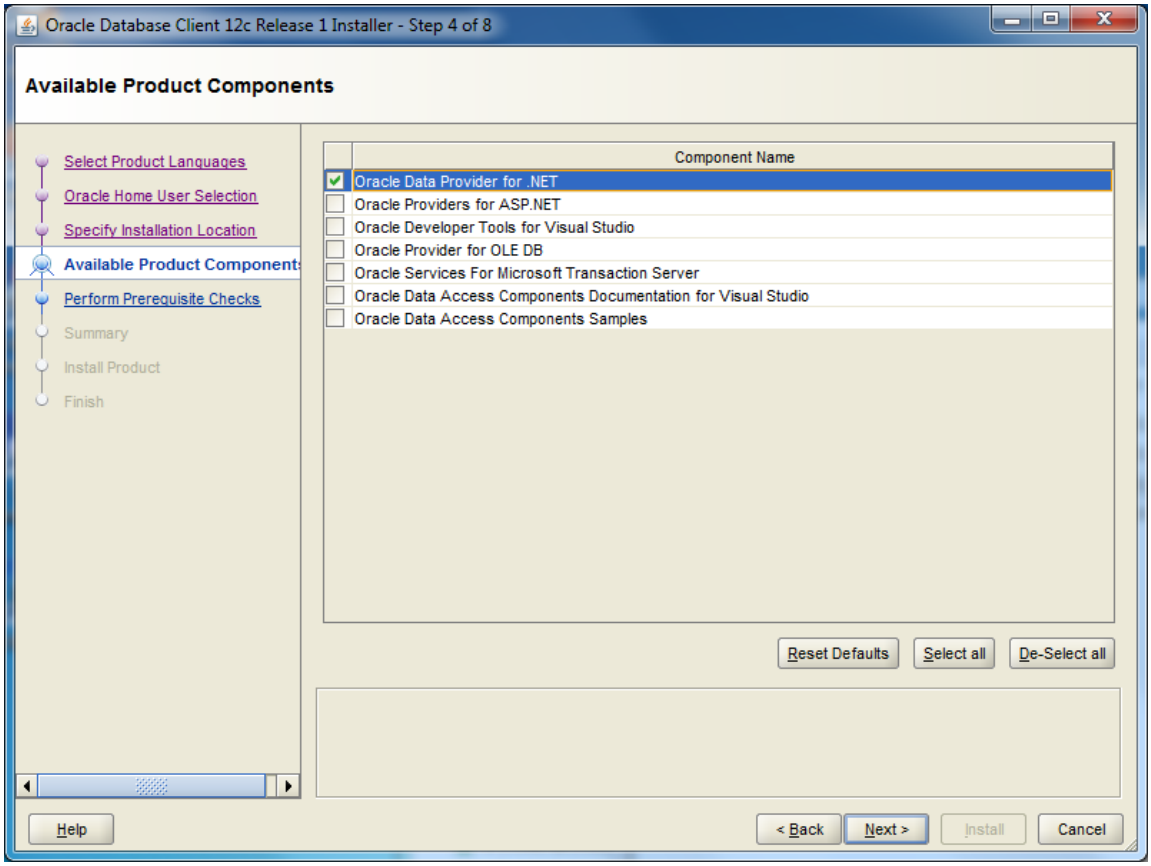
Aplikace OUdot bude využívat centrální databázi Oracle, která již nyní obsahuje zdroje dat, které bude nově vznikající aplikace transformovat do finální podoby mikro datových struktur (včetně jejich meta datového popisu). Další meta data, která budou vznikat jako definice ve vytvářené aplikaci, bude také nutné ukládat s výslednými daty do databáze. Z těchto důvodů budou veškeré struktury datového modelu nejprve definovány v relační databázi Oracle a následně za pomoci Entity Frameworku bude vygenerován Entity Data Model (EDM) vznikající aplikace popsaný v kapitole 3.4.1.

Pro komunikaci mezi platformou .NET a databází Oracle je nutné nainstalovat Oracle Data Provider for .NET[1], aby bylo možné využít výhody Entity Frameworku. Tento software umožňuje vývojářům v .NET využívat pokročilé funkce databáze Oracle. Součástí balíčku je specializovaný nástroj pro vývojáře, který umožňuje obě dostupné možnosti modelování EDM:

- vytvoření entity ve Visual Studiu a jejich následné převedení do Oracle na relační databázový model
- generování EDM z relačního databázového modelu (v případě aplikace OUdot)

Při použití tohoto nástroje od firmy Oracle je nevýhodou nutná instalace balíčku na klientské stanice. Nicméně zde již stačí nainstalovat pouze základní data provider a ostatní pomocné nástroje již nejsou pro provoz vytvořené aplikace nutné. V případě aplikace OUdot tato nevýhoda není příliš podstatná, protože se jedná o podnikovou aplikaci s minimálním počtem uživatelů.

[1] http://www.oracle.com/technetwork/topics/dotnet/index-085163.html



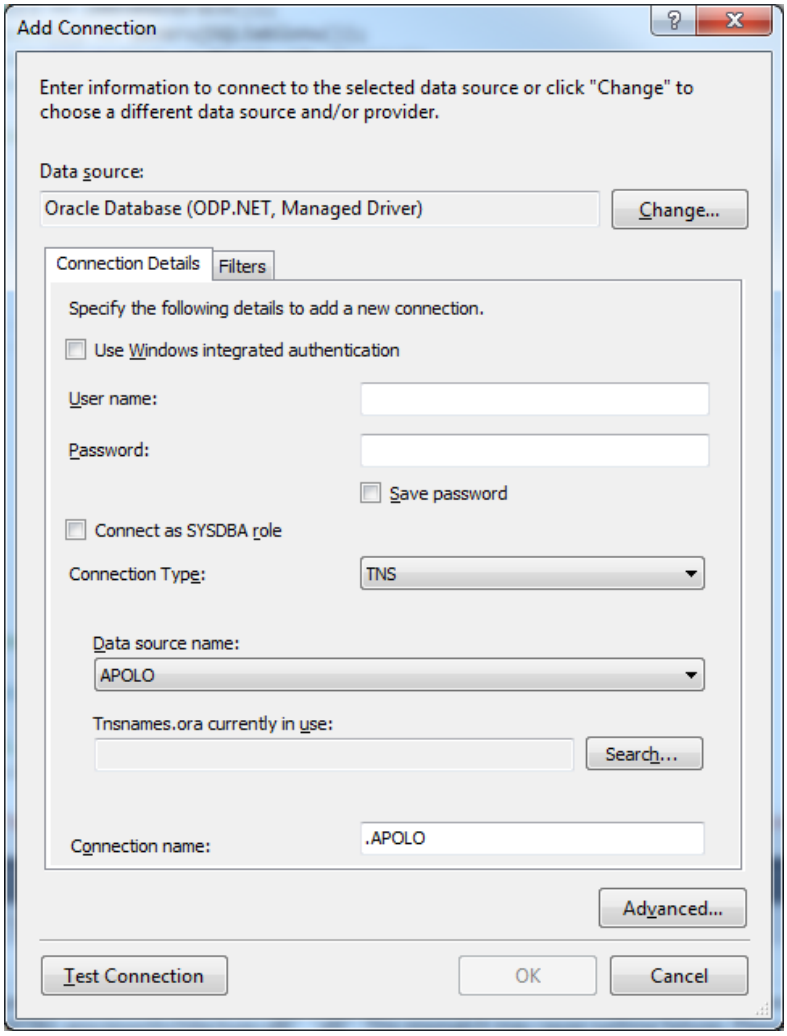
Obrázek 6 - Oracle Data Provider for .NET

4.3.1 Model

Před vygenerováním modelu EDM byly veškeré potřebné datové struktury založeny v databázi. Aby bylo možné požívat Entity Framework je nezbytné do projektu ve Visula Studiu (VS) nainstalovat Entity Framework balíček[2], který provede nezbytné úpravy a nainstaluje potřebné knihovny. To je nejjednodušší pomocí správce balíčků NuGet pro vývojovou platformu .NET.

Dříve než se začne s vytvářením EDM je vhodné si nejprve přidat (případně otestovat) připojení do databáze. Tuto konfiguraci je možné provést ve VS v Server Exploreru, pomocí tlačítka Add Connection, kde se definuje připojení k datovým zdrojům (viz Obrázek 7).

[2] https://www.nuget.org/packages/EntityFramework/



Obrázek 7 - VS, konfigurace připojení k DB

Samotné vytvoření modelu generovaného z databáze není nějak komplikovaný úkon. Ve VS se do solutionu přidá nový prvek, konkrétně se vybere ADO.NET Entity Data Model. Je vhodné jej vložit do separátního jmenného prostoru, v případě OUdotu je tento jmenný prostor nazvána Model, protože aplikace bude vytvořena podle návrhového vzoru MVVM a vygenerovaný EDM bude representovat Modelovou vrstvu tohoto návrhového vzoru. Po připojení k databázi byly vybrány veškeré databázové struktury, se kterými bude aplikace pracovat a vznikl výsledný EDM model, který je zobrazen na Obrázek 8.

Obrázek 8 - OUdot EDM

4.3.1.1 Parciální třídy

Objekty, které vznikly vygenerováním z databáze, jsou ve vývojovém prostředí C# reprezentovány parciálními třídami. Z Obrázek 8 jsou patrné vlastnosti těchto tříd i tzv. navigační vlastnosti, které jsou reprezentací vazeb mezi těmito třídami. Navigační vlastnosti odkazují podle typu vazby na jeden konkrétní objekt, nebo na kolekci objektů.

```
public partial class OU_TAB
{
        public OU_TAB()
        {
            this.OU_PRIMARNI_KLICE = new
ObservableCollection<OU_PRIMARNI_KLICE>();
            this.OU_ODVOZ_UKAZ = new ObservableCollection<OU_ODVOZ_UKAZ>();
        }

        public decimal ID { get; set; }
        public string JMENO { get; set; }
        public string POPIS { get; set; }
        public decimal ID_METOD_OBDOBI { get; set; }
        public bool POMOCNA { get; set; }
        public int PORADI { get; set; }
        public string SQL_GENEROVANI_PK { get; set; }
        public bool SQL_VLASTNI_KOD { get; set; }
        public decimal BDBC_TAB_PRO_PK { get; set; }
        public string SQL_OPRAVA_DAT { get; set; }
        public int GEN_ZALOZ_PK { get; set; }

        public virtual BDBCTABULKY BDBCTABULKY { get; set; }
        public virtual ObservableCollection<OU_PRIMARNI_KLICE> OU_PRIMARNI_KLICE
{ get; set; }
        public virtual ULOHA_METOD_OBDOBI ULOHA_METOD_OBDOBI { get; set; }
        public virtual ObservableCollection<OU_ODVOZ_UKAZ> OU_ODVOZ_UKAZ { get;
set; }
    }
```

Ukázka kódu 1 - EF generovaná třída

Důvodem vzniku vygenerovaných tříd jako parciálních je, že lze pak do těchto tříd přidávat další vlastní logiku na jiném místě v našem projektu a nenarušit tak kód vygenerovaný Entity Frameworkem. To je výhodné zejména pokud by došlo k nějaké třeba i drobné změně a bylo by nutné aktualizovat EDM model – při opětovném generování třídy se vlastní logika neztratí. V následující Ukázka kódu 2 z programu OUdot je přidána vlastní logika do té samé vygenerované třídy, ve které je zobrazena v Ukázka kódu 1. Tato část stejné třídy s vlastní logikou je definovaná ve stejném jmenném prostoru jako třída vygenerovaná, ale nachází se na jiném místě i v jiném souboru.

```
    public partial class OU_TAB
    {
        /// <summary>
        /// Založení nové tabulky
        /// </summary>
        /// <param name="ObdobiTab">Období do kterého zakládám novou
tabulku</param>
        public void Pridej(ULOHA_METOD_OBDOBI ObdobiTab)
        {

            this.SQL_VLASTNI_KOD = false;
            this.ULOHA_METOD_OBDOBI = ObdobiTab;
            DBContext.Instance.Entity.OU_TAB.Add(this);

        }

        /// <summary>
        /// Automatické založení primárního klíče
        /// </summary>
        internal void GenerujSQLZalozeniPK()
        {
            if (this.SQL_VLASTNI_KOD)
                return;
            else
            {
                //Pokud již tabulka obsahuje sql kód, zobrazím varování o
případné ztrátě vlastního textu a případně změnu zruším:
                if (!string.IsNullOrEmpty(this.SQL_GENEROVANI_PK))
                {
                    string zprava = string.Format("Opravdu si přejete vygenerovat
automatický sql kód pro založení hodnot primárního klíče?{0}Dojde ke ztrátě
vlastního napsaného sql kódu.", Environment.NewLine);
                    MessageBoxResult mbr = MessageBox.Show(zprava, "Generování
SQL pro hodnoty PK", MessageBoxButton.YesNo, MessageBoxImage.Warning);
                    if (mbr != MessageBoxResult.Yes)
                    {
                        this.SQL_VLASTNI_KOD = true;
                        return;
                    }
                }

                this.SQL_GENEROVANI_PK = GenerovaniSQL.ZalozeniPK(this,
this.BDBCTABULKY);
            }

        }

        //Celý objekt to string
        public override string ToString()
        {
            return this.JMENO;
        }

    }
```

Ukázka kódu 2 - EF vlastní parciální třída

Z Ukázka kódu 1 je také patrné, že vygenerované kolekce objektů jsou typu *ObservableCollection*. Tato vlastnost byla do Entity Frameworku dodána manuální úpravou, protože Entity Framework vytváří kolekce typu *ICollection*. Úprava kódu Entity Frameworku byla provedena podle návodu ze stránek Microsoft Developer Network[3].

Entity Framework generuje kód z modelu použitím T4 šablon. Šablony dodávané s Visual Studiem nebo stažené z Visual Studio Gallery jsou určeny k všeobecnému použití. To znamená, že entity vytvořené z těchto šablon generují obecné vlastnosti *ICollection<T>*. Nicméně pro WPF, kde se využívá data bindingu, je vhodnější pro vlastnosti typu kolekce používat *ObservableCollection*, protože následně WPF může sledovat změny v těchto kolekcích. Za tímto účelem je vhodné modifikovat šablony, aby používali *ObservableCollection* (Entity Framework Databinding with WPF, 2016).

4.4 View

View neboli grafická rozhraní WPF aplikace se vytváří ve značkovacím jazyce XAML. Tyto View mohou mít také svůj code-behind, kde lze přidávat další logiku a reagovat tak na různé události vytvářené prezentace. Pokud se, ale při psaní aplikace držíme návrhového vzoru MVVM, je žádoucí se vyhnout vkládání kódu do code-behindu. Důvodem je nutnost, aby veškeré logika aplikace zůstala v jiných vrstvách aplikace – zejména ve vrstvě ViewMode, které pomocí commandů a bindingu komunikuje s View. Psaní kódu do code-behindu se však nelze vždy zcela vyhnout a nemusí to být vždy úplně proti zásadám MVVM. Do code-behindu je možné přidávat další kód, který se týká grafického rozhraní (například úprava zobrazení vizuálního objektu atp.), nicméně by neměl obsahovat nic z business logiky programu. Jednou z hlavních zásad při vytváření MVVM aplikace, ale přesto zůstává minimalizace vlastního kódu v code-behindu a snaha přenést tento kód buď do XAML kódu nebo do ViewModelu.

Následující ukázka XAML kódu z části pro definici tabulek v hlavním menu v MainWindow.xaml aplikace OUdot ukazuje, jak je menu sestaveno a propojeno

[3] https://msdn.microsoft.com/en-us/data/jj574514.aspx

na vlastnosti MainViewModel.cs, ale také je zde patrný odkaz na událost v code-behindu. Událost bDefiniceTabulky_Click pouze otevře nové okno s detailní definicí tabulky a nemění nebo nějaký způsobem nemanipuluje s objekty modelu. Pouze předává příslušný objekt, aby mohlo být otevřeno nové okno. Ukázka metody v code-behind, která spouští nové okno pro detail tabulky je předvedena v Ukázka kódu 4.

```
<GroupBox DockPanel.Dock="Top" Header="Tabulky">
    <DockPanel>
        <Button Margin="5,3,3,3" Command="{Binding NovaTabCommand}"
Content="Nová"/>
        <Button x:Name="bDefiniceTabulky" Margin="5,3,3,3"
Click="bDefiniceTabulky_Click" Content="Definice" />

        <ComboBox x:Name="cbTabulky"  ItemsSource="{Binding ObdobiMetod.OU_TAB}"
SelectedItem="{Binding Tabulka}" Margin="3" DisplayMemberPath="JMENO"
SelectedIndex="0" IsSynchronizedWithCurrentItem="True" Tag="{Binding
RelativeSource={RelativeSource AncestorType={x:Type Window},
Mode=FindAncestor}}"/>
    </DockPanel>
</GroupBox>
```

Ukázka kódu 3 - Menu Tabulky v MainWindow.xaml

```
public void OtevriTabulkaOkno(OU_TAB OUTab)
{
    //Vyhledání zda okno s tabulkou již existuje
    var vsechnyLD =
this.dmMyDockingManager.Layout.Descendents().OfType<LayoutDocument>();
    LayoutContent OknoVybTab = vsechnyLD.FirstOrDefault(p => (p.Content.GetType()
== typeof(TabulkaDetailsWindow)) && ((p.Content as
TabulkaDetailsWindow).DataContext as TabulkaDetailsViewModel).Tabulka == OUTab);

    if (OknoVybTab == null)//Vytvářím nové okno
    {
       //Vytvořím a nastavím okno pro detail tabulky
       TabulkaDetailsWindow tdw = new TabulkaDetailsWindow(OUTab);
       //Vložím vytvořené okno jako LeyoutDocument a s aktivním ho
       LayoutDocument ld = new LayoutDocument { Content = tdw };

       //Nabinduji titul okna s názvem tabulky
       Binding titulBinding = new Binding();
       titulBinding.Source = tdw.DataContext;
       titulBinding.Path = new PropertyPath("Tabulka.JMENO");
       BindingOperations.SetBinding(ld, LayoutDocument.TitleProperty,
titulBinding);

       //Nabinduji IsBusy property:
       Binding isBusy = new Binding("IsBusy");
       isBusy.Source = this.DataContext;
       isBusy.Converter = new RozsireniGrafika.InverseBooleanConverter();
       BindingOperations.SetBinding(tdw, TabulkaDetailsWindow.IsEnabledProperty,
isBusy);
```

```
        //Vyhledání LayoutDocumentPane
        LayoutDocumentPane LDP =
        this.dmMyDockingManager.Layout.Descendents().OfType<LayoutDocumentPane>().
        FirstOrDefault();
        LDP.Children.Add(ld);
        ld.IsActive = true;
    }
    else//aktivuji již existující
    {
OknoVybTab.IsActive = true;
    }
}
```

Ukázka kódu 4 - Metoda otevření okna v code-behind MainWindow.xaml

4.4.1 Extended WPF Toolkit

Veškerá grafická rozhraní (view) vytvořená v rámci aplikace OUdot jsou designována pouze pomocí XAML kódu ve Visual Studiu 2015 a bez použití doplňkového programu Blend, který je vhodný pro tvorbu rozhraní s vyššími požadavky na grafický design. Ve vytvořených view je velmi často využíván Extended WPF Toolkit, o kterém je pojednáno v teoretických východiscích – kapitole 3.2. Pomocí tohoto doplňku bylo možné do aplikace vložit nezbytné ovládací prvky bez nutnosti je vytvářet samostatně. V programu jsou využity pouze ovládací prvky spadající pod otevřenou licenci (Microsoft Public Licence). Při vytváření bylo také uvažováno, zda nezakoupit i nástroje a prvky spadající pod placenou licenci Plus Edition, ale při objektivní posouzení nároků na grafické rozhraní bylo usouzeno, že otevřená licence je dostačující.

Přidání Extended WPF Toolkit do projektu proběhlo pomocí správce balíčků NuGet, který opět stáhl a přidal potřebné odkazy na knihovny. Ovládací prvky a různé další nástroje Extended WPF Toolkit je možné využívat v XAML kódu poté, co je přidán potřebný jmenný prostor v definici grafického prvku, tak jak je zvýrazněno v následující Ukázka kódu 5.

```
<Window x:Class="OuDot.ViewModel.PomocnaOkna.KlonovatOUWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"
        Title="Nastavení klonování OU" Height="174" Width="300">
```

Ukázka kódu 5 - Definice jmenného postoru Extended WPF Toolkit

V předchozí Ukázka kódu 5 z aplikace OUdot je jmenný prostor Extended WPF Toolkit definován pod zkratkou „xctk“. Tato zkratka může být libovolná, avšak ve vytvářeném programu OUdot se vždy používá pro Extended WPF Toolkit. Následně je tato zkratka využívána v kódu při definici ovládacích prvků nebo dalších funkcí z Extended WPF Toolkit, tak jak je zobrazeno v Ukázka kódu 6.

```
<xctk:IntegerUpDown x:Name="iudPocetNovychOU" Margin="5" Grid.Column="1"
Minimum="1" DefaultValue="1" Maximum="99" Value="1"></xctk:IntegerUpDown>
```

Ukázka kódu 6 - Použití Extended WPF Toolkit v XAML

4.4.2 Konvertory

Data, která se mají zobrazovat uživateli, se propojují do grafického rozhraní (view) pomocí bindigu. To je možné vidět například v Ukázka kódu z vytvářené aplikace. Zde je vidět binding u ovládací prvku ComboBox:

1. U jeho vlastnosti ItemSource, kam je bindována kolekce objektů OU_TAB, ze kterých uživatel vybírá.
2. U vlastnosti SelectedItem, kam je bindován vlastnost Tabulka, do které se následně promítá vybraný objekt uživatelem v ComboBoxu.

Dalším hojně využívaným prvkem při tvorbě MVVM aplikace bývají konvertory. Ty také bylo velmi výhodné využívat při vývoji aplikace OUdot. Konvertory se používají zejména při bindigu, kdy je zapotřebí přizpůsobit například vybraný objekt danému zobrazení nebo naopak transformovat vstup od uživatele do podoby vlastnosti, která je bindovaná. Tyto konvertory se vytvářejí jako samostatné třídy s implementací rozhraní IValueConverter nebo IMultiValueConverter a následně se použijí v XAML kódu v definici bindingu.

Konvertory použité v aplikaci OUdot byly vytvořeny ve jmenném prostoru RozsireniGrafika a všechny se nachází v souboru Konvertory.cs. Ukázka kódu 7 je příkladem třídy sloužící ke konverzi jak zobrazovaných dat, tak dat vložených uživatelem aplikace. Po implementaci rozhraní IValueConverter, jsou k dispozici dvě funkce. Convert, která podle dodané logiky konvertuje objekt směrem do grafického rozhraní a ConvertBack, která naopak podle přidané logiky konvertuje data směrem k bindované

vlastnosti. Dále, jak lze také vidět v Ukázka kódu 7, je možné přidat ValueConversion, který definuje datové typy bindovaných vlastností, na které lze konvertor využít.

```
/// <summary>
/// Konvertuje enumerátor Spoj na hodnotu a naopak
/// </summary>
[ValueConversion(typeof(SpojEnum), typeof(int))]
public class SpojEnumToStringConverter : IValueConverter
{
    #region IValueConverter Members

    public object Convert(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {

        if (value == null) return null;

        return (SpojEnum)Enum.Parse(typeof(SpojEnum), value.ToString(), true);

    }

    public object ConvertBack(object value, Type targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {
        if (value == null) return null;

        return value.ToString();
    }

    #endregion
}
```

Ukázka kódu 7 - Použití IValueConverter

Použití vytvořeného konvertoru z Ukázka kódu 7 v XAMLu je obsahem následující Ukázka kódu 8, kde:

- Na první řádce je definice jmenného prostoru RozsireniGrafika, který obsahuje vytvořený konvertor.
- Další řádka kódu ukazuje definici konvertoru ve zdrojích (Resources) vytvářeného view.
- Zbylé řádky kódu demonstrují použití konvertoru při bindingu.

```
xmlns:rg="clr-namespace:OuDot.RozsireniGrafika"
..
<rg:SpojEnumToStringConverter x:Key="SpojEnumToStringConverter"/>
..
SelectedItem="{Binding OdvozUkaz.SPOJ, Converter={StaticResource
SpojEnumToStringConverter}, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged,
ValidatesOnDataErrors=True}"
```

Ukázka kódu 8 - Použití konvertoru v XAML kódu

4.4.3 Validace

Validaci objektů lze v aplikaci, která je stavěná podle návrhového vzoru MVVM, provádět různými způsoby. Existuje několik různých druhů rozhraní, které se implementují u objektů, které mají být validovány. V případě aplikace OUdot bylo k validaci objektů využito rozhraní IDataErrorInfo, které se doporučuje v případě používání data bindingu, protože jeho implementace je následně velmi snadná. Také je tento interface vhodný z důvodů okamžitého informování grafického rozhraní, pokud například nějaká vlastnost validovaného objektu obsahuje nepřípustné hodnoty.

V následujících ukázkách kódu je zobrazen postup implementace rozhraní ve vytvořené aplikaci. Nejdříve je implementováno rozhraní ke třídě, která se validuje, to je zobrazeno v Ukázka kódu 9. V případě OUdot aplikace bylo potřeba toto rozhraní implementovat na třídy z vrstvy Model, které byli vygenerované Entity Frameworkem, aby následně při ukládání dat do databáze byla již ověřena správnost těchto dat. Proto, jak bylo zmíněno v kapitole 4.3.1.1, jsou validace umístěny na zvláštním místě v projektu u parciálních tříd. Implementace rozhraní IDataErrorInfo vytvoří dvě veřejné vlastnosti, Error a tzv. Indexer (určený názvem vlastnosti). Obě tyto vlastnosti jsou read-only. Stěžejní vlastností IDataErrorInfo je zmíněný Indexer, zde se doplňuje veškerá logika sloužící ke kontrole správnosti jednotlivých vlastností objektu. Řetězec znaků, který vlastnost vrací, je pak chybovým hlášením. V případě, že je tento řetězec prázdný, je celý objekt validní neboli neobsahuje žádné validační chyby.

```
public partial class OU_CATEGORY : IDataErrorInfo
{
    #region IDataErrorInfo Members

    string IDataErrorInfo.Error
    {
        get { throw new NotImplementedException(); }
    }

    string IDataErrorInfo.this[string columnName]
    {
        get
        {
            if (this.OU_ODVOZ_UKAZ == null) return null;//
            List<ValidacniChyba> chyby = new List<ValidacniChyba>();
            switch (columnName)
            {

                case "HODNOTA":

                    if (this.OU_ODVOZ_UKAZ.OU_CATEGORY.Any(c => (c.HODNOTA ==
this.HODNOTA) && (c != this)))
                    {
                        chyby.Add(new ValidacniChyba("Hodnota musí být
jedinečná.", "OU_CATEGORY", this, columnName, this.OU_ODVOZ_UKAZ));
                    }

                    break;
                case "POPIS":
                    if (string.IsNullOrEmpty(this.POPIS))
                    {
                        chyby.Add(new ValidacniChyba("Pole nesmí být prázdné.",
"OU_CATEGORY", this, columnName, this.OU_ODVOZ_UKAZ));
                    }
                    break;

                default:
                    break;
            }

            var chybyOld = Validace.ValidacniChyby.Where(v => (v.Objekt == this)
&& (v.ChybnaProperty == columnName)).ToList();
            Validace.ValidacniChyby.RemoveRange(chybyOld);
            Validace.ValidacniChyby.AddRange(chyby);
            if (chyby.Count > 0)
            {
                return chyby[0].Chyba;
            }
            else
                return null;
        }
    }

    #endregion
}
```

Ukázka kódu 9 - Implementace IDataErrorInfo

Nyní přejděme k nastavení validací v XAML kódu. V následující Ukázka kódu 10 je, v návaznosti na předchozí Ukázka kódu 9, nastavení bindingu pro vlastnost HODNOTA z třídy OU_CATEGORY. Ke spuštění validace, je nejprve nutné nastavit UpdateSourceTrigger, v ukázce je nastaven na hodnotu PropertyChanged, to spouští validace při každé jednotlivé změně vlastnosti. Také musí být nastavena vlastnost bindingu ValidatesOnDataErrors na hodnotu True.

```
<DataGridTextColumn x:Name="hODNOTAColumn" Binding="{Binding
HODNOTA,UpdateSourceTrigger=PropertyChanged,ValidatesOnDataErrors=True}"
Header="HODNOTA">
    <DataGridTextColumn.EditingElementStyle>
        <Style TargetType="{x:Type TextBox}">
            <Setter Property="MaxLength" Value="3"/>
        </Style>
    </DataGridTextColumn.EditingElementStyle>

</DataGridTextColumn>
```

Ukázka kódu 10 - XAML (ValidatesOnDataErrors) IDataErrorInfo

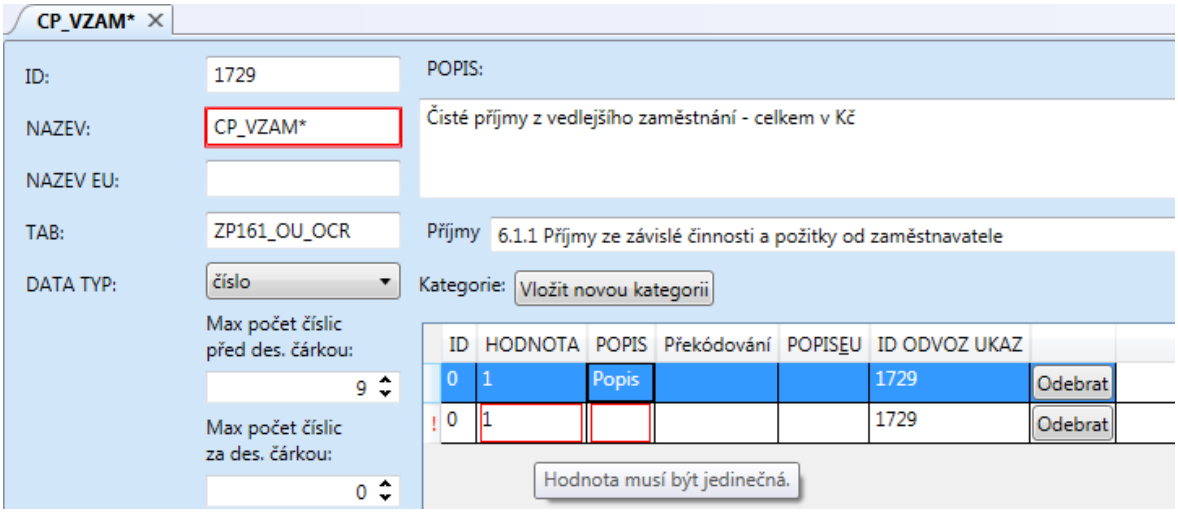
Dále je také vhodné nastavit, jakým způsobem se má zobrazit chyba na ovládacím prvku pokud vlastnost, která je na tento prvek vázána, neprošla validační kontrolou.

```
<!--ToolTip s Errorem pokud je při validaci chyba-->
<Style TargetType="Control" x:Key="myErrorTemplate">
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip"
Value="{Binding RelativeSource={RelativeSource Self},
Path=(Validation.Errors)[0].ErrorContent}"/>
            <!--Nastavení barvy ohraničení na červenou-->
        </Trigger>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="BorderBrush"  Value="Red" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Ukázka kódu 11 - XAML (myErrorTemplate) IDataErrorInfo

Ukázka kódu 11 je příkladem z vytvářené aplikace pro definici stylu ovládacího prvku, pokud obsahuje validační chybu. Tento kód byl definován ve zdrojích (Resources) vytvořeného view. Takto vytvořená šablona se následně automaticky aplikuje na všechny druhy ovládacích prvků. Jsou zde vidět definice dvou různých Trriggerů, které nastavují vlastnosti prvku ToolTip a BorderBrush, v případě, že Validation.HasError vrátí hodnotu

True. Následně je vlastnost ToolTip navázána na chybovou hlášku a vlastnost BorderBrush (barevné ohraničení prvku) nastavena na hodnotu Red.



Obrázek 9 - Validace v UI

Na Obrázek 9 - Validace v UIjsou vidět ovládací prvky, které neprošli validační kontrolou. Podle nastaveného stylu v Ukázka kódu 11 se tyto prvky zobrazují s červeným rámečkem a také je možné vidět zobrazený ToolTip s informací o chybě.

4.5 MVVM

V předchozích kapitolách jsou ukázány praktické příklady a způsoby při tvorbě vrstev Model a View podle návrhového vzoru MVVM. Nyní se přesuneme ke stěžejnímu bodu tohoto návrhového vzoru, tedy k ViewModelu, který tyto dvě vrstvy propojuje a obsahuje potřebnou business logiku. Nástroj MVVM Light, který je dostupný pod otevřenou licencí MIT je velice vhodný způsob, jak zavést do projektu veškeré podpůrné i nezbytné součásti nutné při tvorbě MVVM aplikace. Tento nástroj byl také využit při tvorbě aplikace OUdot.

4.5.1 **MVVM Light**

MVVM Light Toolkit[4] je sada komponent, které pomáhají programátorům začít s Model - View - ViewModel v Silverlight, WPF, Xamarin, .NET Core a Windows Phone. Jedná se o lehký a praktický framework, který obsahuje pouze základní potřebné komponenty. Hlavním účelem této sady nástrojů je urychlit vznik a vývoj MVVM aplikací. Stejně jako ostatní implementace MVVM i sada nástrojů pomáhá uživatelům oddělit View od Modelu, který tvoří aplikaci. Tyto aplikace jsou následně přehlednější, snadněji se udržují a lze je snadno rozšířit. Umožňuje také lépe a snadněji testovat aplikace a zároveň mít mnohem tenčí vrstvu uživatelského rozhraní, která je běžně obtížnější pro automatické testování (MVVM-Light Info, 2017).

MVVM Light Toolkit poskytuje sadu knihoven, které se starají o propojení a nastavení struktury projektu a poskytuje další pomocné třídy, aby psaní MVVM aplikací bylo jednodušší. Je-li MVVM Light přidán do projektu pomocí nástroje Visual Studia, Nuget, většina práce nutná k vytvoření struktury MVVM je udělána automaticky. Nuget vytvoří kostru ViewModelu a ViewModeLocatoru. ViewModeLocator není součástí MVVM návrhového vzoru samotného, ale je používán MVVM Light Toolkitem. ViewModeLocator pomáhá provázat XAML prvky s jejich ViewModely a také spravovat závislosti mezi vytvořenými ViewModely (How to use MVVM Light Toolkit for Windows Phone, 2013).

Následující popis hlavních pomocných tříd a volitelných tříd z knihovny GalaSoft.MvvmLight.Extras MVVM Light Toolkitu byl zpracován podle zdroje: (How to use MVVM Light Toolkit for Windows Phone, 2013).

Hlavní pomocné třídy:

- Třída ViewModelBase má být použita jako základní třída, ze které ostatní ViewModely dědí.

[4] https://mvvmlight.codeplex.com/

- Třída Messenger (a další typy Message) umožňují komunikaci v rámci aplikace a to zejména mezi ViewModely.
- Třídy RelayCommand slouží ke zjednodušení předávání commandů (příkazů) z View do ViewModelu.

Volitelné třídy:

- EventToCommand behaviour umožňuje vývojářům bindovat jakoukoli událost jakéhokoli ovládacího prvku uživatelského rozhraní na rozhraní ICommand.
- Třída DispatcherHelper je odlehčená třída, která pomáhá s tvorbou více-vláknových aplikací.

Do projektu vytvářeného pro Český statistický úřad byl MVVM Light Toolkit nainstalován pomocí nástroje Nuget. Tím byli automaticky vytvořena kostra MVVM aplikace a do projektu přidány veškeré pomocné knihovny, které usnadňují vývoj takovéto aplikace. V projektu nástroj automaticky vytvoří třídy: ViewModelBase, ViewModelLocator a vzorovou třídu pro ViewModel nazvanou MainViewModel a tak teoreticky předdefinovanou pro MainWindow aplikace. Veškeré tyto třídy byly v rámci projektu umístěny do jmenného prostoru ViewModel, který zároveň obsahuje také všechna vytvořená okna (view) aplikace. Pro návrhový vzor MVVM není nutné v projektu separovat View a ViewModely, ačkoli je to zvykem. Výhodou ponechání View a VieModelu je větší přehlednost a jednoduší práce při bindingu.

4.5.2 **ViewModelBase**

Základní třída ViewModelBase, ze které ostatní ViewModely dědí, je ve většině případů pouze základna pro implementaci rozhraní INotifyPropertyChanged a tím vytvoření metody OnPropertyChanged (tento název je použit v programu OUdotu, často pro tuto metodu bývají používány také názvy OnNotificationChanged nebo RaisePropertyChange). Tato metoda je následně používána ve všech ViewModelech a slouží k tomu, že informuje view o tom, že došlo ke změně dané vlastnosti ve ViewModelu.

```
public class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string propertyName =
null)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Ukázka kódu 12 - ViewModelBase

Ukázka kódu 12 je definici třídy ViewModelBase ve vytvářené aplikaci OUdot. Tato třída by měla implementovat logiku, která je pro všechny ViewModely společná. Typický společným prvkem je rozhraní INotifyPropertyChanged, které je při tvorbě MVVM aplikace naprosto nezbytné. Zajišťuje notifikaci view při změně vlastnosti ve ViewModelu.

Při implementaci rozhraní INotifyPropertyChanged není nutné již dodávat žádnou logiku, pokud je implementace provedena automatickým generováním kódu při definici rozhraní, kterou nabízí Visual Studio samo. Atribut [CallerMemberName] není povinný, ale umožňuje využívat metodu OnPropertyChanged bez parametru, pokud je použita v definici vlastnosti, jak je možné vidět v následující Ukázka kódu 13. Není pak následně nutné psát OnPropertyChanged ("názevVlastnosti").

```
private ULOHA_OBDOBI _obdobiTeren;

public ULOHA_OBDOBI ObdobiTeren
{
    get { return _obdobiTeren; }
    set
    {
        if (_obdobiTeren != value)
        {
            _obdobiTeren = value;
            OnPropertyChanged();
        }
    }
}
```

Ukázka kódu 13 - OnPropertyChanged použití

Ukázka kódu 13 je typickým použitím metody OnPropertyChanged v definici vlastnosti ViewModelu. Zde je metoda přidána do setu vlastnosti a tím při její změně

spouští notifikaci. V ukázce je také možno vidět, že metoda i změna hodnoty privátní proměnné se děje pouze tehdy, pokud nová hodnota je rozdílná od předchozího stavu. To zajišťuje podmínka if (_obdobiTeren != value). Tímto jsou odfiltrovány zbytečné notifikace, ke kterým by mohlo dojít, ale hodnota vlastnosti by se nezměnila.

Metodu OnPropertyChanged je také možné použít mimo definici vlastnosti – to je možné vidět na příkladu v Ukázka kódu 14. Zde je metoda ZmenaGenerace() ze třídy OdvozUkazDetailsViewModel. Tato metoda reaguje na změnu generace odvozeného ukazatele a automaticky doplňuje hodnotu vlastnosti PORADI_VYPOCTU o jedna vyšší než je maximální hodnota v generaci. Pro promítnutí změny do uživatelského rozhraní, je nutné spustit metodu OnPropertyChanged("OdvozUkaz"), tentokrát s příslušným názvem vlastnosti, ve které došlo ke změně.

```
        /// <summary>
        /// Na změnu generace vyhledá nejvyšší hodnotu pořadí v dané generaci a
vloží tuto hodnotu + 1 do pořadí výpočtu
        /// </summary>
        internal void ZmenaGenerace()
        {
            if (OdvozUkaz.GENERACE != null)
            {
                var ostOUzeStejneGenerace =
OdvozUkaz.OU_TAB.OU_ODVOZ_UKAZ.Where(o => o.GENERACE == OdvozUkaz.GENERACE && o
!= OdvozUkaz);
                int x = 0;
                if (ostOUzeStejneGenerace.Count() > 0)
                {
                    x = ostOUzeStejneGenerace.Max(o => o.PORADI_VYPOCTU);
                }
                OdvozUkaz.PORADI_VYPOCTU = x + 1;
                OnPropertyChanged("OdvozUkaz");
            }
        }
```

Ukázka kódu 14 - OnPropertyChanged použití 2

Jedna z výhod Visual Studia je možnost vytvoření vlastního snippetu, což je uložená šablona opakujícího se kódu, která se pod zkratkou dá vložit do kódu a případně vhodně upravit. Toho bylo také využito při vývoji aplikace OUdot. Obsahem Ukázka kódu 15, je definice snippetu pro generování vlastnosti s metodou OnPropertyChanged, tak jak je zobrazeno v Ukázka kódu 13.

```
<Declarations>
   <Literal>
      <ID>type</ID>
      <ToolTip>Property type</ToolTip>
      <Default>int</Default>
   </Literal>
   <Literal>
      <ID>property</ID>
      <ToolTip>Property name</ToolTip>
      <Default>Value</Default>
   </Literal>
   <Literal>
      <ID>defaultValue</ID>
      <ToolTip>The default value for this property.</ToolTip>
      <Default>intValue</Default>
   </Literal>
</Declarations>
<Code Language="csharp">
    <![CDATA[
          private $type$ $defaultValue$;
          public $type$ $property$
          {
              get { return $defaultValue$; }
              set
                  {
                    if ($defaultValue$ != value)
                    {
                           $defaultValue$ = value;
                           OnPropertyChanged();
                    }
                  }
          }
   $end$]]>
</Code>
```

Ukázka kódu 15 - Snippet MVVM vlastnost

Ukázka kódu 15 obsahuje ve své horní části deklaraci proměnných, které je nutné doplnit při použití snippetu a spodní část obsahuje C# kód, který je vygenerován s odkazy na proměnné z horní části deklarace.

4.5.3 ViewModelLocator

Při použití návrhového vzoru MVVM je běžnou praxí, že View vyhledávají své ViewModely v dependency injection (DI) containeru. DI container vkládá instanci ViewModelu do View voláním konstruktoru View, které přijímá ViewModel jakož to parametr; toto schéma se nazývá inversion of control (IoC). MVVM Light Toolkit po instalaci pomocí Nugetu automaticky předpřipraví třídu ViewModelLocator k použití s tímto schématem, zde se objevuje třída SimpleIoc, která je také součástí nástroje a slouží k registraci používaných ViewModelů.

ViewModelLocator je nástroj, který umožňuje zachovat výhody DI containaru v MVVM aplikaci a zároveň upravovat vizuální design aplikace (s možností vidět v něm promítnuty bindované vlastnosti z VeiwModelu). To je někdy nazýváno „blendabilitou“ aplikace (s odkazem na Expression Blend). ViewModelLocator také detekuje, zda se aplikace nachází v design módu – pokud se nenachází v design módu, vrací příslušný ViewModel z DI containeru. V případě, že se nachází v design módu, ViewModelLocator vrací fiktivní ViewModel, který je obvykle vyplněn testovacími daty.

```
/// <summary>
/// This class contains static references to all the view models in the
/// application and provides an entry point for the bindings.
/// </summary>
public class ViewModelLocator
{
    /// <summary>
    /// Initializes a new instance of the ViewModelLocator class.
    /// </summary>
    public ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

        ////if (ViewModelBase.IsInDesignModeStatic)
        ////{
        ////    // Create design time view services and models
        ////    SimpleIoc.Default.Register<IDataService, DesignDataService>();
        ////}
        ////else
        ////{
        ////    // Create run time view services and models
        ////    SimpleIoc.Default.Register<IDataService, DataService>();
        ////}

        SimpleIoc.Default.Register<MainViewModel>();
    }

    public MainViewModel Main
    {
        get
        {
            return ServiceLocator.Current.GetInstance<MainViewModel>();
        }
    }

    public static void Cleanup()
    {
        // TODO Clear the ViewModels
    }
}
```

Ukázka kódu 16 - ViewModelLocator

Ukázka kódu 16 je příkladem ViewModelLocatoru vygenerovaného nástrojem MVVM Light. V případě aplikace OUdot je zatím využit pouze pro registraci MainViewModelu. To umožňuje v podstatě vytvoření MainViewModelu jako unikátu a jeho snazší použití v jiných částech aplikace. Ostatní používané ViewModely v aplikaci zde nejsou registrovány, protože se používají ve více instancích a také jejich konstruktor obsahuje parametr. Jejich registrace ve třídě ViewModelLocator by byla komplikovaná a pro vytvářené řešení nebyla nutná.

```
<Application x:Class="OuDot.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="ViewModel/MainWindow.xaml"
             Startup="Application_Startup"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             d1p1:Ignorable="d"
             xmlns:d1p1="http://schemas.openxmlformats.org/markup-
compatibility/2006">
  <Application.Resources>
    <ResourceDictionary>
      <vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" xmlns:vm="clr-
namespace:OuDot.ViewModel" />
        </ResourceDictionary>
  </Application.Resources>
</Application>
```

Ukázka kódu 17 - ViewModelLocator v App.xaml

Nástroj MVVM Light, také automaticky přidává třídu ViewModelLocator do zdrojů aplikace, která se nachází v souboru App.xaml, jak je možné vidět v Ukázka kódu 17. Díky tomu lze pod vygenerovaným klíčem „Locator“ bindovat ve view příslušný ViewModel registrovaný ve ViewModelLocatoru.

Ukázka kódu 18 obsahuje definici hlavního okna MainWindow.xaml – zde je možné vidět jakým způsob je bindován vlastnost Main z ViewModelLocatoru na vlastnost DataContext MainWindow. Tato vlastnost Main vrací unikátní instanci zaregistrované třídy MainViewModel, jak je patrné z ukázky kódu Ukázka kódu 16. Tímto způsobem bylo v aplikaci OUdot propojeno hlavní okno aplikace se svým ViewModelem.

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:OuDot.ViewModel"
        xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"
     xmlns:ItemsFilter="http://schemas.bolapansoft.com/xaml/Controls/ItemsFilter"
        xmlns:xcad="http://schemas.xceed.com/wpf/xaml/avalondock"
        xmlns:m="clr-namespace:OuDot.Model"
        xmlns:complex="clr-
namespace:DaisleyHarrison.WPF.ComplexDataTemplates;assembly=ComplexDataTemplates"
        xmlns:rg="clr-namespace:OuDot.RozsireniGrafika"
        xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
        xmlns:cmd="http://www.galasoft.ch/mvvmlight"
        xmlns:d=http://schemas.microsoft.com/expression/blend/2008
        xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
mc:Ignorable="d" x:Class="OuDot.ViewModel.MainWindow"
        Title="OuDot" Height="356" Width="738"
        DataContext="{Binding Main, Source={StaticResource Locator}}">
```

Ukázka kódu 18 - MainWindow DataContext binding

4.5.4 ViewModel

ViewModel obsahuje většinu aplikační logiky, která seskupuje datové třídy a promítá data do uživatelského rozhraní. ViewModel se na uživatelské rozhraní naváže tak, že se instance ViewModelu ve View přiřadí do vlastnosti DataContext. Jakým způsobem se bindují jednotlivé vlastnosti ViewModelu a provádí notifikace o jejich změně, je popsáno v kapitolách 4.4.2 a 4.5.2. Tato kapitola je zaměřena na příklad ViewModelu obsahujícího parametr a jeho způsob propojení se svým view. Také je zde ukázka vytvoření a použití commandu.

V případě, že přijde požadavek na otevření nového okna pro detail tabulky v aplikaci OUdot, jak je zobrazeno v Ukázka kódu 4, je nejdříve vytvořena nová instance okna pro detail tabulky. Konstruktor, který je spuštěn při vytvoření instance je obsahem ukázky kódu Ukázka kódu 19, v této ukázce je také vidět, jakým způsobem je v code-behind přiřazen ViewModel do DataContextu okna a tím vytvořeno propojení mezi View a jeho ViewModelem. Zároveň je zde také předáván parametr s instancí třídy z modelu OU_TAB, která bude obsahem otvíraného okna.

Zároveň s přiřazením do DataContextu v ukázce kóduUkázka kódu 19, je vytvořena nová instance třídy TabulkaDetailsViewModel. Tento příkaz tedy také spustí konstruktor, který je zobrazen v následující Ukázka kódu 20.

```
public TabulkaDetailsWindow(OU_TAB VybranaTabulka)
{
    InitializeComponent();
    this.DataContext = new TabulkaDetailsViewModel(VybranaTabulka);
}
```

Ukázka kódu 19 - TabulkaDetailsWindow konstruktor

```
public RelayCommand NastavitPKCommand { get; private set; }
public RelayCommand SmazatPKCommand { get; private set; }
public RelayCommand GenerujSQLCommand { get; private set; }
public RelayCommand SmazatTabulkuCommand { get; private set; }

public TabulkaDetailsViewModel(OU_TAB VybranaTabulka)
{
    if (!DesignerProperties.GetIsInDesignMode(new
System.Windows.DependencyObject()))
    {
        // Code runs "for real"
        Tabulka = VybranaTabulka;

        //Pokud obsahuje OU nepovilej upravy
        Tabulka.OU_ODVOZ_UKAZ.CollectionChanged +=
OU_ODVOZ_UKAZ_CollectionChanged;
        PovolitUpravy = (Tabulka.OU_ODVOZ_UKAZ == null ||
Tabulka.OU_ODVOZ_UKAZ.Count == 0);

        ObdobiTeren =
DBContext.Instance.Entity.ULOHA_OBDOBI.Find(Settings.Default.IdTerenUlohaObd);
        TabTeren = Tabulka.BDBCTABULKY;

        //seřazení primárních klíčů podle pořadí
        Tabulka.OU_PRIMARNI_KLICE = new
ObservableCollection<OU_PRIMARNI_KLICE>(Tabulka.OU_PRIMARNI_KLICE.OrderBy(p =>
p.PORADI));

        PrirazeneOU = new
System.Windows.Data.ListCollectionView(Tabulka.OU_ODVOZ_UKAZ);
    }
    else
    {
        // Code runs in Blend --> create design time data.
    }

    NastavitPKCommand = new RelayCommand(() => NastavitPK());
    SmazatPKCommand = new RelayCommand(() => SmazatPK());
    GenerujSQLCommand = new RelayCommand(() => GenerujSQL());
    SmazatTabulkuCommand = new RelayCommand(() => SmazatTabulku());
}
```

Ukázka kódu 20 - ViewModel konstruktor + RelayCommand

Po tom co proběhne i kód konstruktoru v Ukázka kódu 20 je dokončeno provázání mezi View a ViewModelem a okno detailu tabulky je zobrazeno. Tímto jednoduchým

způsobem bez použití ViewModelLocatoru jsou v aplikaci OUdot propojeny ViewModely vyžadující k vytvoření instance parametr.

Konstruktor v Ukázka kódu 20 obsahuje podmínku if(!DesignerProperties.GetIsInDesignMode(new System.Windows.DependencyObject())), ta umožňuje plnit vlastnosti, které jsou bindovány reálnými daty sloužící k běhu aplikace nebo testovacími daty sloužícími při návrhu grafického designu.

```
<Button Command="{Binding NastavitPKCommand}" Margin="2">Nastavit</Button>
```

Ukázka kódu 21 - Button command

Způsob převedení událostí vyvolaných uživatelem aplikace, tedy v grafickém rozhraní, do příslušného ViewModelu je v MVVM aplikacích prováděn pomocí commandů. Jednoduchým příkladem z vytvořené aplikace je v Ukázka kódu 21. Ve WPF jsou již hlavní události ovládacích prvků uzpůsobeny pro MVVM a tak například tlačítko obsahuje vlastnost Command. Když se na ni naváže odpovídající vlastnost z ViewModelu, může tato vlastnost spouštět metodu či kód na ní v konstruktoru ViewModelu navázaný.

Ukázka kódu 20 s příkladem ViewModelu obsahuje také speciální vlastnosti typu RelayCommand (do projektu dodán nástrojem MVVM Light). Ty slouží k spravování událostí z uživatelského rozhraní. Událost na stisknutí tlačítka z Ukázka kódu 21 je tedy v Ukázka kódu 20 v konstruktoru ViewModelu propojena přes RelayCommand na metodu NastavitPK() (Ukázka kódu 22). Metoda je následně spuštěna pokaždé, když je stisknuto tlačítko „Nastavit“ v uživatelském rozhraní.

```
internal void NastavitPK()
{
    try
    {
        if (Tabulka.OU_PRIMARNI_KLICE.Any())
        {
            throw new Exception("Tabulka již primární klíče obsahuje.");
        }
        if (Tabulka.OU_ODVOZ_UKAZ.Any())
        {
            throw new Exception("Tabulka obsahuje odvozené ukazatele.");
        }
        if (TabTeren == null)
        {
            throw new Exception("Nebyla vybrána tabulka.");
        }
        //Seřadit podel pořadí
        var serazenePK = TabTeren.BDBC_PRIMARNI_KLICE.OrderBy(p=>p.PORADI);
        Tabulka.BDBCTABULKY = TabTeren;
        foreach (BDBC_PRIMARNI_KLICE bdbcPK in serazenePK)
        {

            OU_PRIMARNI_KLICE newPK = new OU_PRIMARNI_KLICE();
            newPK.VlozNovyPK(bdbcPK.JMENO, bdbcPK.PORADI, Tabulka);
        }
        GenerujSQL();
    }
    catch (Exception ex)
    {
        string zprava = string.Format("Chyba nastavení primárních klíčů:{0}{1}",
Environment.NewLine, ex.Message);
        MessageBox.Show(zprava, "Nastavení primárního klíče",
MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

Ukázka kódu 22 - Metoda NastavitPK

Pokud metoda, která obsluhuje command z uživatelského rozhraní vyžaduje parametr, lze tento úkol provést jednoduchým přidáním typu k vlastnosti RelayCommand a samozřejmě také bindováním tohoto parametru v XAML kódu. Tento postup je možné vidět na příkladu z aplikace OUdot v Ukázka kódu 23. CommandParameter v ukázce obsahuje pouze klíčové slovo Binding, ale pokud v XAML kódu není specifikováno více, znamená to, že je bindován na stávající datový zdroj.

```
<Button x:Name="bOdeberCat" Content="Odebrat" Command="{Binding
DataContext.OdeberKategoriiCommand, ElementName=oU_CATEGORYDataGrid}"
CommandParameter="{Binding}" IsEnabled="{Binding DataContext.PovolitZmenu,
ElementName=oU_ZDROJEDataGrid}" Click="bOdeberCat_Click" />
..
public RelayCommand<object> OdeberKategoriiCommand { get; private set; }
..
OdeberKategoriiCommand = new RelayCommand<object>((k) => OdeberKategorii(k));
```

Ukázka kódu 23 - Command s parametrem

4.6 Nasazení

Vytvořená aplikace OUdot je nasazována do plného provozu postupnými kroky. Implementace všech jejích základních částí je hotová. V první fázi se aplikace nasadí na vytvoření mikro dat pro úlohu SILC (Výběrové šetření příjmů a životních podmínek domácností) a také pro úlohu VŠIT (Výběrové šetření informačních technologií). V této fázi lze očekávat věcné připomínky k běhu aplikace, které nebyly specifikovány v prvotním zadání. V průběhu této fáze by mělo docházet k jejich postupnému zapracování. Některé již byly zapracovány v této práci.

Po skončení první fáze by mělo dojít k přezkoumání použitých SQL kódů k tvorbě mikro dat. Pro opakující se kódy by mělo dojít v rámci aplikace k vytvoření dalších funkcí, které automaticky generují SQL kód podle zadaných parametrů.

Vytvoření instalačního balíčku je zajištěno pomocí softwaru NSIS (Nullsoft Scriptable Install System). NSIS je profesionální open source systém pro vytváření instalačních programů Windows. Je to software založený na skriptování a umožňuje vytvářet logiku, která zvládne i ty nejsložitější instalační úkoly. Mnoho zásuvných modulů a skriptů již bylo komunitou vytvořeno a jsou volně k dispozici. Umožňuje:

- vytvářet webové instalátory,
- komunikovat s operačním systémem Windows a dalšími softwarovými komponenty,
- instalovat nebo aktualizovat sdílené komponenty a další.

V případě vytvoření instalačního balíčku pro aplikaci OUdot nebylo zapotřebí využití složitých funkcí. Při distribuci nové verze je vytvořen standardní instalační balíček, který pouze testuje, aby aplikace nebyla spuštěna, pokud je instalována nová verze a také,

aby měl vytvořený instalační balíček shodnou verzi jako vydávaný program. Toho je následně využito při automatických aktualizacích programu. Instalační balíček pak přepisuje všechny potřebné soubory ve složce, kam je instalovaný, podle logiky v Ukázka kódu 24.

```
#--------------------------------------------------------
#   Co se instaluje
#--------------------------------------------------------
Section  "-OuDot"

    SetOverwrite on ; vždy přepisuj

    ;kopirovani souborů
    SetOutPath $INSTDIR
    File "${Zdroj}\ServiceCS2.csdb" ;servisní připojení
    File "${Zdroj}\*.dll"
    File "${Zdroj}\OuDot.exe"
    File "${Zdroj}\OuDot.exe.config"

    SetOutPath "$INSTDIR\help"
    File "${Zdroj}\Help\*.pdf" ;Nápověda
    CreateShortCut "$DESKTOP\OuDot.lnk" "$INSTDIR\OuDot.exe"

    SetOverwrite off

SectionEnd ; "-OuDot"
```

Ukázka kódu 24 - NSIS

4.6.1 Aktualizace

Vytvořený instalační balíček je ukládán do sdílené složky na serveru určené pro oddělení šetření v domácnostech. První instalace musí být provedena pracovníkem IT správy ČSÚ, který zajistí, aby měl nový uživatel nainstalovaný další nezbytný software, jako je Oracle Data Provider for .NET a přístup do centrální databáze Oracle.

Aplikace OUdot pak již při vydání nové verze a jejím umístění na server automaticky při spuštění ověřuje, zda má uživatel nainstalovanou nejnovější verzi a pokud tomu tak není, vyzve jej k instalaci nové verze. Kód, který je spuštěn na pozadí při startu aplikace a zajišťuje proces automatické aktualizace, je zobrazen v následující Ukázka kódu 25.

```
#region Aktualizace
void bgwKontrolaVerzi_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    if (e.Error == null && !String.IsNullOrEmpty((string)e.Result))
        this.Aktualizace((string)e.Result);
}
/// <summary>
///  Spuštění aktualizace programu po vyhledání novější verze
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
void bgwKontrolaVerzi_DoWork(object sender, DoWorkEventArgs e)
{
    //Hledání novější verze
    e.Result = this.NovaVerze(false);
}
/// <summary>
/// Aktualizace programu
/// </summary>
/// <param name="InstalSoubor">Instalační soubor</param>
private void Aktualizace(string InstalSoubor)
{
    //1) Dotaz zda instalovat novou verzi
    if (!String.IsNullOrEmpty(InstalSoubor))
    {
        if (MessageBox.Show("Byla nalezena nová verze programu. Chcete ji
nainstalovat?", "Aktualizace programu", MessageBoxButton.YesNo,
MessageBoxImage.Question) != MessageBoxResult.Yes)
        {
            return;
        }
    }
    else
        return;

    //2) Ukončit program a spustit instalaci
    try
    {
        String InstalNewPath = Path.Combine(@"d:\OuDot",
Path.GetFileName(InstalSoubor));
        File.Copy(InstalSoubor, InstalNewPath, true);
        Process Aktualizace = new Process();
        Aktualizace.StartInfo.FileName = InstalNewPath;
        Aktualizace.StartInfo.Arguments = "/zpozdeni=A";
        Aktualizace.Start();
    }
    catch (Exception ex)
    {
        string zprava = string.Format("Chyba aktualizace:{0}{0}{1}",
Environment.NewLine, ex.Message);
        MessageBox.Show(zprava, "Aktualizace", MessageBoxButton.OK,
MessageBoxImage.Error);
    }
    Application.Current.Shutdown();
}
```

```
/// <summary>
/// Vrátí cestu k instalačnímu souboru, pokud byla nalezena novější verze
/// </summary>
/// <param name="Msg">Zobrazovat chybové zprávy</param>
/// <returns></returns>
private string NovaVerze(bool Msg)
{
    String AdrUlozisteSprTaz = @"\\10.1.2.16\data\DATA\ZPRAC\CAPI\OuDot";
    //1) Zjistit dostupnost úložiště
    this.NetUseUloziste();
    if (!Directory.Exists(AdrUlozisteSprTaz))
    {
        if (Msg == true)
            MessageBox.Show("Nelze se připojit na úložiště.", "Aktualizace
programu", MessageBoxButton.OK, MessageBoxImage.Stop);

        return "";
    }


    //2) Získat číslo verze ze souboru v úložišti
    String[] InstalSoubory = Directory.GetFiles(AdrUlozisteSprTaz,
"OuDot_Setup.exe");
    if (InstalSoubory.Length == 0)
        return "";
    FileVersionInfo myFileVersionInfo =
FileVersionInfo.GetVersionInfo(InstalSoubory.FirstOrDefault());

    System.Version MyVersion =
System.Reflection.Assembly.GetExecutingAssembly().GetName().Version
    //3) Porovnat verze, a pokud je na úložišti novější pak nabídnout instalaci
    if ((myFileVersionInfo.FileBuildPart > MyVersion.Build) ||
(myFileVersionInfo.FileBuildPart == MyVersion.Build &&
myFileVersionInfo.ProductPrivatePart > MyVersion.MinorRevision))
        return InstalSoubory.FirstOrDefault();


    return "";
}
/// <summary>
/// Připojení úložiště
/// </summary>
internal void NetUseUloziste()
{
    ..
}
#endregion Aktualizace
```

Ukázka kódu 25 - Automatická aktualizace

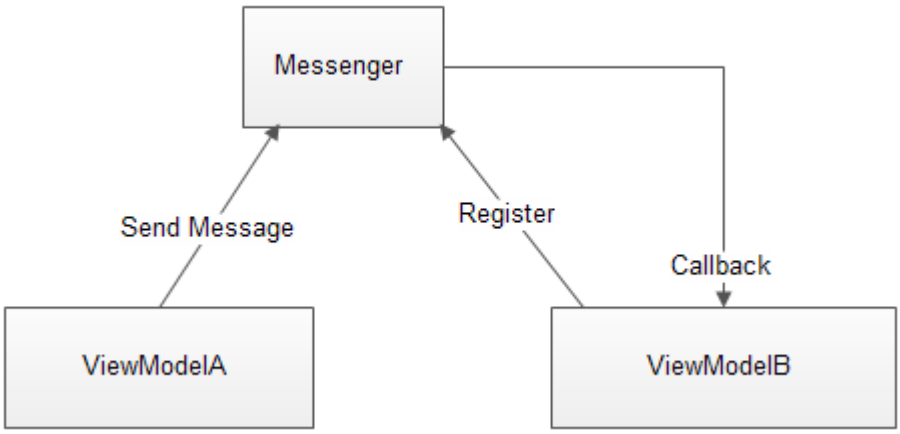
4.7 Další rozvoj

Při vývoji aplikací nebo programů obecně platí, že téměř nikdy nelze označit dílo za kompletně hotové (tedy takové, ve kterém již nejsou třeba provádět další zásahy

a vylepšení). To platí i v případě aplikace OUdot, kde se do budoucna počítá s dalším rozvojem a vylepšeními.

Co se týče návrhového vzoru MVVM použitého v aplikaci jsou v blízké budoucnosti plánována zdokonalení, aby aplikace neobsahovala prvky, kterým by se při využití MVVM mělo vyhnout. Jedním z těchto zdokonalení MVVM v aplikaci bude nahrazení MessageBoxů, sloužících k informování uživatele o chybách, nepovolených operacích či informativních hlášeních. MessageBoxy by měly být nahrazeny systémem zpráv z nástroje MVVM Light.

Třída Messenger, zmíněná v kapitole 4.5.1, umožňuje vyměňování zpráv mezi objekty. Je především určena k zasílání zpráv mezi ViewModely. Každý z takto propojených ViewModelů může komunikovat s ostatními, aniž by mezi sebou potřebovali nějaké vazby (MVVM Light Messenger, 2017).



Obrázek 10 - MVVM Light Messenger (MVVM Light Messenger, 2017).

Pomocí této třídy lze také spouštět notifikace pro uživatele, které jsou zpracovávány ve třídě ViewModeLocator, který následně zajišťuje zobrazení příslušného View s informací uživateli.

Souvisejícím vylepšením MVVM návrhového vzoru ve vytvořené aplikaci bude také zdokonalení systému otevírání nových oken a jejich správa v code-behindu hlavního okna. V rámci zdokonalení bude docházet k přesunutí této logiky z code-behindu do ViewModelu. Zde budou spravovány zobrazované objekty a jejich zobrazování uživateli se přesune do ViewModelLocator, přes který se bude bindovat DataContext zobrazovaného View.

Další vylepšení se týkají spíše rozvoje doplňkových částí aplikace, přičemž úpravy jsou plánovány po nasazení aplikace na všechny úlohy zpracovávané v rámci šetření domácnosti. Také v rámci diskuze s uživateli, po použití aplikace, bude upravováno grafické rozhraní aplikace, aby lépe vyhovovalo jejich potřebám.

5 Výsledky a diskuse

V této kapitole je provedeno zhodnocení jednotlivých částí vytvořené aplikace OUdot a doplňkových nástrojů, které byli při tvorbě použity. Při výběru použitých pomocných vývojových nástrojů se kladl důraz na finanční nenáročnost. Z toho důvodu jsou veškeré tyto nástroje plně pod otevřenými licencemi. Všechny použité technologie a doplňkové nástroje se při vývoji osvědčili a lze je doporučit k tvorbě podobných MVVM aplikací.

Při tvorbě modelové vrstvy aplikace, objektově orientovaného modelu mapovaného na relační databázový model, byla použita technika vygenerování z relační databáze. Tato technika je vhodná, pokud vyvíjená aplikace bude využívat struktury, které jsou již v databázi definovány. Doplnění chybějících struktur vznikající aplikace a následné vygenerování EDM modelu byl snadný a rychlý úkol. Drobné problémy, které se při tvorbě modelu vyskytly, byly zapříčiněny nevhodným nastavení Oracle clienta a nesprávnou konfigurací Oracle Data Provideru for .NET. Pro správnou instalaci vhodného balíku a jeho následnou konfiguraci je potřeba znát detaily o instalované verzi databáze Oracle.

Vrstva uživatelského rozhraní neboli View byla vytvořena pomocí technologie WPF ve značkovacím jazyce XAML. Z důvodů graficky nenáročného rozhraní bylo k vývoji využito pouze Visual Studio, nikoliv specializovaný software Blend, protože grafické rozhraní aplikace sleduje pouze běžnou formulářovou strukturu. Značkovací jazyk XAML je pro vývoj velice vhodný a přehledný. Pomocí šablon a konvertorů bylo dosaženo cíleného designu.

V zájmu snadnějšího následování návrhového vzoru MVVM existují nástroje, které do aplikace pomáhají dodat potřebnou logiku. V případě aplikace OUdot byl vybrán MVVM Light Toolkit, který lze na základě zkušenosti doporučit pro vývoj menších business aplikací. Nástroj automaticky vytvořil prostředí pro přehlednější a účinnější bindování vrstvy ViewModelu se svým View. Také bylo zjištěno, že nástroj MVVM Light pomáhá vytvořit aplikaci, která se lépe testuje. Jednou z těchto součástí je jednoduché rozlišení, zda se aplikace nachází v debug módu nebo release módu. Tím umožňuje vkládat do view testovací data či testovat pouze bussines logiku.

Aplikace OUdot se nyní nachází v první fázi ostrého provozu, kdy je nasazena na zpracování mikro dat pro vybrané úlohy SILC a VŠIT. Již v této fázi byly zapracovány některé požadavky od uživatelů, které se týkali zejména uživatelského rozhraní a snadnější ovladatelnosti aplikace.

Z pohledu MVVM by mělo také dojít k určitým vylepšením či dalšímu rozvoji. Po skončení první fáze by měl být nahrazen systém notifikace uživatele. Ten je nyní založen na informování uživatele pomocí MessageBoxů, ty by měly být nahrazeny systémem Messanger nástroje MVVM Light. V rámci vyčištění code-behindu jednotlivých view by mělo dojít k vylepšení systému provázání ViewModelů, jejichž konstruktor obsahuje parametr a instancují se tak s různým data kontextem.

Poznatky z teoretické části práce byly aplikovány při vývoji aplikace OUdot, na které byly demonstrovány postupy vývoje MVVM aplikace. Tyto vytvořené postupy a použité nástroje lze doporučit pro vývoj menších business aplikací v prostředí .NET a technologie WPF.

6 Závěr

První z dílčích cílů této diplomové práce se podařilo splnit již v teoretické části práce, která byla zaměřena na popis použitých technologií při vývoji aplikace. Nejdříve byla teoretická část zaměřena na popis platformy Microsoft .NET a jejích součástí, které byly použity v praktické části. V dalších částech teoretických východisek byla pozornost věnována popisu technologií spojených s návrhovým vzorem MVVM, jako jsou WPF a Entity Framework. Obsahem této části byl také teoretický popis konstrukce a fungování samotného návrhového vzoru MVVM.

Hlavním cílem práce bylo vytvořit WPF aplikaci v prostředí .NET dle požadavků definovaných Českým statistickým úřadem. Dle zadání má aplikace zpracovávat data z výběrových domácnostních šetření a podle definic zadaných uživatelem vytvářet mikro-datové soubory. Aplikace byla vytvořena podle návrhového vzoru MVVM s využitím objektově-relačního mapování pomocí nástroje Entity Framework.

V praktické části diplomové práce, pak byli popisovány nejdůležitější součásti a řešení problémů při tvorbě MVVM aplikace. Vývoj jednotlivých vrstev MVVM byl popisován na ukázkách kódu z aplikace vytvořené v rámci práce.

Vytvořená aplikace OUdot se nyní nachází v první fázi nasazení do ostrého provozu, aby mohla být později použita na všechna domácnostní šetření. V první fázi byla aplikace nasazena na šetření životních podmínek (SILC) a informačních technologií (VŠIT). Výstupy z nové aplikace OUdot jsou porovnávány se starým způsobem zpracování a podle nynějších výsledků se mikro-datové soubory shodují. Tímto byl splněn hlavní cíl diplomové práce.

Naplnění druhého dílčího cíle, popis postupů při implementaci návrhového vzoru MVVM bylo docíleno v kapitole Vlastní řešení. Zde byly popsány postupy vytvoření jednotlivých vrstev návrhového vzoru MVVM a jejich propojení. Veškeré tyto postupy byly také demonstrovány na praktických ukázkách z aplikace OUdot.

Závěrem z této práce je, že obecně při vývoji aplikace postavené na technologii WPF, lze doporučit následovat návrhový vzor MVVM. Další technologie usnadňující implementaci MVVM, zmíněné v této práci, lze doporučit spíše pro relativně menší a finančně nenáročné projekty.

7 Seznam použitých zdrojů

7.1 Knižní publikace

LIKNESS, J., 2012. *Designing Silverlight Business Application*. Upper Saddle River: Pearson Education. ISBN 9780321810410.

MACDONALD, M., 2010. *Pro WPF in C# 2010: Windows presentation foundation in.NET 4*. New York: N.Y.: Distributed to the book trade worldwide by Springer-Verlag. ISBN 1430272058.

NATHAN, A., 2010. *WPF 4 unleashed*. Indianapolis: Ind.: Sams. ISBN 9780672331190.

SKEET, J., 2014. *C# in depth*. Third editions. Shelter Island: NY: Manning. ISBN 9781617291340.

TENNY, L. a Z. HIRANI, 2010. *Entity Framework 4.0 Recipes: A Problem-Solution Approach*. New York: Apress. ISBN 9781430227038.

VIRIUS, M., 2011. *Programování pro.NET*. Praha: České vysoké učení technické. ISBN 9788001048641.

VIRIUS, M., 2012. *C# 2010: Hotová řešení*. Brno: Computer Press. ISBN 9788025137307.

7.2 Internetové zdroje

Common Language Runtime (CLR), 2016. *MSDN Microsoft* [online] [cit. 2016]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/8bs2ecf4(v=vs.110).aspx

ČÁPKA, D., 2016. Úvod do C# a.NET frameworku. *ITnetwork.cz* [online] [cit. 2016]. Dostupné z: http://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework/

Entity Framework Databinding with WPF, 2016. *MSDN Microsoft* [online] [cit. 2016]. Dostupné z: https://msdn.microsoft.com/en-us/data/jj574514.aspx

HAMILTON, N., 2008. Computerworld. In: *The A-Z of Programming Languages: C#* [online]. 1. 10. 2008 [cit. 2016. Dostupné z: http://www.computerworld.com.au/article/261958/a-z_programming_languages_c_/?

HASAN, N., 2012. History of C# Programming. *All About C# Programming* [online] [cit. 2016]. Dostupné z: http://aboutcsharpprogramming.blogspot.cz/2012/09/history-of-c-programming.html

How to use MVVM Light Toolkit for Windows Phone, 2013. *Microsoft TechNet* [online] [cit. 2017]. Dostupné z: https://social.technet.microsoft.com/wiki/contents/articles/27237.windows-phone-how-to-use-mvvm-light-toolkit-for-windows-phone.aspx

Introduction to the C# Language and the.NET Framework, 2016. *MSDN Microsoft* [online] [cit. 2016]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/z1zx9t92.aspx

KOCAN, M., 2001. Co je a k čemu je Microsoft.NET. *Živě* [online] [cit. 2016]. Dostupné z: http://www.zive.cz/clanky/co-je-a-k-cemu-je-microsoft-net/sc-3-a-103190/default.aspx

KREFT, K. a A. LANGER, 2003. Artima. In: *After Java and C# - what is next?* [online]. 3. 7. 2003 [cit. 2016. Dostupné z: http://www.artima.com/weblogs/viewpost.jsp?thread=6543

LERMAN, J., 2010. MSDN Microsoft. In: *Data Points - Deny Table Access to the Entity Framework Without Causing a Mutiny* [online].2010 [cit. 2016. Dostupné z: https://msdn.microsoft.com/en-us/magazine/ff898427.aspx

MVVM Light Messenger, 2017. *DotNetPattern* [online] [cit. 2017]. Dostupné z: http://dotnetpattern.com/mvvm-light-messenger

MVVM-Light Info, 2017. *Stack Overflow* [online] [cit. 2017]. Dostupné z: http://stackoverflow.com/tags/mvvm-light/info

OSBORN, J., 2000. Windows DevCenter. In: *Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg* [online]. 1. 8. 2000 [cit. 2016. Dostupné z: http://www.windowsdevcenter.com/pub/a/oreilly/windows/news/hejlsberg_0800.html

PRAJAPATI, S., 2015. WPF: Versions, History and Major Enhancements. *Code Project* [online] [cit. 2016]. Dostupné z: http://www.codeproject.com/Articles/1035800/WPF-Versions-History-and-Major-Enhancements

Understanding.Net Framework 4.5 Architecture, 2013. *DotNetTricks* [online], verze 2014 [cit. 2016]. Dostupné z: http://www.dotnet-tricks.com/Tutorial/netframework/NcaT161013-Understanding-.Net-Framework-4.5-Architecture.html

WPF Architecture, 2016. *MSDN Microsoft* [online] [cit. 2016]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms750441(v=vs.110).aspx

8 Seznam obrázků

9 Seznam tabulek

10 Seznam ukázek kódu

11 Přílohy

Zdrojové soubory aplikace OUdot se nacházejí na přiloženém DVD. Zde se nachází složka celého projektu vytvořeného ve vývojovém prostředí Visual Studio 2015.