VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

NÁVRHOVÝ VZOR MODEL-VIEW-VIEWMODEL VE WPF APLIKACÍCH
MODEL-VIEW-VIEWMODEL DESIGN PATTERN IN WPF APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Patrik Švikruha**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Ivo Lattenberg, Ph.D.**
SUPERVISOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Patrik Švikruha ***ID:*** 164423
Ročník: 3 ***Akademický rok:*** 2015/16

NÁZEV TÉMATU:

Návrhový vzor Model-View-ViewModel ve WPF aplikacích

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte základní principy vícevrstvé architektury. Seznamte se podrobněji s návrhovým vzorem Model-Vie--ViewModel. Vytvořte desktopovou aplikaci sloužící jako databáze elektrotechnických součástek v konstrukční dílně. Aplikace bude poskytovat jak skladové informace, tak i technické informace o součástkách (datasheet, fotku, parametry). Měla by umožňovat vyhledávání dle vybraných kritérií. Bude obsahovat modul pro správu (role admin - skladník) a uživatelský modul (role user - konstruktér). Aplikaci vytvořte s pomocí WPF a použijte návrhový vzor Model-View-ViewModel implementovaný pomocí frameworku Catel. Data budou uložena v databázi Microsoft SQL, pro přístup k datům použijte Entity Framework.

DOPORUČENÁ LITERATURA:

[1] PETZOLD, C. Mistrovství ve Windows Presentation Foundation, Nakladatelství Computer Press, a.s. 2008, 928 stran, ISBN 9788025121412

[2] WATSON, B., C# 4.0 - řešení praktických programátorských úloh, Zoner Press, 2010, 656 s., ISBN 978-8--7413-094-6

[3] VIRIUS, M., C# 2010 Hotová řešení, Computer Press, 2012, 424 s., ISBN 978-80-251-3730-7

Termín zadání: 1.2.2016 ***Termín odevzdání:*** 1.6.2016

Vedoucí práce: doc. Ing. Ivo Lattenberg, Ph.D.
Konzultant bakalářské práce:

doc. Ing. Jiří Mišurec, CSc., *předseda oborové rady*

ABSTRAKT

Práca sa zaoberá vysvetlením návrhu aplikácie podľa viac-vrstvových prezentačných vzorov, ktoré popisujú rozdelenie aplikácie na vrstvy a tým uľahčujú jej testovateľnost, udržateľnosti a prenositeľnosť. V práci sa zameriavam na prezentačný vzor pre Model-View-ViewModel ktorý je primárne určený pre WPF aplikácie. Tento prezentačný vzor je možné vytvoriť pomocou frameworkov ktoré sa používajú buď ako NuGet balíčky alebo ako Projektové šablóny. V práci sa zameriavam na framework Catel, ktorý je možné použiť, vďaka jeho robustnosti, skrz všetky vrstvy aplikácie. Primárnym výstupom práce je aplikácia, na ktorej demonštrujem návrhový vzor s integráciou frameworku Catel. Teóriu a následný postup som sa snažil opísať tak, aby ho pochopil aj čitateľ, ktorý nemá žiadne skúsenosti s návrhom viac-vrstvovej aplikácie.

KĽÚČOVÉ SLOVÁ

Model-View-ViewModel, WPF aplikácia, Návrhový vzor, Prezentačný vzor, Architektonický vzor, Architektúra Model-View-Controller, Viac-vrstvá architektúra, XAML, .NET Framework, C#, framework Catel, SCRUM, Agilne metodiky

ABSTRACT

The thesis deals with the explanation of the proposal application according to multi-layered presentation models which describe the distribution of the application layers thereby its facilitating, testability, sustainability and transferability. In this work i focus myself on presentating model for Model-View-ViewModel which is primary designated for WPF application. This presentating model is possible to be built by frameworks used as NuGet open-source package managers or Project templates. In this thesis I focus on framework Catel, which can be used thanks to it´s robustness through all layers of application. The primary output of this thesis is a real application, on which I demonstrate design template with the intagration of framework Catel. I tried to describe whole theory and following process as easy as possible, that even an reader, who has no experiences with designing a multilayer application, could understand it.

KEYWORDS

Model-View-ViewModel, WPF application, Architecture pattern, Presentation model, Architecture Model-View-Controller, Multitier architecture, XAML, .NET Framework, C#, framework Catel, SCRUM, Agile software development

PATRIK, Švikruha *Návrhový vzor Model-View-ViewModel ve WPF aplikacích*: bakalárska práca. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 83 s. Vedúci práce bol doc. Ing. Ivo Lattenberg, Ph.D.

PREHLÁSENIE

Prehlasujem, že som svoju bakalársku prácu na tému „Návrhový vzor Model-View-ViewModel ve WPF aplikacích“ vypracoval(a) samostatne pod vedením vedúceho bakalárskej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor(ka) uvedenej bakalárskej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil(a) autorské práva tretích osôb, najmä som nezasiahol(-la) nedovoleným spôsobom do cudzích autorských práv osobnostných a/nebo majetkových a som si plne vedomý(-á) následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., o právu autorském, o právoch súvisejúcich s právom autorským a o zmeně niektorých zákonov (autorský zákon), vo znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhé, hlavy VI. diel 4 Trestného zákoníka č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

POĎAKOVANIE

Na úvod by som sa rád srdečne poďakoval vedúcemu bakalárskej práce pánovi doc. Ing. Ivanovi Lattenberg, Ph.D. za odborné vedenie, konzultácie, cenné rady a v neposlednom rade mojej priateľke Petre za trpezlivosť a morálnu podporu.

Brno . . . .

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

POĎAKOVANIE

Výzkum popsaný v tejto bakalárskej práci bol realizovaný v laboratóriách podporených projektom SIX; registračné číslo CZ.1.05/2.1.00/03.0072, operačný program Výzkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .
podpis autora(-ky)

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ÚVOD

Návrhový vzor (design pattern) predstavuje pri softwarovom vývoji obecnú šablónu riešenia problému. Typický príklad návrhového vzoru je napríklad vzťah v OOP (Object-oriented programming) interakcia medzi triedami a objektami (bez toho aby bola implementovaná daná trieda).

Návrhové vzory sa používajú na oddelenie užívateľského rozhrania, dátovej časti a časti spracovania dát. Riešia teda problém tzv. „špagetového kódu“, kedy sa nachádza celý program v jednom súbore. Takýto kód je veľmi neprehľadný a prípadná zmena jednej časti kódu často vedie k prepísaniu programu a zmeny aplikácie ako celku. Z týchto skúseností vznikli návrhové vzory, ktoré logicky rozdeľujú aplikáciu do viacerých vrstiev a snažia sa o čo najefektívnejšiu modularitu aplikácie.

V mojej práci sa budem zaoberať návrhovým vzorom Model-View-ViewModel a jeho implementáciou vo WPF aplikácii, v ktorej budem používať framework Catel.

Návrhový vzor MVVM predstavuje šablónu riešení modelových situácii a tým uľahčuje a zrýchľuje vývoj business aplikácii.

Spomínaný framework Catel sa špecializuje na vývoj takýchto aplikácii a implementuje v sebe množstvo nástrojov ako sú validácie, UserControl, rôzne UI elementy, integráciu vzorov ako Unit Of Work alebo Repository. Framework Catel teda budem používať skrz všetky vrstvy aplikácie a budem sa snažíť o čo najširší záber demonštrácie jeho výhody.

Vypracovanie práce sa snažím koncipovať z praktického uhľa pohľadu vývojára a v praktickom výstupe - aplikácii sa budem snažiť aj o dodržovanie zásad objektového programovania.

Časť I

Teoretická časť

1 .NET FRAMEWORK

Na začiatok je vhodné definovať úlohu a význam používania frameworkov ako takých. Framework je komplexné softvérové riešenie, ktorého úlohou je pomôcť vývojárom sústrediť sa na vývoj a odbremeniť ich od písania tzv. "boiler-plate" kódu (kód ktorý sa opakuje) napríklad pomocou knižníc, ktoré obsahujú metódy a triedy pre prácu s určitými prvkami. V skratke môžeme povedať, že framework teda poskytuje funkcionalitu, ktorú by sme bez jeho použitia museli implementovať. Často sa zamieňajú pojmy framework a knižnica, rozdiel medzi nimi je ale značný, a to v spôsobe akým sa používajú. Knižnice sú starší koncept a dá sa povedať, že je to sada metód ktoré voláme z nášho kódu. Framework, naproti tomu volá náš kód na miestach kde sme ho použili. Tento princíp sa nazýva „Inversion of Control".[17]

Príklad uľahčenia práce frameworkom

```
//.Net frameworku 3.5 and lower
        private string name;
        public string Name
        {
                get { return name; }
                set { name = value; }
        }
//.Net frameworku 3.5 and higher
        public string Name { get; set; }
```

Microsoft začal vyvíjať .NET Framework od 90. rokov pod názvom Next Generation Windows Services (NGWS) a jeho prvá beta verzia bola vydaná v roku 2000. V auguste roku 2000 Microsoft, Hewlett-Packard a Intel pracovali na štandardizácii Common Language Infrastructure (CLI). V decembri 2001 bola ratifikovaná ECMA štandardom.[18] Aktuálna verzia .NET Frameworku je 4.6.1 z 30. Novembra 2015.[28]

Od verzie 4.5 sú vydávané paralelne 2 verzie frameworku. .NET Framework 4.6.1 môžeme označiť za plnú verziu frameworku obsahujúcu WinForms, ASP.NET, ADO.NET, WPF, WCF, LINQ, Entity Framework. Odľahčená Open-Source verzia s názvom .NET Framework Core 5 bola uvoľnená 12. Novembra 2014 je Cloud-optimized a Cross-platformová.[27]

.NET Framework je súčasť systému Windows, ktorá obsahuje virtuálny systém spustenia s názvom CLR (Common Language Runtime) a unifikovanú sadu knižníc tried. CLR predstavuje komerčnú implementáciu medzinárodného štandardu CLI spoločnosti Microsoft, ktorá je základom pre vytváranie, vykonávanie a vývoj prostredí, v ktorých spoločne fungujú jazyky a knižnice.

Platforma .NET nepredpisuje nešpecifikuje použitie programovacieho jazyka, avšak vždy sa kód preloží do medzijazyka CLI.

Common Intermediate Language (CIL) je v informatike najnižšie človekom čitateľný programovací jazyk definovaný špecifikáciou CLI používaný projektami .NET Framework, Mono a prerušený projekt DotGNU. CIL bol pôvodne behom uvoľňovania beta .NET frameworku známy aj ako Microsoft Intermediate Language (MSIL). Vzhľadom k štandardizácii C# a Common Language Infrastructure je byte-kód novo a oficiálne označovaný ako CIL. Jazyky, ktoré sa zameriavajú na CLI kompatibilné prostredie, sú zostavované do byte kódu (Bytecode). CIL patrí medzi objektovoorientované jazyky výhradne zásobníkového typu (anglicky stack-based). Prevádzaný je prostredníctvom virtuálneho stroja.

Spolu s platformou .NET Framework bol predstavený aj vysokoúrovňový objektovo orientovaný-programovací jazyk C#, ktorý je založený na jazykoch C++ a Java. C# zväčša priamo odráža vlastnosti vrstvy CLI, ktorá leží pod ním a je navrhnutý tak, aby umožňoval maximálne využitie všetkých vlastností, ktoré CLI poskytuje. Väčšina typov zavedených v C# priamo korešponduje s hodnotovými typmi implementovanými v CLI.

Common Type System (CTS) je unifikovaný typový systém používaný všetkými jazykmi pod platformou .NET Framework. Všetky typy, vrátane primitívnych dátových typov ako je napr. Integer, sú potomkami triedy `System.Object` a dedia sa od nej aj všetky jej metódy (napr. `ToString()`). Typy v CTS sa delia do dvoch skupín, a to hodnotové a referenčné. Hodnotové dátové typy sú z výkonnostných dôvodov alokované na zásobníku.

Hodnotové typy sa delia na 3 časti:

- Primitívne dátové typy - jedná sa o celočíselné a reálne primitívne dátové typy (`Byte`, `Integer`, `Char`, `Float`, `Double`, `Decimal`)
- Štruktúry: jedná sa o užívateľsky definované dátové typy, ktoré pripomínajú triedy, ale nemôžu dediť ani byť dedené a sú podobne ako všetky hodnotové typy alokované na zásobníku na halde.
- Výčtové typy (`Enum`) : v C# je `Enum` iba množina predom definovaných hodnôt (napr. `Enum DayOfWeek`, s hodnotami Monday, Thuesday, atď.) bez možnosti definovať si vnútri enum metódy, atribúty, indexery alebo implementovať rozhranie.

Referenčné dátové typy neuchovávajú na rozdiel od hodnotových samotnú hodnotu, ale odkaz na miesto v pamäti, kde je požadovaná hodnota uložená. Všetky odkazované typy sú alokované na halde.[25]

1.1 Jazyk C#

C# je multiparadigmatický, imperatívny, deklaratívny, funkcionálny, generický a objektovo-orientovaný (class based) programovací jazyk. Bol navrhnutý tímom Andersa Hejlsberga a prvý krát bol predstavený s beta verziou .Net Frameworku v roku 2000. V decembri 2001 bol ratifikovaný ECMA štandardom.[16] Pri tvorbe jazyka C# boli kladené nasledovné ciele v štandarde ECMA:

- Bol navrhnutý ako jednoduchý, moderný objektovo orientovaný jazyk pre všeobecné použitie.
- Jazyk a jeho implementácie by mali poskytovať pre nasledovné princípy softvérového inžinierstva podporu - silná typová kontrola, kontrola ohraničenia polí, detekciu pokusov na využitie neinicializovaných premenných a automatická správa pamäte (tzv. Garbage collector).
- Na prenositeľnosti zdrojového kódu je kladený vysoký dôraz.
- Jazyk je plánovaný na ekonomické využívanie pamäte a procesorového času, nie je však kompletne zameraný na výkonnosť a veľkosť výsledného binárneho kódu, ako napr. jazyky C alebo assembler.

Vychádza z jazykov C a C++, z ktorých čerpá syntax a z jazyka Java, z ktorého čerpá niektoré vlastnosti (ako napr. Garbage collector). C# bol vytváraný tak, aby bol jednoduchým, moderným, objektovo orientovaným jazykom pre všeobecné použitie. Jazyk a jeho implementácie by mali poskytovať podporu pre nasledovné princípy softvérového inžinierstva ako silná typová kontrola, kontrola ohraničenia polí, detekciu pokusov na využitie ne-inicializovaných premenných a automatickú správu pamäte. Dôležitými vlastnosťami je tiež robustnosť, odolnosť a produktivita.

C# ako programovací jazyk priamo odráža vlastnosti vrstvy CLI, ktorá leží pod ním a bol priamo navrhnutý tak, aby umožňoval využitie všetkých vlastností, ktoré poskytuje CLI, na rozdiel od jazykov, ktoré majú vlastnú syntax a využívajú len podmnožinu vlastností CLI (napr. Visual Basic).

Typy zavedené v jazyku C# poväčšine priamo korešpondujú s hodnotovými typmi implementovanými v CLI. To však neznamená, že určuje podmienky, ktorými sa má generovanať kód z kompilátora. Teda kompilátor jazyka C# nemusí prekladať kód na platformu CLI, respektíve vôbec nemusí generovať medziprekladový jazyk CIL(MSIL) ani žiaden iný formát. Teoreticky je možné vytvoriť kompilátor jazyka C#, ktorý bude prekladať priamo do strojového kódu, tak, ako tradičné kompilátory jazyka (napr. C++, Fortran, atď.).[19][29]

Výpis Hello World v konzolovej aplikácii

```
using System;
namespace ConsoleApplication{
  class MasterClass{
    static void Main(string[] args){
      Console.WriteLine("Hello World!");
    }
  }
}
```

Rozdiely jazykov C a C++ s C#:

- Neexistujú globálne premenné. Všetky metódy a atribúty musia patriť danej triede.
- Názvy premenných sa nesmú v blokoch opakovať. Vedie to k zlepšeniu čitateľnosti kódu a zamedzuje nejednoznačnostiam pri čítaní zdrojového kódu.
- Miesto globálnych funkcii (napr. `printf()` v jazyku C) musia byť všetky metódy deklarované s príslušnou triedou. Triedy sú ďalej organizované do menných priestorov (`namespace`).
- Existuje iba jednoduchá dedičnosť.

1.2 XAML - Extensible Application Markup Language

Extensible Application Markup Language (XAML) pôvodným názvom Extensible Avalon Markup Language (podľa pôvodného názvu WPF - Avalon) je značkovací jazyk vychádzajúci z XML ktorý je používaný k opisu grafického rozhrania v aplikáciách Microsoft novej generácie. XAML sa používa od verzie .NET Frameworku 3.0 hlavne v technológiách Windows Presentation Foundation, Workflow Foundation a Silverlight.

V WPF a Silverlight aplikáciách sa XAML používa na vytvorení užívateľského rozhrania, zatiaľ čo pri WF sa používa na definovanie samotných workflows. Všetko čo je vytvorené pomocou XAML je možné interpretovať pomocou štandardných .Net jazykov ako C# alebo VB.NET. Pre prácu s XAML bola vytvorená aplikácia z balíku Microsoft Expression Tools, ktorý sa používa dohromady s Visual Studio. XAML je možné upraviť aj v poznámkovom bloku alebo XAMLPad editore.

Výpis Hello World pomocou XAML

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007"
 ↪  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <TextBlock>Hello World!</TextBlock>
</Canvas>
```

XAML môže byť skompilovaný do binárneho súboru (Bimary XAML - .baml), ktorý potom môže byť vložený a používaný ako resource v .NET projekte. Za behu aplikácie si framework zoberie tento súbor a vykreslí užívateľské rozhranie.[6]

1.3 Windows Forms

Windows Forms (WF) je prvým frameworkom z .NET-u, ktorý umožňuje jednoduchú tvorbu formulárových aplikácii pomocou grafického designeru. Nachádza sa v ňom sada pripravených komponent pre väčšinu situácii. V súčasnej dobe sa WF stále používa a neoznačuje sa za zastaraný, pretože jeho použitie je veľmi jednoduché a v praxi sa stále využívajú aplikácie, ktoré sú vo WF napísané.

Aplikácie WF obsahujú spravidla dva súbory, jeden označený napr. *MyClass.cs* a druhý *MyClass.Designer.cs*. Samotný formulár je teda instancia triedy (objekt), ktorá je vydedená z triedy Form, ktorá je typu Partial (nachádza sa vo viacerých súboroch - *MyClass.cs* a *MyClass.Designer.cs*). Súbor *MyClass.Designer.cs* obsahuje kód, ktorý je automaticky generovaný integrovaným vývojárskym prostredím a ktorý je oddelený od aplikačnej logiky, ktorú vytvára Programátor. Konštruktor tejto triedy obsahuje metódu `InitializeComponent()`, ktorá inicializuje všetky prvky na formulári a nastaví im príslušné vlastnosti.[5]

1.3.1 Rozdiely medzi WF a WPF

WF používa komponenty, WPF používa elementy, ktoré nemajú vlastnosť `Location` (tá bola pri WF určená na určenie absolútnej pozície). Miesto toho elementy používajú vlastnosť `Margin`. Ak máme element napr. vpravo hore, tak na osadenie nám stačí určiť vlastnosti `Margin` odsadenie sprava a zhora.

Vo WPF je možné použiť popri elementoch aj komponenty z WF, a to pomocou `WindowsFormsHost`. Hlavný rozdiel pri pridávaní elementov vo WPF oproti WF je ten, že v WF použijeme kód `this.Controls.Add(tb);` zatiaľ čo pri WPF nemáme komponenty, ktoré ale zastupujú elementy `UIElements`. Tieto elementy môžeme pridávať do gridu alebo rôznych druhov panelov, avšak nie je možné pridávať elementy do samotného formu. Priamo do formu môžeme pridať iba jeden element a tým je grid.[22]

1.4 Windows Presentation Foundation

Windows Presentation Foundation (WPF), pôvodne Avalon, môžeme definovať ako grafický subsystém určený na renderovanie užívateľského rozhrania v aplikáciach založených na platforme Windows alebo aj ako framework určený pre komplexnú

tvorbu formulárových aplikácii pomocou elementárnych grafických prvkov (pojmu komponenty sa snažím vyhnúť kvôli kolízii s komponentami pri WF). Behové knižnice WPF (runtime libraries) obsahujú všetky verzie .Net Frameworku od verzie 3.0. čiže v OS od Windows Vista a Windows Server 2008 vyššie.

WPF obsahuje širokú paletu formulárových prvkov a veľmi obsiahle API. Úlohou WPF je zjednotenie prvkov užívateľského rozhrania - 2D a 3D grafiku, vektorovú a rastrovú grafiku, runtime animácie, previazanie dát (tzv. DataBinding), audio a video. WPF používa prezentačný vzor Model-View-ViewModel a sadu knižní poskytujúcich API Microsoft DirectX.

Hlavnou myšlienkou WPF je skladanie jednotlivých elementárnych grafických prvkov dohromady. Môžeme si to vysvetliť napr. na elemente `Button`, ktoré je v WPF možné rozložiť na elementy `Button` a `TextBlock`. Toto rozloženie poskytuje nový pohľad na stavbu aplikácie resp. užívateľského rozhrania a nové možnosti ako s jednotlivými elementami pracovať. WPF je založená na vektorovej grafike, čo znamená že ľahko reaguje na zmenu rozlíšenia obrazovky a nastavenia DPI. S použitím layout enginu, ktorý zaisťuje rozvrhnutie a vykreslenie jednotlivých prvkov v okne je následne jednoduché dosiahnuť vzhľad, ktorý reaguje na zmeny veľkosti okna a tiež na lokalizované texty v jednotlivých elementoch.

WPF vzniklo na základe tlaku na graficky bohatšie aplikácie, kde už starší projekt WindowsForms(WF) nestačil hlavne z týchto dôvodov:

- na trhu dominujú mobilné zariadenia a WF aplikáciu je problém prispôsobiť kvôli slabej podpore DPI. Nie je teda možné používať tu istú aplikáciu na mobile, tablete a na stolnom počítači s Full HD rozlíšením. Pri WPF je zavedená jednotka dĺžky DIP (Device Independent Pixel) a čisto vektorová grafika aby vyzerala aplikácia na každom zariadení rovnako.
- WF ukladá pri každej komponente jej absolútnu pozíciu, čo sa ale pri návrhu zložitých formulárov nehodí. Pre tvorbu prezentačnej vrstvy sa na webe osvedčil jazyk HTML a C# sa rovnako ako JAVA inšpiroval a zaviedol definíciu formulárov pomocou derivácie jazyka XML - XAML. Daňou za kvalitnejšiu aplikáciu je pri WPF zložitejší návrh formulára a zvládnutie ďalšieho, aj keď jednoduchého jazyka (XAML).
- Používanie staršieho zobrazovacieho grafického rozhrania Windows (GDI) je pomalšie a prináša množstvo obmedzení a preto sa pri WPF od neho upustilo. WPF používa Direct3D rozhranie pre akcelerovanú grafiku a tým sú WPF aplikácie rýchlejšie a menej zaťažujú CPU. Vďaka nezávislosti na GDI je možné sa odpútať od jednoduchej palety základných komponentov operačného systému a vytvoriť grafický bohatú aplikáciu. Môžme napríklad vkladať obrázky do tlačidiel, spriehľadňovať elementy (čo bolo pri WF zložité), vytvárať skin-y pre aplikácie alebo jednoducho animovať pomocou storyboard-ov.

WPF ponúka architektúru DataBindingu, ktorá je veľmi výkonný nástroj pre manipuláciu a prácu s dátami v užívateľskom rozhraní zaisťujúcu synchronizáciu dát medzi UI a dátovými objektami.

Existují tri typy DataBindingu a to:

- "one-time DataBinding" – stiahne dáta a ignoruje aktualizácie na zdroji.
- "one-way DataBinding" – komunikácia prebieha iba jedným smerom (read-only dátový pristup).
- "two-way DataBinding" – zdroj a aj klient spolu komunikujú a aktualizujú sa navzájom.

K dispozícii je tiež podpora notifikácie zmien a využitie dáta konvertorov medzi dátovým zdrojom a cieľom. DataBindingu ponúka tiež triedy pre manipuláciu s kolekciami dát vo forme `CollectionView`, ktoré ponúkajú radenie, grupovanie navigácii a filtrovanie poskytnutých dát. Previazanie dát podporuje niekoľko rôznych zdrojov a nechýba samozrejme ani podpora dotazovacieho jazyka LINQ, kde môžeme k jeho výstupu pristupovať ako k zdroji dát.

WPF ponúka niekoľko možností ako pracovať s textom a zobraziť užívateľovi text vo forme dokumentu, napríklad vo forme Flow dokumentu, ktorý ponúka jednoduché zväčšovanie formátovaného textu a umiestnenie textu do stĺpcov alebo vkladanie obrázkov a iných elementov WPF. Je tiež možné vytvoriť fixný dokument za použitia XPS rovnako ako v Office a tiež je k dispozícii nové API pre prácu s tlačiarňami, tlačiarenskými servermi a tlačovými frontami.

Všetky elementy WPF majú východzí vzhľad, ktorý je definovaný autorom daného prvku. Každý vývojár, designer však môže tento vzhľad zmeniť a prispôsobiť si ho bez ohľadu na jeho štandardné chovanie. Napríklad je teda možné zmeniť kompletný vzhľad tlačidla bez toho, aby sme tlačidlo pripravili o jeho udalosti - napr. `Onclick`.

Šablóna vo WPF je mechanizmus, ktorý umožňuje zmeniť UI každému elementu. Máme teda možnosť definovať šablónu UI a túto šablónu priradiť k jednotlivým prvkom, o ktorých vykreslenie sa stará WPF. Veľmi podobne môžeme použiť štýly, ktoré umožňujú definíciu jednotlivých nastavení na jednom mieste a podobným spôsobom priradenia sa tieto vlastnosti premietnu na sadu prvkov. Štýly sa chovajú veľmi podobne ako kaskádové štýly CSS pri HTML.

Každá WPF aplikácia funguje tak, že v popredí prebieha okno a na pozadí prebieha proces. Je teda žiadúce, aby bol proces na okne nezávislí a táto nezávislosť sa nám odráža aj na prezentačnom vzore MVVM, ktorý WPF používa. View slúži pre zobrazenie a interakciu s informáciami z daného procesu, ViewModel na spracovanie príkazov a posúva dáta, prípadne volá webservisy vrstvy Model. Je teda možné zavrieť okno(View) bez jeho vplyvu na proces a tiež jednoduché zdieľanie

dát/informácii (Model) medzi viacerými oknami. Z tohto vyplýva, že pri tvorbe pomocou Modelu WPF je vhodnejšie, aby inštanciu ViewModelu vytváral View. Model MVVM nie je nutné pri WPF aplikáciách dodržovať, avšak pri robustnejších aplikáciách sa oplatí prepracovanejší návrh, ktorým následný vývoj značne urýchli.[23][7][26]

1.5 Entity Framework

Entity Framework je open-source ORM (Object-relational mapping) framework, čo znamená, že z databáze miesto poľa hodnôt dostávame objekty, ku ktorým sa viažu aj metódy. Tabuľky v databázy vidíme teda ako kolekcie rôznych objektov (entít), vďaka čomu s nimi dokážeme pracovať s bežnými prostriedkami daného jazyka. ORM teda oddelí relačnú databázu a vytvorí z nej Entity.

Prístup ORM, má však aj svoju temnú stránku a tou je, že dochádza k pomerne veľkej degradácii výkonu databázy, vzhľadom k tomu, že generované SQL commands môžu byť neefektívne.

Entity framework je ale naproti tomuto výkonovo odladený, a preto je jeho spolupráca s .NET platformou veľmi efektívna. Bol aj súčasťou .NET frameworku do verzie Entity Framework 5, ale od verzie Entity Framework 6 sa jedná o samostatný projekt. Entity framework vytvára takú úroveň abstrakcie, ktorá oddelí programátora od databáze a dovoľuje mu pracovať s objektovými modelmi dát a vďaka odľahčeniu práce programátora tým urýchľuje vývoj softvéru.

Entity framework ponúka 3 možnosti vývoja a to EF designer from database (kedy vygeneruje triedy podľa existujúcej DB) - táto metóda sa hodí pre existujúce DB, ďalej EF designer model (Drag n Drop vývoj, kde si cez GUI vytvoríme entity a určíme vzťahy medzi nimi - EF nám vygeneruje DB tabuľky a aj triedy) a posledná možnosť Code First model, ktorý tvorí tabuľky a vzťahy podľa existujúcich tried a vzťahov medzi nimi. Code First však môže vygenerovať aj triedy z tabuľky a následne môžeme meniť relačný databázový model zmenou tried jednotlivých Entít.

Pre programátora je veľmi vhodná cesta pomocou *Code-First* vývoja, kde po vytvorení DbContextu, nastavení *Connection Stringu* a povolení automatickéj migrácie a povolenia automatického prepisovania dát stačí zmeniť *Entity Model* - triedy a tabuľky sa v databáze automaticky prispôsobia daným entitám.

2 TEORETICKÝ NÁVRH APLIKÁCIE

2.1 Historický vývoj návrhu aplikácie

Na začiatku 60. rokov minulého storočia vznikla softvérová kríza ktorej charakteristickými znakmi bolo predlžovanie a predražovanie projektov, nízka kvalita návrhu programov, a z toho vyplývajúca zlá produktivita práce vývojára a neefektívnosť vývoja.

Príčin bolo niekoľko, ale hlavná príčina bola absencia jednotného vzoru vývoja aplikácii. Aplikáciu navrhoval a schvaľoval priamo vývojár, čo viedlo k individuálny riešeniam návrhov, ktoré viedli k nepresným odhadom vývoja, nákladov a rozsahu. Taktiež bolo rozširovanie a udržovanie aplikácii zložité a nejednotné.

Koncom 60. rokov začal vznikať odbor softwarové inžinierstvo. Vznikli pojmy ako „návrh zhora dole“, „modularita“, atď. Začali vznikať prvé aplikácie ktoré mali interakciu s užívateľom, vznikajú znovu-použiteľné časti programov (moduly) a začína sa upúšťať od projektov „šitých na mieru“. Koncom 70. rokov vznikajú a začínajú sa používať dodnes využívané techniky ako špecifikácia, návrh, architektúra, testovanie, modely životného cyklu, návrhové vzory, atď.

V 80. rokoch prichádza k nástupu softvérovo-inžinierske metodiky vychádzajúce z objektovo-orientované programovanie ako objektovo orientované návrhy, snaha o interoperabilitu softvérových produktov, štandardizácia a upustenie od jednorazových riešení, zdokonalenie komponentových technológii a vznik návrhových architektúr a prezentačných modelov. [1][4]

2.2 Návrhové vzory

Pojem návrhový vzor sa v terminológii softvérového inžinierstva používa pre všeobecný popis riešenia daného problému. Vzory začali vznikať už v 60. rokoch minulého storočia, avšak prvé použiteľné návrhové vzory sa objavili až v roku 1987 v Orlande na konferencii OOPSLA (Object-Oriented Programming, Systems, Languages & Applications).

Asi najväčším míľnikom bolo v tomto smere založenie skupiny GoF (Gang of Four) pánmi Erichom Gamma-om, Richardom Helmom, Ralphom Johnsonom a Johnom Vlissidesom na začiatku 90. rokov. Táto skupina predstavila v roku 1991 na konferencii ECOOP (European Conference on Object-Oriented Programming) prvé návrhové vzory ako napr. Observer, Composite, Decider... Táto skupina vydala knihu „Design Patterns: Elements of Reusable Object-Oriented Software“, ktorá sa pova-

žuje za „bibliu“ v oblasti objektovo-orientovaných návrhových vzorov. Kniha obsahuje 23 základných vzorov rozdelených do troch skupín:

- Creational Patterns (Vzory pre tvorbu objektov)
- Structural Patterns (Štrukturálne vzory)
- Behavioral Patterns (Vzory chovania)

2.2.1 Creational Patterns (Vzory pre tvorbu objektov)

- **Abstract Factory (Abstraktná továreň)** - Definuje rozhranie pre vytváranie objektov, ktoré budú implementovať rovnaké vlastnosti a metódy. Na rozdiel od interface-u môže aj implementovať metódy alebo prístupnosť metód. Z abstraktnej továrne nie je možné spraviť inštanciu.
- **Builder (Staviteľ)** - Vytvára komplexné objekty pomocou oddelenia procesu ich tvorby tak, aby sa mohol tento proces použiť aj pre iné reprezentácie.
- **Factory Method (Továrenská metóda)** - Vytvára objekty bez špecifikácie triedy ktorej inštanciu nám má továreň vrátiť.
- **Prototype (Prototyp, Klon)** - Vytvára objekty skopírovaním existujúceho objektu.
- **Singleton (Jedináčik)** - Dovolí vytvoriť iba jednu inštanciu danej triedy.
- **Lazy Initialization (Odložená inicializácia)** - Inicializuje objekt až keď sa prvý krát použije.
- **Multiton** - Dovolí vytvoriť jednu jednu inštanciu triedy k danému kľúču.
- **Object pool (Objektový bazén)** - Umožňuje prepoužitie objektov, ktoré sa momentálne nepoužívajú.
- **Resource aquisition is initialization** - Zaisťuje, že zdroje sa uvoľnia po uplynutí životného cyklu objektu.

2.2.2 Structural Patterns (Štrukturálne vzory)

- **Adapter (Adaptér)** - Ak potrebujeme, aby spolu pracovali dve triedy, ktoré nemajú kompatibilné rozhranie. Adaptér vytvorí vlastné rozhranie pre spoluprácu triedy A a triedy B.
- **Bridge (Most)** - Oddelí abstrakciu od implementácie tak, aby mohli byť rozdielne
- **Composite (Strom)** - Zanorenie objektov do stromovej štruktúry, ktoré umožňuje pracovať s daným stromom ako s jedným objektom.
- **Decorator (Dekorátor)** - Dokáže pridávať alebo prepisovať existujúce metódy objektom.
- **Facade (Fasáda)** - Vytvára jednotné rozhranie k sade rozhraní v podsystéme.

- **Flyweight (Muší váha)** - Používa sa ak existuje príliš mnoho objektov, ktoré sú si veľmi podobné.
- **Proxy** - Používa sa ako zástupný objekt pre kontrolu prístupu k inému objektu.

2.2.3 Behavioral Patterns (Vzory chovania)

- **Chain of responsibility (Reťaz zodpovednosti)** - Deleguje príkazy a zreťazuje príjemcov spracujúcich príkazy.
- **Command (Príkaz)** - Zapúzdruje požiadavok do objektu a tým umožňuje napríklad vytvárať operácie ako späť.
- **Interpreter (Interpret)** - implementuje špeciálny jazyk.
- **Iterator (Iterátor)** - Spôsob ako pristupovať k elementom skupiny objektov bez toho, aby sme odhalili jeho základnú reprezentáciu.
- **Mediator (Prostredník)** - Poskytuje komunikáciu medzi dvoma triedami (komponentami), bez toho aby navzájom poznali svoju vnútornú štruktúru.
- **Memento** - Poskytuje možnosť vrátiť (obnoviť) objekt do jeho predchádzajúceho stavu.
- **Observer (Pozorovateľ)** - Ak má objekt mnoho závislostí, tento vzor poskytuje spôsob ako dať vedieť o zmene stavu.
- **State (Stav)** - Poskytuje možnosť objektu zmeniť chovanie, ak sa zmení jeho vnútorný stav.
- **Strategy (Stratégia)** - Zapúzdruje algoritmus tak, aby ho bolo možné použiť v runtime.
- **Template method (Šablonová metóda)** - Definuje skeleton algoritmu ako abstraktnú triedu a povoľuje potomkom triedy upravovať jej určité kroky bez zmeny štruktúry.
- **Visitor (Návštevník)** - Oddeľuje algoritmus z objektovej štruktúry oddelením hierarchie metód do jedného objektu - to umožňuje definovať nové operácie bez zmeny triedy pôvodného objektu.
- **Inversion of control (Dependency Injection, Obrátené riadenie)** – Zvyšuje modularitu programu a robí ho rozšíriteľným.
- **Null Object (Prázdný objekt)** – Prázdny objekt, ktorý funguje ako základný stav objektu a ktorý je náhradou stavu null.
- **Servant (Služobník)** – Definuje spoločnú funkcionalitu pre skupinu tried ktoré nemajú spoločnú rodičovskú triedu.
- **Specification (Špecifikácia)** – Obchodná logika, ktorá sa kombinuje reťazením obchodných pravidiel na základe booleanovskej logiky.

V dnešnej dobe sú však niektoré návrhové vzory integrované v samotných programovacích jazykoch ako napríklad v jazyku C# (Iterator, Abstract Factory, Prototype, Resource aquisition is initialization...).[33] V súvislosti s Entity Frameworkom treba ešte spomenúť vzor Respository, ktorý sa stará od vyššiu úroveň abstrakcie a umožňuje vývojárovi pracovať s Repository (kolekcia modelov jedného druhu) ako s „listom“. Ďalším vzorom je Unit Of Work, ktorý je kolekciou Repositárov.[20]

2.3 Architektúra

Pojem architektúra alebo aj architektonický vzor sa v terminológii softvérového inžinierstva používa pre popis vzoru aplikácie. Popisuje štruktúru aplikácie, jej rozdelenie a všeobecnú funkčnosť jednotlivých častí, čiže ovplyvňuje aj vývoj aplikácie. Keďže Architektúra len teoreticky popisuje všeobecný návrh aplikácie, zaviedli sa prezentačné modely, ktoré určujú jednotlivým prvkom štruktúry ich presnú funkcionalitu, obsah a komunikáciu medzi jednotlivými prvkami modelu. Architektúra teda všeobecne popisuje jednotný postup pri tvorbe aplikácie. [2]

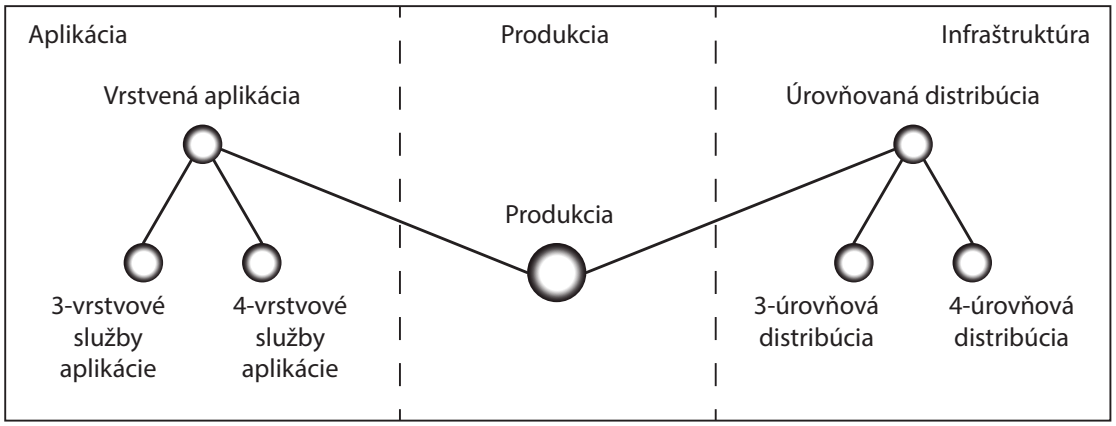
2.4 Viac-vrstvá architektúra

Viac-vrstvá architektúra (používaný aj pojem N-vrstvá architektúra) je návrhový vzor (model) aplikácii, kde sa aplikácia rozdelí na niekoľko logických vrstiev (komponentov), spravidla kvôli výpočtovej infraštruktúre, ale samozrejme aj kvôli udržateľnosti kódu. Príkladom môže byť internetový obchod, kde aplikačná logika beží na webovom serveri (prípadne aplikačnom serveri), dáta (vlastnosti produktov, údaje o užívateľoch, atď.) sú na databázovom serveri a klientska časť (UI) beží v prehliadači užívateľa.

Viac-vrstvá architektúra vytvára model, ktorý umožňuje vytvárať flexibilné aplikácie a znovu použiteľné časti - napr. môžeme zmeniť UI bez toho aby sme zasahovali do aplikačnej logiky, alebo naopak zmeniť aplikačnú logiku bez zásahu do UI. Je to veľmi užitočná vlastnosť napr. pri nasadení na produkciu, kedy nemusíme meniť celú aplikáciu, ale len jej určitú komponentu.

V terminológii softvérovej architektúry si musíme definovať pojmy layer a tier, pretože sú často vysvetľované ako identické. Layer (vrstva) je logická štruktúra mechanizmu elementov (aplikačná logika, dáta, UI) ktoré tvoria softwarové riešenie, zatiaľ čo tier (úroveň) je fyzická štruktúra mechanizmu v systémovej infraštruktúre (Databázový server, Aplikačný server, atď.).[8]

Dnes existuje množstvo frameworky s pred-pripravenými komponentami (drag & drop vývoj), ktoré je možné usporiadať pomocou vizuálneho nástroja v moder-



Obr. 2.1: Schéma aplikácie na produkcii

nom integrovanom vývojárskom editore (IDE). Vývoj aplikácie je zo začiatku rýchly a z pohľadu vývojára pohodlný a jednoduchý. Problémy však nastávajú ak napr. chceme aplikáciu pokryť automatizovanými testami (tzv. Unity test) alebo chceme zaistiť aby sa kód neopakoval.[8]

Pohodlie drag&drop vývoja je teda v praxi vždy sprevádzané komplikáciami, ktoré sa najviac prejavia pri údržbe danej aplikácie. Z toho vyplýva, že je výhodnejšie z pohľadu životného cyklu aplikácie venovať energiu a čas kvalitnému návrhu aplikácie (napriek oveľa väčšej dobe vývoja ako pri drag&drop vývoji) pre následné jednoduchšie ladenie, údržbu a rozšíriteľnosť. Existuje niekoľko architektonických vzorov pre prezentačnú vrstvu (prezentačné vzory, GUI architektúry, atď.), ktoré sa snažia o najudržateľnejšiu aplikáciu. [3]

Prenos dát medzi vrstvami je taktiež súčasťou architektúry a tieto rozhrania sú založené na štandardných protokoloch a technológiách ako napr. Java RMI, .NET Remoting, sokety, UDP alebo rôzne webové služby.

2.4.1 Výhody viac-vrstvovej architektúry

- Oddelením jednotlivých vrstiev môžeme jednoducho vymeniť celú vrstvu - napr. zmeniť databázu z MySQL na Microsoft SQL alebo Oracle, vymeniť UI, biznis logiku a to bez zásadného vplyvu na ostatné vrstvy. Tieto zmeny je možné spraviť za chodu aplikácie bez toho, aby užívateľ zaznamenal výpadok.
- Vrstvy je možné rozdeliť medzi niekoľko vývojárov alebo vývojárskych tímov a tým sa značne zefektívni a urýchli vývoj danej aplikácie.
- Vrstvy sú prenositeľné - je možné ich použiť ako moduly v inej aplikácii.

2.4.2 Nevýhody viac-vrstvovej architektúry

- Oddelenie jednotlivých vrstiev, týka sa to hlavne úrovní tiers (teda HW), je zároveň nevýhoda. Treba totiž prevádzkovať monitoring a údržbu niekoľko rôznych zariadení (napr. servery, databázy, atď.).
- Je nutné mať k dispozícii vybudované integračné testovacie prostredie a v ňom po každej inštalácii patch-a, driver-u, firmware-u alebo nového OS dôkladne aplikáciu otestovať.
- Množstvo technológii, ktoré sú potrebné pre prácu s viac-vrstvovým modelom.

Aby sme maximálne využili výhod viac-vrstvovej architektúry, je treba dodržovať základné pravidlá:

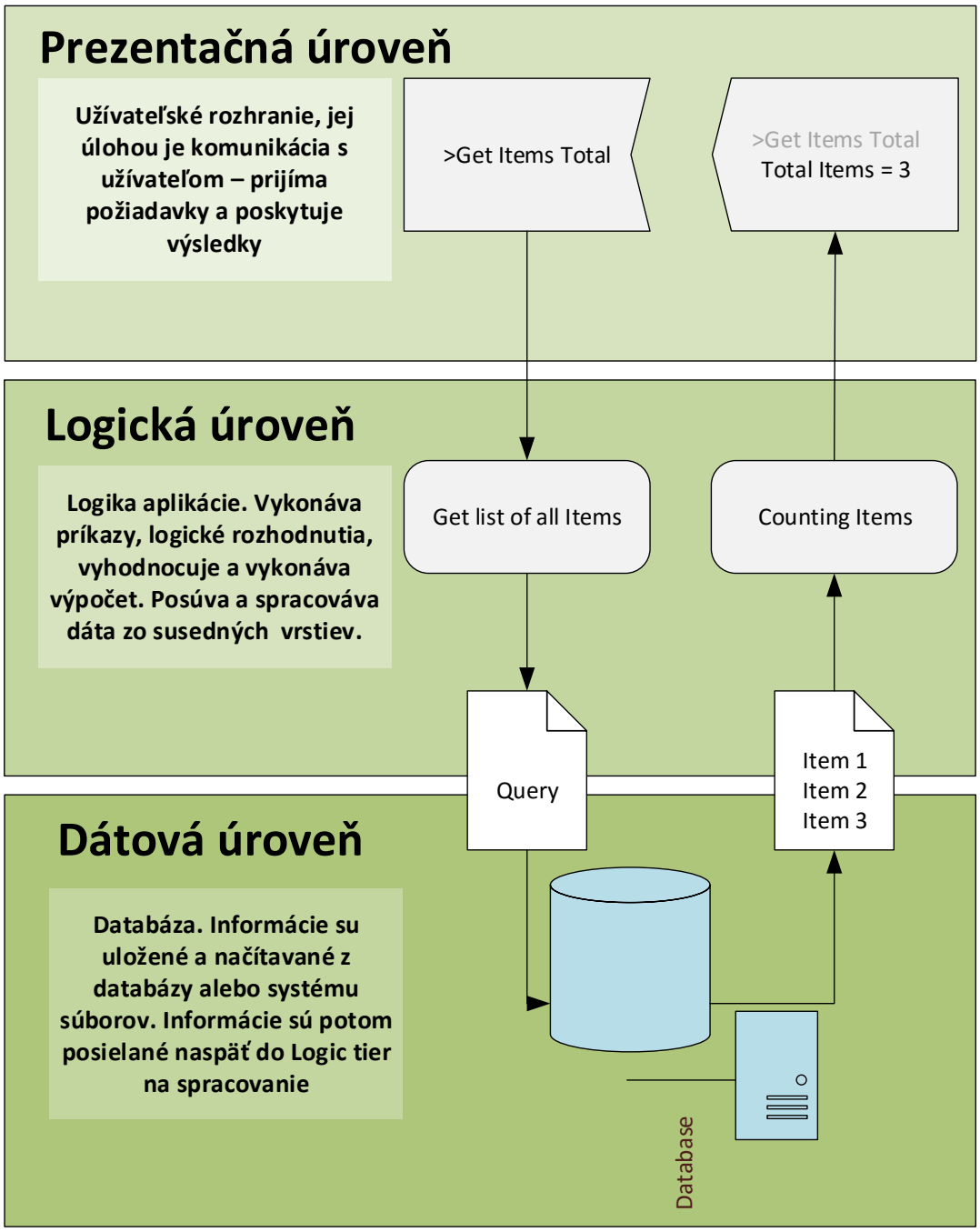
1. Každá vrstva by mala komunikovať len s najbližšou vrstvou.
2. Väzby medzi vrstvami by mali byť čo najvoľnejšie, aby bolo možné danú vrstvu kedykoľvek bez problému vymeniť.
3. Vrstvy by spolu mali komunikovať medzi sebou prostredníctvom správ čo najmenej. Treba predpokladať, že ich v budúcnosti bude nutné rozdeliť a premiestniť do iných úrovní.
4. Vytváranie nových úrovní, len ak je to naozaj potrebné.
5. Jednotlivé vrstvy prevádzkovať pokiaľ možno na rovnakých verziách OS.[9]

2.4.3 3-vrstvová architektúra

Najčastejším príkladom implementácie viac-úrovňovej architektúry je 3-úrovňová architektúra (Three-tier architecture), na ktorej princípe je založených veľa webových a niektoré desktopové aplikácie.

Zloženie:

1. Prezentačná úroveň - Presentation tier *(Zobrazuje informácie pre užívateľa (napr, GUI), slúži na interakciu s užívateľom, nie však na spracovanie dát)*
2. Logická úroveň - Logic tier (application tier, business logic, middle tier) *(Obsahuje aplikačnú logiku - vnútornú logiku, funkcie, zpracovanie dát...)*
3. Dátová úroveň - Data tier *(Obsahuje dáta - najčastejšie databáza, sieťový súborový systém, webová služba, alebo iná aplikácia. Táto vrstva okrem uchovávania a sprístupnenia dát tiež zaručuje ich konzisteniciu)*

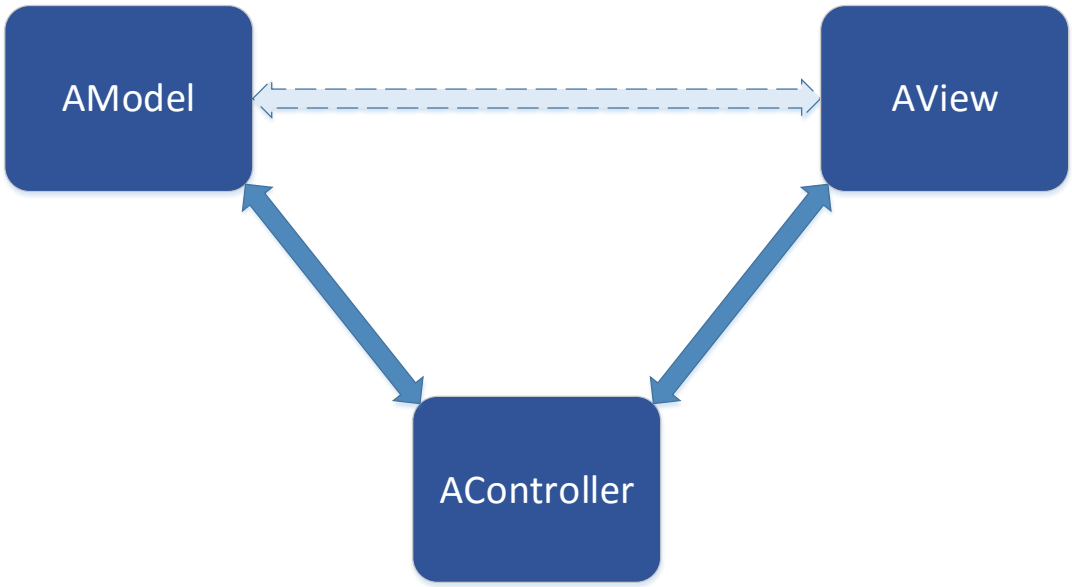


Obr. 2.2: 3-úrovňová architektúra

Na základe tejto 3-vrstvovej architektúry je založený všeobecný vzor návrhu, a to architektúra Model-View-Controler (MVC), z ktorého vychádzajú prezetačné modely Model-View-Controler (MVC), Model-View-Presenter (MVP) a model Model-View-ViewModel (MVVM).[24][8]

2.5 Architektúra Model-View-Controller

Architektúra MVC rozdeľuje aplikáciu na 3 logické vrstvy (osvojilo sa označenie triády) tak, aby bola možná ich úprava s minimálnym vplyvom na ostatné vrstvy aplikácie. Tieto 3 vrstvy sa nazývajú *Model*, *View* a *Cotroller*, ale pre rozlíšenie oproti prezentačným vzorom budem používať pojmy *AModel*, *AView* a *AController*.



Obr. 2.3: Architektúra Model-View-Controller

AModel prezentuje vnútornú logiku aplikácie a je zodpovedný za ukladanie a načítavanie dát. *AView* je zodpovedný za vykresľovanie užívateľského rozhrania a interakciu s užívateľom. *AController* sa stará v aplikácii o tok udalostí aplikácie a obecnú aplikačnú logiku. V architektúre MVC je väzba medzi jednotlivými vrstvami iba všeobecná a líši sa od jednotlivých variácii. Všeobecne je definovaná obojsmerná väzba medzi všetkými troma vrstvami navzájom tak, ako je znázornené na obrázku 2.3. Väzba medzi vrstvami však môže byť aj jednosmerná a medzi *AModel* a *AView* nemusí existovať vôbec.

Keďže architektúra je iba všeobecný vzor, tak pri reálnom vývoji nastáva niekoľko otázok, ktoré architektúra rieši len veľmi všeobecne a nejednoznačne:

- Prezentačná logika je *AView* alebo *AControler*?
- Má mať *AView* priamu väzbu na *AModel*?
- Patrí vnútorná logika aplikácie do AControlleru alebo do *AModelu*?

- Aké úzké má byť previazanie medzi *AView* a *AControllerom*?

Z týchto otázok si môžeme vyvodiť ďalšiu otázku, ktorá znie - Je teda architektúra MVC užitočná, aj keď nevie dať vývojárom jednoznačnú odpoveď na konkrétne problémy? Pre odpoveď na tieto otázky je nutné si najskôr definovať pojmy architektúra MVC a MVC prezentačný vzor (architektonický vzor), ktoré sa veľmi často zamieňajú.

MVC je taktiež veľmi často interpretovaný ako návrhový vzor, je však presnejšie ho definovať ako návrhový vzor zložený z návrhových vzorov (vzor *Observer*,vzor *Strategy*) - preto je presnejšie používať termín Architektúra MVC. Z architektúry MVC vychádzajú rôzne variácie - architektonické vzory (prezentačné vzory) ako Model-View-Cotroller, Model-View-Presenter a Model-View-ViewModel, ktoré presne definujú jednotlivé vrstvy a väzby medzi nimi a priamo odpovedajú na otázky položené v predchádzajúcom odseku. Podľa architektonických vzorov je možné navrhnúť aplikáciu rozdelenú na viac vrstiev. V prezentačnom vzore MVVM teda *Model* prezentuje *AModel*, *View* prezentuje *AView*, a *ViewModel* prezentuje *AController*.[3]

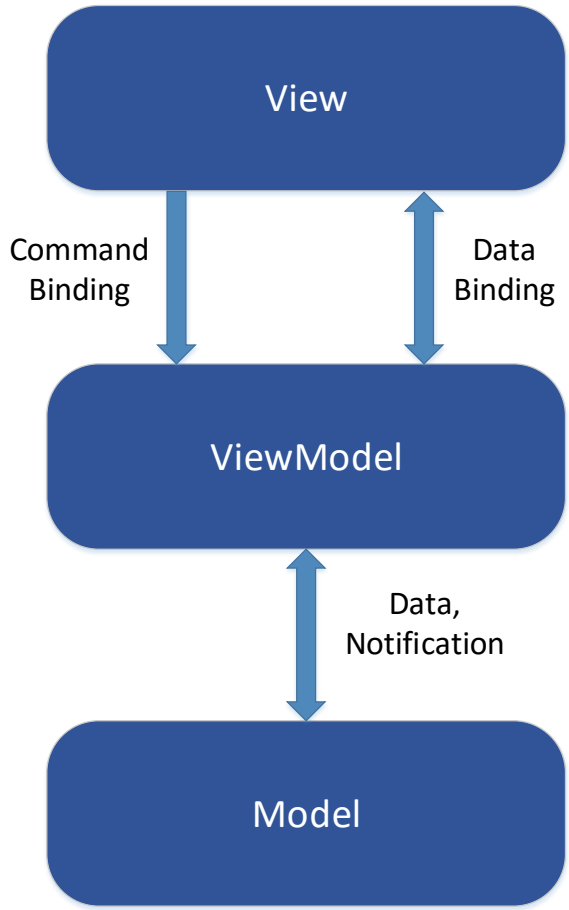
2.6 Prezentačný vzor Model-View-ViewModel

Model-View-ViewModel, označovaný aj ako Presentation model je prezentačný vzor, ktorý bol navrhnutý softvérovým architektom Johnom Gossmanom pre WPF aplikácie. Samotné WPF bolo navrhnuté tak, aby plne odpovedalo prezentačnému vzoru MVVM a plne sa vyžíval obojsmerný *Data-binding* a *Command*, čiže interakcia s užívateľom bude prebiehať práve na vrstve *View* - náhrada za riadenie udalosťami (*Controller* pri architektúre MVC).

Prezentačný vzor MVVM teda vychádza z architektúry MVC a skladá sa z 3 základných častí, ktoré sa volajú *Model*, *View* a *ViewModel*. *View* a *Model* spolu nemajú žiadnu priamu väzbu, ale komunikujú spolu cez medzivrstvu *ViewModel*.

2.6.1 Model

Najnižšou vrstvou je *Model* a označuje všetko čo je „pod úrovňou“ užívateľského rozhrania, čiže prístup k dátam, notifikácie o zmene, volanie back-end logiky, herný algoritmus, atď. *Model* nesmie nič vedieť o stave ovládacích prvkov.



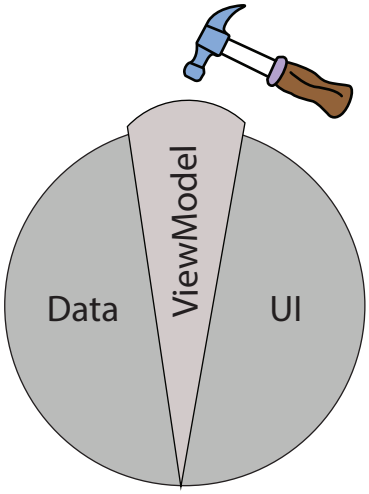
Obr. 2.4: Architektúra Model-View-ViewModel

2.6.2 View

Najvyššou vrstvou je *View*, ktorý má za úlohu vzhľad užívateľského rozhrania. Prakticky je *View* vytvorené popisom rozhrania pomocou jazyka XAML (ktorý je vytvorený na základe hypertextového značkovacieho jazyka XML) a s minimálnym množstvom code-behind. *View* komunikuje s *ViewModelom* pomocou *Commands* a *Data-binding-u*, čo je hlavná výhoda aplikácie napísanej pomocou návrhového vzoru MVVM. *View* sa označuje aj ako UI, formulár alebo prezentačná vrstva.

2.6.3 ViewModel

Spájajúca vrstva sa nazýva *ViewModel* (niekedy sa označuje aj ako *Model of a View*) a prostredníctvom nej spolu komunikujú *Model* a *View*. Táto vrstva tiež uchováva stav aplikácie.



Obr. 2.5: Oddelenie dát a UI pomocou ViewModel-u

ViewModel je samostatná trieda bez znalosti *View*, ktorá býva zväčša umiestená oddelene a má 2 základné '"kamene'". Kolekciu `ObservableCollection<T>`, ktorá implementuje rozhranie `INotifyCollectionChanged`, ktorá hlási či je pridaný alebo odobraný nejaký prvok danej kolekcie a rozhranie `INotifyPropertyChanged`, ktoré popisuje udalosť, ktorá nastane, ak sa zmení niektorá z vlastností *ViewModelu*. Pre stručnosť si teda môžeme poznamenať, že `INotifyCollectionChanged` nám aktualizuje *View* a `Collection<T>` nám aktualizuje *ViewModel*.

***ViewModel* teda:**

- Udržuje informácie o stave aplikácie.
- Je pomocou *Data-bindingu* prepojený s ovládacími prvkami a poskytuje im svoj obsah.
- Filtruje dáta v závislosti na stave aplikácie.
- Reaguje na príkazy z UI (stlačenie prvku `Button`, klávesovej skratky, atď.) pomocou *Command*-ov
- Predstavuje prostredníka medzi *Model*-om a *View*-om ktorý prispôsobuje *Model* pre potreby *View*-u.
- Je zodpovedný za stav *View*, bez nutnosti priameho zisťovania hodnôt pre jednotlivé prvky vo *View*

- Volá *Model* alebo jeho služby cez jeho/ich interface.
- Vystavuje verejné vlastnosti, ktoré sú nabindované cez XAML - aby však mohlo dochádzať ku komunikácii medzi *View* a *ViewModel* je vhodné, aby boli tieto verejné vlastnosti publikované ako notifikačné.
- Jednoduché typy sú vystavované pomocou typu `DependencyProperty`, prípadne je na nich implementované rozhranie `INotifyPropertyChanged`.
- Kolekcie prvkov sú vytvárané pomocou `ObservableCollection<T>` pre príkazy, ktoré vedú k volaniu metód *Model*-u (služieb) a sú sprístupnené pomocou *Command* cez implementované rozhranie `ICommand`.

Prezetačný vzor MVVM nám teda uľahčuje testovateľnosť aplikácie a pomáha udržať čistý kód skrz všetky vrstvy aplikácie. Môžeme si napr. vytvoriť inštanciu *ViewModel*-u (bez skutočného UI) a spustiť proti nej Unit testy. Nezávislosť na platforme - rovnaký *ViewModel* je možné použiť na viacerých platformách (pokiaľ podporujú XAML), pretože *ViewModel* nemá žiadnu závislosť na konkrétnej platforme. Nezávislosť na realizácii UI - pokiaľ sa rozhodnem zmeniť vzhľad aplikácie *View* - je to jednoduché, pretože *ViewModel* nemá žiadnu závislosť na *View*. *ViewModel* je tzv. *Value Converter*, čo znamená že je zodpovedný za konvertovanie dát objektov z *Model*-u do takej podoby, že objekty sú ľahko ovládateľné a spracovateľné. *ViewModel* dokáže eliminovať až 99,9% potrieb zmien formátu. Môžeme tiež povedať že *ViewModel* drží pohromade celú zobrazovaciu logiku. *ViewModel* môže implementovať prístup do back-end logiky v prípadoch že View podporuje možnosť zmeny dát.[10][30][31]

Inštancia *ViewModel*-u sa zväčša vytvára v code behinde *View*-u, menej často v *Model*-i.

2.7 UML Diagramy

Pri vývoji softvéru bol jednou z hlavných príčin neefektívnosti pri navrhovaní a rozširovaní aplikácii nejednotnosť výrazových prostriedkov, ktorými sa popisovala špecifikácia, návrh, architektúra, testovanie, modely životného cyklu, atď. Tak vznikali rôzne metodiky, ktoré však boli nejednotné, až pokým v 90. rokoch minulého storočia firma Rational Software nevytvorila základ štandardu Unified Modeling Language (UML).

V reakcii na to štandardizačná skupina OMG (Object Management Group), do ktorej patria firmy ako (Hewlett-Packard, IBM, Sun Microsystems, Apple Computer, Microsoft, Oracle), prijala ako štandard UML vo verzii 1.1, a do ktorej začlenila aj ďalšie prvky z rôznych metodík. Dôvodom štandardizácie bolo zjednodušenie

komunikácie a dokumentácie v projektoch, kde sa pracuje s rôznymi technológiami. UML môžeme teda definovať ako grafický jazyk, ktorým vytvárame návrh, následnú špecifikáciu a dokumentáciu vyvíjaného softvéru. [13]

S jeho pomocou je možné odhadnúť cenu v počiatočnej fáze projektu a na základe analýzy tiež pomáha pri vytváraní štruktúry riešenia. Taktiež je veľmi užitočným nástrojom pri komunikacii s klientom, keďže mu dokážeme veľmi jednoducho graficky priblížiť dané riešenie.[32] V súčasnej dobe sa UML skladá zo 14 diagramov a nachádza vo verzii 2.5 a je z júna 2015.[34]

Štrukturálne diagramy (Structural UML diagrams)

- Diagram tried (Class diagram)
- Diagram komponentov (Component diagram)
- Diagram zloženej štruktúry (Composite structure diagram)
- Diagram nasadení (Deployment diagram)
- Diagram balíčkov (Package diagram)
- Diagram objektov - Diagram inštancii (Object diagram)
- Diagram profilov (Profile diagram)

Diagramy chovania (Behavioral UML diagrams)

- Diagram aktivít (Activity diagram)
- Diagram komunikácie (Communication diagram)
- Diagram interakcií (Interaction overview diagram)
- Diagram sekvencí (Sequence diagram)
- Diagram stavu (State diagram)
- Diagram časovania (Timing diagram)
- Diagram použitia (Use case diagram)

3 TEORETICKÝ VÝVOJ APLIKÁCIE

3.1 Metodiky vývoja softvéru

V počiatkoch vývoja softvéru sa používal postup "Code and fix model", ktorý je jednoduchý avšak neexistuje pri ňom návrh a požiadavky na dané softvérové riešenie. Jeho podstatou je priama implementácia a následná oprava chýb v riešení (modeli). Tento model sa tiež spoliehal na presné definovanie požiadaviek zákazníkom na dané softvérové riešenie už v počiatku vývoja. V dnešnej dobe je takýto vývoj zdĺhavý, neefektívny a v praxi neuskutočniteľný, pretože sa vytvárajú robustné softvérové riešenia, kde je potrebná spolupráca v tíme, relatívne rýchly vývoj funkčného prototypu a pružná reakcia na zmeny v zadaní klienta.

Preto vznikli metodiky, ktoré komplexne riešia postupy pri vývoji softvérového riešenia. Tieto metodiky doporučujú postup pri návrhu, komunikáciu so zákazníkom, formy mítingov, štýl implementácie (či sa tvorí funkcionalita po častiach alebo ako celok), rozdeľovanie vývojárov do skupín a celkov, rozdeľovanie zodpovednosti za danú funkcionalitu medzi týmy alebo jednotlivcov a môžu doporučovať napríklad aj rozmiestnenie vývojárov v kancelárii. V súčasnej dobe sa často prelínajú role pri vývoji aplikácii, najmä pri malých týmoch, kde musí jeden člen zvládať napr. tzv. frontend aj backend.

Prístupy metodík vo vývoji softvérového riešenia:

- **Vodopádový prístup** - Je to sekvenčný vývojový proces, ktorý postupuje vývojom od zadania požiadavky, cez návrh riešenia, implementácie, testovania po údržbu. Tento model bol popísaný dr. Winston W. Royce-om, ktorý ho popísal ako model nefungujúceho, zle navrhnutého modelu.[12]
- **Prototypový prístup** - Nie je samostatná metodika, ale skôr vývojový podproces využívané v určitej metodike (často používaný s iteratívnym prístupom), v ktorom sa vytvárajú funkčné prototypy častí systému.
- **Inkrementálny prístup** - Je to kombinácia sekvenčného a iteratívneho procesu, kde je prvotný koncept vykonaný sekvenciami malých „vodopádov" a následne iteratívnym prototypovým prístupom vytvárame konečné riešenie. Tento prístup sa dá definovať aj ako vývoj pomocou funkčných častí, ktoré pridávame k existujúcim častiam a tým rozširujeme funkcionalitu celého riešenia. Je ešte nutné poznamenať, že v iteratívnom vývoji veľmi často dochádza k prepisovaniu „zlého" kódu namiesto jeho obchádzania.
- **Špirálový prístup** - Je to proces, v ktorom sa pre každý cyklus vykonajú štyri základné fázy v špirále, a to analýza (stanoví cieľ a rozsah cyklu), vyhodnotenie

(identifikácia a riešenie rizík), vývoj (implementácia) a plánovanie (plánovanie rozsahu ďalšieho cyklu).

3.1.1 Metodika Rapid Application Development

Je to metodika, ktorá v sebe spája iteratívny a prototypový prístup. Rozdiel oproti inkrementálnemu prístupu je v úlohe vytvárať miesto prototypov častí systému počas iterácii prototypy celého systému. V RAD je nutné mať aktívne zapojeného zákazníka a oboznamovať ho s postupom pomocou priebežne vytváranej dokumentácie a demonštráciou prototypov systému. Ak sa iterácia nestíha dokončiť v danom časovom intervale, upúšťa sa od požiadaviek miesto posunutia termínu.

3.1.2 Objektovo orientované metodiky

Táto metodika spája dáta a funkcie do objektov, ktoré pomocou predpísaných modelov čo najvernejšie zobrazujú realitu. K základným modelom patria statické modely (napríklad Class diagram) a dynamické modely (napríklad Use Case diagram).

Medzi objektovo-orientované metodiky patria:

- **Rational Unified Process (RUP)** je metodika vytvorená spoločnosťou Rational Software Corporation (teraz spoločnosť IBM), ktorá je primárne vhodná pre rozsiahlejšie projekty a väčšie vývojárske tímy, keďže kladie dôraz na analýzu, plánovanie, riadenie zdrojov a dokumentáciu. Nevýhoda tejto metodiky je tá, že je komerčná.
- **Unified Software Development Process** je metodika, o ktorej sa dá povedať že je bezplatným klonom metodiky RUP.

3.1.3 Agilné metodiky

Pod pojmom agilná metodika alebo agilný vývoj si môžeme predstaviť techniky, ktoré sú založené na iteratívnom a inkrementálnom vývoji. Agilny vývoj si teda môžeme predstaviť tak, že aplikácia je vyvíjaná po fázach - iteráciách, v ktorých sa postupným inkrementovaním funkcionality vyvíja výsledné softvérové riešenie, kde sú dielčie úlohy rozdeľované týmom, ktoré sú samo-organizované (self-organizing) a viac-odborové (cross-functional). V praxi to znamená, že metodika má rýchly a flexibilný (adaptívny) vývoj, vďaka čomu sme schopný rýchlo reagovať na zmeny v zadaní projektu, dodržovať časovo ohraničené intervaly. Podmienkou sú však iterácie vývojového cyklu. [15] Oproti ostatným vyššie spomínaným metodikám je agilný vývoj orientovaný skôr na rýchly vývoj a stručnú dokumentáciu ako na byrokratickú réžiu projektu. Z toho dôvodu patrí táto metodika medzi najrýchlejšie, čo sa týka rýchlosti implementácie požiadavok do výsledného softvérového riešenia.[14]

Najčastejšie používané agilne metodiky:

- **Extrémne programovanie (eXtreme Programing)**
- **Štíhly vývoj (Lean development)**
- **Vývoj riadený testami (Test Driven Development)**
- **SCRUM Development process**

Veľmi často sa pri agilnych metodikách skloňuje pojem Unit Testing. Jedná sa o automatizované testovanie jednotlivých častí implementácii (tzv. jednotiek alebo modulov), ktoré je možné testovať nezávisle na zbytku implementácie. Unit Testing je jednou z hlavných častí metodík eXtreme Programing a Test Driven Development.

Extrémne programovanie (eXtreme Programing)

Je vývoj, v ktorom sa využíva realistický prístup vývoja softvéru a všetko sa dotahuje do "extrémov", napr. maximálna jednoduchosť - aká najjednoduchšia implementácia vyhovuje požiadavkám, postupné prepracovávanie architektúry, ak sa osvedčuje, tak aj testovateľnosť, krátke iterácie a častá komunikácia so zákazníkom, párové programovanie - za jedným počítačom sedia dvaja programátori, kde jeden píše a druhý kontroluje.

Štíhly vývoj (Lean development)

Je vývoj, ktorý eliminuje zdroje plytvania v priebehu „výrobného" procesu, rozhodnutia sa prijímajú tak neskoro, ako je možné a sú určované dopytom, nespolieha sa na „best practices" riešenia, ale overuje ich aj v praxi a konštruktívne kritizuje „výrobný" proces, čo napomáha zlepšovaniu procesov. Táto metodika má napriek eliminácii plytvania za úlohu poskytnúť zákazníkovi maximálnu kvalitu daného softvérového riešenia. Táto metodika sa veľmi dobre uplatňuje pri údržbe.

Vývoj riadený testami (Test Driven Development)

Je vývoj, kde je najväčší dôraz kladený na testovanie, a tým je zaistená maximálna kvalita dodaného softvérového riešenia. Myšlienka Test-Driven Development (TDD) je vytvorenie testu pred samotnou implementáciu a implementácia iba takého množstva kódu, aby prešla testom. Prístup je teda opačný ako pri tradičných metodikách, neopravujeme chyby, ale predchádzame im. Po úspešnom testovaní nastáva fáza refaktorizácie kódu. Veľkou výhodou tejto metodiky je možnosť jej integrácie s inými metodikami, pričom najčastejšie sa v dnešnej dobe používa s metodikou SCRUM.[14]

SCRUM Development process

V súčastnosti sa jedná o najrozšírenejšiu agilnou metodiku, ktorá prebieha v pevne daných časových intervaloch (Sprints). Každý sprint trvá maximálne mesiac a po dokončení každého sprintu by mal byť tím schopný dodať funkčný a otestovaný medziprodukt, ktorý by mal byť potencionálne schopný pre nasadenie na produkcii. Každý sprint sa skladá zo skupiny **SCRUM tímov** (nie je podmienka, môže byť aj 1 tím), časových rámcov (**Time-Boxes**), **artefaktov** a súhrnu pravidiel vzťahov medzi časťami SCRUM-u. Každý SCRUM tím je zostavený tak, aby využíval samo-organizáciu a bola zoptimalizovaná flexibilita a produktivita dosahovania cieľov. V každom SCRUM tíme musia byť nasledovné role:

- **SCRUM Master** - rola, ktorá je zodpovedná za pochopenie, optimalizáciu a dodržiavanie procesu.
- **Product Owner** - rola, ktorá zodpovedá za dosiahnutie maximálnej hodnoty vytvorenej tímom (zákazník, alebo jeho zástupca).
- **Team Member** - členovia, ktorí implementujú dane riešenie.

Metodika SCRUM využíva tzv. Time boxes (časové rámce) k vytvoreniu pravidelnosti vývoja. Time box obsahuje míting pre plánovanie - **(Release Planning Meeting)** a míting plánovania **sprint**-u (**Sprint Planning Meeting**). Každý sprint ďalej obsahuje tri druhy mítingov:

- **Daily Scrum** - alebo denný míting, ktorý slúži vedúcim ku kontrole a miernej optimalizácii práce jednotlivých členov. Schôdzka by sa mala konať v stoji, čím sa zabezpečí jej rozumná dĺžka trvania.
- **Sprint Review and Sprint Planning**, hodnotenie a plánovanie slúži ku kontrole postupu práce voči cieľu a optimalizácii prostriedkov na konci daného sprint-u.
- **Sprint Retrospective** míting, ktorý sa uskutočňuje na zhodnotenie dokončeného **sprint**-u, stanovenie prípadných zmien v projekte a snaží sa ďalší **sprint** zefektívniť.

SCRUM využíva v každom sprint-e Time boxes a tzv. artefakty:

- **Product Backlog** je zoznam požiadaviek na dané softvérové riešenie, ktoré sa vykonávajú v danej postupnosti, spisuje ich **Product owner** a podľa potreby v ňom môže jednotlivé položky zoznamu uprednostnovať pred ostatnými.
- **Sprint Backlog** má rovnakú úlohu ako **Product backlog**, ale na úrovni **sprint**-u a má za úlohu pretaviť výsledok **sprint**-u do nasaditeľného riešenia, vytvára ho však **SCRUM Master**.
- **Release Burndown** je to graf, ktorý slúži na grafické znázornenie zostávajúcej práce v čase, aktualizuje sa pomocou **Sprint Burndown**-ov.

- **Sprint Burndown** Má rovnakú úlohu ako **Release Burndown**, ale len na úrovni **sprint**-u, pravidelne ho aktualizuje **SCRUM Master**.

Každý sprint obsahuje nasledovné časti:

- **Stories** - jednotlivé položky z **Product Backlog**-u, ktoré vytvára **SCRUM Master**.
- **Tasks** - pridelené úlohy pre **Team Member**-ov, ktoré prideľuje **SCRUM Master** podľa ich zamerania.

V krátkosti sa to dá zhrnúť tak, že celý vývoj je rozdelený do **sprint**-ov, ktoré sú rozdelené do **Stories**, ktoré sú položky z **Sprint Backlog**-u, ktoré určuje **SCRUM Master** na základe požiadaviek **Product Owner**-a, ktoré sa nachádzajú v **Product Backlog**-u. **Stories** ďalej porozdeľuje **SCRUM Master** jednotlivým **Team Member**-om **Tasks**, ktoré obsahujú zadania jednotlivých dielčích úloh. Projekt sa zahajuje **Release Planning** mítingom, kde sa určí počiatočný cieľ projektu. Každý **sprint** sa zahajuje **Sprint Review** mítingom, kde sa naplánujú jednotlivé **Stories** do **Tasks** a ukončuje sa zhodnotením postupu na **Sprint Retrospective** mítingu. Každý deň sa zahajuje **Daily Scrum** mítingom, kde **SCRUM Master** zhodnotí postup a zadá prioritne **Task**-y pre jednotlivých **Team Member**-ov.[14]

3.2 Framework Catel

Tento framework je zameraný na vývoj desktopových a webových aplikácii, ktoré používajú prezentačný vzor MVVM (WPF, Silverlight, Windows Phone, WinRT, Xamarin.Android and Xamarin.iOS) a vzor MVC (ASP.NET MVC). Veľkou výhodou frameworku je jeho modularita a to, že je možné používať jeho časti nezávisle na ostatných „moduloch“(Catel.MVVM, Catel.Controls, Catel.EntityFramework6, Catel.Extension.Prism, Catel.Serialization.Json). Tento framework je teda vhodný pre použitie skrz celú aplikáciu na všetkých jej vrstvách a tým rapídne urýchľuje jej vývoj. Framework využíva techniku Dependency injection, ktorá je dôležitá pri vytváraní viac vrstvových aplikácii. Táto technika poskytuje spôsob spracovania závislosti medzi jednotlivými objektami. Napríklad objekt, ktorý spracováva informácie o zákazníkoch môže závisieť na inom objekte (alebo objektoch), ktorý pristupuje k dátovému úložisku a na základe toho, mu umožní vykonať nejaké akcie ako napríklad aktualizácia ponuky, zmazanie užívateľa, atď.[21]

Tab. 3.1: Základné informácie o Frameworku Catel

Oficiálny názov	Catel
Aktuálna „stable“ verzia	Catel 4.4.0
Aktuálna „unstable“ verzia	Catel 4.5.0-unstable0281
Dátum vydania „stable“ verzie	27. septembra 2015
Distribúcia	Open-Source
Oficiálne stránky	`http://catelproject.com/` `https://github.com/catel/catel`
Spôsob použitia	NuGet package
NuGet Package url	`https://www.nuget.org/packages/Catel.MVVM/`

Vlastnosti:

- Podporuje Silverlight, WPF, Windows Phone a Xamarin aplikácie .
- Podpora prezentačných vzorov MVVM a MVC.
- Podporuje dynamickú Dependency injection.
- Podpora špecifických WP7 ViewModel services.

Moduly frameworku Catel, ktoré predstavím v nasledovných podkapitolách sú zoradené od najnižšej položenej vrstvy aplikácie (Data tier) po najvyššiu (Presentation tier).

- Catel.Core
- Catel.EntityFramework6
- Catel.MVVM
- Catel.Controls

3.2.1 Catel.Core

Je to tzv. jadro frameworku ktoré obsahuje užitočné triedy ako:

- `WeakEventListener` -> trieda zabraňujúca tzv. „memory leaks“, čiže zbytočnému čerpaniu systémových prostriedkov.
- `MessageMediator` -> trieda pre vytvorenie návrhového vzoru Mediátor.
- `Memento` -> trieda implementújúca vzor *Memento*
- `ArgumentClass` -> trieda pre kontrolu vstupu `Argument.IsNotNull(argument)`, atď.
- `ModelBase` -> abstraktná trieda vhodná pre použitie v *Model*-ových triedach, implementuje napr. `INotifyPropertyChanging`, vlastnosť `IsDirtyProperty`, ďalej rôzne validátory hodnôt, implementuje serializovatelnosť objektov atď.

3.2.2 Catel.EntityFramework6

Modul pre spoluprácu s Entity Framework-om 6 (Existuje aj verzia pre EF5) ktorý obsahuje triedy:

- `EntityRepositoryBase` -> trieda implementujúca vzor *Repository*.
- `UnitOfWork` -> trieda implementujúca vzor *Unit of Work*.

3.2.3 Catel.MVVM

Modul umožňuje oproti iným MVVM frameworkom komunikáciu medzi *View* a *Model*-mi pomocou jednoduchých atribútov a nie sú potrebné tzv. správy. Tento modul implementuje triedy pre jednoduchú prácu s *Command* a *Property*.

- `ViewModelBase` -> trieda ktorá implementuje ModelBase a eventy pre prácu s ViewModelom .
- `PropertyData` -> trieda ktorá zjednodušuje registráciu Property.
- `Command` -> trieda ktorá zjednodušuje registráciu *Command*-ov.

3.2.4 Catel.Controls

Rieši problémy s tzv. *User Control* vo MVVM. *ViewModel*-y pre *User Control* sú vytvárané za behu na základe kontextu *ViewModel*-u daného *View*, z ktorého ich voláme ako `UserControl<TViewModel>` a `DataWindow<TViewModel>`, ktorý však vytvára inštanciu nového okna. Obrovská výhoda v spojení s triedami odvodenými z `ViewModelBase` je jednoduché použitie *Dependency property*. Vo *View* používa svoju špecifickú *namespace* `catel`.

Porovnanie klasických zápisov vs. zápisy frameworkom

```
<!-- štandardný začiatok zápisu pomocou XAML-->
<Window
        x:Class="DESPREK.WPFAplication.View.UserLogin"/>
<!-- začiatok zápisu "View" frameworkom Catel pomocou XAML-->
<catel:Window
        x:Class="DESPREK.WPFAplication.View.UserLogin"/>
<!-- User Control tzv. DataWindow - vytvára nove 'okno'-->
<catel:DataWindow
        x:Class="DESPREK.WPFAplication.View.UserLogin"/>
<!-- User Control-->
<catel:UserControl
        x:Class="DESPREK.WPFAplication.View.UserLogin"/>
```

Časť II

Praktická časť

4 POŽIADAVKY NA SOFTVÉR (SOFTWARE REQUIREMENT SPECIFICATION)

4.1 Úvod

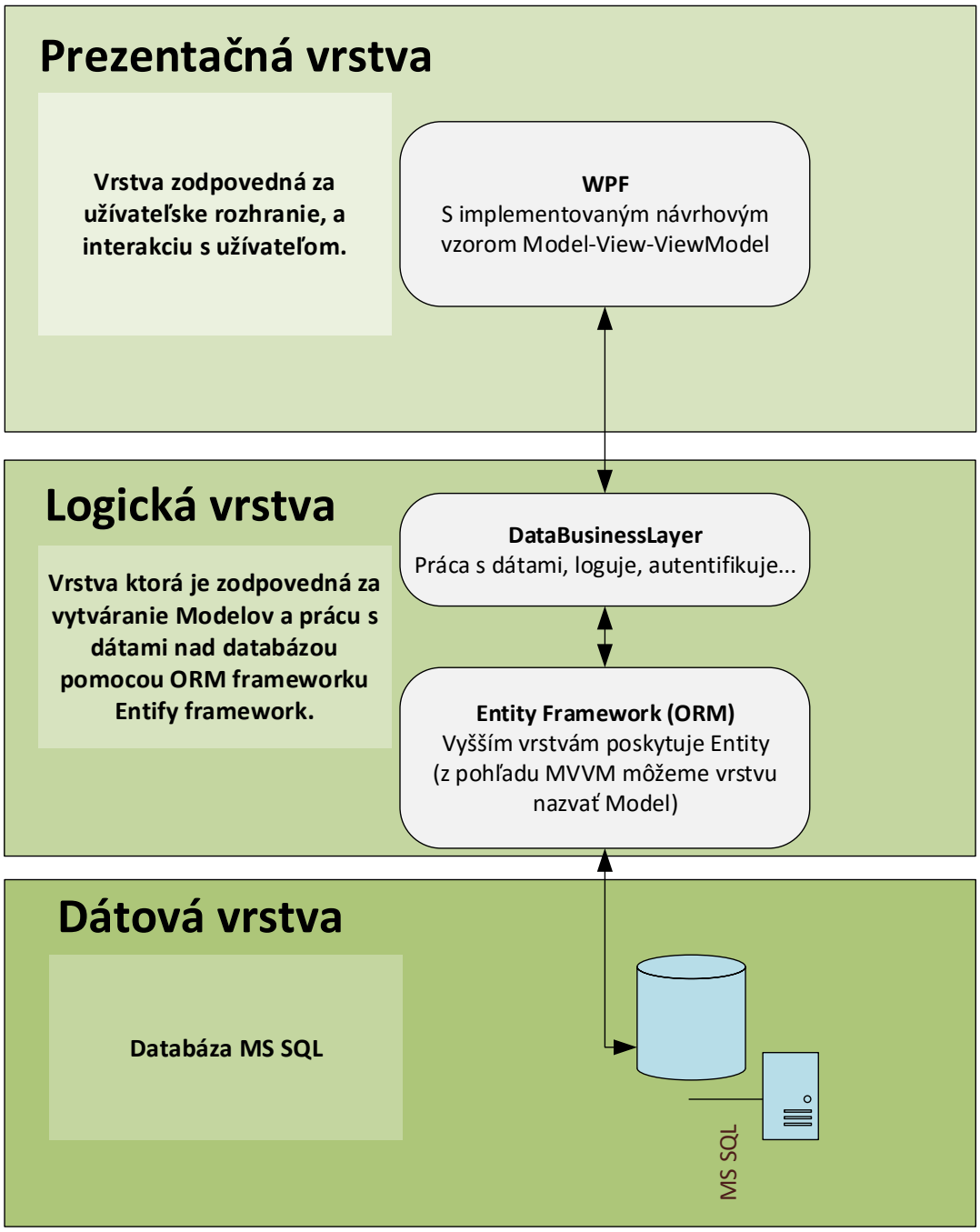
Účel tohoto dokumentu je predstaviť a detailne popísať desktopovú aplikáciu DESPREK - Databázu elektronických súčiastok. Dokument ďalej podrobne popisuje požiadavky, rozdelenie a úlohy jednotlivých vrstiev, návrh databáze a užívateľského rozhrania.

4.1.1 Rozsah

Desktopová aplikácia bude slúžiť ako databáza elektrotechnických súčiastok v konštrukčnej dielni. Aplikácia slúži na vyhľadávanie podľa špecifických filtrov a jednoduchú zmenu počtu kusov zmenu stavov jednotlivých súčiastok. Požadovaný prístup do aplikácie je na základne zamestnaneckého loginu a hesla. Každá súčiastka bude obsahovať základné informácie špecifické pre jej typ. Aplikácia bude obsahovať tri role (user, admin a superAdmin) s rozličnou funkcionalitou. Aplikácia bude napísaná pomocou WPF s použitím návrhového vzoru Model-View-ViewModel. Dáta budu uložené v MS SQL databáze a namapované do aplikácie pomocou ORM frameworku Entity Framework.

4.2 Rozdelenie aplikácie na vrstvy

Aplikácia bude rozdelená na 3 vrstvy, dátovú, logickú a prezentačnú. Dátová vrstva bude obsahovať MS SQL server. Logická vrstva bude obsahovať Entity framework, ktorý vytvára vyššiu úroveň abstrakcie dát a umožňuje pracovať s tabuľkami relačnej databáze ako s objektami (vytvára entity - Modely). Entity framework bude poskytovať entity vrstve DataBusinessLayer, ktorá bude ešte vyššou úrovňou abstrakcie a bude poskytovať prácu s dátami (CRUD), ako aj plne oddeľovať dátovú a prezentačnú vrstvu. DataBusinessLayer bude taktiež zabezpečovať logovanie činnosti a prístup k dátam na základe autentifikácie. Nasledujúca schéma zobrazuje vyššie popísané rozdelenie na vrstvy 4.1.



Obr. 4.1: Rozdelenie aplikácie DESPREK

4.3 Požiadavky na funkcionalitu

Pri spustení aplikácie sa zobrazí autentifikačný pohľad, na základe ktorého sa užívateľ prihlási. Na základe prihlásenia sú definované tri druhy pohľadov. Prvý a základný má slúžiť návrhárovi (rola User) na vyhľadávanie súčiastok podľa filtrov, druhý pohľad slúži skladníkovi (rola Admin) a zvyšuje funkcionalitu možnosťami ako pridávanie, editácia a mazanie súčiastok. Posledná rola je manažér (rola SuperAdmin), ktorá slúži na správu užívateľov.

4.3.1 Vyhľadávanie na základe filtrov

Filtre budú rozdelené do dvoch skupin. Prvá skupina je filter na základe typu súčiastky (kondenzátor, cievka, rezsitor, všetky typy). Druhá skupina je rozdelená na dve podskupiny, a to na všeobecné druhy filtrov ako (Názov, Katalógové číslo, Minimálna hodnota (požaduje sa možnosť zadávať hodnoty napr 6E-6)) a špecifické druhy filtrov pre rôzne typy súčiastok. Špecifické druhy filtrov sú uvedené v tabuľke špecifických filtrov 4.1. Pre filtre je tiež požadovaná validácia zadanej hodnoty (pre číselné filtre).

Tab. 4.1: Špecifické filtre

Druh súčiastky	**Filter 1**	**Filter 2**	**Filter 3**
Rezistor	Max. AC U	Max. menovitý výkon	Min. izolačné U
Kondenzátor	Max. DC U	Max. pracovná teplota	Min. životnosť
Cievka	Max. AC I	Materiál jadra	-

4.4 Dátove modely

Každá súčiastka bude obsahovať všeobecné vlastnosti uvedené v tabuľke všeobecných vlastností 4.2. Ku všeobecným vlastnostiam sú požadované vlastnosti podľa typu uvedené v tabuľkách pre model rezistor 4.4, model kondenzátor 4.3 a model cievka 4.5. Pri všetkých modeloch je požadovaná minimalizácia a konzistencia ukladaných dát. Niektoré vlastnosti nemusia obsahovať žiadnu hodnotu[1]. Ďalšie požadované modely sú model užívateľa 4.6, model logeru 4.7 a model správy pre oznámenie činnosti užívateľovi 4.8.

[1]*n/a pri vlastnosti znamená že vlastnosť može byť prázdna

Tab. 4.2: Všeobecné vlastnosti

Názov vlastnosti	**Príklad**
Unikátne katalógové číslo	R15 -rezistor, C15 - kondenzátor, L16 - cievka
Názov	PISM-3R3M-04
Hodnotu	1,5 μH, 1,5 kΩ, 8 μF
Zostávajúce kusy	min. 0
Datasheet *n/a	odkaz na pdf alebo docx-doc súbor
Obrázok *n/a	
Výrobcu	FASTRON
Toleranciu	15 %
Obal	SMD, SMD-MELF, vývodový
Spôsob použitia	radiálne / axiálne

Tab. 4.3: Model Kondenzátor (CapacitorModel) - špecifické vlastnosti

Názov vlastnosti	**Príklad**
Maximálne pracovné DC napätie	= 100 V
Typ	elektrolytický, sľudový
Rozteč vývodov	10 mm
Výška *n/a	10 mm
Priemer *n/a	ϕ 10 mm
Životnosť *n/a	7800h
Minimálnu pracovnú teplotu	-50°C
Maximálnu pracovnú teplotu	90°C

Tab. 4.4: Model Rezistor (ResistorModel)- špecifické vlastnosti

Názov vlastnosti	**Príklad**
Maximálne pracovné AC napätie	100 V
Izolačné napätie	100 V
Priemer	ϕ 15 mm
Výška *n/a	10 mm
Šírka *n/a	15 mm
Dĺžka *n/a	35 mm

Tab. 4.5: Model Cievka (InductorModel) - špecifické vlastnosti

Názov vlastnosti	**Príklad**
Maximálny pracovný AC prúd	100 A
Materiál jadra *n/a	ferit
DCR (odpor pri DC napätí)	1,5 μΩ
Priemer *n/a	ϕ 15 mm
Výška *n/a	10 mm

Tab. 4.6: Model užívateľ (UserModel)

Názov vlastnosti	**Príklad**
Meno	Fero
Priezvisko	Skladníkovič
Login	fero.s
Heslo	v DB bude hash Hesla

Tab. 4.7: Model logu (LoggerMessage)

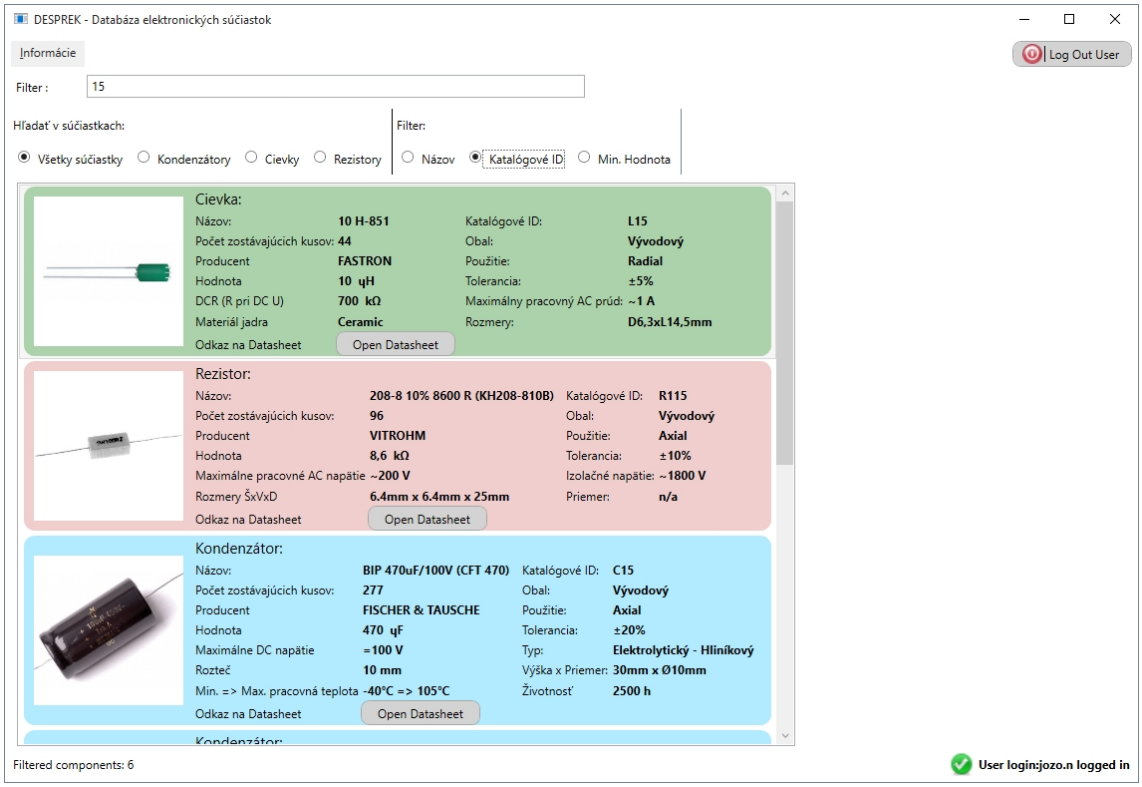
Názov vlastnosti	**Príklad**
Dátum	2016-05-15 19:10:18.683
Činnosť	Log in
Výsledok	Sucessfull
Popis	User login:superAdmin logged in

Tab. 4.8: Model správy oznámenia činnosti užívateľovi (ResultMessage)

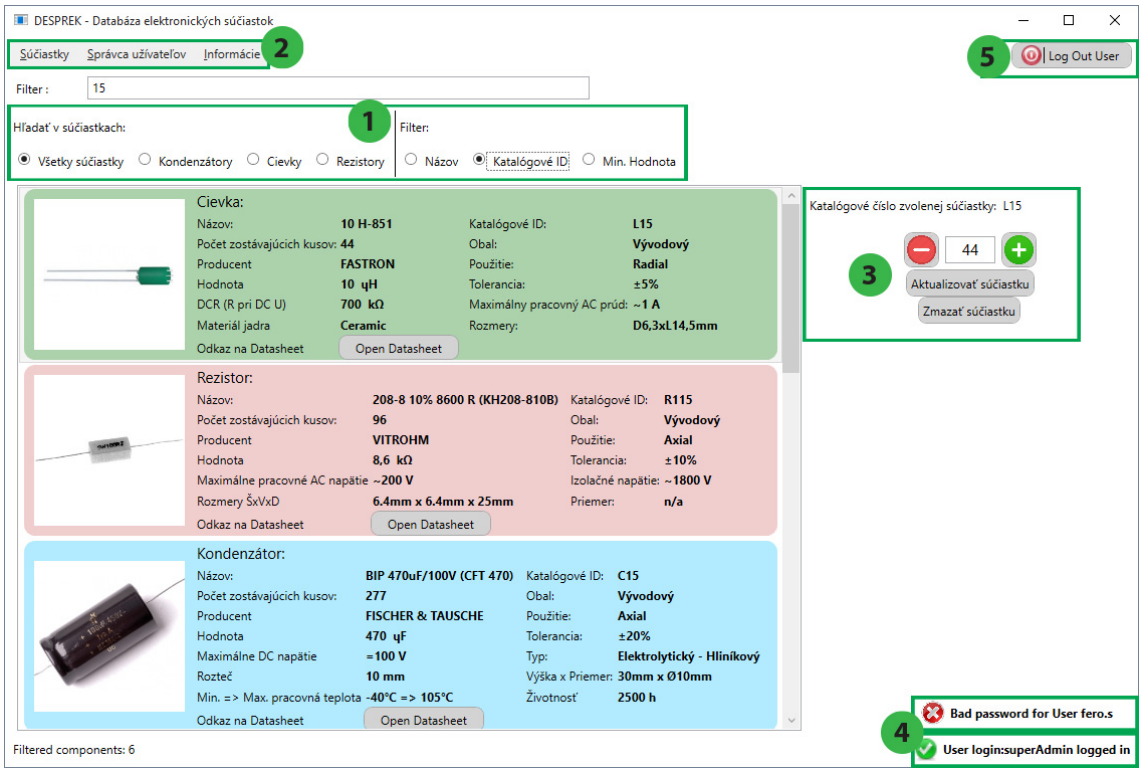
Názov vlastnosti	**Príklad**
Výsledok	Sucessfull
Popis	User login:superAdmin logged in

4.5 Návrh a popis funkcionality prvkov UI

Obr. 4.2: Prihlasovacie okno



Obr. 4.3: Základný vzhľad s možnosťami filtrácie (rola User)



Obr. 4.4: Vzhľad pre užívateľa s rolami superAdmin a Admin

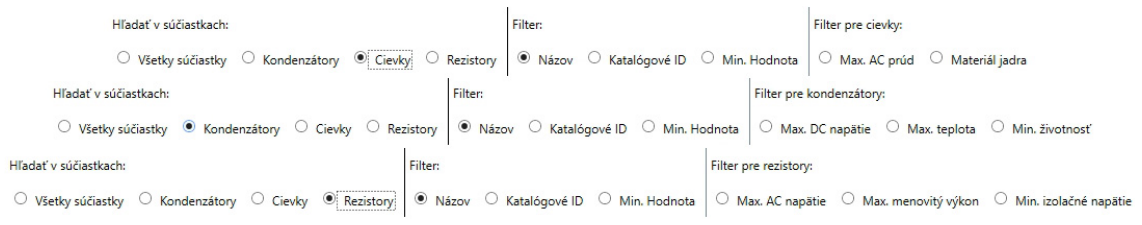
ui.1. Filtre pre rôzne druhy súčiastok. Filtre sú znázornené na pohľade číslo 4.5.

ui.2. Moduly aplikácie pre pridávanie súčiastok - pohľad číslo 4.6, a správu užívateľov - pohľad číslo 4.9 a nakoniec pre modul Informácie - pohľad číslo 4.13.

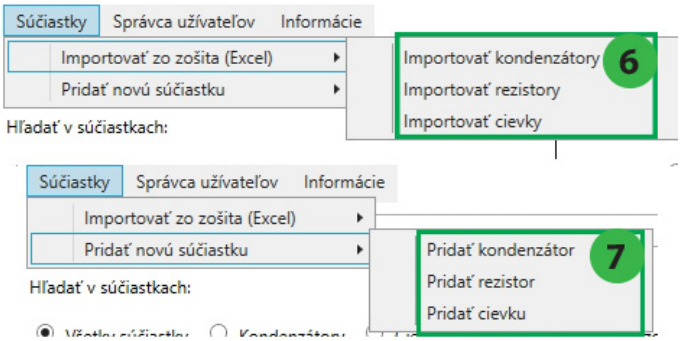
ui.3. Modul s modifikáciou počtu kusov, kde sa zmeny uložia pri ukončení aplikácie alebo tlačidlom Aktualizovať súčiastku. Tlačidlo zmazať súčiastku zmaže súčiastku a aktualizuje databázu.

ui.4. Oznámenia činnosti pre užívateľa - úspešná vs. neúspešná operácia.

ui.5. Odhlásenie užívateľa z aplikácie, ktoré preruší pripojenie na databázu, avšak ponechá aplikáciu vo východzom stave s prihlasovacím oknom - pohľad číslo 4.2.



Obr. 4.5: Filtre pre rôzne typy (**1.** bod)



Obr. 4.6: Modul pre pridávanie (**2.** bod)

ui.6. Príkazy, ktoré budú otvárať dialógové okno s možnosťou vybrať súbor - požadované formáty(*.xls;*.xlsx) z priloženého súboru a následne asynchrónne importuje zvolené dáta daného typu súčiastky do databázy.

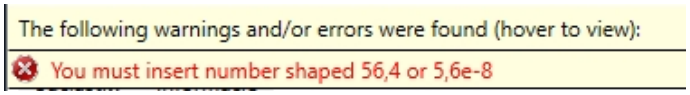
ui.7. Príkazy pre pridávanie súčiastok Po špecifickom type. Dizajn je uvedený v pohľade číslo 4.8. Nad pohľadmi sú požadované validátory hodnôt, prípadné prázdne hodnoty, tak ako je uvedené v tabuľkách číslo 4.2, 4.3, 4.4 a 4.5.

ui.8. Príkazy, ktoré otvoria dialógové okno a prekopírujú zvolený súbor do nastavenej lokácie, kde budú uložené obrázky - požadované formáty(*.bmp;*.jpg;*.gif) a datasheety súčiastok - požadované formáty (*.doc;*.docx;*.pdf).

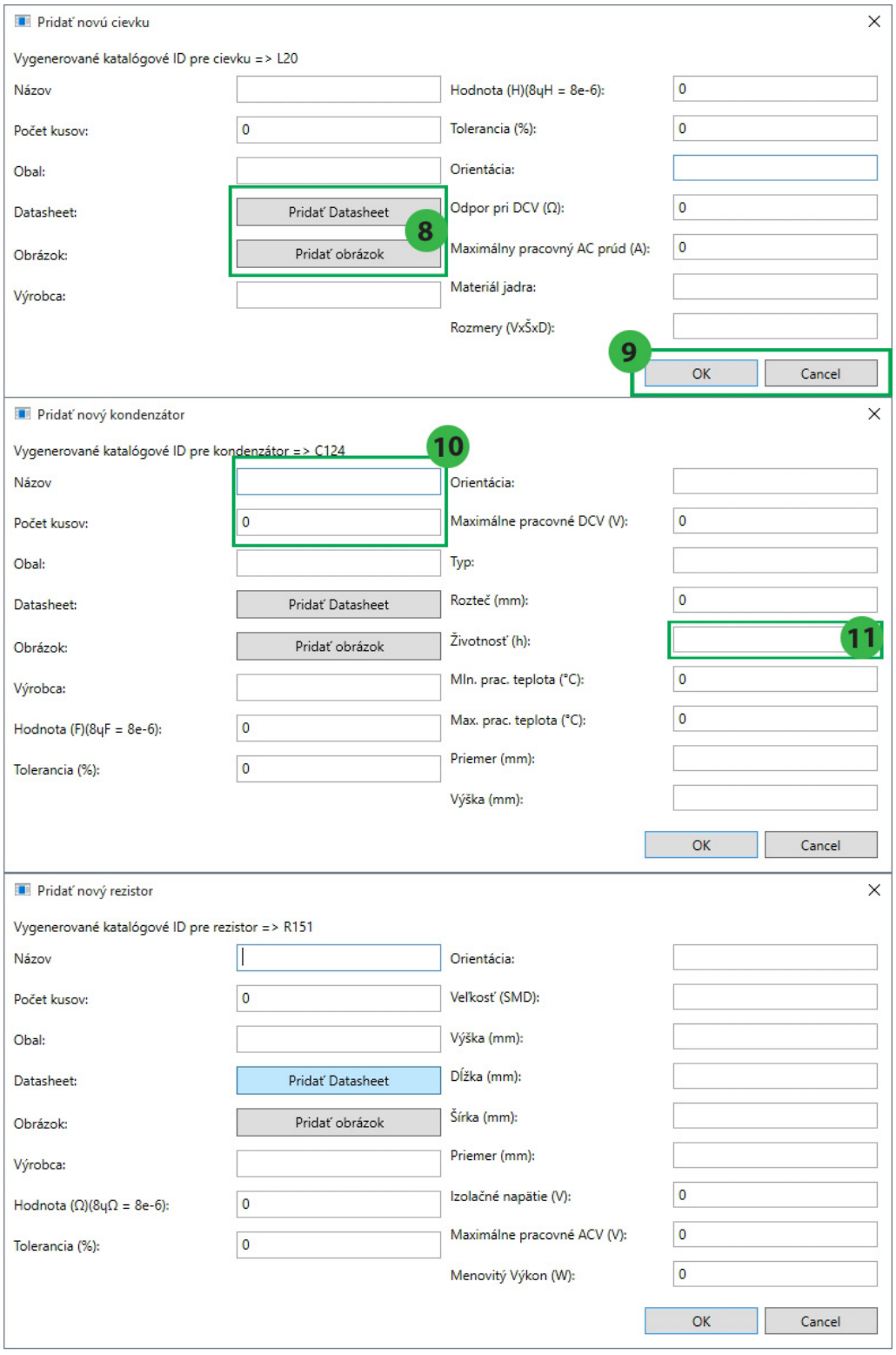
ui.9. Tlačidlo Ok ukladá zmeny, tlačidlo Cancel zahadzuje zmeny.

ui.10. Validátor bude fungovať na základe typu vlastnosti, do ktorej sa bude vkladať - pohľad chybne zadanú hodnotu čísla 4.7 . Ak sa bude jednať o číslo (float, double, int...), tak sa nastaví hodnota v poli nastaví na 0.

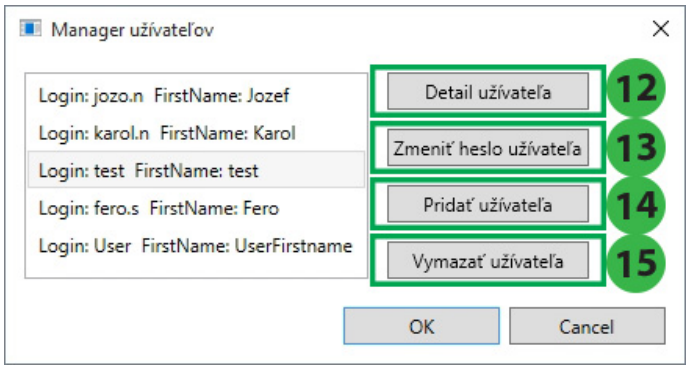
ui.11. Číselna hodnota, ktorá nemusí byť vyplnená, ak sa však vyplní, je nutná validácia na číslo - pohľad číslo 4.7.



Obr. 4.7: Validátor číselných hodnôt (**10.** bod)



Obr. 4.8: Vzhľady pre pridávanie nových súčiastok (**7.** bod)



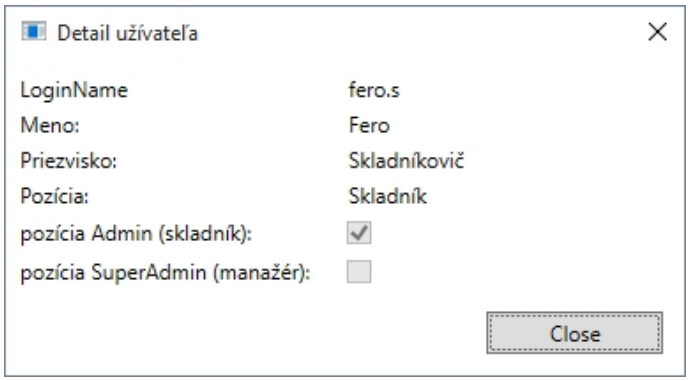
Obr. 4.9: Manažér užívateľov (**2.** bod)

ui.12. Príkaz, ktorý bude zobrazovať detail užívateľa - pohľad číslo 4.10.

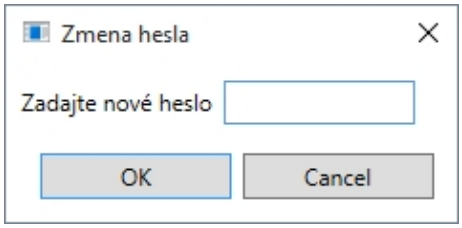
ui.13. Príkaz, ktorý bude zobrazovať okno zmeny hesla pre užívateľa - pohľad číslo 4.11.

ui.14. Príkaz, ktorý bude zobrazovať okno pre pridanie nového užívateľa - pohľad číslo 4.12.

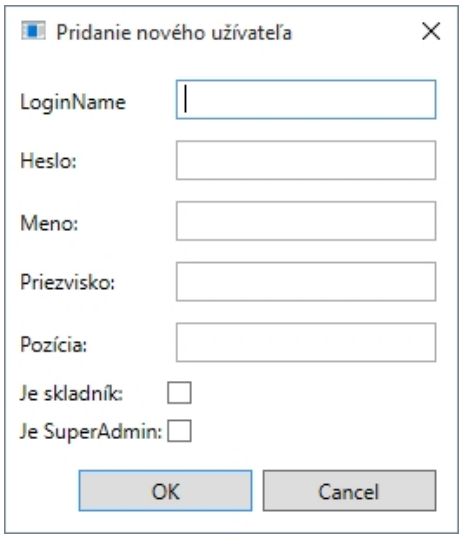
ui.15. Príkaz, ktorý bude zmaže užívateľa z aplikácie.



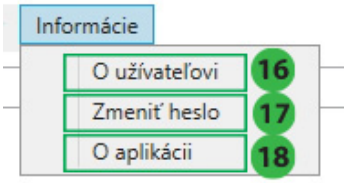
Obr. 4.10: Detail užívateľa (**12.** bod)



Obr. 4.11: Zmena hesla užívateľa (**13.** bod)



Obr. 4.12: Pridanie nového užívateľa (**14.** bod)



Obr. 4.13: Informácie pre užívateľa (**2.** bod)

ui.16. Príkaz, ktorý bude zobrazovať detail prihláseného užívateľa - pohľad číslo 4.10.

ui.17. Príkaz, ktorý bude zobrazovať okno zmeny hesla pre prihláseného užívateľa - pohľad číslo 4.11.

ui.18. Prikaz, na zobrazenie okna s informáciami o aplikácii.

5 IMPLEMENTÁCIA

Na začiatku som si kvôli modularite, testovateľnosti a správnemu rozdeleniu aplikácie vytvoril solution s nasledovnými projektami:

- **DESPREK.Application** - WPF Application using Catel
- **DESPREK.BusinessLayer** - Class Library
- **DESPREK.BusinessLayer.Test** - Unit Test Project
- **DESPREK.Helper** - Class Library
- **DESPREK.Models** - Class Library
- **DESPREK.Models.Test** - Unit Test Project
- **DESPREK.Services** - Class Library

5.1 DESPREK.Models

Prvý projekt, ktorý popíšem je projekt, v ktorom sa vytvárajú modely jednotlivých entít. Ako prvé som nainštaloval nuget balíček `Catel.Core` pomocou príkazu:

Pridanie NuGet balíčku Catel.Core

```
PM> Install-Package Catel.Core -Version 4.4.0
```

Tento balíček obsahuje triedu `Catel.Data.ModelBase`, z ktorej som si odvodil triedu `Model`, ktorá je otcovskou triedou všetkých mojich modelov:

Trieda Model

```
public abstract class Model : ModelBase
{
        [Key]
        [Column(Order = 1)]
        public Guid ID { get; private set; }
        public Model()
        {
                this.ID = Guid.NewGuid();
        }
}
```

Táto trieda obsahuje konštruktor, ktorý vytvorí unikátny kľúč typu Guid a k nemu priradené anotácie `[Key]`, ktorá vraví Entity Frameworku, že sa jedná o primárny kľúč v tabuľke, a `[Column(Order = 1)]` ktorá vraví, že bude tento kľúč zaradený v tabuľke na prvom mieste.

Trieda Component a PassiveComponent

```
public abstract class Component : Model
{
        [Required]
        public string UniqueCatalogNumber { get; set; }
        [Required]
        public string Name { get; set; }
        public int RemainingPieces { get; set; }
        public string Package { get; set; }
        public string DatasheetUrl { get; set; }
        public string ImageUrl { get; set; }
        public string Producent { get; set; }
}
public abstract class PassiveComponent : Component
{
        [Required]
        public float Value { get; set; }
        public string Unit { private get; set; }
        [NotMapped]
        public string GetValueWithUnitInEngineeringNotation =>
 ↪  ConvertorToEngineeringFormat.Convert(Value, Unit);
        public int Tolerance { get; set; }
        public string Usage { get; set; }
}
```

Tieto dve triedy slúžia ako rodič pre všetky súčiastky a priraďujú im základné spoločné vlastnosti popísané v kapitole 4.3. V triede sú použité ďalšie dva druhy anotácii a to `[Required]`, ktorá vraví, že táto vlastnosť sa nemôže uložiť do databázi s hodnotou `null` a anotácia `[NotMapped]`, ktorá prikazuje Entity Frameworku danú vlastnosť vylúčiť z namapovania do databázi. Táto vlastnosť volá metódu, ktorá vráti hodnotu v tzv. „*inžinierskej anotácii*“ a ktorá bude popísaná v nasledovnej kapitole v projekte **DESPREK.Helper** 5.2.

V projekte sú obsiahnuté okrem tried špecifických pre dané typy súčiastok - `CapacitorModel`, `InductorModel` a `ResistorModel`, ktoré dedia z otcovskej abstraktnej triedy `PassiveComponent` aj triedy, ktoré slúžia pre autentifikáciu užívateľa - `UserModel` a trieda pre logy - `LoggerMessage`. Obidve tieto triedy dedia z abstraktnej triedy `Model`. Všetky doposiaľ spomenuté triedy budú v nasledujúcej kapitole v projekte **DESPREK.Services** uložené pomocou Entity frameworku v databáze. Projekt obsahuje ešte triedy `ResultMessage`, ktorá slúži na oznamovanie činnosti vrstvy DataBusinessLayer užívateľovi a triedu `ImportData`, ktorá napĺňa modely súčiastok dátami z Exclu pomocou triedy `ExcelParser`, ktorá bude popísaná v projekte **DESPREK.Helper** 5.2.

5.2 DESPREK.Helper

Projekt obsahuje pomocné triedy `ConvertorToEngineeringFormat`, `Extensions`, `ExcelDataParser`, `SecurePasswordHasher`, a `Constants`. Statická trieda a metóda `ConvertorToEngineeringFormat.Convert` má za úlohu vrátiť reťazec, ktorý obsahuje zadané číslov inžinierske anotácii - t.j. napr. 15 μΩ.

Trieda Extensions implementuje rozšírujúcu metódu pre triedy implementujúce rozhrania `IEnumerable` alebo `ICollection`.

Ukážka rozširujúcej metódy pre triedy implementujúce ICollection

```
public static bool IsNullOrEmpty<T>(this ICollection<T> collection)
{
        if (collection == null)
        {
                return true;
        }
        return collection.Count < 1;
}
```

Pomocná trieda `SecurePasswordHasher` obsahuje hashovaciu funkciu, ktorá pomocou algoritmu SHA1 vytvorí 160 bitový otlačok, ktorý skonvertuje do hexadecimálno tvaru a vytvorí 40 znakový hash. Trieda slúži pre bezpečnú prácu v aplikácii s užívateľským účtom a pre bezpečné uloženie hesla v databáze.

Trieda SecurePasswordHasher

```
public static class SecurePasswordHasher
{
        public static string Hash(string nonHashedPassword)
        {
                var hash = new
 ↪  SHA1Managed().ComputeHash(Encoding.UTF8.GetBytes(nonHashedPassword));
                return string.Join(string.Empty,hash.Select(b =>
 ↪  b.ToString("x2")).ToArray());
        }
}
```

V praxi sa používa pri vytváraní hash-u kryptografická soľ, ktorá znižuje šance prelomenia hashu pomocou brute force útoku s použitím rainbow tables. Tento príklad je v rámci zjednodušenia bez kryptografickej soli.

Ďalšou pomocnou triedou je trieda `ExcelDataParser`, kde som nainštaloval balíček *ExcelDataReader*.

Pridanie NuGet balíčku ExcelDataReader

```
PM> Install-Package ExcelDataReader -Version 2.1.2.3
```

Tento balíček implementuje metódy pre parsovanie dát z *Excel*-u a podporuje okrem binárne uložených súborov - **.xls**, aj textovo uložený formát pomocou značkovacieho jazyka xml - **.xlsx**. Táto trieda slúži metódam, ktoré následne napĺňajú *Model*-y, ktoré sú definované v projekte **DESPREK.Models** kapitola 5.1. Statická trieda `Constants` obsahuje konštanty, ktoré sa používajú v aplikácii ako napr. „n/a“, ktorá slúži pri importe ako náhrada *null*.

5.3 DESPREK.Services

Na začiatku som nainštaloval pomocou *Powershell Package Manager Console Entity Framework* a *Catel.Extensions.EntityFramework6*, pomocou ktorého príkazu sa automaticky nainštaloval aj balíček Catel.Core.

Pridanie NuGet balíčku EntityFramework a Catel.Extensions.EntityFramework6

```
PM> Install-Package EntityFramework -Version 6.1.3
PM> Install-Package Catel.Extensions.EntityFramework6 -Version 4.4.0
```

Tento projekt slúži na komunikáciu s databázou pomocou ORM frameworku EF naplnenie entít dátami z relačnej databázy MS SQL. Na začiatku som vytvoril repozitáre pre jednotlivé modely - `UsersRepository`, `ResistorsRepository`, `LogsRepository`, `InductorsRepository` a `CapacitorsRepository`.

Ukážka implementácie repozitárov pre Model UserModel

```
public interface IUsersRepository : IEntityRepository<UserModel, Guid>
{}
public class UsersRepository : EntityRepositoryBase<UserModel, Guid>,
 ↪  IUsersRepository
{
        public UsersRepository(DbContext dbContext)
        : base(dbContext)
        {}
}
```

Ďalej som vytvoril triedu `DesprekDbContext`, ktorá slúži na vytvorenie spojenia s databázou, vytvorenie tabuliek podľa entíty, naplnenie modelov a triedu `DesprekDbContextConfiguration`, ktora slúži pre konfiguráciu pri automatickej migracie pomocou *Code-First* vývoja. V konštruktore triedy `DesprekDbContext` sa pomocou triedy `ServiceLocator` z frameworku Catel registrujú jednotlivé repozitáre, ktoré sú vytvárané ako `Singleton`-y vzorom `Repository`. Prepísaná metóda `OnModelCreating` sa volá vždy pri zmene modelu. Nastavuje napr. názvy tabuliek, povoľuje automatickú migráciu, atď.

Trieda DesprekDbContext a DesprekDbContextConfiguration

```
public class DesprekDbContext : DbContext
{
        public DesprekDbContext():base("DesprekDBConnection")
        {
                var serviceLocator = ServiceLocator.Default;
                serviceLocator.RegisterType<IInductorsRepository,
 ↪  InductorsRepository>();
                serviceLocator.RegisterType<IResistorsRepository,
 ↪  ResistorsRepository>();
                serviceLocator.RegisterType<ICapacitorsRepository,
 ↪  CapacitorsRepository>();
                serviceLocator.RegisterType<IUsersRepository,
 ↪  UsersRepository>();
                serviceLocator.RegisterType<ILogsRepository, LogsRepository>();
        }
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
                Database.SetInitializer(new
 ↪  MigrateDatabaseToLatestVersion<DesprekDbContext,
 ↪  DesprekDbContextConfiguration<DesprekDbContext>>());
                base.OnModelCreating(modelBuilder);
                modelBuilder.Entity<UserModel>().ToTable("Accounts")
                .IgnoreCatelProperties();
                modelBuilder.Entity<LoggerMessage>().ToTable("Logs")
                .IgnoreCatelProperties();
                modelBuilder.Entity<ResistorModel>().ToTable("Resistors")
                .IgnoreCatelProperties();
                modelBuilder.Entity<InductorModel>().ToTable("Inductors")
                .IgnoreCatelProperties();
                modelBuilder.Entity<CapacitorModel>().ToTable("Capacitors")
                .IgnoreCatelProperties();
        }
}
public class DesprekDbContextConfiguration<T> : DbMigrationsConfiguration<T>
 ↪  where T : DbContext
{
        public DesprekDbContextConfiguration()
        {
                this.AutomaticMigrationsEnabled = true;
                this.AutomaticMigrationDataLossAllowed = true;
        }
}
```

Pred spustením automatickej migrácie a inicializovaním Code-First vývoja je nutné vytvoriť *Connection String* s platnými prihlasovacími údajmi do databázy.

Connection String

```
<connectionStrings>
        <add name="DesprekDBConnection" connectionString="Data
    ↪ Source=PadreASUS;Initial Catalog=DesprekDatabase;Integrated
    ↪ Security=True;MultipleActiveResultSets=True"
    ↪ providerName="System.Data.SqlClient" />
</connectionStrings>
```

Následne som pomocou konzoly a zvoleného „*Default project*-u“ výstupu konzoly povolil automatické migrácie. Následne som vytvoril novú migráciu a aktualizoval databázu s modelmi.

Povolenie a vytvorenie migrací pomocou Powershell konzole

```
PM> Enable-Migrations
PM> Add-Migration initMigration
PM> Update-Database
```

Týmito príkazmi sa povolia migrácie, vytvorí sa nová s názvom „*initMigration*“ a následne sa aktualizuje databáza s našimi modelmi. O nasledujúce migrácie sa nemusíme starať a databáza sa bude sama aktualizovať podľa zmeny v modeloch.

5.4 DESPREK.Models.Test

Tento projekt je typu *Unit Test Project*. V ňom som vytvoril triedu `UnitTestModels` s metódami pre testovanie vytvárania modelov a import súčiastok z Excel-u.

Unit test UnitTestModels

```
[TestClass]
public class UnitTestModels
{
        [TestMethod]
        TestImportResistorsFromExcel()
        [TestMethod]
        TestImportInductorsFromExcel()
        [TestMethod]
        TestImportCapacitorsFromExcel()
        [TestMethod]
        CreateUsersAccount()
}
```

5.5 DESPREK.BusinessLayer

Projekt, ktorý je v schéme rozdelenia aplikácie na vrstvy priamo pod *ViewModel*-om. Ako zdôrazním, že projekt obsahuje konfiguračný súbor `Settings.settings`, ktorý obsahuje konštanty ako `PathToLogs` pre logovací súbor v prípade výpadku servera, konštanty ako `ExcelSheetNameOfCapacitors`, `ExcelSheetNameOfResistors` a `ExcelSheetNameOfInductors`, ktoré slúžia na lokalizovanie hárkov zošita pri importe dát z Excelu. Ďalej projekt obsahuje triedu `LogToFileWriter`, ktorá slúži v prípade výpadku databáze na logovanie činnosti do súboru. Posledná a najdôležitejšia trieda projektu sa nazýva `DataBusinnessLayer`. Táto trieda sa stará o prácu CRUD nad dátami, autentifikáciu užívateľov voči databáze a logovanie činnosti.

V projekte sú inštalované balíčky *Catel.Extensions.EntityFramework6* a *EntityFramework*. Framework Catel poskytuje `UnitOfWork`, ktorý slúži ako zoznam repozitárov a nie je nutné vytvárať pre každý repozitár nový kontext do databázy. Keďže má táto trieda viac ako 1000 riadkov, v nasledujúcej ukážke naznačím iba časť metód a vlastnosti, ktoré táto trieda implementuje.

Ukážka triedy DataBusinnessLayer

```
1  public class DataBusinnessLayer : ModelBase, INotifyPropertyChanged
2  {
3          // prihlasenie a odhlasenie uzivatela
4          public ResultMessage LogInUser(UserModel loggingUser)
5          public ResultMessage LogOutUser()
6
7          public ObservableCollection<ResistorModel> ResitorsList =>
   ↪   LoadObservableCollectionOfResistors();
8          private ObservableCollection<ResistorModel>
   ↪   LoadObservableCollectionOfResistors()
9          {
10                 // metoda vracajuca vsetky rezistory ak je uzivatel prihlaseny
11         }
12         #region CRUD metody nad uzivatelmi
13         public ResultMessage AddUser(UserModel user)
14         public ResultMessage DeleteUser(UserModel userToDelete)
15         public ResultMessage UpdateUser(UserModel userToModify)
16         // asynchronne metody
17         public async Task<ResultMessage> AddUserAsync(UserModel user)
18         public async Task<ResultMessage> DeleteUserAsync(UserModel userToDelete)
19         public async Task<ResultMessage> UpdateUserAsync(UserModel userToModify)
20         #endregion
21         #region CRUD metody nad suciastkami
22         public ResultMessage DeleteItem(PassiveComponent component)
23         public ResultMessage DeleteAllResistorsInDatabase()
24         public ResultMessage AddPassiveComponent(PassiveComponent component)
```

Ukážka triedy DataBusinnessLayer

```
25          private ResultMessage
    ↪ AddRangeOfPassiveComponents(IEnumerable<PassiveComponent> components)
26          public ResultMessage UpdateExistingPassiveComponent(PassiveComponent
    ↪ component)
27          public InductorModel CreateInductorModel() => IsUserAutentificated ? new
    ↪ InductorModel() {UniqueCatalogNumber =
    ↪ CreateNewUniqueCatalogNumberForInductor()} : null;
28          private string CreateNewUniqueCatalogNumberForCapacitor()
29          {
30                  var uniqueNumbers = CapacitorsList.Select(capacitor =>
    ↪ capacitor.UniqueCatalogNumber).ToList();
31                  int maxNumber = GetLastCatalogNumber(uniqueNumbers);
32                  return String.Format("C{0}",maxNumber+1);
33          }
34          private static int GetLastCatalogNumber(List<string> uniqueNumbers)
35          {
36                  var maxNumber = 0;
37                  foreach (short number in uniqueNumbers.Select(uniqueNumber =>
    ↪ Convert.ToInt16(uniqueNumber.Substring(1, uniqueNumber.Length -
    ↪ 1))).Where(number => number > maxNumber))
38                  {
39                          maxNumber = number;
40                  }
41                  return maxNumber;
42          }
43          public async Task<ResultMessage>
    ↪ AddPassiveComponentAsync(PassiveComponent component)
44          public async Task<ResultMessage>
    ↪ UpdateExistingPassiveComponentAsync(PassiveComponent component)
45          public async Task<ResultMessage> ImportCapacitorsFromExcelAsync(string
    ↪ pathToFile)
46          private async Task<ResultMessage>
    ↪ AddRangeOfPassiveComponentsAsync(IEnumerable<PassiveComponent> components)
47          #endregion
48          #region metody pre aktualizaciu databaze a logovanie, implementacia
    ↪ INotifyPropertyChanged
49          private ResultMessage SaveBusinessLayerToDatabase(LoggerMessage log,
    ↪ string changedProp1, string changedProp2, bool updateAllLists = false)
50          private void LogToDatabase(LoggerMessage item)
51          private void OnPropertyChanged(string propertyName)
52          {
53                  PropertyChanged?.Invoke(this, new
    ↪ PropertyChangedEventArgs(propertyName));
54          }
55          public event PropertyChangedEventHandler PropertyChanged;
56          #endregion
57  }
```

Ako je vidieť z predchádzajúcej ukážky, každá metóda, ktorá ma accessor public, má návratový typ `ResultMessage`, v ktorom je obsiahnutý výsledok a popis činnosti, ktorú trieda `DataBusinnessLayer` vykonala. Užívateľ tak má prehľad, čo aplikácia robí, ako je to popísane v bode **ui.4**. a obrázku 4.4. Zároveň je aj zabezpečené logovanie činnosti do databázy.

5.6 DESPREK.BusinessLayer.Test

Tento projekt je znova typu *Unit Test Project*. V ňom je vytvorená trieda na testovanie vrstvy *BusinnessLayer* s názvom `UnitTestBusinesslayer` a testovacími metódami.

Unit test UnitTestBusinesslayer

```
[TestMethod]
public void CompletTestSyncMethod()
[TestMethod]
public void CreateTestData()
[TestMethod]
public void CompletTestAsyncMethod()
[TestMethod]
public void RemoveDataFromDatabase()
[TestMethod]
public void ImportResitorsFromExcelToDatabase()
[TestMethod]
public void ImportCapacitorsFromExcelToDatabase()
[TestMethod]
public void ImportInductorsFromExcelToDatabase()
[TestMethod]
public void CreateUsers()
[TestMethod]
public void GetDataUsingDataBusinnessLayer()
[TestMethod]
public void UpdateDatabaseItems()
[TestMethod]
public void DeleteAllUserWithoutDevUser()
[TestMethod]
public void DeleteIndividualItem()
[TestMethod]
public void ImportResitorsFromExcelToDatabaseAsync()
[TestMethod]
public void ImportCapacitorsFromExcelToDatabaseAsync()
[TestMethod]
public void ImportInductorsFromExcelToDatabaseAsync()
```

5.7 DESPREK.Application

Projekt, ktorého výsledkom sú spustiteľné súbory so samotnou aplikáciou. Projekt som vytvoril na základe šablóny *WPF Application with Catel*. Táto šablóna vygeneruje stromovú štruktúru pre jednoduchú prácu s vzorom MVVM a importuje balíčky *Catel.Core*, *Catel.MVVM* a *Catel.Controls*. Do projektu som nasledovne importoval NuGet balíček *EntityFramework*.

Nasledovná ukážka je stromová štruktúra s popisom vygenerovaná šablónou:

```
DESPREK.Application
├── Properties
│   ├── AssemblyInfo.cs
│   └── Settings.settings ..... Konfiguračný súbor s konštantami
├── References ..... Referencie na balíčky a ostatné projekty
├── Converters ..... Priečinok s konvertormi
├── Icons ..... Priečinok s obrázkami
├── Models ..... Priečinok s modelmi - projekt DESPREK.MODELS
├── Services Priečinok pre Services - projekt DESPREK.SERVICES
├── ViewModels ..... Priečinok s ViewModel-mi
│   └── MainWindowViewModel.cs ..... Hlavný ViewModel
├── Views ..... Priečinok s View-s
│   └── MainWindow.xaml ..... Hlavný View
├── App.config ..... Konfiguračný súbor zapísaný v XML
├── App.xaml ..... Konfiguračný riadiaci súbor zapísaný v XAML
├── packages.config ..... Konfiguračný súbor NuGet balíčkov
└── Readme.txt
```

Do nasledovnej štruktúry som postupne pridal niekoľko *View*-s.

```
../
└── Views ..... Priečinok s View-s
    ├── ChangePasswordView.xaml ..... View návrh Obr.4.11
    ├── InfoAboutAppView.xaml ..... View s informáciami o aplikácii
    ├── MainWindow.xaml ..... View návrh Obr.4.4
    ├── ManagerForUsersView.xaml ..... View návrh Obr.4.9
    ├── NewCapacitorView.xaml ..... View návrh Obr.4.8
    ├── NewInductorView.xaml ..... View návrh Obr.4.8
    ├── NewResistorView.xaml ..... View návrh Obr.4.8
    ├── NewUserView.xaml ..... View návrh Obr.4.12
    ├── ResultMessageUserControlView.xaml ..... User control ui.4.
    └── UserDetailView.xaml ..... View návrh Obr.4.10
```

K týmto *View*-s som pridal podľa vzoru MVVM *ViewModel*-i s odpovedajúcimi názvami:

```
../
└── ViewModels ..... Priečinok s ViewModel-s
    ├── ChangePasswordViewModel.cs .. VM ktorý mení heslo pre daného užívateľa
    ├── MainWindowViewModel.cs .. VM ktorý drží stav celej aplikácie a komunikuje ako jediný s vrstvou BusinessLayer
    ├── ManagerForUsersViewModel.cs ..... VM ktorý slúži na správu užívateľov
    ├── NewCapacitorViewModel.cs ..... VM ktorý slúži na pridanie kondenzátoru
    ├── NewInductorViewModel.cs .. VM ktorý slúži na pridanie cievky
    ├── NewResistorViewModel.cs . VM ktorý slúži na pridanie rezistoru
    ├── NewUserViewModel.cs ..... VM ktorý slúži na pridanie nového užívateľa
    ├── ResultMessageUserControlViewModel.cs ..... VM ktorý slúži na pridanie nového užívateľa
    └── UserDetailViewModel.cs ..... VM ktorý slúži na predanie informácii o užívateľovi
```

V projekte som tiež vytvoril niekoľko konvertorov, ktoré sa používajú vo *View*-s pre správnu interpretáciu hodnôt.

```
../
└── Converters ..... Priečinok s konvertormi
    ├── FallbackConverter.cs ..... Konvertor null na „n/a“
    ├── FullImagePathConverter.cs ..... Skladá full path na obrázky
    ├── InvertableBooleanToVisibilityConverter.cs ..... Invertovateľný konvertor bool na Visibility
    └── StringToBooleanConverter.cs ..... Konvertuje string na bool a naopak
```

Aplikácia používa tri „typy“ okien, a to `catel.Window`, `catel.DataWindow` a `catel.UserControl`. V aplikácii som používal zväčša typ `catel.DataWindow`, ktorý spolu s použitím `ViewModel`-u implementuje dependency property a tým umožňuje dynamicky udržovať kontext skrz celú aplikáciu. Nasledujúce ukážky sa budú zameriavať na implementáciu frameworku Catel vo výslednej aplikácii.

Ukážka z pohľadu MainWindowView

```
<catel:DataWindow.Resources>
        <ViewModels:DesignMainWindowViewModel x:Key="MainWindowViewModel"/>
        <Style x:Key="RoundedButtonStyle" TargetType="{x:Type Button}"><!--....--></Style>
        <myConverters:InvertableBooleanToVisibilityConverter x:Key="InvertableBoolToVisibilityConverter"/>
        <CollectionViewSource x:Key="OrderedComponentByName" Source="{Binding FilteredComponents}"> <!--....--> </CollectionViewSource>
</catel:DataWindow.Resources>
<catel:StackGrid x:Name="LoginView" Visibility="{Binding IsAuthenticated, Converter={StaticResource InvertableBoolToVisibilityConverter}, ConverterParameter=Inverted}"> <!--....--> </catel:StackGrid>
```

„Zdroje“, ktoré sú definované na začiatku ohľadu `MainWindowViewModel`. Je tu vidieť pridanie VM `DesignMainWindowViewModel`, ktorý je však iba pre účely dizajnu a pri spustení aplikácie sa pomocou `serviceLocator` prepojí *View* s *ViewModel*-om na základe mennej notácie. Ďalej je vidieť pridanie štýlu pre typ komponentu *Button*, definícia konvertoru a `CollectionViewSource`, ktorý slúži na zoradenie položiek `FilteredComponents` z VM na základe vlastnosti `Name`. Nasleduje ukážka použitia konvertoru `InvertableBoolToVisibilityConverter` s parametrom `Inverted`. V tejto ukážke používam komponentu `catel:StackGrid` ktorá, dokáže s použitím napr. `Height="Auto"` automaticky nastaviť optimálnu veľkosť prvku.

Ukážka pohľadu z MainWindowView

```
<TextBox Grid.Column="1" Text="{Binding SearchFilter}">
        <i:Interaction.Behaviors>
                <catel:UpdateBindingOnTextChanged UpdateDelay="500" />
        </i:Interaction.Behaviors>
</TextBox>
<TextBlock>
        <TextBlock.Text>
                <MultiBinding StringFormat="{}{0}mm x &#216;{1}mm" Converter="{StaticResource FallbackConverter}" FallbackValue="n/a">
                        <Binding Path="Height" />
                        <Binding Path="Diameter" />
                </MultiBinding>
        </TextBlock.Text>
</TextBlock>
<PasswordBox x:Name="PasswordBox">
        <i:Interaction.Behaviors>
                <catel:UpdateBindingOnPasswordChanged Password="{Binding NonHashedPassword, Mode=OneWayToSource}" />
        </i:Interaction.Behaviors>
</PasswordBox>
```

Prvý `TextBlock` slúži ako searchbox pre všetky filtre v aplikácii. Má na sebe „zavesený“ trigger a metódu `MainWindowViewModel.UpdateSearchFilter()`, ktorá každých 500 ms spustí aktualizáciu vyhľadávania. V ďalšiom `TextBlock`-u je ukážka multibindingu, ktorý sa hodí napríklad pri vkladaní dát z viacerých zdrojov do jednej komponenty. Reťazec *&#216* slúži pre zadanie znaku ϕ v decimálnom tvare (D8). Je tu tiež použitý konvertor na zistenie, či sa jedná o prázdnu (`null`) hodnotu a v tom prípade sa vráti reťazec `n/a`.

Komponent `PasswordBox` je takmer vždy miernym orieškom, pretože jeho hodnotu nie je možné nabindovať priamo, pretože sa jeho hodnota uchováva v pamäti šifrovane a jediná možnosť prístupu je skrz CLR vlastnosť. Framework Catel však poskytuje možnosť nabindovať hodnotu do reťazca. **Je to však bezpečnostné riziko**, pretože sa následne uchováva v pamäti ako plain text. Preto som použil mód bindingu `OneWayToSource`, čo znamená že sa bude aktualizovať iba vlastnosť vo *ViewModel*-y, a kde som nebezpečenstvo plain textu vyriešil nasledovne:

Property HashedPassword s módom bindingu OneWayToSource

```
public string HashedPassword
        {
            get { return GetValue<string>(NonHashedPasswordProperty); }
            set { SetValue(NonHashedPasswordProperty,
 ↪  SecurePasswordHasher.Hash(value ?? Empty)); }
        }
public static readonly PropertyData NonHashedPasswordProperty =
 ↪  RegisterProperty(nameof(HashedPassword), typeof(string));
```

Nasledujúce ukážka sa venuje použitiu *User Control* pomocou frameworku Catel:

Ukážka pohľadu z MainWindow -použitie User Control

```
<views:ResultMessageUserControlView  DataContext="{Binding
 ↪  ResultMessageFromBusinessLayer}" ></views:ResultMessageUserControlView>
```

Pohľadu `ResultMessageUserControlView` nabindujeme pomocou kontextu vlastnosť `ResultMessageFromBusinessLayer`. Táto vlastnosť sa predá ako argument konštruktoru triedy `ResultMessageUserControlViewModel`, a ten ho vloží do vlastnosti `Message`:

Ukážka MainWindowViewModel -Command

```
public ResultMessageUserControlViewModel(ResultMessage message)
        {
            Message = message;
        }
```

Vlastnosť je následne pomocou `DependencyProperty` automaticky aktualizovaná s kontextom `MainWindowViewModel`-u. O vytvorenie `DependencyProperty` sa stará

framework Catel.

Ukážka implementácie Command

```
 public Command<object, object> LoginCommand { get; private set; }
 private bool LoginCommandCanExecute(object parameter)=> !IsAuthenticated;
private void LoginCommandExecute(object parameter)
        {
            PleaseWaitHelper.Show("Čakajte prosím, prihlasujem sa");
            // code
            PleaseWaitHelper.Hide();
        }
// registrácia commandu v~konštruktore
LoginCommand = new Command<object, object>(LoginCommandExecute,
 ↪  LoginCommandCanExecute);
```

Vytvorenie *Command*-u pomocou frameworku Catel je veľmi jednoduché. Za povšimnutie stojí `PleaseWaitHelper.Show()`, ktorý spôsobí „zamrznutie“ aplikácie a zobrazí nasledovnú hlášku.

Obr. 5.1: Oznámenie „Čakajte prosím“

Framework Catel tiež implementuje `AsynchronousCommand`, ktorý zabezpečuje asynchrónne vykonávanie metód a hladký beh aplikácie bez „zamŕzania“.

Ukážka implementácie AsynchronousCommand

```
public AsynchronousCommand<object,object> SaveComponentToDbAsyncCommand { get;
 ↪  set; }
private bool CurrentUserIsWarehouser(object parameter) => IsAuthenticated &&
 ↪  IsWarehouser;
private async void SaveComponentToDbAsyncCommandExecute(object parameter)
        {
             ResultMessageFromBusinessLayer = new ResultMessage() {Result =
 ↪  LoggerMessage.Working, Description = String.Format("Updating component
 ↪  UCID: {0}",SelectedComponent.UniqueCatalogNumber)};
            ResultMessageFromBusinessLayer = await
 ↪  dataBusinnessLayer.UpdateExistingPassiveComponentAsync(SelectedComponent as
 ↪  PassiveComponent);
        }
SaveComponentToDbAsyncCommand = new AsynchronousCommand<object,
 ↪  object>(SaveComponentToDbAsyncCommandExecute, CurrentUserIsWarehouser);
```

Na to tomto príklade je vidieť okrem implementácie asynchrónneho *Command*-u aj spôsob napĺňania správy pre užívateľa.

Nasledujúca ukážka používa *Command* typu `TaskCommand`, ktorý sa môže použiť napríklad na vytvorenie *DataWindow* so synchronizovaným kontextom.

Ukážka implementácie TaskCommand

```
public TaskCommand AddNewResistor { get; private set; }
private bool CurrentUserIsWarehouserForTask() => CurrentUserIsWarehouser(null);
private async Task AddNewResistorExecuteAsync()
        {
            var resistor = dataBusinnessLayer.CreateResistorModel();
            var typeFactory = this.GetTypeFactory();
            var resistorViewModel =
    typeFactory.CreateInstanceWithParametersAndAutoCompletion
    <NewResistorViewModel>(resistor);
            if (await _uiVisualizerService.ShowDialogAsync(resistorViewModel) ??
    false)
            {
                ResultMessageFromBusinessLayer = new ResultMessage() { Result =
    LoggerMessage.Working, Description = String.Format("Aditing component with
    catalog Number {0}",resistor.UniqueCatalogNumber);
                ResultMessageFromBusinessLayer = await
    dataBusinnessLayer.AddPassiveComponentAsync(resistor);
            }
        }
// registrácia commandu v~konštruktore
AddNewResistor = new TaskCommand(AddNewResistorExecuteAsync,
    CurrentUserIsWarehouserForTask);
```

Na začiatku sa vytvorí nový model súčiastky - `CreateResistorModel()` a pomocou metódy `CreateInstanceWithParametersAndAutoCompletion` so zadaným typom `<NewResistorViewModel>` a argumentom `(resistor)` sa vytvorí instancia `NewResistorView`, ktorá je inicializovaná na základe volania systémovej metódy `System.Activator.CreateInstance(System.Type)` a vytvára instanciu VM `NewResistorViewModel` a injektuje do jeho konštruktora argument `(resistor)`. Po otvorení nového pohľadu, naplnenia modelu a stlačenia tlačidla OK (východzie nastavenie tlačidiel pre `DataWindow` je `DataWindowMode.OkCancel`) sa tento model injektuje späť do VM, z ktorého pochádza.

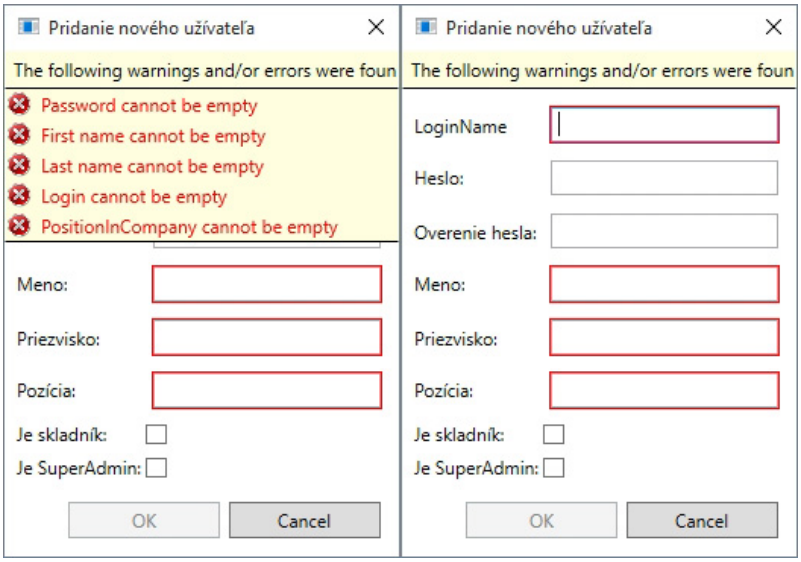
Pri vytváraní business aplikácii sú častým problémom validácie dát. Samotný vzor MVVM je na validácie dát pred-pripravený a framework Catel dokáže jednoducho spojením DataWindow a validácii nad modelmi napríklad zablokovať OK tlačítko, na základe chybne zadaných hodnôt, ako je to demonštrované na obrázku 5.2.

Ukážka validácie pre vlastnosť FirstName

```
protected override void ValidateFields(List<IFieldValidationResult>
 ↪  validationResults)
{
        if (string.IsNullOrEmpty(User.FirstName))
        {
                validationResults.Add(FieldValidationResult.
 ↪  CreateError(nameof(User.FirstName), "First name cannot be empty"));
        }
}
```

Zápis vo View

```
<Label Content="First name" />
<TextBox Text="{Binding FirstName, ValidatesOnDataErrors=True,
 ↪  NotifyOnValidationError=True}" />
```



Obr. 5.2: Ukážka validácie

5.8 Logovanie v Databáze

V SRS je špecifikovaná požiadavka na logovanie činnosti v databáze 5.3. Z kapitoly 5.5 vieme, že o logovanie sa stará práve logická vrstva, čiže trieda `DataBusinessLayer`. Na nasledujúcej ukážke je vidieť mechanizmus ukladania správy s návratovou hodnotou typu `ResultMessage`. Tento mechanizmus je zodpovedný za informovanie užívateľa a zároveň za logovanie do súboru 5.4, ak sa spojenie s databázou nepodarí nadviazať.

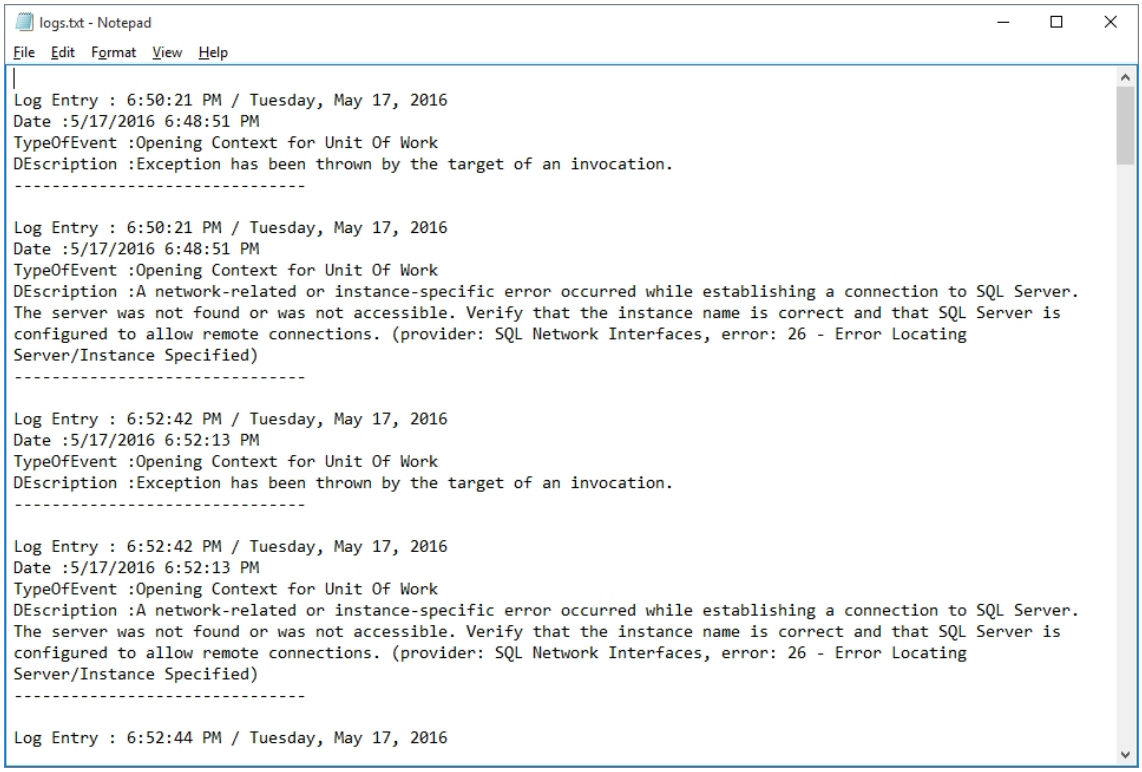
Ukážka vytvárania logu a

```
private async Task<ResultMessage> SaveBusinessLayerToDatabaseAsync(LoggerMessage log, string changedProp1, string changedProp2, bool updateAllLists = false)
{
        LogToDatabase(log);
        await UnitOfWorkBusinessLayer.SaveChangesAsync();
        // volanie metód OnPropertyChanged
        return new ResultMessage()
                {
                        Result = LoggerMessage.Sucessfull,
                        Description =
                        string.IsNullOrEmpty(log.DescriptionResultMessage) ? log.Description : log.DescriptionResultMessage
                };
}
private bool IsUnitOfWorkDisposed()
{
        if (IsDisposed)
        {
                LoggerMessage log = new LoggerMessage(LoggerMessage.Error)
                {
                        Result = LoggerMessage.Failed,
                        Description = LoggerMessage.UnitOfWorkIsDisposed
                };
                LogToFileWriter.WriteLogToFile(log);
        }
        return IsDisposed;
}
```

Results | Messages

	ID	TypeOfEvent	Result	Description	Date
1	5568...	Disposing Unit Of Work	Sucessfull	Connection to Database is disposed	2016-05-30 11:11:34.863
2	227A...	Log out	Sucessfull	User login:superAdmin logged out	2016-05-30 11:11:34.843
3	59E...	Deleting Item	Sucessfull	Delete 1 items of ebee13e0-dc6c-4480-b452-38f1baf3d183	2016-05-30 11:11:34.793
4	16C...	Deleting Item	Sucessfull	Delete 1 items of 481910f4-2408-4f7c-9d68-29a72202cfb1	2016-05-30 11:11:34.757
5	46F...	Log in	Sucessfull	User login:superAdmin logged in	2016-05-30 11:11:34.143
6	8A25...	Opening Context for Unit Of Work	Sucessfull	Connection to DB is open	2016-05-30 11:11:34.137
7	EA6...	Disposing Unit Of Work	Sucessfull	Connection to Database is disposed	2016-05-30 11:11:34.110
8	8999...	Log out	Sucessfull	User login:superAdmin logged out	2016-05-30 11:11:34.093
9	4680...	Updating User	Sucessfull	Update UserLogin: arnold.s ID: 8740716c-8186-4652-903a-20ee6bbd99f0 with user superAdmin, ID:abea3d04-df6c-4d41-bec...	2016-05-30 11:11:34.060
10	ECC...	Updating item	Sucessfull	Update item, with ID 9c02f2bd-d2f5-4599-abc7-07dda5f476de	2016-05-30 11:11:34.017
11	5D8...	Updating item	Sucessfull	Update item, with ID 8aaff41f-6600-4fd5-9836-082bf32c66b1	2016-05-30 11:11:33.970
12	18D...	Log in	Sucessfull	User login:superAdmin logged in	2016-05-30 11:11:33.437
13	875C...	Opening Context for Unit Of Work	Sucessfull	Connection to DB is open	2016-05-30 11:11:33.427
14	F3FE...	Disposing Unit Of Work	Sucessfull	Connection to Database is disposed	2016-05-30 11:11:33.340
15	57E...	Log out	Sucessfull	User login:superAdmin logged out	2016-05-30 11:11:33.323
16	3111...	Log in	Sucessfull	User login:superAdmin logged in	2016-05-30 11:11:32.900
17	D1B...	Opening Context for Unit Of Work	Sucessfull	Connection to DB is open	2016-05-30 11:11:32.893
18	195...	Disposing Unit Of Work	Sucessfull	Connection to Database is disposed	2016-05-30 11:11:32.887
19	0CB...	Log out	Sucessfull	User login:superAdmin logged out	2016-05-30 11:11:32.883
20	4BC...	Adding New User	Sucessfull	Added new user with UserLogin: karol.n ID: a75d2006-ca04-4857-bf06-5135d0f70d58 with user superAdmin, ID:abea3d04-df6...	2016-05-30 11:11:32.877
21	8D1...	Adding New User	Sucessfull	Added new user with UserLogin: jozo.n ID: a1104865-e00c-45aa-afec-e28a66fd53cc with user superAdmin, ID:abea3d04-df6...	2016-05-30 11:11:32.873
22	ABC...	Adding New User	Sucessfull	Added new user with UserLogin: arnold.s ID: 8740716c-8186-4652-903a-20ee6bbd99f0 with user superAdmin, ID:abea3d04-d...	2016-05-30 11:11:32.870
23	AED...	Adding New User	Sucessfull	Added new user with UserLogin: fero.s ID: 24917319-f699-4594-930a-4e75eac734b5 with user superAdmin, ID:abea3d04-df6...	2016-05-30 11:11:32.863

Obr. 5.3: Ukážka logovania do databáze



Obr. 5.4: Ukážka logovania do súboru

6 ZÁVER

Návrh a vývoj viac-vrstvých aplikácii pomocou prezentačných vzorov je dnes bežná prax pri tvorbe robustných business aplikácii. Rozdelenie má za cieľ viacero pozitívnych faktorov, medzi ktoré patrí udržateľnosť, testovateľnosť, znovu-použiteľnosť modulov a jednoduché rozdelenie práce na vývoji aplikácie medzi viacero týmov alebo viacero členov týmu. Jedna z nevýhod rozvrstvenia aplikácie je zdĺhavejší návrh, ktorý je však neskôr kompenzovaný výhodami tohoto rozčlenenia.

Technológia WPF v spojení s návrhovým vzorom MVVM je dnes pomerne populárna, avšak do popredia sa začína predierať technológia UWP (Universal Windows Platform) ktorá podobne ako WPF používa silu jazyka XAML, jedná sa však o dve rozdielne, aj keď principiálne podobné technológie. Dá sa predpokladať že si WPF udrží svoju pozíciu ešte nejaký čas, pretože má širokú vývojársku základňu je celkom slušne optimalizované.

Framework Catel, ktorý som používal v mojej praktickej časti - aplikácii je v súčastnosti asi najobsiahlejší a najprepracovanejší (nielen MVVM) framework, ktorý je možné použiť skrz všetky vrstvy aplikácie. Jeho obrovskou výhodou je jeho modulárnosť a množstvo praktických prvkov ako napríklad `PleaseWaitHelper`.

V mojej práci som sa snažil zamerať nielen na pokročilú prácu s dátami a vytváraní užívateľsky prívetivej aplikácie ale aj o dodržovanie zásad SOLID kódu a celkovej úprave projektov. Na záver by som chcel ešte dodať, že pri vytváraní vzhľadu aplikácie som sa snažil aj o vytváranie UX dizajnu a jednoduchú orientáciu skrz ovládacie prvky aplikácie.

LITERATÚRA

[1] Association for Computing Machinery *Computing Degrees & Careers* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://computingcareers.acm.org/?page_id=12>.

[2] Borek B. *Alternativy k MVC a závěrečné poznámky* [online]. [cit. 1. 12. 2015] Dostupné z URL: <https://www.zdrojak.cz/clanky/alternativy-k-mvc-a-zaverecne-poznamky/>.

[3] Borek B. *Úvod do architektury MVC* [online]. [cit. 1. 12. 2015] Dostupné z URL: <https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/>.

[4] BROOKS, Frederick Phillips, ml. *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley, 1995. ISBN 0-201-83595-9.

[5] Čápka D. *1. díl - Úvod do Windows Forms aplikací* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.itnetwork.cz/csharp/windows-forms/c-sharp-tutorial-windows-forms-okenni-aplikace-uvod>.

[6] Čápka D. *2. díl - Jazyk XAML v C# .NET WPF* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.itnetwork.cz/csharp/wpf/c-sharp-tutorial-wpf-jazyk-xaml>.

[7] Čápka D. *1. díl - Úvod do WPF (Windows Presentation Foundation)* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.itnetwork.cz/csharp/wpf/c-sharp-tutorial-wpf-uvod-a-prvni-formularova-aplikace>.

[8] Čermák M. *Vícevrstvá architektura: popis vrstev* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.cleverandsmart.cz/vicevrstva-architektura-popis-vrstev/>.

[9] Čermák M. *Vícevrstvá architektura: výhody a nevýhody* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.cleverandsmart.cz/vicevrstva-architektura-vyhody-a-nevyhody/>.

[10] Dajbych V. *MVVM: Model-View-ViewModel* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel>.

[11] Developer's Guide to Microsoft Prism Library 5.0 for WPF *15: Extending the Prism Library 5.0 for WPF* [online]. [cit. 1. 12. 2015] Dostupné z URL: <https://msdn.microsoft.com/en-us/library/gg430866(v=PandP.40).aspx>.

[12] Royce W. Winston, *Managing the deplyment of large software systems* [online]. [cit. 4. 22. 2015] Dostupné z URL: <http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>.

[13] Kasalová, Z., *Porovnání CASE nástrojů objektově orientované analýzy a návrhu* [online]. [cit. 4. 23. 2015] Dostupné z URL: <https://is.muni.cz/th/50767/fi_m/DP-Zuzana_Kasalova.pdf>.

[14] Kacvinský, M., *Agilní metodologie řízení vývoje softwaru* [online]. [cit. 4. 23. 2015] Dostupné z URL: <http://is.muni.cz/th/207629/fi_m/kacvinsky-dp.pdf>.

[15] Signatári Agile Manifesto, *Manifesto for Agile Software Development* [online]. [cit. 5. 9. 2015] Dostupné z URL: <http://agilemanifesto.org/>.

[16] ECMA *Standard ECMA-334: C# Language Specification* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.

[17] Rotem-Gal-Oz A. *If you build it - will they come?* [online]. [cit. 5. 11. 2016] Dostupné z URL: <https://web.archive.org/web/20070504053354/http://www.ddj.com/blog/architectblog/archives/2006/07/frameworks_vs_l.html>.

[18] ECMA *Standard ECMA-335: Common Language Infrastructure (CLI)* [online]. [cit. 1. 12. 2015] Dostupné z URL: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.

[19] Evjen B., Glynn J., Nagel Ch., Skinner M., Watson K. *C# 2008 - Programujeme profesionálně* Brno: Computer Press, 2009. ISBN 978-80-251-2401-7.

[20] Dykstra, T. *Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)* [online]. [cit. 5. 12. 2016] Dostupné z URL: <http://goo.gl/SqSASU>.

[21] Haque Azad M. *Dependency Injection in WPF using Unity for Dummies* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://www.codeproject.com/Articles/137968/Dependency-Injection-in-WPF-using-Unity-for-Dummie>.

[22] Johnson T. *1. Díl WPF - Základní rozdíly oproti WF* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://www.itnetwork.cz/csharp/wpf/tutorial-wpf-c-sharp-zakladni-rozdily>.

[23] Jirava J. *Odhalení WPF - I.* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://xaml.cz/wpf/odhaleni-wpf-1/>.

[24] MSDN *Chapter 5: Layered Application Guidelines* [online]. [cit. 30. 11. 2015] Dostupné z URL: <https://msdn.microsoft.com/en-us/library/ee658109.aspx>.

[25] MSDN *Introduction to the C# Language and the .NET Framework* [online]. [cit. 30. 11. 2015] Dostupné z URL: <https://msdn.microsoft.com/cs-cz/library/z1zx9t92.aspx>.

[26] Petzold,Ch. *Mistrovství ve Windows Presentation Foundation* Brno: Computer Press, 2006. ISBN 978-80-251-2141-2.

[27] Landwerth I. [MSFT] *.NET Core is Open Source* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://blogs.msdn.com/b/dotnet/archive/2014/11/12/net-core-is-open-source.aspx>.

[28] The .NET Fundamentals Team *.NET Framework 4.6.1 is now available!* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://blogs.msdn.com/b/dotnet/archive/2015/11/30/net-framework-4-6-1-is-now-available.aspx>.

[29] Osborn J. *Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://www.windowsdevcenter.com/pub/a/oreilly/windows/news/hejlsberg_0800.html>.

[30] Blogs MSDN *Aplikace pro více zařízení (2.) – MVVM* [online]. [cit. 30. 11. 2015] Dostupné z URL: <https://blogs.msdn.microsoft.com/vyvojari/2013/03/20/aplikace-pro-vce-zazen-2-mvvm/>.

[31] Thakkar K. *Difference between MVC vs. MVP vs. MVVM* [online]. [cit. 30. 11. 2015] Dostupné z URL: <http://blogs.k10world.com/technology/difference-between-mvc-vs-mvp-vs-mvvm/>.

[32] Čápka D. *Úvod do UML* [online]. [cit. 15. 3. 2016] Dostupné z URL: <http://www.itnetwork.cz/navrhove-vzory/uml/uml-uvod-historie-vyznam-a-diagramy//>.

[33] Algoritmy.net *Návrhové vzory* [online]. [cit. 15. 3. 2016] Dostupné z URL: <https://www.algoritmy.net/article/51224/Navrhove-vzory>.

[34] Object Management Group *Unified Modeling Language™ (UML®)* [online]. [cit. 15. 3. 2016] Dostupné z URL: <http://www.omg.org/spec/UML/>.

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

API	Application Programming Interface
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CTS	Common Type System
CSS	Cascading Style Sheets
CRUD	Create-Read-Update-Delete
DB	Database
DPI	Dots per inch
EF	Entity Framework
GUI	Graphical User Interface
GDI	Graphical Device Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
NGWS	Next Generation Windows Services
OOP	Object-oriented programming
ORM	Object-relational mapping
OS	Operačný systém - Operating system
UI	User Interface
UML	Unified Modeling Language
WF	Windows Forms
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

ZOZNAM PRÍLOH

A ZDROJOVÉ SÚBORY

A.1 Štruktúra súborov prílohy

PrilohaBP_PatrikSvikruha_2016.zip
- _Datasheets ..... Priečinok s datasheetmi súčiastok
- _Images ..... Priečinok s obrázkami súčiastok
- _Logs ..... Priečinok textovým výstupom logeru
  - logs.txt ..... Súbor pre zápis logov popísaný v kapitole 5.8
- _TestData ..... Priečinok s testovacími datami
  - TestData.xlsx ..... Zošit s troma s testovacími datami
- DESPREK.Solution ..... Priečinok so solution a projektami A.1.1

A.1.1 DESPREK.Solution

Solution je vytvorené v Visual Studio a projekty sú vo verzii .Net Frameworku 4.5.2.

DESPREK.Solution ..... Priečinok so solution a projektami
- DESPREK.Application ..... projekt *WPF* aplikácie A.1.2
- DESPREK.BusinessLayer ..... projekt *Class Library* A.1.3
- DESPREK.BusinessLayer.Test ..... projekt *Unit Test* A.1.4
- DESPREK.Helper ..... projekt *Class Library* A.1.5
- DESPREK.Models ..... projekt *Class Library* A.1.6
- DESPREK.Models.Test ..... projekt *Unit Test* A.1.7
- DESPREK.Services ..... projekt *Class Library* A.1.8
- packages ..... Priečinok s NuGet balíčkami
  - Catel.Core.4.4.0
  - Catel.Extensions.Controls.4.4.0
  - Catel.Extensions.EntityFramework6.4.4.0
  - Catel.Fody.2.14.0
  - Catel.MVVM.4.4.0
  - CommonServiceLocation.1.0.0
  - CommonServiceLocator.1.0
  - EntityFramework.6.1.3
  - ExcelDataReader.2.1.2.3
  - Fody.1.29.3
  - MicrosoftExpressionInteractions.3.0.40218.0
  - Obsolete.Fody.4.0.5
  - SharpZipLib.0.86.0
- DESPREK.sln ..... Súbor solution Desprek

A.1.2 Projekt DESPREK.Application

- DESPREK.Application ..... projekt *WPF* aplikácie
  - Converters ..... Priečinok s konvertormi
    - FallbackConverter.cs
    - FullImagePathConverter.cs
    - InvertableBooleanToVisibilityConverter.cs
    - StringToBooleanConverter.cs
  - Icons ..... Priečinok s obrázkami
  - Properties
    - AssemblyInfo.cs
    - Settings.settings ..... + code behind Settings.Designer.cs
  - ViewModels ..... Priečinok s ViewModel-mi
    - ChangePasswordViewModel.cs
    - MainWindowViewModel.cs
    - ManagerForUsersViewModel.cs
    - NewCapacitorViewModel.cs
    - NewInductorViewModel.cs
    - NewResistorViewModel.cs
    - NewUserViewModel.cs
    - ResultMessageUserControlViewModel.cs
    - UserDetailViewModel.cs
    - MainWindowViewModel.cs
  - Views. Priečinok s View-s, „+p“ ´=> pridanie partial triedy (.cs)
    - ChangePasswordView.xaml ..... +p
    - InfoAboutAppView.xaml ..... +p
    - MainWindow.xaml ..... +p
    - ManagerForUsersView.xaml ..... +p
    - NewCapacitorView.xaml ..... +p
    - NewInductorView.xaml ..... +p
    - NewResistorView.xaml ..... +p
    - NewUserView.xaml ..... +p
    - ResultMessageUserControlView.xaml ..... +p
    - UserDetailView.xaml ..... +p
    - MainWindow.xaml ..... +p
  - App.config ..... Konfiguračný súbor zapísaný v XML
  - App.xaml ..... Konfiguračný riadiaci súbor zapísaný v XAML
  - DESPREK.Application.csproj
  - packages.config ..... Konfiguračný súbor NuGet balíčkov

A.1.3 Projekt DESPREK.BusinessLayer

```
DESPREK.BusinessLayer ..... PROJEKT Class Library
├── Properties
│   ├── AssemblyInfo.cs
│   └── Settings.settings ..... + CODE BEHIND SETTINGS.DESIGNER.CS
├── App.config
├── DataBusinnessLayer.cs
├── DESPREK.BusinessLayer.csproj
├── LogToFileWriter.cs
└── packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV
```

A.1.4 Projekt DESPREK.BusinessLayer.Test

```
DESPREK.BusinessLayer.Test ..... PROJEKT Unit Test
├── Properties
│   └── AssemblyInfo.cs
├── App.config
├── DESPREK.BusinessLayer.Test.csproj
├── packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV
└── UnitTestBusinesslayer.cs
```

A.1.5 Projekt DESPREK.Helper

```
DESPREK.Helper ..... PROJEKT Class Library
├── ConvertorToEngFormat
│   └── ConvertorToEngineeringFormat.cs
├── ExcelParser
│   ├── ExcelDataParser.cs .2 ExtensionClass
│   └── Extensions.cs
├── Properties
│   └── AssemblyInfo.cs
├── SecurePasswordHasher
│   └── SecurePasswordHasher.cs
├── Constants.cs
├── DESPREK.Helper.csproj
└── packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV
```

A.1.6 Projekt DESPREK.Models

- DESPREK.Models ..... PROJEKT *Class Library*
  - Base
    - Implementation
      - Component.cs
      - Model.cs
      - PassiveComponent.cs
    - Models
      - Logger
        - LoggerMessage.cs
      - PassiveComponents
        - CapacitorModel.cs
        - InductorModel.cs
        - ResistorModel.cs
      - ResultMessage.cs
      - UserModel.cs
  - ImportExport
    - ImportData.cs
  - Properties
    - AssemblyInfo.cs
  - DESPREK.Models.csproj
  - packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV

A.1.7 Projekt DESPREK.Models.Test

- DESPREK.Models.Test ..... PROJEKT *Unit Test*
  - Properties
    - AssemblyInfo.cs
  - DESPREK.Models.Test.csproj
  - packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV
  - UnitTestModels.cs

A.1.8 Projekt DESPREK.Services

- DESPREK.Services ..... PROJEKT *Class Library*
  - Migrations
    - 201603102318211_initial migration 11 03 2016.cs
    - 201603102318211_initial migration 11 03 2016.Designer.cs
    - 201603102318211_initial migration 11 03 2016.resx
    - Configuration.cs
  - Properties
    - AssemblyInfo.cs
  - Repository
    - Interfaces
      - ICapacitorsRepository.cs
      - IInductorsRepository.cs
      - ILogsRepository.cs
      - IResistorsRepository.cs
      - IUsersRepository.cs
    - CapacitorsRepository.cs
    - InductorsRepository.cs
    - LogsRepository.cs
    - ResistorsRepository.cs
    - UsersRepository.cs
  - App.config
  - DESPREK.Services.csproj
  - DesprekDBContext.cs
  - DesprekDbContextConfiguration.cs
  - packages.config ..... KONFIGURAČNÝ SÚBOR NUGET BALÍČKOV