VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2017 Radim Kafka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Radim Kafka**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Ingeteam a.s.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Konzultant bakalářské práce: Bc. Pavel Kocich

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017 .....

Ingeteam
Technologická 371/1
708 00 Ostrava-Pustkovec

Rád bych poděkoval firmě Ingeteam a.s., Česká republika za umožnění absolvování odborné praxe, a mému konzultantovi Bc. Pavlovi Kocichovi za cenné rady během odborné praxe.

Abstrakt

Tato bakalářská práce pojednává o absolvování odborné praxe ve firmě Ingeteam a.s., Česká republika. Práce popisuje zadané úkoly a jejich praktické řešení s ukázkou kódu. V závěru je shrnuto využití znalostí při vykonávání této praxe a její celkové zhodnocení.

Klíčová slova: WPF, C#, XAML, Ingeteam

Abstract

This bachelor thesis describes individual professional practice in the company Ingeteam a.s., Česká republika. The thesis describes given tasks and their solutions with code demonstration. In the conclusion of the thesis is summary of applicated knowledge and overall evaluation.

Key Words: WPF, C#, XAML, Ingeteam

Obsah

Seznam použitých zkratek a symbolů

HMI	–	Rozhraní mezi člověkem a strojem
MVVM	–	Model,View, ViewModel
PLC	–	Programovatelný logický automat
ST	–	Structured Text
UI	–	Uživatelské rozhraní
WPF	–	Windows Presentation Foundation
XAML	–	Extensible Application Markup Language
XML	–	Extensible Markup Language

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Má odborná praxe proběhla ve firmě Ingeteam a.s., Česká republika. Během svého působení jsem pracoval na několika projektech, které jsou popsány v následující části dokumentu. Bakalářskou práci formou praxe jsem vybral z důvodů, že si uvědomuji, jak je praxe důležitá, obzvlášť v oboru informatiky, pro budoucí uplatnění na trhu práce, a taky jsem si chtěl odzkoušet mé znalosti nabyté během studia na vysoké škole.

1.1 Popis odborného zařízení firmy

Společnost Ingeteam a.s., Česká republika, patřící do skupiny Ingeteam představuje významného a tradičního dodavatele komplexních řešení a služeb v oblasti průmyslové automatizace. Od roku 1993, kdy byla naše společnost založena jako Ingelectric a.s. (na Ingeteam a.s. byla přejmenována v roce 2007), jsme se podíleli na vývoji a realizaci desítek projektů, u nichž kvalita, spolehlivost a přesnost jsou základní podmínkou úspěchu. Rozsahem služeb, od nalezení optimálního zákaznického řešení až po zajištění komplexní dodávky na klíč, prokazujeme naše zkušenosti, spolehlivost a profesionalitu. Využití moderních technologií, hledání inovativních řešení a neustálé zkvalitňování našich služeb jsou základními předpoklady k dalšímu rozvoji naší společnosti a k úspěšné realizaci požadavků našich zákazníků.[1]

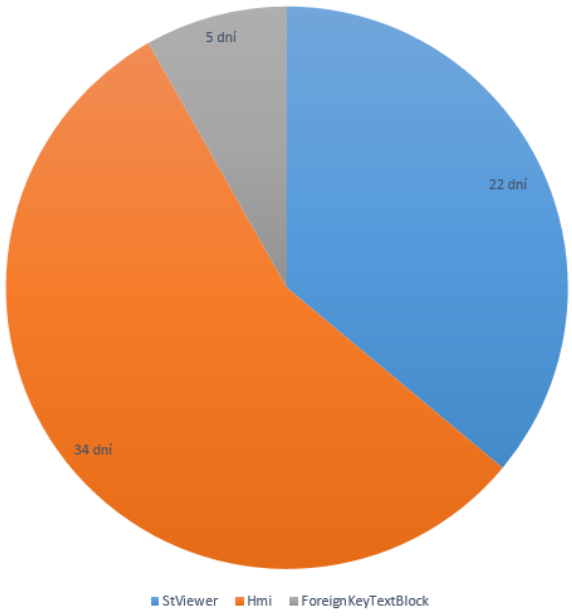
1.2 Pracovní zařazení studenta

Byl jsem zařazen do IT oddělení na pozici programátora C#, kde jsem pracoval pod vedením mého konzultanta Bc. Pavla Kocicha.

[1] Ingeteam [online]. Ostrava, 2008 [cit. 2016-04-28]. Dostupné z: http://www.ingeteam.cz/

2 Seznam úkolů zadaných studentovi s vyjádřením jejich časové náročnostil

Po dobu absolvování odborné praxe ve společnosti Ingeteam a.s, Česká republika jsem pracoval na třech projektech. Při programování jsem využíval vývojářské prostředí Microsoft Visual Studio 2013 a verzovací software git s grafickou nadstavbou TortoiseGIT. První projekt, na kterém jsem pracoval, byl firemní nástroj, ST Viewer. Do firmy jsem docházel dva až tři dny v týdnu, podle rozvrhu ve škole. Celkově jsem na projektu strávil dvacet dva dní. Druhý projekt na, které jsem pracoval, byla vizualizace/HMI automatizovaného procesu měření složení atmosféry ve vysoké peci. Jednalo se o podstatně rozsáhlejší projekt než předchozí, čemuž odpovídá i čas strávený na něm. Celkově jsem na projektu strávil 34 dní. Poslední projekt, na kterém jsem pracoval byla komponenta ForeignKeyTextBlock, která byla použita v DataGridu pro zobrazení vlastnosti konkrétního objektu z kolekce na základě shody vlastnosti s hodnotou, kterou uživatel zadal.



Obrázek 1: Časová náročnost projektů.

3 Popis použitých technologii

3.1 .NET

Je framework vyvíjen společností Microsoft. První verze .NET frameworku byl a vydána v roce 2002. Současná verze .NET frameworku je verze 4.6.2 vydána v srpnu roku 2016. Zdrojový kód je přeložen do mezikódu nazvaným Common Intermediate Language (CIL) tento kód je následně interpretován virtuálním strojem, Common Language Runtime (CLR). Nejznámější jazyky podporované .NETem jsou C# a Visual Basic .NET, mezi další patří funkcionální programovací jazyk F# nebo jazyk velmi podobný Javě, J#. Kde C# patří mezi nejpoužívanější jazyk, a který byl přizpůsobený právě pro .NET. Z důvodu, že všechny jazyky používající .NET se překládají do CIL, tak nabízejí vzájemnou kompatibilitu, takže je např. možné mít aplikaci psanou v jazyce C# a používat knihovny psané ve Visual Basic .NET. Mezi knihovny patřící do .NET frameworku jsou ASP.NET, který slouží pro vývoj webových aplikací, WCF, technologie pro vývoj webových služeb a komunikační infrastruktury aplikací, WPF technologie pro tvorbu uživatelských rozhraní, Language Integrated Query (LINQ), který objektově přistupuje k datům v databázím, XML a objektech implementující rozhraní IEnumerable.

3.2 C#

Jedná se objektově orientovaný, generický, typově bezpečný jazyk vyvíjen společností Microsoft. C# byl založen na jazycích Java a C++. Syntaxe jazyka je podobná jazykům C, C++ a Java. První verze byla vydána v roce 2002 ve verzi 1.0, která obsahovala generičnost, anonymní metody, statické třídy atd. Od verze 1.0 do aktuální verze 6.0, která byla vydána v roce 2015, přibyly prvky jako anonymní typy, lambda výrazy, bindování, asynchronní metody a další. Pomocí jazyka C# lze psát jak desktopové aplikace, tak i webové a mobilní.

3.3 Windows Presentation Foundation

Windows Presentation Foundation(dále jen WPF) je framework pro tvorbu uživatelských rozhraních, který je v .NET frameworku od verze 3.0. WPF je druhým frameworkem pro tvorbu uživatelských rozhraních v .NET frameworku, starší framework pro tvorbu uživatelských rozhraní je Windows Forms, který ještě není označen jako zastaralý. Nicméně WPF je technologicky mnohem vyspělejší, a v současné době téměř není důvod upřednostnit Windows Forms oproti WPF. Vykreslování uživatelského rozhraní probíhá na grafické kartě pomocí Direct3D, což umožňuje vykreslování složitých tvarů a různých barevných schémat a použití vysokého rozlišení.

3.4 Extensible Application Markup Language

Extensible Application Markup Language, zkráceně XAML (čteno zaml) je značkovací jazyk používán pro tvorbu uživatelských rozhraní vydaný v roce 2008. Vychází z jazyka XML s přidanými

značkami pro vytváření uživatelských rozhraní. Syntaxe je identická s XML stejně jako struktura dokumentu, tvořená stromem. XAML slouží pro tvorbu uživatelských rozhraní ve WPF. Jmenné prostory v XAMLu stejně jako v XML jsou definovány pomocí XML Name Space (xmlns), jenž je ekvivalentem klíčového slova using.

3.5 GIT

Jedná se o distribuovaný systém správy verzí, vytvořen Linusem Torvaldsem pro vývoj jádra Linux. Distribuovaný je z toho důvodu, že každý uživatel má celý repozitář (úložiště) lokálně na svém zařízení, na rozdíl od systémů centralizovaných, které mají jeden repozitář, se kterým se uživatelé synchronizují. Příkladem centralizovaného verzovacího systému je Apache Subversion nebo Team Foundation Server od společnosti Microsoft. Hlavní výhoda distribuovaného systému je, v případě, kdy je potřeba se vrátit na starší verzi, není nutné mít připojení k serveru. Základní akce ve verzovacích systémech jsou commit, který aktuální změny uloží do repozitáře. Příkaz pull stáhne repozitář (nebo jen chybějící část), push naopak nahraje repozitář(nebo jen nově přidané část) na server.

4 Popis projektů

4.1 StViewer

Můj první úkol byl naprogramovat firemní aplikaci pro zobrazování ST souborů. Na tomto projektu jsem se naučil nejvíce nových věcí. Firma pro vytváření UI používá framework WPF, se kterým jsem před tím nikdy nepřišel do styku, a oproti frameworku Windows Forms, jež jsem znal, se liší v moha ohledech. Použití návrhového vzoru Model View ViewModel bylo také novinkou.

Jedná se o návrhový vzor, který řeší problematiku oddělení uživatelského rozhraní a logiky u frameworku WPF. Odděluje data, stav aplikace a uživatelské rozhraní. Tento návrhový vzor se skládá ze tří částí Model, View a ViewModel. ViewModel je asi nejpodstatnější částí, tato vrstva uchovává stav aplikace.

View prezentuje uživatelské rozhraní, které je definováno (většinou) pomocí jazyka XAML. Model popisuje data, se kterými aplikace pracuje, např. se může jednat o na mapované databázové tabulky. ViewModel představuje mezi vrstvu, která propojuje View s Modelem, a uchovává stav aplikace pomocí vlastností. ViewModel musí implementovat rozhraní `INotifyPropertyChanged`, které implementuje událost PropertyChanged, díky které se do View propagují změny jednotlivých vlastností (metodu, kterou zaregistrujeme k události PropertyChanged musíme v settru volat s argumentem typu string obsahující název vlastnosti, kterou právě měníme). ViewModel je nezávislý na View a neměl by mít o něm žádnou informaci.

Další, pro mně, novinkou bylo verzování souborů. Pro verzování souborů jsme používali verzovací systém GIT s UI nadstavbou TortoiseGIT.

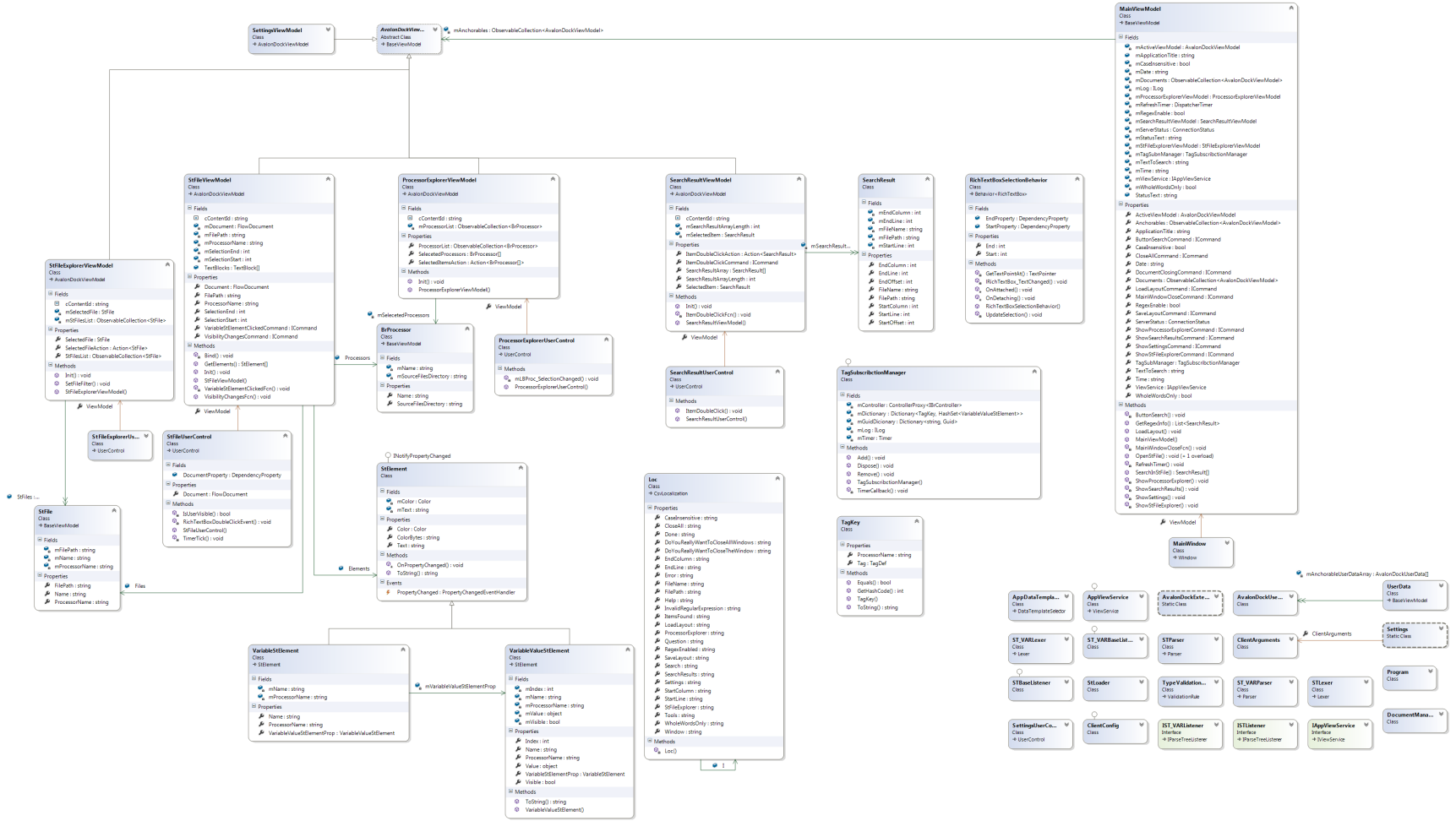
Na projektu jsem pracoval sám, pod vedením mého konzultanta. Aplikace měla za úkol zobrazit PLC procesory a jejich soubory, které se daly následně otvírat. St soubor je zdrojový kód pro PLC v jazyce ST.

Jako první prvky do uživatelského rozhraní jsem přidal dva Listboxy. Jeden, který zobrazoval všechny dostupné PLC procesory a druhý pro zobrazení zdrojových ST souborů daných procesorů. Názvy procesorů jsem získal podle názvů kořenových složek, ze kterých jsem načítal ST soubory. Názvy daných souboru jsem získal stejným způsobem jen o úroveň níže v adresářové struktuře. Aby bylo vidět, který soubor patří, k jakému procesoru do kolekce všech ST souborů jsem přidal tzv. „dummy“ položku, která obsahovala jen název souboru, a programově jsem ošetřil, že pokud objekt nemá cestu k souboru, bude zobrazen jinak než ostatní. Při vybrání procesoru/ů se zobrazily soubory pouze daného/daných procesoru/ů. Toto jsem realizoval pomocí metody `OnSelectedIndexChanged`, která vracela pole indexů vybraných položek. Další prvek, který jsem přidal, byl RichTextbox. RichTextbox je prvek dost podobný TextBoxu, který je ale vhodnější pro zobrazení většího množství textu a nabízí širší paletu formátování a stylování textu. Pomocí události DoubleClick na ListBoxu s ST soubory se daný soubor otevřel a pomocí Parseru, který znal syntax jazyka vracel objekty tříd VariableStElement a VariableValueStEle-

ment, které dědily ze třídy StElement. Díky polymorfismu jsem všechny objekty vložil do kolekce třídy StElement. Kolekci objektů jsem do RichTextboxu vkládal pomocí bindování. RichTextbox se již v pořádku otvíral a začal jsem do něj vkládat text, konkrétně vlastnost Text z třídy StElement. Pokud se jednalo pouze o StElement text se vypsal bílou barvou, pokud se jednalo o VariableStElement text byl žlutý a hned za ním se vypsal VariableValueStElement. VariableStElement reprezentoval název proměnné a VariableValueStElement jeho aktuální hodnotu, která se později četla ze serveru.

Nyní bylo potřeba načítat hodnoty VariableValueStElement ze serveru. S tím to úkolem mi značně pomáhal můj konzultant, protože se využívalo firemních služeb pro komunikaci se serverem. Pro snížení zátěže sítě se stahovaly pouze hodnoty, které byly zobrazeny právě v RichTextboxu. K tomuto účelu jsem vytvořil třídu TagSubscriptionManager, která se starala o právě zobrazené VariableValueStElement.

Jako správný textový prohlížeč musela mít aplikace vyhledávání textu s klasickými možnostmi jako je vyhledávání bez ohledu na velikost písma, regulární výraz nebo klasické vyhledání zadaného řetězce. Vyhledávání probíhalo ve všech souborech všech procesorů a výsledek se vypsal do GridView s názvem souboru, na jakém řádku se nalezený text nachází a na jaké pozici v řádku. K vyhledávání jsem použil třídu Regex, která zajišťuje práci s regulárními výrazy. Po dvojkliku na řádek GridView se otevřel daný soubor (pokud už nebyl otevřený) a najelo se na danou část textu a nalezený výsledek bude označený. V tom to bodě se vyskytl zatím největší problém, a to bylo špatné označování textu. Pomocí debuggingu a vyhledání na internetu jsem přišel chybu, která se nacházela ve Focusu. Focus zajišťuje jaká část uživatelského rozhraní je aktivní. V mém případě byl focus automaticky nastaven na GridView, ve kterém byly zobrazeny výsledky, a tím pádem se nikdy nemohl text v RichTextboxu označit. Vše vyřešilo manuální nastavení Focusu na RichTextbox po otevření okna. Zbývalo už jen zajistit více jazyčnost. Tu zajišťovala statická třída Loc která využívala návrhového vzoru Singleton. Tento návrhový vzor se využívá v případě, kdy potřebujeme v aplikaci pouze jednu instanci třídy. Všechen text, který musel být více jazyčný se vázal na proměnné ze třídy Loc. Proměnné měly nastavené výchozí hodnoty, podle kterých se z csv souboru získaly názvy přeložené v případě, že byl jazyk změněn.



Obrázek 2: Třídní diagram StVieweru.

4.2 HMI

Můj další projekt byl už daleko rozsáhlejší, jednalo se o komerční produkt vizualizace/HMI automatizovaného procesu měření složení atmosféry ve vysoké peci. Aplikace byla připojena k PLC, které řídí samotný proces měření a slouží k ovládání a zobrazování aktuálního stavu a historických měření. Aplikace byla rozdělena na dvě části, klientskou a serverovou. Serverová část komunikovala pomocí protokolu TPC s PLC, a následně data zprostředkovávala klientské aplikaci. Já jsem na této aplikaci měl za úkol pracovat na klientské části, nicméně občas jsem musel zasáhnout i do části serverové. Můj úkol byl vytvářet obrazovky (HMI), které zobrazovaly aktuální stav jednotlivých prvků, které obsluhovalo PLC. Práce na tomto projektu z velké části obsahovala práci v jazyce XAML. Základní znalosti jazyka XAML jsem získal během práce na mém prvním projektu, a na tomto projektu se mé znalosti mnohokrát rozšířily.

4.2.1 Tvoření symbolů

Před začátkem tvoření obrazovek, bylo potřeba vytvořit symboly, které byly následně použity na obrazovkách. Každý symbol byl reprezentován jako `UserControl`, což je ve WPF třída, která slouží pro vytváření UI prvků. K tvoření symbolů jsem využil třídu `Canvas`, do které se vkládají prvky třídy `UIElement`, a které mohou být absolutně pozicovány vůči prvku Canvas. Pro tvoření symbolů jsem využil třídu `Path`, která využívá Path Markup Syntax pro popis geometrických tvarů v jazyce XAML.

```
<Canvas>
   <Path Stroke="Black" Fill="Gray" Data="M 10,100 L 20,100" />
</Canvas>
```

Výpis 1: Příklad Path Markup Syntax v prvku Canvas

Path Markup Syntax využívá tzv. draw commands pro definování jednotlivých geometrických tvarů. Definování tvaru provádíme v atributu Data, který vždy začíná písmenem M, jež představuje počáteční bod, poté můžeme využít jakýkoli draw command např. L slouží pro rovnou čáru do bodu, které mu nastavíme souřadnice xy. Symboly musely zobrazovat aktuální stav prvků. Každý prvek měl svou třídu, která byla vygenerována z definiční listiny daného prvku. Třída obsahovala proměnné, které určovaly aktuální stav prvku. Aby se prvek dynamicky měnil podle aktuálních hodnot použil jsem DataTrigger, který můžeme vnímat jako if, který spojíme s proměnnou, nastavíme hodnotu a určíme, které vlastnosti prvku se změní, a jakou hodnotu nabudou. V případě, že svázaná proměnná se bude rovnat určené hodnotě aplikují se námi zadané změny. To vše musíme definovat ve stylu námi požadovaného prvku.

```
<Path Data="M18,3 10,8 18,12Z" Stroke="Black" StrokeThickness="0.5" Canvas.Top=
    "1.5" Canvas.Left="6">
   <Path.Style>
      <Style TargetType="Path">
```

```
            <Setter Property="Fill" Value="White"/>
          <Style.Triggers>
          <DataTrigger Binding="{Binding Sts.OpndA}" Value="True">
            <Setter Property="Fill" Value="lime"/>
          </DataTrigger>
          </Style.Triggers>
        </Style>
    </Path.Style>
</Path>
```

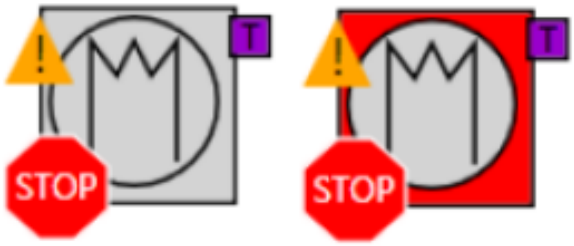
Výpis 2: Ukázka DataTriggeru

Další způsob, jakým je možno docílit dynamické změny je pomocí Converteru. Converter se používá v případě, kdy datový typ vstupní proměnné neodpovídá požadovanému datovému typu. Definice converteru se neprovádí v jazyce XAML, ale pomocí jazyka C#. Abychom mohl využít converter musíme implementovat interface `IValueConverter` pokud jako vstup je pouze jedna proměnná. V případě, že potřebujeme získat výsledek z více proměnných využijeme interface `IMultiValueConverter`. Při použití obou rozhraní musíme implementovat metody Convert a ConvertBack. Výhoda Converteru oproti DataTriggeru je, že nám poskytuje možnost znovu využít converter, bez nutnosti ho psát opakovaně, stačí pouze referenci na něj přidat do Resources prvku a změna v kódu converteru se promítne všude, kde je converter použit, na rozdíl od DataTriggeru, který musíme všude definovat znovu, a tím pádem když provedeme nějakou změnu tak ji musíme provést na všech místech kde je DataTrigger použit. Nadruhou stranu pomocí DataTriggeru můžeme změnit více vlastností prvku, kdežto converter pouze mění jednu proměnnou. V mém případě jsem converter použil pro zobrazení symbolu STOP a Warning, kde má vstupní proměnná byla datového typu bool a požadovaný datový typ byl Visibility. V případě, že proměnná byla rovna 0 výstup z converteru byl Hidden a, když vstup byl roven 1 converter vracel hodnotu Visible.

Mnohé prvky měly některé části podobné např. zobrazení operačního módu prvku, z toho důvodu jsem pro tyto prvky vytvořil vlastní objekty, které jsem následně vkládal do symbolů, to ušetřilo spoustu času v případě, kdy se prvek musel nějakým způsobem měnit. Po vytvoření několika symbolů jsem si všiml, že mnoho symbolů sdílí veškeré chování a graficky se liší jen v geometrickém tvaru uvnitř. Z toho důvodu mi přišlo vhodné si vytvořit styl, ve kterém budou prvky,jenž jsou stejné u symbolů. K vytvoření stylu jsem použil Resource Dictionary. Aby se styl automaticky aplikoval, musel být uložen ve složce Themes s názvem v souboru Generic.

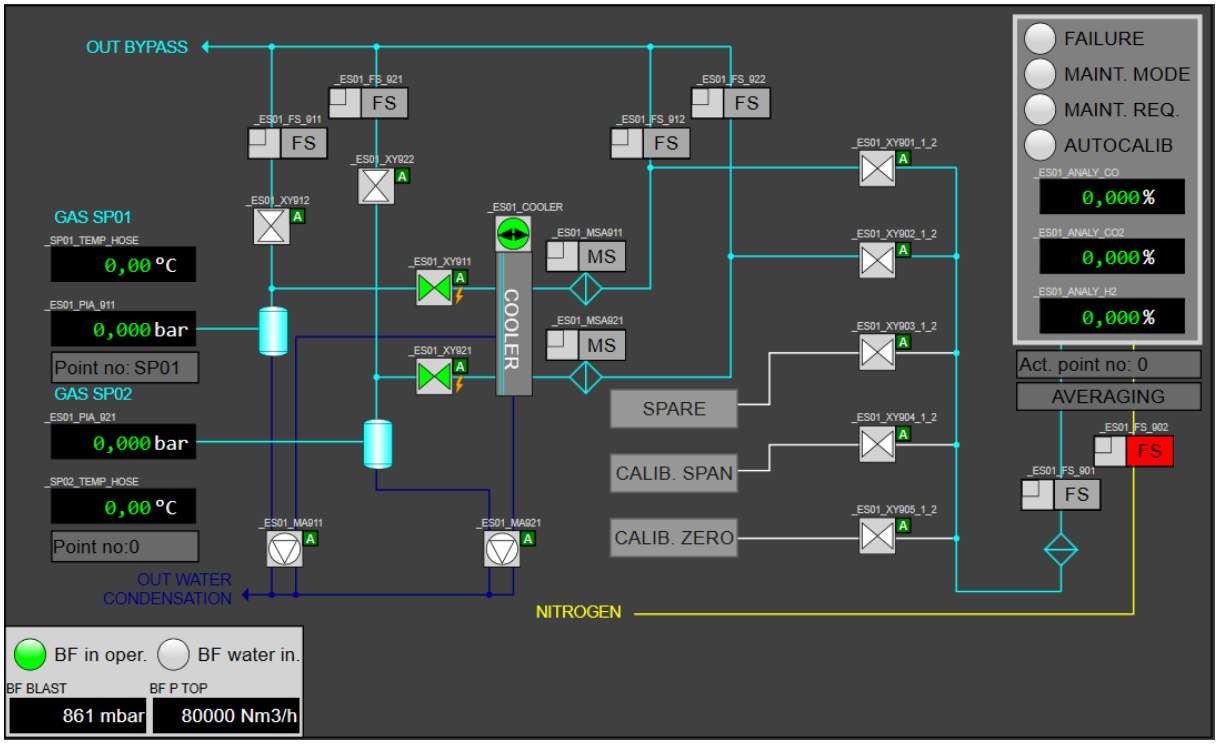
4.2.2 Vytáření obrazovek

Část vytváření obrazovek byla velice časově náročná z důvodu relativně velkého počtu obrazovek. Aplikace obsahovala bez mála 15 obrazovek a každá obrazovka byla jedinečná. Obrazovky jsem vytvářel podle předlohy, kterou jsem obdržel od kolegy, který se staral o práci s PLC na



Obrázek 3: Symbol Heater.

tomto projektu. K mému překvapení symboly, které jsem vytvářel byly využity pouze na dvou obrazovkách, nicméně nadřízeným se zpracování symbolů líbilo natolik, že se budou nadále používat i dalších projektech. Na obrazovkách, kde byly symboly použity jsem opět využil prvek canvas, do kterého jsem vkládal mnou vytvořené symboly, a následně pomocí prvku path jsem vytvářel spoje mezi jednotlivými prvky.



Obrázek 4: Obrazovka Analyser overview.

Vytváření všech obrazovek mělo vesměs pořád stejnou náplň, a to vytváření různých grafických prvků v jazyce XAML. Poslední obrazovka, kterou jsem měl vytvořit, měla představovat sekvence během měření, za úkol jsem měl vymyslet způsob generování obrazovky z XML dokumentu, u kterého jsem měl určit jeho strukturu. Obrazovka měla být rozdělená na dvě poloviny,

kde na levé části byly zobrazeny hlavní sekvence a na straně pravé zobrazené podsekvence, které se měly zobrazit po kliknutí na dané sekvence, které podsekvence obsahovaly. Po konzultaci s kolegou, který se mnou na projektu pracoval, jsme došli k rozhodnutí, že nejlepší způsob bude definovat v XML dokumentu jednotlivé řádky. Pro to jsem vytvořil třídu Stage, která mimo jiné v sobě obsahuje pole datového typu Stage s názvem Items. Jednotlivé vlastnosti třídy Stage jsem označil jako XML atributy, což ulehčilo čtení ze souboru, které prostřednictvím XML serializace.

```
<Stage Name="_GAS_Sequences">
   <Items>
      <Stage Name = "MainSequences">
         <Items>
            <Stage Name="_GAS_StageA">
            /*V tomto bloku se nachazely definice jednotlivych radku*/
            </Stage>
            <Stage Name="_GAS_StageB">
            /*V tomto bloku se nachazely definice jednotlivych radku*/
            </Stage>
         </Items>
      <Stage Name="_GAS_BasicSequences">
         <Items>
            <Stage Name_SP01_SqFailSafe">
            /*V tomto bloku se nachazely definice jednotlivych radku*/
            </Stage>
            <Stage Name_SP01_SqTstN2">
            /*V tomto bloku se nachazely definice jednotlivych radku*/
            </Stage>
         </Items>
      </Stage>
   </Items>
</Stage>
```

Výpis 3: Definice XML

Při samotném vytváření obrazovky jsem se vydal směrem použití behavioru, protože jsem nepřišel na způsob jakým generovat prvky pomocí XAMLu. Behavior, česky chování, se používá v případech, kdy chceme přidat chování (funkcionalitu) k prvku, které může být znovu použitelná, nebo požadovaná funkcionalita nejde vytvořit použitím jazyka XAML. V mém případě se jednalo o kombinaci obou případů. K vytváření behavioru se nevyužívá jazyk XAML ale jazyk C#. Při vytváření musíme dědit s generické třídy Behavior, u které určujeme k jakému prvku behavior patří,v mém případě se jednalo o Grid. Behavior jsem na zval StageGridBehavior a přijímal objekt typu Stage a string s názvem dané stage, která se měla zobrazit. Při vložení Objektu Stage se zavolala funkce CreateStagesGrid, která pomocí cyklu foreach procházela pole Items z objektu Stage, a pro každý Item volala funkci GenerateGrid, která vracela objekt typu Grid. Tento Grid se poté vložil do StackPanelu a nastavilo se mu odsazení 20px z vrchní strany, aby byl výpis přehlednější.

```
private Grid GenerateGrid(Stage aStage)
{
   if (aStage == null)
   return null;
   var lStageItems = aStage.Items;
   if (lStageItems == null)
      return null;
   int lMaxColumns = 0;
   Grid lGrid = new Grid();
   lMaxColumns = aStage.GetItemsMaxLength();
   for ( int i = 0; i < lStageItems.Length; i++)
   {
      GenerateLabels(lGrid, lStageItems[i], lMaxColumns, i);
   }
   for (int i = 0; i < lStageItems.Length; i++)
   {
      lGrid.RowDefinitions.Add(new RowDefinition());
   }
   for (int i = 0; i < lMaxColumns; i++)
   {
      lGrid.ColumnDefinitions.Add(new ColumnDefinition());
   }
   lGrid.Height = lStageItems.Length * 40;
   return lGrid;
}
```

Výpis 4: Funkce GenerateGrid

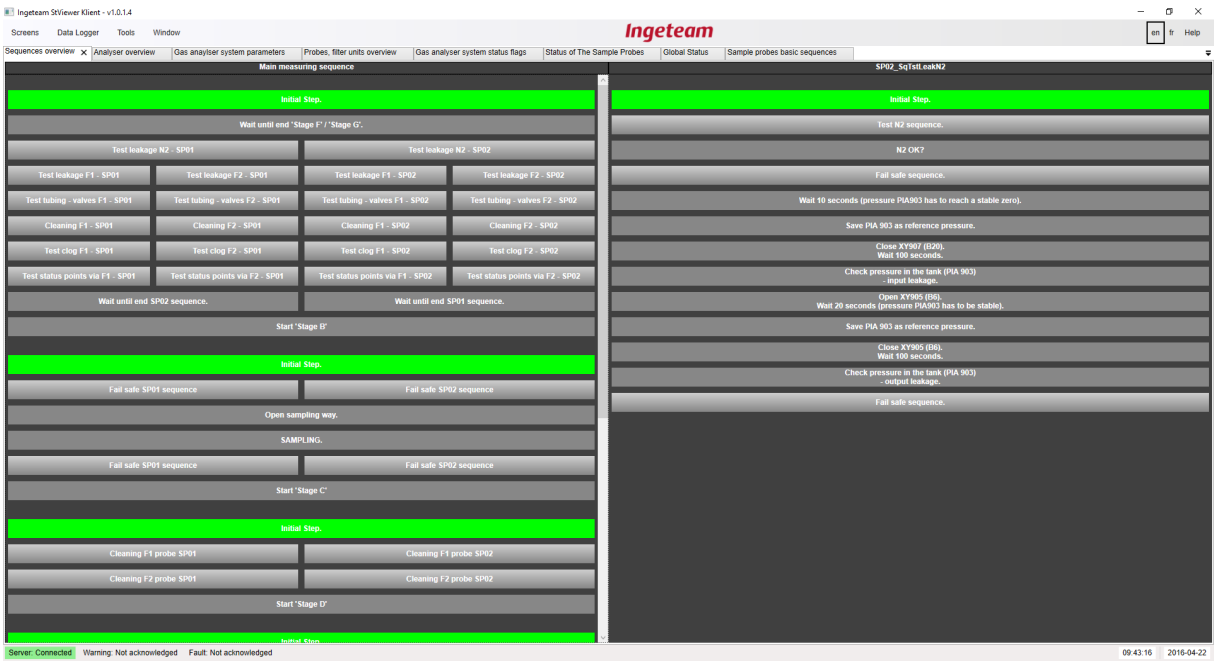
Tím, že v třídě Stage byla kolekce typu Stage, při generaci jsem využil znalostí z předmětu Algoritmy 2 a použil jsem rekurzivní volání funkce, kterou jsem nazval GenerateLabels.

```
private void GenerateLabels(Grid aGrid,Stage aStage,int aColumnSpan, int aRow =
0, int aColumn = 0);
```

Výpis 5: Definice funkce GenerateLabels.

Funkce v případě, že se jednalo o poslední prvek v hierarchii a neměl již žádné Items provedlo se vytvoření Labelu, pokud Items obsahoval položky, zavolala se opět funkce GenerateLabels. Stage, které měly vyplněnou vlastnost GoToSqName měly po kliknutí zobrazit dané podsekvence, aby je šlo lehce poznat, musely být jinak graficky znázorněny. Pro tento účel se nabízela možnost použít barevný přechod na daný label, nic méně mi osobně přechod připadal moc silný a

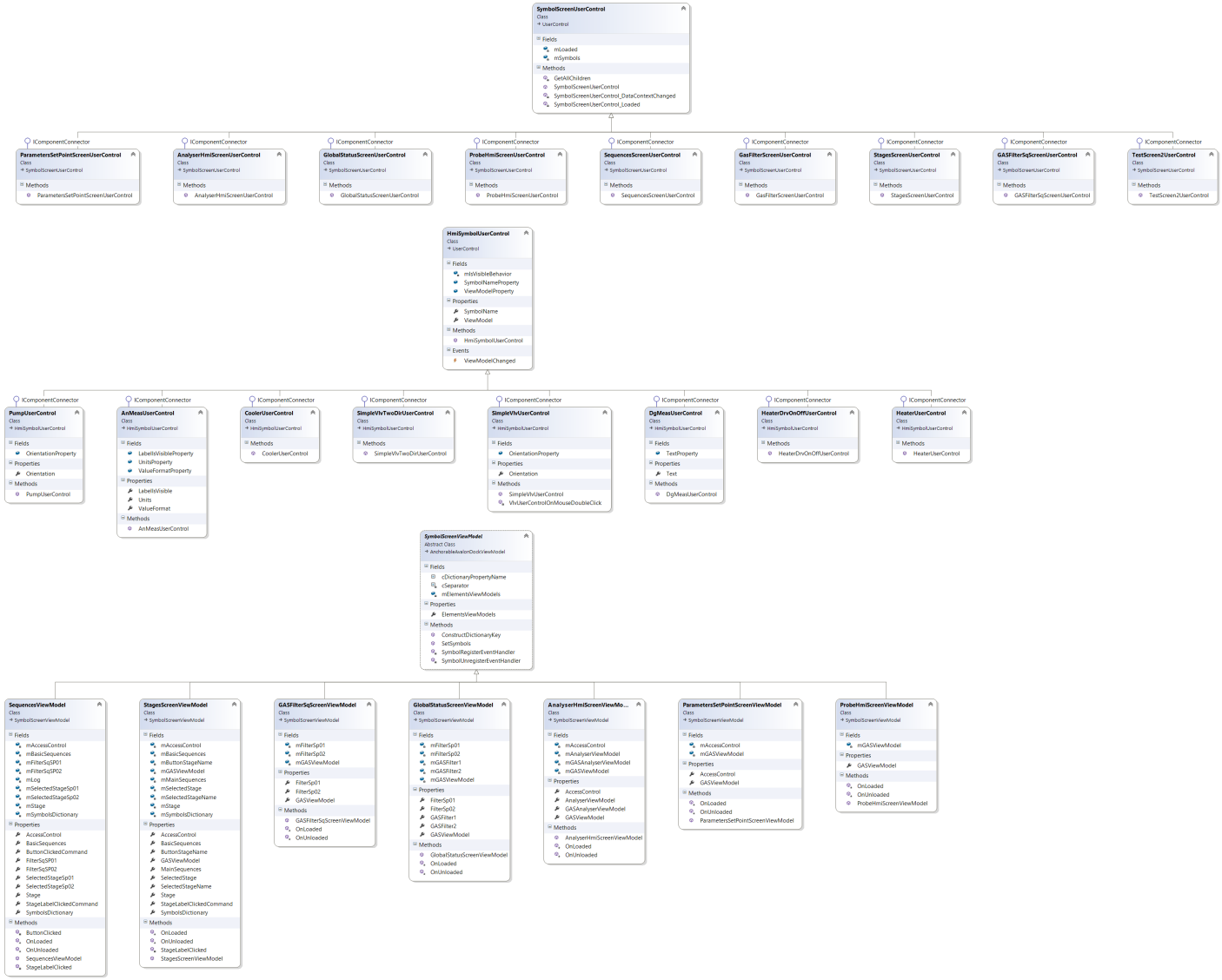
tak jsem zvolil cestu ponechání barvy labelu, a využil jsem možnosti, že label obsahuje vlastnost conent, do které se dá vložit jakýkoli objekt, a tak jsem do něj vložil další label, ve které jsem nastavil barevný přechod a snížil jeho viditelnost, což mělo za následek mnohem měkčí přechod barev. Možnost kliknutí jsem zprvu řešil pomocí události double click, kterou jsem implementoval přímo v behavioru, nicméně to bylo ve sporu s filozofií behavioru. A to, že behaviour by se měl starat pouze o to, jak prvek vypadá. Tak jsem místo události zvolil command. Další odlišnost byla, že některé labely musely obsahovat tlačítko. Opět jsem využil možnosti vložit jakýkoli prvek do vlastnosti content od labelu. Tentokrát jsem použil prvek StackPanel, který řadí za sebe prvky v něm. Celý software musel mít podporu dvojjazyčnost. Objekt Stage obsahoval vlastnosti TextLang1 a TextLang2, které určovaly popisky v obou jazycích. Pro svázání jsem využil firemní funkcionalitu DynLocBinding, která toto zajišťuje. Nicméně v mém testovacím XML souboru nebyly vlastnosti TextLang1 a TextLang2. Abych si otestoval funkčnost, udělal jsem jednoduchou aplikaci, která otevřela XML soubor a pomocí regexu získala text z atributu Name, a před něj vložila TextLang1 s daným názvem a TextLang2, do kterého vložila stejný název s příponou Lang2.



Obrázek 5: Obrazovka Sequences overview.

Poslední požadavek na aplikaci byl, že pozadí obrazovek mělo být stejné a měnitelné z jednoho místa. K tomuto jsem vytvořil třídu Screenbackground, která byla vytvořená podle návrhového vzoru singleton. Měla jedinou vlastnost background datového typu SolidColorBrush.

Třídní diagram vypadá stroze, protože spojení ViewModelů s View bylo definováno pomocí DataTemplateů, ve kterých je určeno jaký View patří ke kterému ViewModelu.



Obrázek 6: Třídní diagram aplikace HMI.

4.3 ForeignKeyTextblock

Tento prvek byl vytvořen za účelem použití v DataGridu v režimu zobrazení. Každá buňka DataGridu má režim pro zobrazení a úpravu. Režim pro úpravu je aktivní pouze v případě, kdy se buňka upravuje, jinak je aktivní režim pro zobrazení. Pomocí DataTemplate (představují šablonu, podle které se prvek vytvoří) lze pro každý sloupec definovat jaký prvek se má zobrazit v režimu úpravy a zobrazení. ForeignKeyTextblock měl za úkol nahradit ComboBoxu v režimu zobrazení. Nutnost vytvořit tento prvek byla způsobena špatnou výkonností ComboBoxu.

ComboBox je prvek, který v sobě ukrývá rolovací seznam libovolné kolekce. Problém nastával v případě, kdy v kolekce ComboBoxu obsahoval velké množství položek (kolem sta tisíce), po té bylo párování hodnot velmi pomalé a při rolování v DataGridu mělo velkou odezvu, což bylo uživatelsky velice nepřívětivé. A tak se vymyslel, že se vyzkouší vytvořit vlastní prvek, který bude párovat vlastnost proměnné z kolekce pomocí klíče. U tohoto prvku se následně může pracovat s algoritmem párování, což jako první přišlo v potaz vkládat seřazenou kolekci a pomocí binárního vyhledávání hledat požadovaný prvek, což se nakonec ukázalo jako zbytečné protože obyčejné lineární vyhledávání mělo dostatečnou výkonnost, aby práce s DataGridem byla uživatelsky přívětivá.

ComboBox obsahuje pro zobrazení hodnot vlastnosti DisplayMemberPath, která určuje název vlastnosti, kterou chceme zobrazit, SelectedValuePath představuje vlastnost pro porovnávání a SelectedValue je hodnota, typicky identifikátor, který určuje čemu se má hodnota ve vlastnosti SelectedValuePath rovnat. Podobný postup jsem využil u mnou vytvořeného prvku s jediným rozdílem, že u ComboBoxu jsou DisplayMemberPath a SelectedValuePath řetězce, podle kterých se následně vlastnost, pomocí reflexe, vyhledává, já jsem namísto to použil delegáta obsahujícího metodu, která vybírala potřebnou vlastnost. Výhoda mého přístupu byla vyhnutí se reflexi, která je obecně považována za pomalou, ale na druhou stranu odepřel možnost vytvořit prvek pomocí XAMLu, což v mém případě nevadilo, protože prvek byl určen pro vytvoření v kódu (C#).

ForeignKeyTextBlock byla generická třída, která dělila z třídy TextBlock a generický prvek musel implementovat rozhraní INotifyPropertyChanged a muselo se jednat od třídu generický datový typ jsem nazval TComboBoxItem. ForeignKeyTextBlock obsahoval následující veřejné vlastnosti. ItemsSource, který představoval vstupní kolekci. Kolekce byla IList a položky kolekce byly typu TComboBoxItem. IList je rozhraní, které rozšiřuje všechny kolekce v C#, což umožnilo vložit jakoukoli kolekci požadovaného datového typu. Další vlastnost byla ForeignKey datového typu `IComparable`. Podle této vlastnosti se párovaly objekty z kolekce. ComparedPropertySelector datového typu `Func<TComboBoxItem, long>` byl použit pro vybrání požadované vlastnosti. Další vlastností byl DisplayedPropertySelector datového typu `Func<TComboBoxItem, object>`. Tato funkce vybírala jaká vlastnost TComboBoxItem byla zobrazena toto bylo potřeba z důvodu, že pokud by se zobrazoval objekt z kolekce zobrazilo by se text, který by vrátila metoda `ToString()`, která by se musela přepsat což by nemusel být úplně ideální řešení. Poslední ve-

řejná vlastnost byla StringFormat datového typu string, pomocí kterého bylo možno zobrazovací text upravit (např. přidat jednotky, závorky). Všechny veřejné vlastnosti byly tzv. DependencyProperty, které je možné následně propojit s proměnnými a obsahují notifikace o změnách. ForeignKeyTextBlock také obsahoval množství privátních vlastností za zmínku stojí vlastnost SelectedItem datového typu ForeignKeyTextBlockSelectedItem.

Tuto třídu jsem vytvořil pro uchovávání právě zobrazeného objektu. Třída obsahovala vlastnost Value datovévého typu INotifyPropertyChanged, který měl pouze privátní setter a veřejnou metodu SetValue s parametrem typu INotifyPropertyChanged. Tato třída byla vytvořena pro předcházení memory leakům, ty by mohly nastat v případě kdyby se neodregistrovala událost PropertyChanged, která byla zaregistrována na prvku, aby se při změně hodnoty změna projevila do UI.

Vlastnosti ItemsSource a ForeignKey obsahovaly PropertyChangedCallback, které jsou možné použít v jednom z přetížených konstruktorů Dependency Property. PropertyChangedCallback je dvou parametrická metoda, kde první parametr je dependency objekt,ke kterému Dependency Property patří, druhý parametr je ChangedEventArgs, třída která obsahuje, novou hodnotu a starou hodnotu. Callback se zavolá při změně hodnoty. V ItemsSource callbacku jsem odregistroval událost CollectionChanged od staré kolekce, a zaregistroval událost CollectionChanged do kolekce nové. Událost CollectionChanged je zavolána když se obsah kolekce změní. Odregistrace byla nutná, jinak by opět docházelo k memory leakům. V Callbacku ForeignKey jsem volal metodu MakeItemActive, o které se zmíním dále v textu.

Třída ForeignKeyTextBlock obsahovala množství metod, které byly všechny privátní. Metody OnLoaded, která se vyvolala ve chvíli, kdy se prvek zobrazil, obsahovala volání funkce MakeItemActive. Metoda OnUnloaded sloužila k odregistrování událost CollectionChanged na vlastnosti ItemsSource a taky volání metody SetValue na právě zobrazeném prvku s argumentem null, což zajistilo odregistrování události PropertyChanged. Další metodou byla MakeIteActive s parametrem IComparable v této metodě se volala další metoda FindItem, která vracela objekt TComboBoxItem, který následně nastavila jako právě zobrazený prvek a dále volala metodu ChangeText s parametrem typu string. Při volání této metody jsem využil DisplayedPropertySelector, do kterého jsem vložil objekt, který mi vrátila metoda FindItem. Výsledkem volání DisplayedPropertySelector byla požadovaná vlastnost, která se měla zobrazit, nad touto vlastností jsem zavolal metodu ToString, aby bylo splněno, že parametr je datového typu string, který MetodaChangeText požadovala. Metoda FindItem obsahovala dva parametry typu `IComparable` a `IList<TComboBoxItem>`.V metodě jsem pomocí cyklu prošel všechny položky a voláním ComparedPropertySelector jsem si z každého prvku v kolekci vybral požadovanou vlastnost, pomocí které se porovnávalo. Metoda ChangeText s parametrem typu string nastavila vlastnost Text, která byla zděděna z TextBocku, na hodnotu parametru, v případě, že vlastnost StringFormat byla nastavena použil se požadovaný formát.

5 Závěr

5.1 Teoretické a praktické znalosti získané v průběhu studia a uplatněné v průběhu praxe

Při práci na obou projektech jsem využil znalostí jazyka C# získané v předmětu Programovací Jazyky II, z předmětu Algoritmy jsem využil znalost rekurze a při tvorbě UI jsem uplatnil znalosti z předmětu Uživatelská rozhraní.

5.2 Znalosti či dovednosti scházející studentovi v průběhu odborné praxe

Na problémy s neznalostí jsem se setkal při používání frameworku WPF, se kterým jsem se za dobu studia nesetkal. Další dovednost, která mi chyběla byla znalost verzovacích softwarů. Nicméně jsem tyto nedostatky v průběhu praxe redukoval.

5.3 Závěrečné shrnutí

Bakalářská práce formou individuální odborné praxe mi přišla jako skvělá zkušenost, a možnost odzkoušet mé znalosti nabité na středí a vysoké škole v praxi. Zjistil jsem, jak vypadá programování aplikací v týmu. Také jsem se dozvěděl, jak je důležitá komunikace při tvoření aplikací,a kolik času dokáže komunikace ušetřit. Také díky této možnosti jsem získal velké množství zkušeností v programování, které určitě využiji nejen ve škole, ale i budoucím profesním životě.

Literatura

[1] 1.díl – Úvod do WPF (Windows Presentation Foundation). itnetwork. [online]. 4.12.2013 [cit. 2017-04-07]. Dostupné z: www.itnetwork.cz/csharp/wpf/c-sharp-tutorial-wpfuvod-a-prvni-formularova-aplikace

[2] 2. díl – Jazyk XAML v C# .NET WPF. itnetwork. [online]. 8.12.2013 [cit. 2017-04-07]. Dostupné z: http://www.itnetwork.cz/csharp/wpf/c-sharp-tutorial-wpf-jazyk-xaml

[3] Ingeteam [online]. Ostrava, 2008 [cit. 2017-04-28]. Dostupné z: http://www.ingeteam.cz/

[4] Windows Presentation Foundation. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-46-26]. Dostupné z: https://cs.wikipedia.org/wiki/Windows_Presentation_Foundation

[5] C Sharp. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-04-26]. Dostupné z: https://cs.wikipedia.org/wiki/C_Sharp

[6] Git. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-04-26]. Dostupné z: https://cs.wikipedia.org/wiki/Git