Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra Informačního Inženýrství

Bakalářská práce

Migrace z VB6 do .NET Frameworku

René Souček

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

René Souček

Informatika

Název práce

Migrace z VB6 do .NET Frameworku

Název anglicky

Code migration from VB6 into .NET Framework

Cíle práce

Bakalářská práce je zaměřena na problematiku migrace z technologie Visual Basic 6 do .NET Frameworku. Hlavní cíl práce je vysvětlení jednotlivých programovacích postupů a výhody, které migrace přináší. Dílčím cílem práce je popsat rozdíly mezi jednotlivými technologiemi na konkrétních příkladech a dále na příkladové studii demonstrovat celý proces.

Metodika

Metodika řešení teoretické části práce spočívá ve studiu odborných informačních zdrojů. Na základě zjištěných poznatků budou popsány rozdíly mezi technologiemi. Praktická část práce bude spočívat v příkladové studii demonstrující přechod z VB6 do .NET Frameworku. Pro zpracování praktické čás budou využity vybrané standardní postupy a metody softwarového inženýrství. Výsledkem případové studie je analýza rozdílů mezi starým a novým řešením. Z jednotlivých výsledků je poté formulován závěr projektu a celé bakalářské práce.

Doporučený rozsah práce

35-40 stran

Klíčová slova

.NET framework, Simple injector, Entity framework, Visual Basic (VB6), Visual Studio

---

Doporučené zdroje informací

DRISCOL, B. Entity Framework 6 Recipes. ISBN-13 (electronic): 978-1-4302-5789-9. FREEMAN, A. Pro ASP.NET MVC 5. ISBN-13 (electronic): 978-1-4302-6530-6.
KORT, W D. Programming in C#. ISBN: 978-0-7356-7682-4.
TROELSEN, A. Pro C# 5.0 and the .NET 4.5 Framework. ISBN-13 (electronic): 978-1-4302-4234-5

Předběžný termín obhajoby

2016/17 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 21. 2. 2017

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 21. 2. 2017

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 06. 03. 2017

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Migrace z VB6 do .NET Frameworku" jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor(ka) uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 14. 3. 2017 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu bakalářské práce Ing. Jiřímu Brožkovi za rady a celkovou ochotu při zpracování práce, Michalu Čumpelíkovi za pomoc s teoretickým zpracováním práce, Lukáši Janu Markovi za poskytnutí veškerých podkladů a práv pro zpracování projektu a Tereze Filipové a Pavlu Barešovi za korekturu bakalářské práce.

Migrace z VB6 do .NET Frameworku

Souhrn

Cílem bakalářské práce je nastínit problematiku migrace ze zastaralé technologie Visual Basic do moderního .NET Frameworku. Tato problematika je zpracována na konkrétním příkladě migrace ze staré k nové technologii, na které se autor několik let podílel. Součástí projektu je popis použitých technologií, jejich výhody a principy fungování. Autor zde popisuje také srovnání doporučených postupů migrace s vlastním vypracovaným postupem.

Práce má za úkol poskytnout informace a postupy ze skutečného projektu, s jakým se člověk může běžně setkat v zaměstnání. Cvičné programovací projekty ve spoustě ohledech neodpovídají skutečným projektům, proto tato práce může sloužit i jako získání zkušeností, které se dají získat až v zaměstnání. V práci jsou také ukázány jednotlivé části nejdůležitějších projektů a jejich funkční popis pro zachování pravidel coding rules.

Práce obsahuje pouze dílčí výsledky, protože projekt migrace ještě není zcela dokončen a lze tudíž prezentovat pouze částečné výsledky.

Klíčová slova: Migrace, .NET Framework, Visual Basic, C#, VB.NET, Entity Framework, Visual Studio, Microsoft, Simple Injector, Unit testing, Rhino Mocks

Code Migration from VB6 to .NET Framework

Summary

The goal of the bachelor thesis is to outline the issue of migration from an outdated technology Visual Basic to modern .NET Framework. This issue has been analysed within a case study of migration from the old technology version to new one, which the author was participated in for a few years. The project includes a description of the technologies, their benefits and workflows. The author also describes the comparison of the best migration practises with the process used in project.

The thesis provides information and procedures from a real project, which can be commonly encountered in working practise. Common practise programming projects are usually in many aspects different from the real projects and therefore this work could contribute as a way to gain experience otherwise found in full-time job. The thesis also shows examples of the most important products of migration and their description connected with standard coding rules.

The bachelor thesis includes only partial results, because the migration project is not fully completed and therefore presented results could not be summarized.

Keywords: Migration, .NET Framework, Visual Basic, C#, VB.NET, Entity Framework, Visual Studio, Microsoft, Simple Injector, Unit testing, Rhino Mocks

Obsah

Seznam obrázků

1. **Visual Basic source code.** ***Visual Basic 6.*** **[Online] 2011. [Citace: 09. 03 2017.] http://www.visual-basic-6.com/visual-basic-source-code.php.**

2. **Visual Studio 2015.** ***Danijel Malik.*** **[Online] 2015. [Citace: 09. 03 2017.] http://danijelmalik.com/visual-studio-2015-release-date-announced/.**

3. **Entity Framework multiple diagrams.** ***MSDN Microsoft.*** **[Online] 2016. [Citace: 09. 03 2017.] https://msdn.microsoft.com/en-us/library/jj519700(v=vs.113).aspx.**

4. **Creating Unit tests for C# code.** ***Codeproject.*** **[Online] 2012. [Citace: 27. 02 2017.] https://www.codeproject.com/Articles/391465/Creating-Unit-tests-for-your-csharp-code.**

5. **Debugging.** ***Visual Studio.*** **[Online] 2017. [Citace: 08. 03 2017.] https://code.visualstudio.com/docs/editor/debugging.**

1 Úvod

Technologický vývoj je jeden z nejdůležitějších článků dvacátého prvního století. Rychlost, se kterou společnosti chrlí nové technologie je závratná, že si to člověk ani neuvědomuje. Když jde do obchodu a kupuje si nový počítač, nenapadne ho, že ve skutečnosti si kupuje zastaralou technologii, která bude zanedlouho střídána novou a novou. Jinak to není ani v IT prostředí. Vývojová prostředí a podpůrné programy jsou stále zdokonalovány a nabízí uživatelům stále nová zlepšení a zjednodušení.

Společnost Microsoft je pravděpodobně největší společnost v oblasti informačních technologií, zasahuje do celé řady odvětví včetně programovacích frameworků. V devadesátých letech patřil jazyk Visual Basic k velice oblíbeným, stejně jako vývojové prostředí Microsoft Visual Basic. Mnoho společností mají své služby, weby a další mechanismy psané právě touto technologií. Nicméně tento jazyk se přestal vyvíjet a po nějaké době přestal být podporován, tudíž se nabízí otázka, co s tímto produktem. Většina společností je nucena přejít k nové technologii vzhledem k nedostatečné nabídce kvalifikované pracovní síly. Přechod k nové technologii může být buď formou migrace staré k nové nebo vypracováním úplně nového systému. Právě o migraci pojednává tato bakalářská práce.

Bakalářská práce má za cíl předvést, jak probíhá migrace ze staré technologie Visual Basic do moderního .NET Frameworku. Popisuje veškeré technologie nutné pro tuto migraci, ukazuje průběh celého projektu. V práci autor popisuje nejdůležitější části a projekty důležité pro tuto migraci a jejich dílčí výsledky. Jedná se o několikaletý projekt, který v době psaní bakalářské práce je z veliké části dokončen a navazují na něj další projekty.

Samotná migrace je dost zavádějící pojem, nejedná se totiž o pouhý přepis kódu z jednoho jazyka do nového. Díky celé řadě změn technologií bylo nutné upravovat logiku, odstraňovat celé projekty a přidávat nové. Pouhý přepis do moderní technologie by vyřešil pouze otázku nabídky pracovní síly na trhu, ale nezlepšil by projekt z technologického a programátorského hlediska. V samotné práci proto pojem migrace bude znamenat i migraci technologickou.

Bakalářská práce je strukturována do několika bloků. V prvním teoretickém bloku jsou popsány technologie, doporučené postupy. Autor zde také rozebírá otázku, proč se do tohoto

projektu vůbec pouštět, jaké z toho plynou výhody a nevýhody. V dalším bloku autor popisuje jednotlivé části migrovaného projektu, ukazuje, na jaké bázi projekty fungují a v čem jsou největší rozdíly oproti původní technologii Visual Basic.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je popis migrace z Visual Basic 6 jazyka do .NET Frameworku, nicméně slovo migrace je v tomto případě značně zavádějící. Migrace sama o sobě je pouze přechod z jedné verze do nové. V tomto případě sice autor popisuje změnu ze staré technologie Visual Basicu 6 do moderních jazyků .NET Frameworku, z hlavní části C#, z menší části Visual Basic.NET. Nicméně tento přechod by byl jen z poloviny prospěšný, kdyby nedošlo také k dalším změnám.

V teoretické části jsou popsány nejdůležitější nové technologie, které byly k projektu potřeba. Ty nejdůležitější jsou samozřejmě .NET Framework, s ním spojené programovací jazyky C# a VB.NET. Vývojové prostředí je použito Microsoft Visual Studio Enterprise 2015. Při změnách v kódu byly využity nové technologie jako například Simple Injector, Service Locator. Pro spojení s databází je použit Entity Framework. Projekt byl testován celou řadou testů, mezi ty významné lze zmínit Unit testing s použitím Rhino Mocks.

V praktické části se pracuje s konkrétními částmi migrovaného kódu, jeho nejdůležitější součásti jsou podrobně popsány a vysvětleny. Celý projekt migrace se skládá z mnoha komponentů a tato část práce je především o ukázkách kódu napříč nejdůležitějšími projekty. Všechny ukázky, které jsou zde v praktické části uvedeny jsou použity z vlastních zdrojů.

Cílem práce není dokončit celý projekt migrace. Vzhledem k tomu, že v době psaní této práce projekt stále probíhá, nelze přesně popsat všechny výsledky projektu migrace. Celý projekt má řadu dílčích výsledků, které jsou popsány v kapitole 5.

2.2 Metodika

Metodika řešení teoretické části práce spočívá ve studiu odborných informačních zdrojů. Na základě zjištěných poznatků byly popsány rozdíly mezi technologiemi. Ke studiu byly využity jak tištěné odborné knihy, tak elektronické zdroje a odborné blogy. Některé zdroje byly použity ve více případech.

Praktická část práce spočívala v příkladové studii demonstrující přechod z VB6 do .NET Frameworku. Pro zpracování praktické části byly využity vybrané standardní postupy a metody softwarového inženýrství. Veškeré zdroje, které byly použity v praktické části pocházely z vlastních zdrojů. Použité postupy v kódu odpovídají stanoveným coding rules, které byly sestaveny přímo pro tento projekt. Z jednotlivých výsledků je poté formulován závěr celého projektu a celé bakalářské práce.

Samotnou práci a použití technologií autor konzultoval s hlavním team leaderem a ostatními členy projektu, který má celý projekt na starosti.

3 Teoretická východiska

V této kapitole se rozebírají veškerá teoretická východiska spojená s migrací. Řeší se zde základní otázka, proč vůbec se do tohoto projektu pouštět a její odpovědi. Ačkoliv tento projekt není jenom o migraci, je třeba znát důvody, proč si jako nový produkt zvolit právě .NET Framework, jaké jsou jeho výhody. To, že se nejedná jen o migraci kódu souvisí se všemi novými použitými technologiemi. V kapitole jsou podrobně popsány technologie Entity Framework, vývojové prostředí Visual Studio. Jsou zde uvedeny i další použité technologie Simple Injector, Service Locator, Unit testing nebo Rhino Mocks. Je veliké množství postupů jak tuto migraci provádět, proto je v této kapitole také srovnání doporučených postupů spolu s autorem aplikovaným postupem.

3.1 Proč vůbec migrovat?

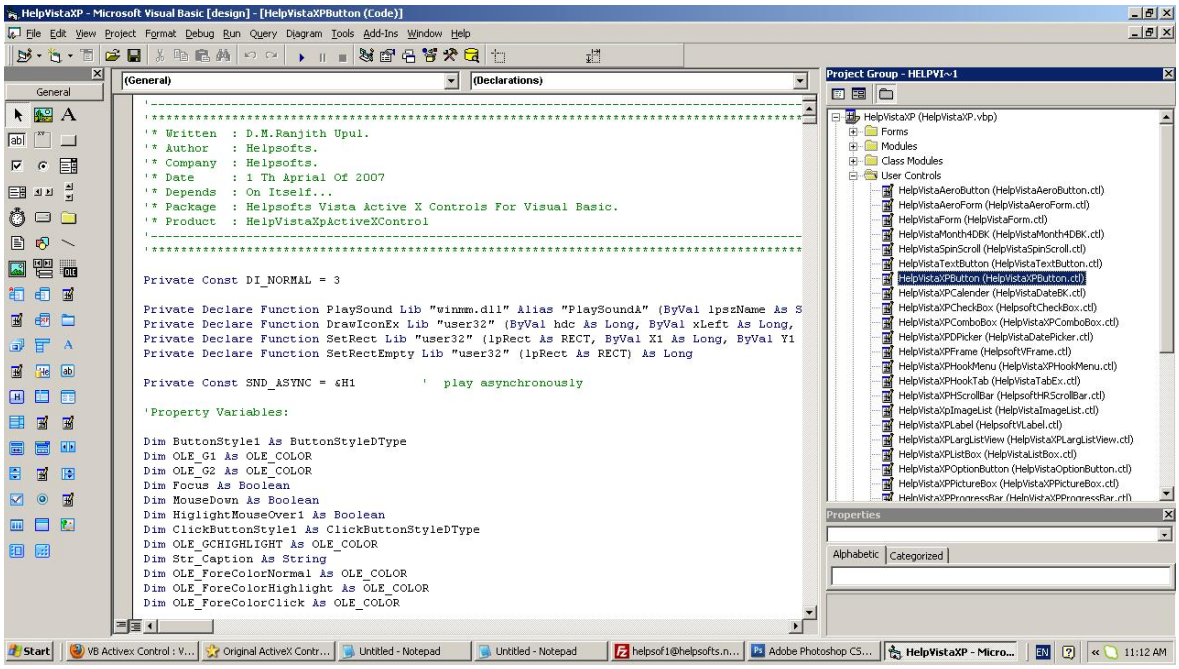
Hned na začátek se nabízí základní otázka. Proč vůbec něco takového dělat? Odpovědí lze najít více a každá z nich má jiný předpoklad. Jedna z nich má ryze praktický charakter. Zatímco před patnácti lety byl Visual Basic jedním z nejznámějších programovacích jazyků, dnes tomu tak ani zdaleka není. Poslední šestá verze jazyka byla vydána v roce 1998 a o deset let později firma Microsoft ukončila podporu tohoto jazyka. Dnešní podíl na trhu je v řádu tisícín procenta narozdíl od C# a VB.NET, které patří k nejrozšířenějším programovacím jazykům. Nabídka i poptávka po .NET technologii je nesrovnatelně větší, tudíž pro firmy mnohem lákavější.

Nejpoužívanější programovací jazyky pro rok 2016 (1):

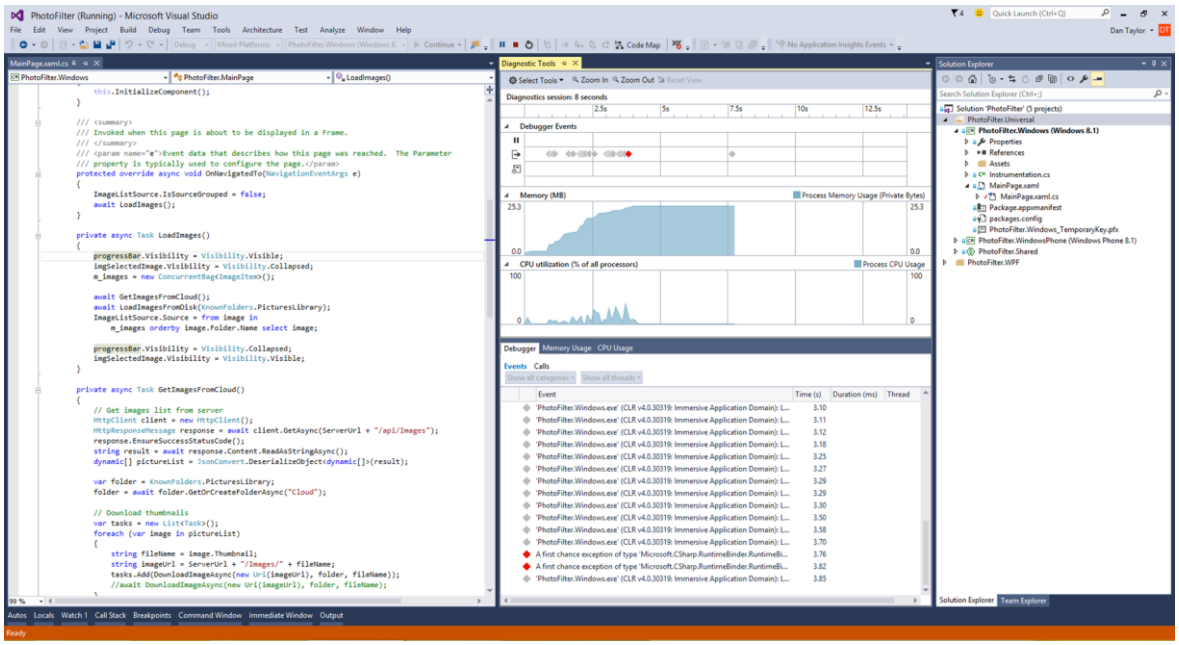
1) Python
2) Java
3) C++
4) C#
5) C
6) Javascript
7) Ruby
8) PHP
9) Haskell

10) Go

S tím také souvisí i další důvody jako například složitost práce a přehlednost. .NET Framework a jeho příslušné jazyky využívají nejnovější vývojová prostředí, programy či knihovny. Projekty jsou díky nim daleko přehlednější i čitelnější a tímpádem se s nimi i snáze pracuje.



Obrázek 3:1 Microsoft Visual Basic (1)



Obrázek 3:2 Microsoft Visual Studio 2015 (2)

3.1.1 Výhody .NET Frameworku

Ačkoliv hlavní část kódu je psána v jazyce C#, některé projekty jsou primárně psané v jazyce Visual Basic pro .NET (dále pouze VB.NET). Asi nejvýznamnější výhoda VB.NET oproti VB6 je podpora objektově orientovaného programování. Je to dáno tím, že .NET Framework je plně objektově orientovaný, všechny ostatní jazyky využívající tuto platformu musí být tudíž také objektově orientované (2).

Další výhoda, kterou je třeba zmínit je odpadnutí nutnosti registrovat všechny assembly pomocí regsvr32. .NET Framework v tomto případě funguje na bázi označení namespace. Díky tomu ve většině případů postačí kopírování příslušných .dll souborů na určené místo. K případnému širšímu nastavení se využívá příslušný XML config soubor, kde se dá vyhledávání assembly souborů značně zjednodušit.

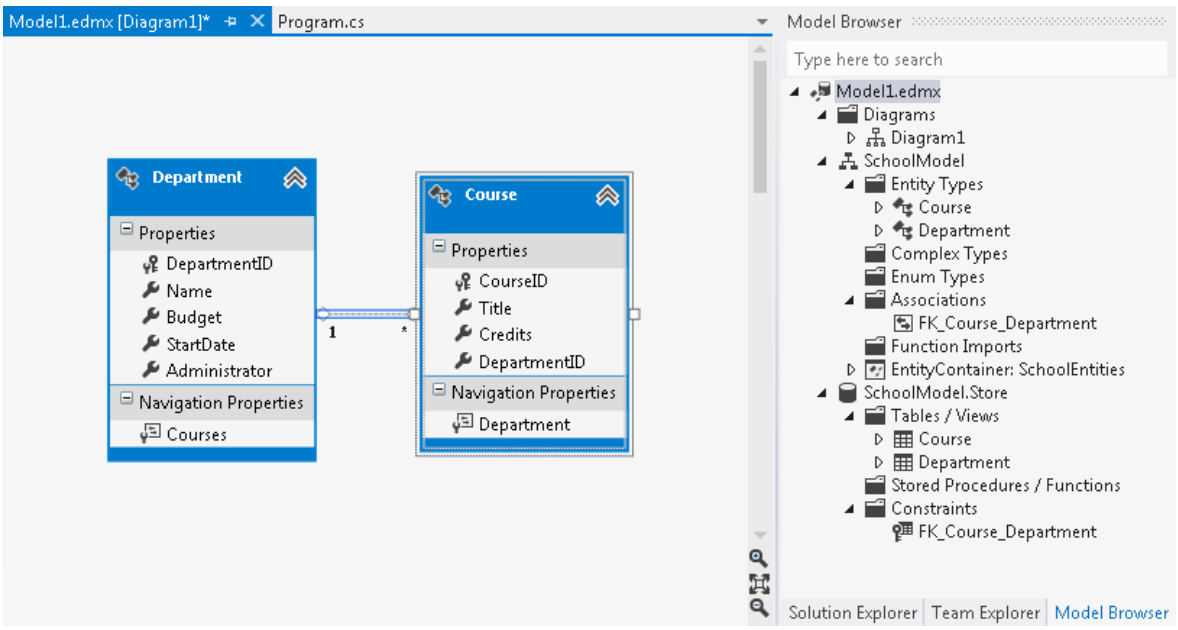
Důležitá součást programovacího jazyka je vývojové prostředí. Jedním z oblíbených vývojových prostředí pro práci s .NET je prostředí Visual Studio od Microsoft (3). Visual Studio umožňuje využívat .NET Framework a jeho příslušné jazyky v nejnovějších verzích včetně poslední verze VS2015.

Nespornou výhodou .NET oproti VB6 je dědičnost. To znamená možnost vytvářet si třídy, které slouží jako základ pro další zděděné třídy (5). Zděděné třídy dědí všechny metody a property základní třídy. Mohou dokonce novou implementací přepisovat metody ze základní třídy pro funkčnost pouze ve svojí třídě. Všechny třídy zde vytvořené jsou defaultně dědičné.

3.1.2 Entity Framework a jeho výhody

Entity Framework je objektově relační mapovací framework využivaný pro spojení mezi relační databází a objektově orientovaným jazykem, v tomto případě s C# a VB.NET (4). Je to velice oblíbený způsob komunikace a oproti např. SqlClientu nebo ADO.NET má několik podstatných výhod.

Díky tomu, že systém za vás automaticky vytváří objekty připsaní kódu proti Entity Frameworku, proces aktualizace databáze je daleko jednodušší. Pro databázové dotazy je možno využívat uložené procedury nebo psát dotazy přímo pomocí LINQ (6). Tyto dotazy jsou zpracovávány za běhu a překládány do syntaxe dané databáze. Entity Framework navíc není vázaný pouze na SQL Server, lze jej například propojit i s databází Oracle, MySQL a dalšími.



Obrázek 3:3 Entity Framework model edmx (3)

Další výhodou Entity Frameworku je jeho přehledný model databázového schématu. Tento model lze zobrazit jako třídu například pomocí Visual Studia. V modelu lze přizpůsobovat design podle vlastní představy (2). Lze také jednoduše aktualizovat, upravovat nebo mazat jednotlivé tabulky.

3.2 Použité technologie, jejich princip a využití

Migrace představuje kromě samotného přepisování značnou technologickou inovaci. Volání pomocí COM objektů je značně zastaralé a komplikované na údržbu. Vývoj technologií se posouvá dopředu obrovskou rychlostí, proto i zde bylo nutno využít několik zásadních nových technologií. Pro propojení projektů je klíčové využití Simple Injectoru a Service Locatoru. Pro přepis kódu bylo použito vývojové prostředí Visual Studio Enterprise 2015 od společnosti Microsoft. Jeden ze způsobů otestování bylo pomocí Unit testů. Zde autor hojně využíval technologie Rhino Mocks, která sloužila k otestování volání mezi projekty. V této kapitole jsou popsány hlavní technologie a jejich princip fungování.

3.2.1 Simple Injector

Simple Injector je jednoduchá .NET knihovna typu Dependency Injection, která funguje od čtvrté verze .NET Frameworku výše. Je propojitelný s frameworky jako například WebAPI, MVC, WCF a dalšími (7). Hlavní výhoda Simple Injectoru je jednoduchá propojenost mezi jednotlivými projekty. Pokud je nutné, aby jedna komponenta používala druhou bez toho, aby na ni musela mít referenci, je třeba použít ho k volání skrz jednotlivé projekty. Simple Injector je open source framework, což znamená, že je volně dostupný pro používání bez nutnosti platit za licenci. Nesporná výhoda Simple Injectoru je jeho rychlost. Programátor se díky tomu nemusí starat, zda procesy v programu nejsou zbytečně zpomalovány. Není proto třeba sledovat výkon knihovny.

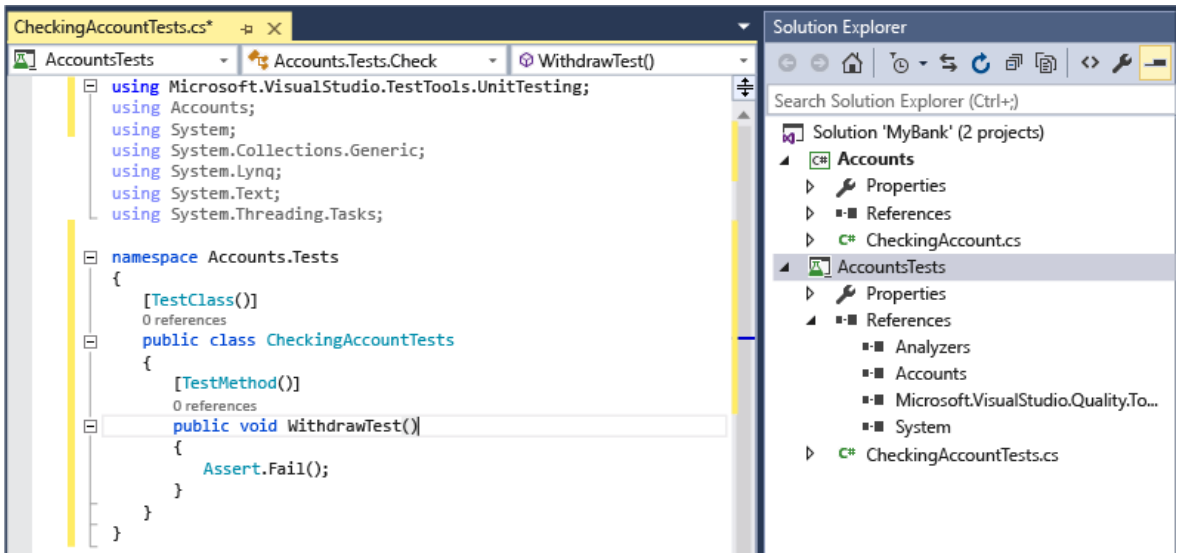
3.2.2 Service Locator

Service Locator je součástí Dependency Injection. Pokud si programátor vyžádá instanci nějaké sužby, úkolem Service Locatoru je dodat hotovou instanci. Pokud má programátor problém s rostoucím počtem parametrů u konstruktoru s rostoucími závislostmi, Service Locator může být řešení pro takové situace. Protože Service Locator dokáže obstarat potřebné služby, není nutné je injektovat zvlášť (8).

3.2.3 Unit testing a Rhino Mocks

Unit testing, jak už z názvu vypovídá, souvisí s testováním. Pokud programátor chce dát pozor na data, která prochází metodami v kódu, potřebuje otestovat, že nimi prochází jenom jím vybraná data, která si zvolí. A přesně k tomu slouží unit testy.

Názory na ně se liší, stejně tak jako míra jejich použití. Jeden programátor na každou metodu napíše 10 unit testů, druhý zase jen jeden. Jejich princip je založen na testování samostatných jednotek kódu, tedy jeden „unit“ (9).



Obrázek 3:4 Unit testing (4)

Testovací třída bývá vždy označena parametrem `[TestClass]`, každá testovací metoda zase parametrem `[TestMethod]`. Kromě tohoto parametru může být metoda označena dalším parametrem `[TestCategory("")]`. Do uvozovek vkládáme název kategorie testu. Jeho využití

je v lepší přehlednosti a rozdělení do kategorií. V ideálním případě by měl test pokrývat jednu podmínku metody. Uvažujme v metodě například podmínku:

```
if (reservationBe.Status == ResReservationStatusEnum.Confirmed)
```

Metoda unit testu by měla být zaměřena právě na tuto podmínku a testovat data v případě splnění či nesplnění podmínky.

V případě, že testovaná metoda volá ve svém kódu jiný projekt, může být značně problematické toto volání testovat pomocí unit testů. Rhino Mocks je technologie, která nám toto umožňuje (10). Její princip funguje tak, že pokud očekáváme volání externího kódu, vytvoříme v unit testu instanci dané třídy pomocí `MockRepository`. Pomocí instance voláme hledanou metodu a sami si určíme její návratovou hodnotu, kterou očekáváme.

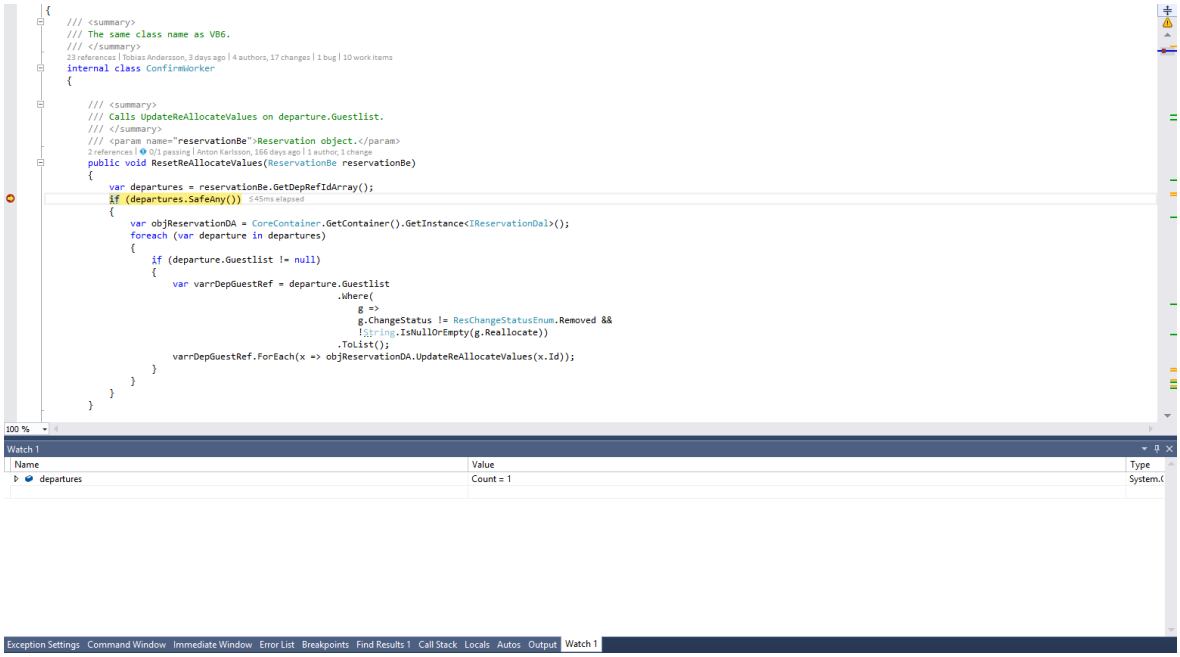
Výhoda Rhino Mocks je propojení s výše zmíněnou technologií Simple Injector. Díky ní se v praxi dá bez problému a s minimálním množstvím kódu v unit testech otestovat volání externích knihoven. Funguje na testovacím paternu AAA (Arrange, Act, Assert). S pomocí Rhino Mocks nelze mockovat třídy `static a sealed`. Rhino Mocks je open source framework.

3.2.4 Visual Studio 2015

Ačkoliv existuje celá řada vývojových prostředí, jedním z nejrozšířenějších programovacích softwarů pro vývoj je Visual Studio. Visual Studio 2015 je vývojové prostředí (IDE) od společnosti Microsoft. Objevuje se na trhu již 20 let a od první verze VS97 se značně vyvinulo. Již od své třetí verze plně podporuje .NET Framework a v současné době se připravuje vydání nové verze 2017 (11). Jeho hlavní funkcí je vývoj programů, webových stránek, web services, mobilních aplikací a dalších programů.

Visual studio podporuje širokou škálu programovacích jazyků. Mezi základní patří jazyky C a C++, dále pak jazyky spojené s .NET Frameworkem jako například C#, Visual Basic .NET nebo F#. Kromě těchto jazyků Visual Studio umí pracovat i s dalšími jazyky. Kupříkladu Ruby, Python, XML, HTML, Javascript a CSS patří mezi ty známější. Jeden z nejrozšířenějších jazyků Java v současné době již není podporován.

Jeden z oblíbených nástrojů Visual Studia je Debugger. Umožňuje nastavit si záchytné body (breakpoints) nebo sledovat proměnné (watch), díky kterým programátor dokáže krokovat program za běhu (12). Pro debugging slouží soubory typu .pdb, které se automaticky generují při kompilaci programu.



Obrázek 3:5 Visual Studio debugging (5)

Na obrázku je v hlavním okně znázorněn zachytávací breakpoint červenou barvou. Řádek označen žlutou barvou ukazuje aktuální polohu, ve které se proces právě nachází. Ve spodním okně je záložka watch. Do ní si lze uložit proměnnou, kterou chce autor sledovat. V záložce value se zobrazuje aktuální hodnota dané sledované proměnné. Při debuggingu lze procházet řádek po řádku a sledovat proces, lze vstupovat do funkcí a debuggovat je zevnitř. Součástí je také funkce Edit and continue, díky které lze upravovat kód za běhu. Jednotlivým proměnným lze nastavovat hodnotu. V oknech locals nebo watch jsou zobrazené aktuální proměnné, u kterých je možno modifikovat jejich hodnotu ve sloupci values.

3.3 Postup migrace

Vzhledem k tomu, že Visual Basic byl ve své době silně rozšířený, je jasné, že migrace k modernější technologii absolvovala řada menších i větších firem. Stejně jako neexistuje jeden programovací jazyk, který je nejlepší na světě po všech stránkách, postup migrace nabízí různé varianty a cesty. Jedna například volí cestu doslovné migrace, tedy přepis veškerého kódu do nové verze beze změny funkčnosti. Jiná varianta radí kompletní změnu celé struktury programů, která zahrnuje kromě nové technologie i novou logiku kódu, celé nové projekty a jejich propojení.

3.3.1 Doporučené postupy

Společnost Microsoft je úzce spojena s programovacím jazykem Visual Basic, proto se sama podílela na vývoji nástroje pro migraci staré technologie na novou. Mnoho společností potřebovalo nástroj, který by jim pomohl s touto migrací (13). Tyto oficiální nástroje pro migraci mají svoji nespornou výhodu. Byly mnohokrát testovány na rozličných produktech. Společnosti, které se pro tyto nástroje rozhodly, mají jistotu, že budou fungovat tak, jak mají. Další výhoda je možnost analýzy časového hlediska, který by byl potřeba pro upgrade aplikace do .NET Frameworku (14). Velká část práce na analýzách a časových odhadech díky tomu může být zautomatizována.

3.3.2 Náš postup práce

Původní cíl projektu zněl celkem jednoznačně: Odstranit všechny knihovny napsané v jazyce Visual Basic 6 a nahradit je knihovnami psanými v modernějším prostředí .NET v jazyce C#. To je samo o sobě velice jednoduše rečeno, avšak bylo třeba vyřešit spousty základních otázek. Jak bude knihovna propojena s databází? Jak bude knihovna volána externími webovými aplikacemi a klienty? Dosavadní způsob volání databáze pomocí Microsoft ActiveX byl značně zastaralý a náročný na aktualizace i údržbu.

```
Private Sub IReservationDA_DeletePcrGroup(ByVal ReservationCode As String,
ByVal PCRGroupCode As String, Optional cn As ADODB.Connection)
```

```
'$$METHOD_ENTRY_BEGIN
Const strMethodName = "IReservationDA_DeletePcrGroup"
On Error GoTo ErrorHandler
Dim arrParamNames() As Variant
Dim arrParamValues() As Variant
'$$METHOD_ENTRY_END

Dim cmd As Command

Set cmd = CreateObjectEx("ADODB.Command")

With cmd
      .CommandText = "dbo.Reservation_DeletePcrGroup"
      .CommandType = adCmdStoredProc

      .Parameters.Append .CreateParameter("ReservationCode", adVarChar,   adParamInput,
      CON_LEN_ReservationCode, ReservationCode)
      .Parameters.Append .CreateParameter("PcrGroupCode", adVarChar, adParamInput,
      CON_LEN_CODE_STD_LTH_10, PCRGroupCode)
End With

Call ExecuteActionCmd(cmd, cn)

'$$METHOD_EXIT_BEGIN
Exit Sub
ErrorHandler:
      arrParamNames = Array("ReservationCode", "PCRGroupCode", "cn")
      arrParamValues = Array(ReservationCode, PCRGroupCode, cn)

Call m_objLog.Trace(CurrentMachineName, APPLICATION_NAME, App.EXEName,   TypeName(Me),
strMethodName, Err.Number, FormatLogState(arrParamNames, arrParamValues, Err,
TypeName(Me), strMethodName), ltlError)

Call sdfErrBubble(Err, TypeName(Me), strMethodName)
'$$METHOD_EXIT_END
End Sub
```

V tomto konkrétním případě byla databáze volána pomocí metody `CreateObjectEx()`, která vytvářela objekt ActiveX a jako parametr dostávala název třídy, kterou autor chtěl vytvořit. Do atributů tohoto objektů se přidal název a typ příkazu a také všechny potřebné parametry databázové procedury. Databázový příkaz se provedl pomocí metody `ExecuteActionCmd()`, ve které se aktivovalo připojení k databázi, byl proveden příkaz a následně se připojení zrušilo. Nové řešení spojení knihovny s databází bylo zvoleno přes Entity Framework, relační mapovací framework, jehož princip je popsaný v kapitole 3.1.2.

Další důležité rozhodnutí se týkalo propojení nové knihovny s ostatními aplikacemi a webovými klienty. Dosavadní model byl založen na stejném způsobu vytváření COM objektů jako v případě databázového spojení.

```
Public Sub New(ByVal EncryptedReservationCode As String)
```

```
Dim resServer As ResSale.IReservation

Dim xmlString As String

Try

resServer = CreateObject("ResBook.ReservationPO")
xmlString = resServer.GetEncrypted(EncryptedReservationCode)

m_ReservationXML.XML = xmlString

' Check if reservation was not found
CheckForErrors()

ReadXML()

m_ClientLanguageCode = m_LanguageCode

' Check if there are any errors
If Me.Errors.Count > 0 Then
      Throw New SloopException(Me.Errors(0))
End If

Catch e As SloopException
      Throw e
Catch e As Exception
       Throw New SloopException(e)

End Try

End Sub
```

COM objekt se i zde vytvářel pomocí metody `CreateObject()` a parametr metody byla konkrétní třída, která se v dané metodě volala. Tento způsob projevoval značné nedostatky například při debuggingu. Pokud chtěl uživatel procházet kód za běhu programu, musel mít všechny projekty spuštěné, jinak nebylo možné danou knihovnu procházet za běhu.

4 Vlastní práce

V této kapitole se prezentuje samotné migrování a jeho zásady. Ačkoliv logika často zůstávala zachována, propojení projektů se změnilo, tudíž se muselo změnit i jejich vzájemné volání. V kapitole je ukázán na příkladu princip volání Sale.NET knihovny z externích aplikací. Jedním z hlavních znaků objektově orientovaného programování je zapouzdřenost a tomuto tématu se také věnuje praktická část. Opět s ukázkami je vysvětlen systém a jeho pravidla, podle kterých se citlivé části kódu zapouzdřují a naopak viditelné části kódu zůstávají ve vrchních vrstvách kódu. Kapitola také pojednává o testování, konkrétně Unit testech, jejich použití a jak se tvoří. V kapitole je také popsáno jak funguje jádro Core.NET, které se používá pro propojení projektů.

4.1 Sale.NET knihovna

Sale.NET knihovna je hlavní knihovna, která obsahuje veškeré volání do databáze. Původní knihovna byla psána v jazyce VB6 a byla první věc, která měla projít migrací. Kromě přepisu z jednoho jazyka do druhého bylo třeba vyřešit několik věcí. První věc bylo setřídění kódu do vrstev, přičemž logika měla zůstat vždy zapouzdřena. Druhá věc byla změna volání knihovny. Dosavadní způsob volání přes COM objekty byl značně zastaralý, proto bylo třeba změnit technologii. Současný způsob je řešen pomocí simple injectoru a jednoduchého containeru, pro který byl vytvořen další samostatný projekt. V externích aplikacích je použit pro volání Sale.NET knihovny.

4.1.1 Vrstvy kódu

Samotný kód je roztříděn do vrstev podle důležitosti. Nejméně přístupné části kódu jsou umístěny v Data Access Layer (zkráceně DAL) vrstvě, zatímco nejpřístupnější části kódu najdeme ve Facade. Jednoduchým demonstrativním příkladem je Reservation skupina tříd. Nejvyšší vrstva `ReservationFacade.cs` vyžaduje interface `IReservation` pro volání přes interface v `DependencyContainer` a také atribut `RegisterDependency` pro registraci do DC. Samotné metody obsahují pouze rozcestník do dalších úrovní kódu v závislosti na tom, jestli chceme volat starý VB6 kód pomocí COM objektu nebo jestli chceme volat .NET a také catch

pro zachytávání vyjímek ošetřený vlastní třídou `GlobalExceptionHandler.cs`. Metody jsou public z důvodu jejich volání napříč projekty.

```
namespace ResBook
{

    /// <summary>
    /// The main implementation of IReservation interface.
    /// </summary>
    [ExcludeFromCodeCoverage]
    [RegisterDependency]
    public class ReservationFacade : IReservation
    {
        #region Confirm

        /// <inheritdoc />
        public string Confirm(string reservationXml)
        {
            try
            {
                if (CodeUsages.UseNotTestedCode)
                {
                    var reservationInputXsd =
                    SerializationHelper.Deserialize<ReservationBe>(reservationXml);
                    var reservationInternal = new ReservationInternal();
                    reservationInternal.Confirm(reservationBe: ref
                    reservationInputXsd, originalRequest: reservationXml);

                    return SerializationHelper.Serialize(reservationInputXsd);
                }
                // ReSharper disable once RedundantIfElseBlock
                else
                {
                    var resPo = (IReservation)new ReservationPoCom();
                    var reservationInputXsd =
                    SerializationHelper.Deserialize<ReservationBe>(reservationXml);
                    var reservation = resPo.Confirm(reservationXml:
                    reservationInputXsd);

                    return SerializationHelper.Serialize(reservation);
                }
            }
            catch (Exception ex)
            {
                GlobalExceptionHandler.HandleException(ex, this, false);
                throw;
            }
        }
```

O úroveň nižší vrstva se nazývá `ReservationInternal.cs` obsahující pouze internal a private metody. Metody jsou volány pouze uvnitř třídy (`private`) nebo v rámci projektu (`internal`). V této části už je v kódu obsažena většina logické části kódu a volání DAL vrstvy. DAL vrstvu autor vždy volá pomocí třídy `CoreContainer.cs`. Stejným způsobem jsou volány všechny projekty z externích aplikací a projektů.

```
internal void UpdateGuestList(ReservationBe reservationBe)
     {
       var objReservationDA = CoreContainer.GetContainer()
       .GetInstance<IReservationDal>();
       var objModHelpers = CoreContainer.GetContainer().GetInstance<IModHelpers>();
       using (var resdMainConnection = CoreContainer.GetContainer()
       .GetInstance<IResDbConnection>())
            {
                using (var transaction = resdMainConnection.BeginTransaction())
                {

                    SaveReservationInternal.HandleResPassengerTable(reservationBe:
                    reservationBe);
                    SaveReservationInternal.HandleDepGuestlistTable(reservationBe:
                    reservationBe);
                    objReservationDA.CopyGuestlistToHistory(
                        reservationCode: reservationBe.Reservation.ReservationCode,
                        dbConnection: resdMainConnection);

                    transaction.SafeCommit();

                    objModHelpers.UpdateCabinNumbersAndCabinBerths(reservationBe:
                    reservationBe);
                }
            }

       // Make sure that any changes of loyalty member information are handled
       if (reservationBe.Status == ResReservationStatusEnum.Confirmed)
       {
              var modMain = CoreContainer.GetContainer().GetInstance<IModMain>();
              modMain.PostEvent("PPM", "DIRECT", ModPPMConstants.EV_MANUAL_EARN,
              string.Format("ReservationCode={0},EventType=EV_MANUAL_CALCULATION",
              reservationBe.Reservation.ReservationCode));
       }
}
```

Nejvíce zapouzdřenou částí projektu je vrstva DAL, která je připojena pomocí `Entity Framework` k databázi a ve které probíhají LINQ a procedurální dotazy. Samotné dotazy jsou vždy zabaleny pomocí funkce `using` pro automatické uvolňování paměti.

```
/// <inheritdoc />
[IntegrationTestParameters("10051815")]
public ReservationSpGetReservationDbBe GetTimeStampInfo(
       string reservationCode,
       string timestamp = "",
       IResDbConnection dbConnection = null)
{
       var timestampArr = timestamp.ConvertHexToByte();

       using (var db = RES_Entities.Create(dbConnection))
       {
           var reservation = (timestampArr != null) ?
             (from res in db.RESERVATIONS
              where (res.CODE == reservationCode && res.TIMESTAMP == timestampArr)
              select new ReservationSpGetReservationDbBe
               {
                   TIMESTAMP = res.TIMESTAMP,
                   STATUS_CODE = res.STATUS_CODE,
                   VERSION = res.VERSION
               }).ToList() :
             (from res in db.RESERVATIONS
              where (res.CODE == reservationCode)
              select new ReservationSpGetReservationDbBe
              {
                    TIMESTAMP = res.TIMESTAMP,
                    STATUS_CODE = res.STATUS_CODE,
                    VERSION = res.VERSION
               }).ToList();

           return reservation.Any() ? reservation.Single() : null;
       }
}
```

4.1.2 Volání napříč projekty

V předchozí kapitole byly ukázány způsoby vrstvení jednotlivých částí kódu podle stupně zapouzdřenosti. Nyní nastává posun o úroveň výše. Aby bylo možné zavolat knihovnu z externího projektu, je třeba vytvořit instanci dané třídy pomocí `CoreContainer`, což je jednoduchý dependency container, který využívá technologii Simple Injector. Pomocí vytvořené instance uložené do proměnné autor volá už danou metodu standartním způsobem, přičemž obvyklá návratová hodnota je serializována do XML. Tento konkrétní případ je bez návratové hodnoty. Metoda pouze upravuje jistou hodnotu už dříve uloženou v databázi.

V ukázce kódu jsou zásadní 2 věci:

1) Try/Catch/Finally příkaz
2) If podmínka pro použití originální implementace

```
Public Sub UpdateCancellationFee(ByVal ParametersXML As
UpdateCancellationFeeParameters) Implements IReservation.UpdateCancellationFee

Dim resServer As ResSale.IReservation = Nothing
'Dim EmptyDoubleArray() As Double
'Dim EmptyLongArray() As String

Try
       If (CodeUsages
       .UseOriginalImplementation("IReservation.UpdateCancellationFee")) Then
              resServer = CreateObject("ResBook.ReservationPO")
              resServer.UpdateCancellationFee(ParametersXML.GetXML)
       Else
              Dim reservationPo As Stenaline.Core.Sloop.Sales.ResBook.IReservation =
              CoreContainer.GetContainer().GetInstance(Of
              Stenaline.Core.Sloop.Sales.ResBook.IReservation)()
              reservationPo.UpdateCancellationFee(ParametersXML.GetXML)
       End If

       RaiseEvent AfterPriceCalculation()
 Catch e As SloopException
       Throw e
 Catch e As Exception
       Throw New SloopException(e)
 Finally
       ReleaseCOMRef(resServer)
 End Try
 End Sub
```

Každé volání externí knihovny je standartně pojištěno uložením do příkazu Try/Catch/Finally, kde se zachytávají veškeré errory, které se objevily v průběhu volání. Dále je zde podmínka, která zjisťuje, zda chceme použít původní originální implementaci knihovny nebo novou .NET verzi. Tento přepínač se nastavuje ve web.config souboru, který je součástí každého externího projektu. Celá metoda je vždy ukončena uvolněním COM reference příkazem `ReleaseCOMRef(resServer)` v poslední části `Finally`.

4.2 Core.NET

Už podle názvu je jasné, že Core.NET je jádro celé soustavy všech projektů. Ještě donedávna bylo volání projektů mezi sebou značně nesjednocené, část knihoven se volala pomocí COM objektů, jiné části zase byly kopírovány jako součást celé řady projektů. Core.NET je projekt, který má za úkol toto volání sjednotit. Jeho princip je založen na Service Locator a využití Simple Injector. Core.NET obsahuje třídu `CoreContainer`, která je využívána ve všech externích projektech pro zavolání Sale.NET knihovny.

V třídě `CoreContainer` se využívá metoda `GetContainer()`. Tato metoda vrací dependency container ve standartním nastavení, jak je z popisku metody zřejmé.

```
/// <summary>
/// Gets the core dependency container with standard production settings.
/// </summary>
/// <returns>Core dependency container with standard production
settings.</returns>
public static IDependencyContainer GetContainer()
{
    // first we need to catch any exceptions with assembly resolving
    // Internal class then can use SimpleInjector from renamed file.
    Hook.HookAssemblyResolve();
    return InternalContainer.GetContainer();
}
```

Samotný container je umístěn interně ve třídě `InternalContainer`. Ještě před jeho zavoláním proběhne v metodě `Hook.HookAssemblyResolve()` kontrola, že všechny assembly jsou správně načtené. Třída `Hook` pomáhá v případě, že je problém s nalezením požadovaného assembly.

```
public static IDependencyContainer GetContainer()
{
     Initilize(ContainerInitializationType.None);
     return new InternalContainer();
}
```

V internal třídě containeru probíhá nejprve Inicializace a poté se vrací nový container. Inicializace containeru potřebuje cestu ke konfiguračnímu souboru, kterou získává v metodě `GetFullConfigurationPath()`.

```
private static string GetFullConfigurationPath()
{
     var currentPath = InternalCoreTools.GetExecutingAssemblyDirectory();
     var ret = Path.Combine(currentPath, _configurationFileName);
     return ret;
}
```

Pokud je cesta známá a platná, probíhá samotná registrace ve třídě `RegistrationWorker.RegisterAllClasses(container: _container)`.

```
public static void RegisterAllClasses(Container container)
{
     if (CodeUsages.UseRemoteCall)
     {
```

```
            container.InterceptWith<GeneralInterceptor>(type => type !=
            typeof(GeneralInterceptor) &&
            !type.GetCustomAttributes(false).OfType<NoRemoteCallAttribute>().Any()
                  );
    }

    foreach (var item in Configuraton.Definitions)
    {
        var method = typeof(Container).GetMethods()
                  .Single(m => m.ToString() == "Void
                  Register[TService,TImplementation]()");
        var genericMethod =
        method.MakeGenericMethod(Type.GetType(item.TypeInterface),
        Type.GetType(item.TypeImplementation));
        genericMethod.Invoke(obj: container, parameters: null);
    }
}
```

Metoda za pomoci Simple injectoru vezme každou třídu a zaregistruje ji do containeru pomocí příkazu `genericMethod.Invoke()`. Celá registrace do containeru probíhá při spuštění každého projektu, který je připojený na jádrový projekt Core.NET.

Pokud dochází k volání knihovny Sale.NET, výše v textu bylo popsáno pravidlo, jakým způsobem se volání zapisuje. K vytvoření instance se používá `IDependencyContainer`, který obsahuje metodu `TService GetInstance<TService>()`.

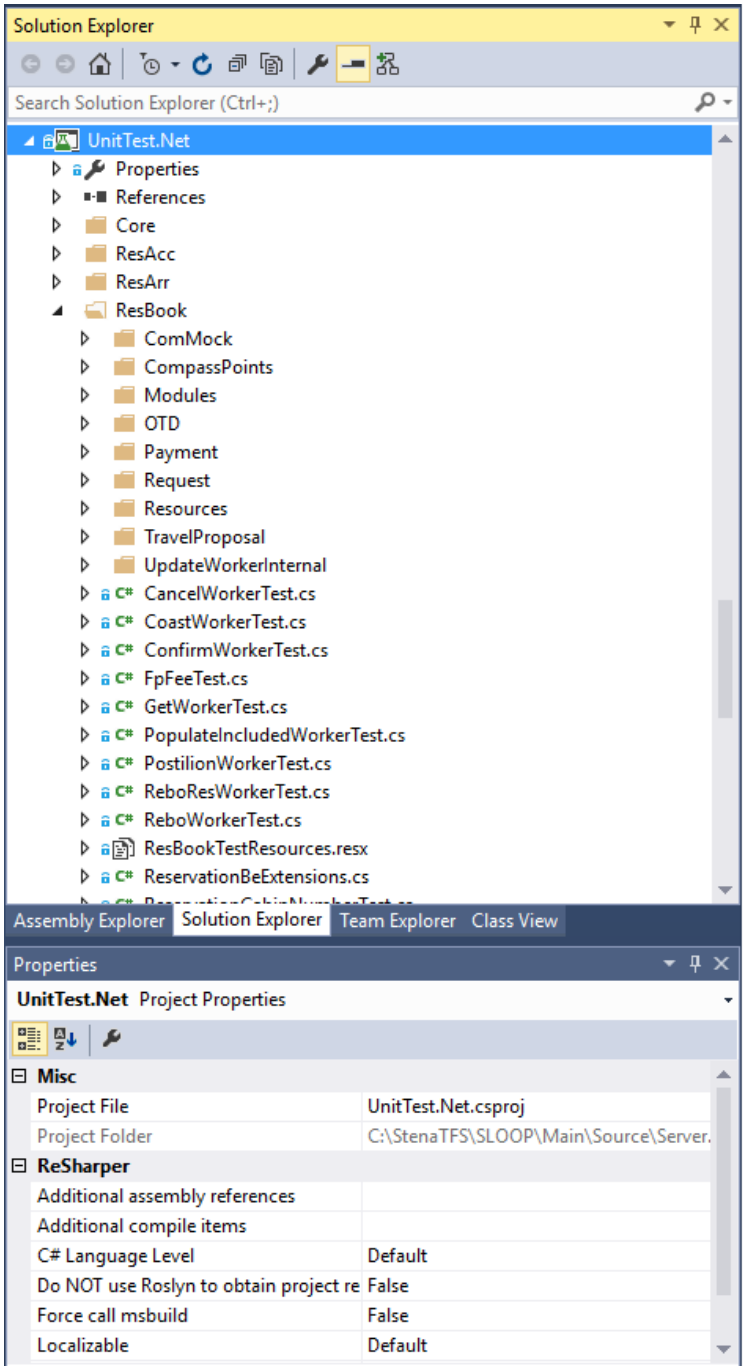
```
/// <summary>
/// Gets an instance of the given TService.
/// </summary>
/// <typeparam name="TService">Type of object requested.</typeparam>
/// <returns>The requested service instance.</returns>
public TService GetInstance<TService>() where TService : class
{
    return _container.GetInstance<TService>();
}
```

Tato metoda vrací instanci daného interface. Metoda je generická, tudíž se její návratová hodnota může lišit. Stanovuje se za běhu programu, podle potřeby daného procesu.

4.3 Testování

Jedna z důležitých součástí celého projektu je jeho testování. Ačkoliv Unit testing často není brán jako nejdůležitější část, v tomto projektu má svoje místo i význam. Jedna z podmínek při

vzniku projektu bylo, aby alespoň 75% kódu bylo pokryto testy. Proto byl v knihovně Sale.NET vytvořen samostatný projekt pouze pro Unit testy.



Obrázek 4:1 Unit tests system (vlastní zdroje)

Jeho struktura je rozdělena do samostatných složek, přičemž každá složka odpovídá jedné z částí knihovny Sale.NET. V složkách se nachází třídy s Mock objekty pro volání napříč

projekty a samotné třídy s testovacími metodami. Součástí Unit test projektu je také třída AssemblyInfo.cs, ve které se nachází základní nastavení projektu pro testování.

```
using System.Reflection;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("UnitTest.Net")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Microsoft")]
[assembly: AssemblyProduct("UnitTest.Net")]
[assembly: AssemblyCopyright("Copyright © Microsoft 2014")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("7893f46d-e181-43de-a3b0-79022cb8d699")]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

V této třídě lze nastavit základní nastavení pro testování pomocí atributů. Kromě základních titulů a jmen se dá v této třídě nastavovat například volání COM objektů pomocí atributu `[assembly: ComVisible(false)]` přepínáním jeho hodnoty na true/false. Lze také nastavovat viditelnost a přístupnost `internal` metod. Stačí v třídě AssemblyInfo.cs od daného projektu vložit atribut `[assembly: InternalsVisibleTo("UnitTest")]`, přičemže do parametru autor vkládá jméno testového projektu.

Samotné testové třídy jsou řešeny poměrně jednoduchou formou, ačkoliv na první pohled trochu zdlouhavou pro psaní. Metoda `TestSetUp()` je určena k inicializaci testové třídy.

```
#region initialize

/// <summary>
/// Set up one test.
/// </summary>
[TestInitialize]
public void TestSetUp()
{

       ContainerCore = CoreContainer.CreateContainerForTests();
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IComHelper>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<ICompassPoints>());
       ContainerCore.Register<IDataAuthorization, DataAuthorisationMock>();
       ContainerCore.Register<IFuncAuthorization,
       PPMFuncAuthFuncAuthorizationPOMock>();
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IModHelpers>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IModMain>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IReservationDal>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IModPPMSaleHelpers>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IPaymentInternal>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IPerformanceTrace>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IProduct>());
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<IRegistryHelper>());
       //for mock CancelWorker>()
       ContainerCore.RegisterSingle(MockRepository.GenerateMock<ICapacityDal>());
}

#endregion // initialize
```

V této metodě probíhá registrace Všech interface pomocí Rhino Mocks třídy `MockRepository`. Bez této registrace nelze při testování volat externí projekty pomocí `ContainerCore`. Inicializace probíhá na začátku spuštění testů a platí pro všechny testy z testovací třídy. Stanovujeme ji na začátku metody atributem `[TestInitialize]`.

Testové metody jsou vždy na začátku označeny atributem `[TestMethod]`. To zaručuje, že se metoda zobrazí v Test Explorer toolboxu, ze kterého se dají testy spouštět.

```
[TestCategory("DepartureWorker")]
[TestMethod]
public void GetResultOk()
{
         CoreContainer.CreateContainerForTests();
         var capacityDalMock = MockRepository.GenerateMock<ICapacityDal>();
         CoreContainer.GetContainer().RegisterSingle(capacityDalMock);
         var generalDal = MockRepository.GenerateMock<IGeneralDal>();
         CoreContainer.GetContainer().RegisterSingle(generalDal);


         capacityDalMock.Stub(i => i.GetPrdCifGroupCodesForDepartureCapacity(
              Arg<string>.Is.Anything,
              Arg<IResDbConnection>.Is.Anything))
                 .Return(new List<CapacityGetPrdCifDbBe>()
                .Repeat.Times(1,1));
```

```
            generalDal.Stub(x => x.GetSalesOwnerInfo(
                 Arg<string>.Is.Anything,
                 Arg<IResDbConnection>.Is.Anything))
                    .Return(new[] { new GetSalesOwnerInfoDbBe() }.ToList());

            var worker = new DepartureWorker();

            const string depId1 = "DepID1";
            const string depId2 = "DepID2";

            var departures = new List<Departure>
            {
                new Departure
                {
                    DepartureID = depId1
                },
                new Departure
                {
                    DepartureID = depId2
                }
            };

            var result = worker.GetResult(departures);
            Assert.AreEqual(4, result.SafeCount());
            Assert.AreEqual(depId1, result[0].DEPARTURE_ID);

}
```

Každá metoda by standartně měla být určena pro otestování jedné části kódu obvykle vyznačené nějakou podmínkou. V metodě se vytvoří instance testované třídy, ze které se metoda volá. Pokud má metoda povinné parametry, vytváří autor také objekt s příslušnými testovacími hodnotami, který vkládá jako parametr metody. V této konkrétní metodě se testuje metoda `GetResult()` z třídy `DepartureWorker`. Uvnitř metody se volají externí třídy `ICapacityDal` a `IGeneralDal`, které se mockují příkazem `MockRepository.GenerateMock<>()` a registrují se pomocí příkazu `CoreContainer.GetContainer().RegisterSingle()`. U daného mocku se pomocí metody `Stub()` očekává, že testovaná metoda bude externě přes interface volat metodu `GetPrdCifGroupCodesForDepartureCapacity()` a jako její parametry bude očekávat jakékoliv hodnoty autorem zvoleného typu díky `Arg<string>.Is.Anything`. Autor sám si určuje návratovou hodnotu metody a počet opakování (kupříkladu při při použití `while` nebo `for each`) řádkem `Return(new List<CapacityGetPrdCifDbBe>() .Repeat.Times(1,1))`.

5 Výsledky a diskuse

5.1 Sale.NET knihovna

V kapitole o knihovně Sale.NET bylo podrobně vysvětleno, jak knihovna funguje a na jaké bázi je postavena. Jednotlivé podkapitoly se podrobně zaměřily na nejdůležitější části knihovny. Všechny ukázky kódu v této kapitole byly použity z vlastních zdrojů. Oproti předchozí verzi je kód mnohem čitelnější. Jeho údržba je nyní značně jednodušší, v případě bugů se nyní dají mnohem jednodušeji chyby dohledat a odstranit.

5.1.1 Vrstvy kódu

V podkapitole Vrstvy kódu autor vysvětloval systém zapouzdření kódu a jeho pravidla. Každá vrstva byla popsána na konkrétní ukázce kódu. Bylo popsáno, proč se které třídy označují `public` a které naopak jsou `internal`. Bylo zde demonstrováno využití LINQ pro dotazy z databáze.

5.1.2 Volání napříč projekty

V druhé podkapitole Volání napříč projekty bylo popsáno propojení Sale.NET knihovny s externími aplikacemi. Všechny projekty využívají tuto knihovnu a na názorném přikladu autor popsal, jak se tato knihovna v projektech volá. Je zde ukázka používání jazyka VB.NET.

5.2 Core.NET

V kapitole Core.NET bylo vysvětleno, že se jedná o jádro používané mezi všemi projekty. Byl podrobně popsán celý proces volání Sale.NET knihovny pomocí jádra Core.NET. Spolu s ukázkami kódu byly vysvětleny jednotlivé metody a jejich význam a pořadí v procesech. V kapitole byla také popsána funkce Simple injector containeru.

5.3 Testování

Kapitola Testování pojednávala systému testů a jejich skladbě. Sale.NET využívá pro testy Unit testing spolu s technologií Rhino Mocks. Tyto technologie byly podrobně popsány a na ukázkách kódu byl podrobně popsán princip a pravidla při psaní Unit testů.

6 Závěr

Bakalářská práce si kladla za cíl popis projektu migrace z Visual Basic do .NET Framework. Tento projekt v době psaní bakalářské práce byl stále v průběhu, tudíž nebylo možné popsat do detailu všechny výsledky projektu. Nicméně hlavním cílem práce byl podrobný popis celého projektu. Byly zde popsány všechny důležité technologie, které byly v průběhu projektu migrace využívány, byly zde vysvětleny důvody samotné migrace a jak si v takovém případě počínat. V praktické části autor ukázal nejdůležitější komponenty, vysvětlil také na praktických příkladech, jak tyto komponenty fungují a jak se tvoří.

Dalším cílem bakalářské práce bylo prezentovat reálný produkt na trhu. Projekt není jenom práce jednoho člověka za účelem získání titulu ve škole, jedná se o několikaletý proces, na kterém se podílelo několik desítek programátorů a manažerů, projekt, který má své plnohodnotné místo na IT trhu, na který bylo vynaloženo veliké množství práce i peněz. Skutečný projekt často neodpovídá představě programátora ani klienta, který o něj zažádal. Nicméně ukazuje, jak funguje v závislosti na obchodní i marketingové stránce, dále ukazuje, že takový projekt není jen o naprogramování několika řádek kódu, ale i o celé řadě jednání, dohad, kompromisů, upřednostňování významnějších úkolů před méně významnými.

V době, kdy tento projekt vznikal, autor začínal s programováním a učil se na něm základy objektově orientovaného programování a práci s .NET Frameworkem. Na všech výše zmíněných projektech a částech kódu se autor podílel, ať už ve větší či menší míře. Projekt migrace se nyní úspěšně chýlí ke konci a všechny knihovny, které byly doteď psané v jazyce VB6 nyní fungují na bázi .NET Frameworku.

Bakalářská práce je inspirací pro začínající i pokročilejší programátory v .NET Frameworku díky zajímavému řešení situací, které se během projektu vyskytovaly. Projekt může inspirovat jako návod pro použití technologií, postupů migrace, technik nebo implementací částí kódu.

7 Seznam použitých zdrojů

1. **10 nejpoužívanějších jazyků 2016.** ***Codeeval.*** **[Online] 2016. [Citace: 02. 02 2016.] http://blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016.**

2. **Brian, Driscoll.** ***Entity Framework 6 Recipes.*** **2013. 978-1-4302-5789-9.**

3. **.NET Framework advantages.** ***Itelern.*** **[Online] 2016. [Citace: 12. 12 2016.] http://itelearn.com/blog/2016/07/20/dot-net-framework-advantages-and-disadvantages/.**

4. **Introduction to Entity Framework.** ***MSDN Microsoft.*** **[Online] 2016. [Citace: 02. 03 2017.] https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx.**

5. **Benefits of .NET Framework.** ***C-sharpcorner.*** **[Online] 2009. [Citace: 12. 12 2016.] http://www.c-sharpcorner.com/uploadfile/puranindia/benefits-of-the-net-framework/.**

6. **Model Entity Framework.** ***Entity Framework Tutorial.*** **[Online] 2016. [Citace: 01. 03 2017.] http://www.entityframeworktutorial.net/model-first-with-entity-framework.aspx.**

7. **Simple Injector Basics.** ***Simple Injector.*** **[Online] 2014. [Citace: 06. 03 2017.] https://simpleinjector.org/index.html.**

8. **Service locator.** ***Rarous Webblog.*** **[Online] 2009. [Citace: 26. 02 2017.] https://www.rarous.net/weblog/357-service-locator.aspx.**

9. **Creating Unit tests for C# code.** ***Codeproject.*** **[Online] 2012. [Citace: 27. 02 2017.] https://www.codeproject.com/Articles/391465/Creating-Unit-tests-for-your-csharp-code.**

10. **Rhino Mocks Tutorial.** ***Dotnet Pattern.*** **[Online] 2016. [Citace: 08. 03 2017.] http://dotnetpattern.com/Rhino-Mocks-Tutorial.**

11. **Introducing Visual Studio.** ***MSDN Microsoft.*** **[Online] 2008. [Citace: 08. 03 2017.] https://msdn.microsoft.com/en-us/library/fx6bk1f4(v=vs.90).aspx.**

12. **Debugging.** ***Visual Studio.*** **[Online] 2017. [Citace: 08. 03 2017.] https://code.visualstudio.com/docs/editor/debugging.**

13. **Migrating Visual Basic.** ***MSDN Microsoft.*** **[Online] 2005. [Citace: 08. 03 2017.] https://msdn.microsoft.com/en-us/library/ff648898.aspx?f=255&MSPPError=-2147217396.**

14. **Visual Basic 6 migration tool.** ***MSDN Microsoft.*** **[Online] 2005. [Citace: 10. 03 2017.] https://msdn.microsoft.com/en-us/library/ff648898.aspx.**