VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

REFAKTORIZACE SÍŤOVÉHO FORENZNÍHO NÁSTROJE NETFOX DETECTIVE

NETFOX DETECTIVE TOOL REFACTORIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **HANA SLÁMOVÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20380/2017/xslamo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Slámová Hana**
Obor: Informační technologie
Téma: **Refaktorizace síťového forenzního nástroje Netfox Detective**
Netfox Detective Tool Refactorization
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s nástrojem Netfox Detective a proveďte jeho analýzu zaměřenou na správné použití návrhových vzorů.
2. Vytvořte sady Unit Testů ověřující aktuální chování nástroje.
3. Identifikujte nevhodně implementované třídy a proveďte refaktorizaci.
4. Srovnejte zjištění z analýzy původního stavu s refaktorovaným.
5. Ověřte a diskutujte, zdali vývoj nástroje probíhá v souladu s doporučenými praktikami a nástroj je při vývoji automaticky testován, případně toto zajistěte.

Literatura:

- Strauss, D. (2017). *C# 7 and .NET Core Cookbook*. Birmingham: Packt Publishing.
- Price, M. (2017). *C# 7 and .NET Core modern cross-platform development : create powerful cross-platform applications using C# 7, .NET Core, and Visual Studio 2017 or Visual Studio Code*. Birmingham, UK: Packt Publishing.
- MATOUŠEK, P., PLUSKAL, J., RYŠAVÝ, O., VESELÝ, V., KMEŤ, M., KARPÍŠEK, F. and VYMLÁTIL, M. (2015). *Advanced Techniques for Reconstruction of Incomplete Network Data*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering., vol. 2015, no. 157, pp. 69-84. ISSN 1867-8211.

Pro udělení zápočtu za první semestr je požadováno:

- body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pluskal Jan, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práce pojednává o návrhových vzorech použitých v síťovém nástroji Netfox Detective. V práci je obsažena jejich analýza a identifikace problémů v jejich implementaci. Dále je popsána implementace unit testů a procesu jejich automatizace. Tato práce také obsahuje popis kontribuce nového kódu do projektu Netfox Detective.

Abstract

This semestral thesis deals with usage of design patterns in a network tool Netfox Detective. The thesis contains analysis and identification of issues regarding their implementation. Also, a description of writing unit tests for a part of application is present and its automation. This thesis describes Netfox Detective contribtion process as well.

Klíčová slova

návrhový vzor, refaktorizace, Netfox Detective, unit testy, strategie větvení

Keywords

design pattern, refoctoring, Netfox detective, unit tests, branching strategies

Citace

SLÁMOVÁ, Hana. *Refaktorizace síťového forenzního nástroje Netfox Detective*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Refaktorizace síťového forenzního nástroje Netfox Detective

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením Ing. Jana Pluskala a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány a uvedeny v seznamu literatury na konci práce.

.....
Hana Slámová
17. května 2018

Poděkování

Na tomto místě bych chtěla poděkovat vedoucímu mé bakalářské práce Ing. Janovi Pluskalovi za odbornou pomoc, cenné připomínky a poskytnuté rady, které mi napomohly k vytvoření této bakalářské práce.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá refaktorizací části aplikace Netfox Detective. Ačkoliv k restruktualizaci kódu dochází od počátků vývoje softwaru, teprve William Opdyke použil v roce 1990 ve svém článku slovo refactoring v tomto kontextu. Dále v roce 1991 William Griswold popsal tento proces pro funkcionální a procedurální jazyky ve své disertační práci, čímž vznikla první akademická práce na toto téma. O rok později jej následoval William Opdyke se svou dizertací zaměřenou na refaktorovaní programů napsaných v objektově orientovaných jazycích [3].

Refaktorování programu je dnes nezbytnou součástí vývojového cyklu. Při refaktorování provádíme takové změny, které pouze mění strukturu kódu, nikoliv jeho funkcionalitu. Tato činnost nám přináší vyšší kvalitu kódu, zlepšuje architekturu, a tím pádem umožňuje rychlejší budoucí vývoj. Kód je udržovatelný, rozšiřitelný, testovatelný, čitelnější. Metody a postupy využívané při refaktorizaci, popsal Martin Fowler v knize *Refactoring: Improving the Design of Existing Code* [36], která je dodnes používáná jako referenční příručka k aplikaci refaktorování.

K refaktorování patří i návrhové vzory. Při vývoji systému identifikujeme problém, který by nám elegantně vyřešil návrhový vzor. Refaktorizace pak představuje cestu, jak k aplikaci tohoto návrhového vzoru dojít.

Při refaktorování je nezbytné mít připravenou sadu funkčních testů. Pokud ji nemáme, je jednoduché zanést do programu chyby.

V úvodu této práce bude čtenáři představen pojem návrhový vzor, k čemu slouží a co nám přináší. Dále v kapitole č. 2 představím výběr návrhových vzorů, které jsou použity v aplikaci Netfox a následně v kapitole č. 3 bude představena jejich analýza využití. V této části také pojednávám o implementaci návrhových vzorů v prostředí .NET Framework. V další kapitole č. 4 je popsána refaktorizace části aplikace Netfox Detetcive. V kapitole č. 5 se čtenář dozví, co je to tzv. *mockování*, co jsou to *Unit testy*. Také je zde možno nalézt popis problémů, se kterými jsem se setkala při vytváření sady unit testů a jak tyto problémy řešit. V dalších dvou kapitolách č. 6 a č. 7 popisuji proces rozšiřování kódu, možnosti posuzování kódu či vysvětluji pojem jako *continuous integration* a analyzuji použití těchto procesů a nástrojů v aplikaci Netfox Detective. Nakonec v kapitole č. 8 shrnuji výsledky této práce.

Kapitola 2

Návrhové vzory

Při vývoji softwarového produktu se vývojář setkává s problémy specifickými pro daný projekt, ale také s takovými, které patří do kategorie úloh, se kterými se vývojář má nemalou šanci setkat při práci snad na každém projektu. Naštěstí, k některým problémům z této kategorie již byla nalezena ideální řešení, která představují právě návrhové vzory [39].

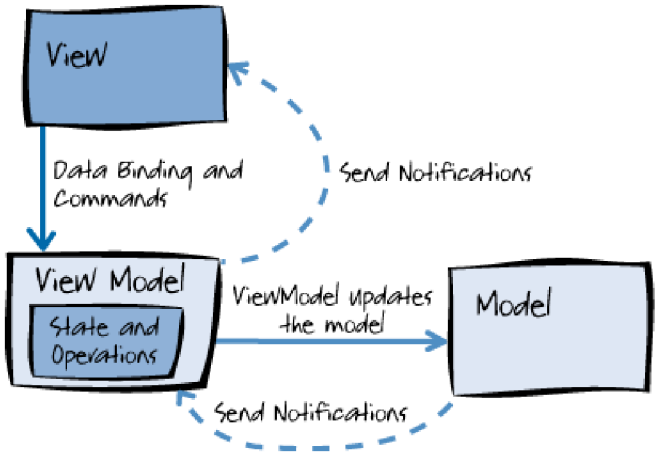
Pro jasnější pochopení co je to návrhový vzor, jak by měla vypadat jeho definice, můžeme čerpat z knihy *Design Patterns: Elements of Reusable Object-Oriented Software* [37], kde popis návrhového vzoru se skládá ze čtyř částí:

- **Problém, který návrhový vzor řeší** nám říká, kdy daný vzor použít. Může popisovat strukturální problém, tedy problém uspořádání objektů a instancí v aplikaci, problém s tvorbou objektů, či behaviorální, který řeší například jak spolu objekty komunikují.
- **Řešení problému** popisuje jednotlivé elementy řešení, vazby mezi nimy a jejich zodpovědnosti. Nepopisuje konkrétní implementaci problému, jelikož návrhový vzor má sloužit jako šablona, která je aplikovatelná v několika odlišných situacích.
- **Výhody a nevýhody**, které přináší aplikace návrhového vzoru by měly být brány v potaz při úvahách zda daný vzor použít. Následky mohou představovat využití paměti, časovou náročnost, obtížnost implementace vzoru v programovacím jazyce, či vliv na flexibilitu, rozšiřitelnost, a použitelnost aplikace na jiných systémech. Na druhou stranu mohou přinést čitelnější, testovatelnější, udržovatelnější a rozšiřitelnější kód.
- **Název** je nedílnou součástí popisu. Jedním, či dvěma slovy představuje problém a jeho řešení.

Při studiu návrhových vzorů se lze setkat i s pojmem *anti-pattern*, neboli česky *anti návrhový vzor*. Anti vzorem označujeme řešení k opakujícímu se problému, které po aplikaci přináší zcela určitě negativní důsledky, nebo jsou všeobecně vnímány jako neefektivní [32]. Můžeme tedy říct, že anti návrhové vzory jsou opakem návrhových vzorů. Zatímco návrhové vzory jsou považováný za dobré programátorské techniky, anti vzory nikoliv [32]. Příkladem anti návrhového vzoru může být například tzv. *Swiss Army Knife* neboli *švýcarský nožík*. Jedná se o třídu, která implementuje příliš mnoho nesouvisejících rozhraní [22]. Toto může vést k náročnějšímu pochopení k čemu třída slouží, jak má být využívána. Další důsledky implementování několika rozhraní může vést k obtížnějšímu ladění či údržbě aplikace.

2.1 Model-View-ViewModel

Model-View-ViewModel je jednim z návrhových vzorů, který slouží k oddělení logiky aplikace od uživatelského rozhraní. Lze jej využít ve všech aplikacích využívající XAML technologie (Windows Presentation Foundation, Windows Forms, Silverlight) [25] .



Obrázek 2.1: Obrázek představuje popis komunikace komponent návrhové vzoru MVVM. Převzato z článku [25]

MVVM se skládá ze tří komponent:

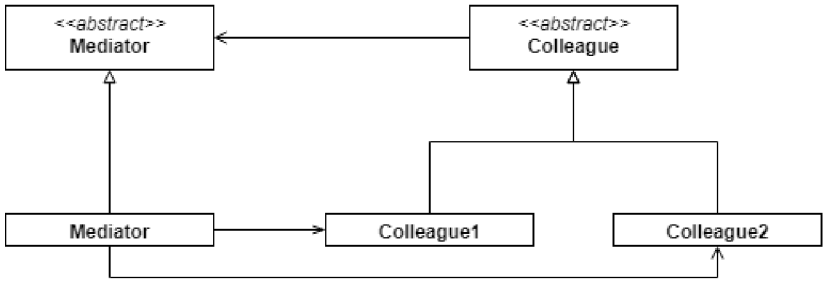
- **Model** představuje data, které aplikace využívá. Kromě samotných dat může model obsahovat i logiku pro validaci dat.
- **View** reprezentuje uživatelské rozhraní. Úkolem je definovat rozložení a způsob zobrazení dat. Toto je ideálně napsáno pouze v již zmíněném jazyce XAML. Třídy typu View by neměly vědět o třídách typu Model a naopak. Ke komunikaci mezi těmito dvěma komponentami slouží ViewModel.
- **ViewModel** implementuje vlastnosti [1], které pak View může využívat. ViewModel se také stará o formát dat, který View zpracovává.

Komunikace mezi View a ViewModel probíhá pomoci tzv. *data binding*. Jednoduše řečeno, jedná se o způsob komunikace mezi UI a business logikou. Díky tomuto, pokud se data změní, elementy, které jsou vázány k těmto datům získají o této změně notifikaci a mohou na ni tedy příslušně reagovat. Například pokud uživatel změní hodnotu v `TextBox` elementu, hodnota ve ViewModelu je ihned aktualizována [5]. Aby ViewModel byl schopen upozornit View na změnu, potřebují jeho vlastnosti, o kterých chce View informovat o změne, vyvolávat událost `PropertyChanged`. K tomuto je potřeba aby ViewModel implementoval `INotifyPropertyChanged`. Pro kolekce platí to samé, stačí používat `ObservableCollection<T>`, která již toto umožňuje díky implementaci rozhraní `INotifyCollectionChanged`.

[1] Vlastnost, v kontextu složení třídy, můžeme v anglické literatuře najít pod pojmem *property*. Jedná se o proměnnou objektu, která je veřejná.

2.2 Messenger

Návrhový vzor *Messenger*, můžeme nalézt také pod názvem *Mediator*, definuje objekt přes který ostatní objekty komunikují. Díky tomu, že objekty nekomunikují každý s každým, ale přes prostředníka, nedochází k tzv. *tight coupling*, tedy k úzce provázanému kódu komponent. Můžeme říct, že tento návrhový vzor tedy zabraňuje tzv. *spaghetti code* [14].



Obrázek 2.2: Colleague1 a Colleague2 obsahují instanci Mediatoru, přes který spolu komunikují. Instance Mediatoru ví o všech instancích Colleague, tedy má možnost jim zprávy přeposílat. Obrázek vlastní, obsah převzat z [15].

Příklad implementace lze vidět ve výpisu č. 2.1 a č. 2.2. V uvedené implementaci se využívá vlastnosti jazyka C#, tzv. *event* [2]. Implementaci třídy `Colleague` zde představuje třída `Person`.

Třída `Mediator` zpřístupňuje event `MessageReceived`, který slouží k registraci metod, které budou spuštěny v případě poslání zprávy. K posílání zpráv třída `Mediator` zpřístupňuje metodu `void Send(string message, string from)`.

```
public delegate void MessageReceivedEventHandler(string message,
    string from);

public class Mediator
{
    public event MessageReceivedEventHandler MessageReceived =
        (message, from) => {};

    public void Send(string message, string from)
    {
        MessageReceived(message, from);
    }
}
```

Listing 2.1: Třída `Mediator` z příkladu implementace návrhového vzoru Mediator. Převzato z [15]

Třída `Person` uchovává instanci třídy `Mediator`. K posílání zpráv používá již zmíněnou metodu `void Send(string message, string from)`. Její použití lze vidět v metodě `void Send(string message)`. Metoda
`void Receive(string message, string from)` obsahuje kód, který bude spuštěn v pří-

[2] Event, jakým způsobem je implementován v jazyce C# je vlastně implementací návrhového vzoru Observer.

padě odeslání zprávy přes totožnou instanci třídy `Mediator`, kde je tato metoda registrována. Registrace je provedena v konstruktoru třídy `Person`.

```
public class Person
{
    private Mediator _mediator;

    public string Name { get; set; }

    public Person(Mediator mediator, string name)
    {
        Name = name;
        _mediator = mediator;
        _mediator.MessageReceived += Receive;
    }

    private void Receive(string message, string from)
    {
        Console.WriteLine("Messag: {0} received from:
            {1}",message,from);
    }

    public void Send(string message)
    {
        _mediator.Send(message, Name);
    }
}
```

Listing 2.2: Třída `Person` z příkladu implementace návrhového vzoru Mediator v jazyce C#. Převzato z [15]

2.3 Singleton

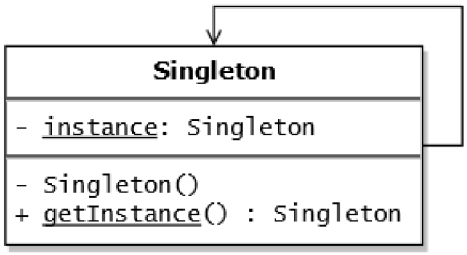
Při použití návrhového vzoru *Singleton* (česky *Jedináček*) dosáhneme toho, že instance daného typu je pouze jedna a je k ní umožněn globální přístup.

Problém, který nám tento návrhový vzor pomáhá řešit je tedy potřeba mít pouze jedinou instanci objektu po celý běh programu a tuto instanci mít globálně přístupnou. Zjednodušeným příkladem může být souborový systém v operačním systému.

Základní implementace *Singleton* je jednoduchá. Máme třídu s privátním konstruktorem a veřejnou statickou metodu, která jako jediná má přístup k jedinečné instanci. Většinou při přístupu k instanci se kontroluje, zda je instance již vytvořena. Pokud ano, vrátí se daná instance, pokud ne, vytvoří se, uloží a vrátí právě vytvořená instance. Tento mechanizmus kontroly bývá implementován právě v přístupové metodě.

V prostředí .NET Framework lze využít vlastnosti modifikátoru `static`. Díky tomuto modifikátoru proběhne inicializace ihned jakmile se přistoupí k jakémukoliv statickému členu třídy. Není tedy potřeba řešit vytvoření instance v její přístupové metodě a kontrolovat zda již není vytvořena [11].

Dále lze využít i modifikátoru `readonly`, kterým říkáme že k inicializaci dojde buď v konstruktoru nebo při statické inicializaci jako v našem příkladě. Implementace využívající těchto vlastností lze vidět ve výpisu č. 2.3.



Obrázek 2.3: Diagram tříd návrhového vzoru Singleton. Třída obsahuje instanci sebe sama, která není přímo zpřístupněna ostatním třídám. Přístup k ní je řízen přes metodu. K zaručení jedinečnosti instance je potřeba zabránit možnosti dědění z této třídy. Dále je nežádoucí mít možnost vytvářet nové instance třídy, proto je potřeba mít konstruktor privátní. Zdroj [20].

Ve více vláknovém programování, je potřeba zaručit vyhrazený přístup k instanci. Pokud tak nebude učiněno, mohlo by se stát, že instancí bude vytvořeno více, a tedy dojde k porušení podstaty tohoto návrhové vzoru. K předejití tohoto možného problému existuje několik řešení, ať již obecných nebo vyplývajících přímo z vlastností .NET Framework. Jedním řešením z první zmíněné skupiny je například tzv. *double-checked locking* [3] s použitím modifikátoru `volatile` [6]. Řešením z druhé skupiny je využití právě modifikátoru `static`.

```
public sealed class Singleton
{
   private static readonly Singleton _instance = new Singleton();

   private Singleton() {}

   public static Singleton Instance
   {
      get { return instance; }
   }
}
```

Listing 2.3: Implementace návrhového vzoru Singleton. Zdroj [11]

V některých situacích může být vytvoření objektu časově náročné, a tudíž nechceme aby konstrukce objektu proběhla pokud to není nutné. Tomuto způsobů se říká *lazy loading*, neboli česky opožděná inicializace.

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> _instance = new
       Lazy<Singleton>(() => new Singleton());

    private Singleton() {}

    public static Singleton Instance
    {
       get { return _instance.Value; }
    }
```

[3] Double-checked locking `https://en.wikipedia.org/wiki/Double-checked_locking`

```
}
```

Listing 2.4: Implementace návrhového vzoru Singleton pro vícevláknové apliakce. Zdroj [10]

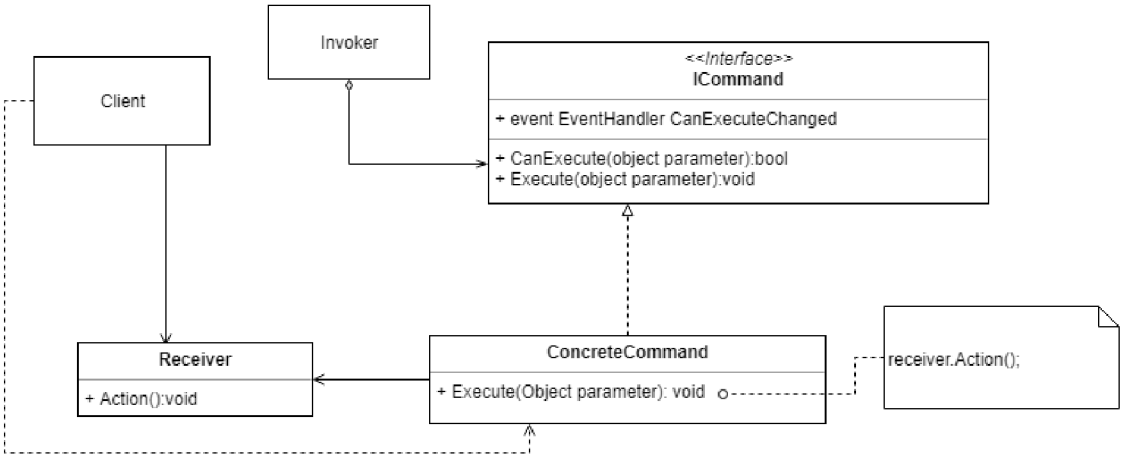
Určitá forma opožděné inicializace je již implementována ve výpisu č. 2.3, kde nedojde k inicializaci statických proměnných do té chvíle dokud se nepřistoupí k jakémukoliv statickému členu třídy.

Pokud chceme ale provést konstrukci objektu pouze tehdy, když přímo tento objekt potřebujeme, je potřeba využít další konstrukce jazyka C#. K tomuto nám pomůže třída `Lazy<T>`[4], která umožňuje tuto funkcionalitu [10]. Jelikož nyní statickou proměnnou bude instance `Lazy<Singleton>` a nikoliv instance typu `Singleton`, mohlo by se zdát, že nemáme zaručený výlučný přístup k instanci a tak čelíme opět problému vytvoření více instancí jedináčka. Naštěstí, třída `Lazy<T>` defaultně podporuje výlučný přístup. Implementaci s opožděnou inicializací lze vidět ve výčtu č. 2.4.

2.4 Command

Návrhový vzor *Command* nám umožňuje zapouzdřit a parametrizovat požadavek, bez toho aby klient věděl o způsobu jeho zpracování. Klient pouze definuje daný požadavek [4].

Využití tohoto návrhového vzoru můžeme vidět například u aplikací s grafickým uživatelským rozhraním (dále jen GUI). Knihovna reprezentující GUI používá objekty jako tlačítko či textové pole, které nějakým způsobem reagují na interakci od uživatele. Toto chování ale definuje jiný projekt, který definuje funkcionalitu celé aplikace [37, p. 263].



Obrázek 2.4: Diagram tříd návrhového vzoru Command. Uvedený diagram je přizpůsoben k využití rozhraní poskytovaného .NET Framework. Inspirace pro obrázek převzata z [37, p. 266].

Ve výpisu č. 2.5 níže, můžeme vidět část implementace návrhového vzoru Comand zobrazeného v obrázku č. 2.4. Třída `DeleteWorkspaceCommand` je implementací rozhraní `ICommand`. Třídu `Receiver` z diagramu tříd č. 2.5 reprezentuje implementace rozhraní `IFileSystem`.

```
class DeleteWorkspaceCommand: ICommand
```

[4] `Lazy<T>` https://msdn.microsoft.com/en-au/library/dd642331(v=vs.110).aspx

```
{
    private readonly IFileSystem _fileSystem;

    public DeleteWorkspaceCommand(IFileSystem fileSystem)
    {
        this._fileSystem = fileSystem;
    }

    public bool CanExecute(object parameter) { return parameter is
       Workspace;}

    public void Execute(object parameter)
    {
        var workspace = parameter as Workspace;

        _fileSystem.Delete(workspace);
    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
}
```

Listing 2.5: Implementace návrhového vzoru Command v prostředí .NET Framework. Zdroj vlastní.

2.5 Dependency Injection

Dependency Injection (dále jen DI) neboli česky *vkládání závislostí*, je v objektově orientovaném programování technika pro vkládání závislostí mezi jednotlivými komponentami programu tak, aby jedna komponenta mohla používat druhou, aniž by na ni měla v době sestavování programu referenci [28].

DI se dá chápat i ne jako přímo návrhový vzor, ale souhrnný název pro několik návrhových vzorů a principů, umožňujících psaní volně vázaného kódu [52].

2.5.1 Typy dependency injection

Způsobů jak předat závislosti třídě máme tři:

Constructor injection Vkládání přes konstruktor vyžaduje definování závislostí v konstruktoru třídy.

Nevýhodou tohoto přístupu je, že po proběhnutí konstruktoru jsou všechny závislosti instanciovány okamžitě, a to i přesto, že v budoucnu k nim nebude vyžádán přístup.

Vhodné je označit field [5], který uchovává instanci závislosti jako readonly. Tímto zaručíme, že jakmile konstruktor proběhne, field nemůže být dále modifikován.

Property injection vyžaduje vytvoření modifikovatelné vlastnosti požadovaného typu závislosti. Tento způsob se nazývá také *Setter injection*.

[5] Field je proměnná, která není veřejná ale privátní.

Pokud použijeme tento způsob, měli bychom mít k dispozici defaultní implementaci této závislosti. Pokud tak neučiníme a přistoupíme k dané vlastnosti, hrozí nám vyskytnutí `NullReferenceException` vyjímky. Tudíž, tento způsob se doporučuje použít pouze v případě, kdy chceme volajícím umožnit přepsat defaultní implementaci. Použitím tohoto způsobu také volajícím sdělujeme, že tato závislost není povinná.

Pro defaultní implementaci můžeme využít návrhového vzoru *Null Object*. Pak tedy vlastnost představuje tzv. *extensible point*. V tomto řešení lze vidět aplikaci principu Open-closed vysvětleného v další podkapitole.

Method injection Vkládání závislostí přes metodu je odlišné od předcházejících způsobů v momentu kdy se vložení vykoná. Nevykoná se jak u předcházejících způsobů v době instanciace objektu, ale v době volání dané metody. Tento způsob se nejvíce hodí využít v situacích, kdy závislost je potřeba být odlišná pro jednotlivá volání metody.

2.5.2 SOLID

DI k dosažení volně vázaného kódu používá tzv. SOLID principy. Jedná se o pět pravidel, které při jejich dodržování napomáhají vytvářet snadno rozšiřitelný a udržovatelný software.

SOLID je akronym, který vytvořil Robert C. Martin a uvedl ve své knize *Agile principles, patterns, and practices in C#* [40], ačkoliv jednotlivé principy vznikly již dříve.

Single responsibility principle Princip jedné zodpovědnosti říká, že třída by měla mít jedinou zodpovědnost, jediný důvod ke změně. Výhodou tohoto principu, pokud se dodržuje, je kromě snadnějšího udržování kódu i snadné pojmenovávání tříd. Název třídy pak jasně vystihuje danou zodpovědnost.

Open closed principle Princip otevřenosti a uzavřenosti říká, že každá třída by měla být otevřená pro rozšíření ale uzavřená pro modifikaci. Tedy je možné přidat nový kód, bez zásahu již do existujícího.

Další princip **Liskov substitution principle** , česky Liskovové princip zaměnitelnosti, říká, že bysme měli být schopni zaměnit implementaci rozhraní jinou implementací toho samého rozhraní, bez toho, abychom dále museli měnit kód.

Interface segregation principle Tento princip upřednostňuje používání více malých úzce zaměřených rozhraní, nežli jedno univerzální rozhraní. Dodržováním tohoto principu se tak vyhneme problému, kdy musíme implementovat například metodu i přesto, že objekt implementující toto složité rozhraní by tuto metodu mít implementovanou neměl.

Posledním principem je **Dependency inversion principle**. Princip inverze závislostí tvrdí, že moduly na vyšší úrovni by neměly záviset na modulech nízkoúrovňových. Oba by měly záviset na abstrakcích (rozhraní, abstraktní třídy). A dále, abstrakce by neměly záviset na implementačních detailech, ale naopak - detaily by měly záviset na abstrakcích [40] .

2.6 Shrnutí

Hlavní výhodu, kterou používání návrhových vzorů přináší, je snazší komunikace v týmu. Je snazší a rychlejší popsat část architektury aplikace jedním nebo dvěma slovy, než popisovat objekty, vztahy mezi nimi a jejich zodpovědnosti. Další výhodu vidím, že nabízí použití již prověřeného řešení. Většina návrhových vzorů existuje již několik let, a tak lze o nich nalézt spoustu užitečných informací z praxe na internetu [51], v článcích z konferencí [47], či literatuře [40]. Můžeme se dočíst, že například návrhový vzor Singleton, považují někteří programátoři a softwarový architekti za spíše anti návrhový vzor a upřednostňují využívat

jiné návrhové vzory [47] . Nevýhodu vidím v tom, že pro jeden návrhový vzor existuje několik názvů či modifikací. Toto může být pro nové uživatele matoucí a tak je odradit od hlubšího studia a následného používání.

Ačkoliv jsem uvedla několik způsobů vkládání závislostí, preferuji první způsob, protože jasně dokumentuje jaké další komponenty pro instanciaci třídy jsou potřeba a protože vkládané závislosti jsou většinou potřebné pro funkcionalitu závislé třídy.

Kapitola 3

Využití návrhových vzorů

V kapitole č. 2 byla představena teorie k několika návrhovým vzorům, které je možno identifikovat v aplikaci Netfox Detective. Byl stručně popsán problém, který se snaží vyřešit i způsob jak přistupují k tomuto problému. Jejich struktura byla popsána pomocí UML diagramů, následně byla předvedena i vzorová implementace v jazyce C#.

Tato kapitola se věnuje využití návrhových vzorů zmíněných v kapitole č. 2. Uvádím zde přímo jejich implementaci v aplikaci Netfox Detective a zabývám se, zda implementace neobsahuje nedostatky.

3.1 Model-View-ViewModel

V aplikaci Netfox se využívá tohoto návrhové vzoru na několika místech. K analýze jsem si vybrala část kódu, který se zabývá zobrazení tzv. Workspace. Workspace je pracovní prostředí, které se může skládat z několika Investigation. Investigation může představovat například analýzu `pcap` souborů.

Po práci s Workspace se používá několik tříd:

- Workspace
- WorkspaceVm
- WorkspaesManagerService
- WorkspaceManagerVm

Vazby mezi nimi lze vidět v obrázku č. 3.1.

Obrázek 3.1: Zjednodušený diagram tříd zobrazující vazby mezi třídami sloužící k implementaci Workspace. Zdroj vlastní.

3.1.1 Třída Workspace

Tato třída definuje především vlastnosti vhodné pro serializaci informací o Workspace atributem `[DataMember]`. Dále také implementuje metody, pro vytváření, mazání a načítání

tříd typu `Investigation`. Obsahuje i kolekci, kde uchovává Investigation náležící danému Workspace. Jak se lze dočíst, tato třída má několik zodpovědností, čímž porušuje Single responsibility princip (viz 2.5.2).

Domnívám se, že nejlepším využitím této třídy by bylo, kdyby sloužila pouze pro uchování dat, která jsou označena atributem `[DataMember]`. Správu Investigation by měla mít na starost jiná třída.

3.1.2 Třída WorkspaceVm

Tato třída se používá jako Model (výčet 3.1) ale je označena jako ViewModel. Zároveň kromě uchovávání dat poskytuje i serializaci objektů typu `Workspace`. Toto nepovažuji za vhodné, neboť jedna třída zodpovídá opět za více nežli jednu věc. Řešením je například vytvořit jinou třídu, která bude zodpovídat pouze za uchovávání dat, a třídu, která se bude starat o serializaci. Tímto k porušení výše zmíněného principu nedojde.

```
public ICommand COpenWorkspace =>
new RelayCommand<WorkspaceVm>(async vm
   => await this.WorkspacesManagerService.OpenWorkspace(vm),
      workspace => true);
```

Listing 3.1: Využití třídy `WorkspaceVm` jako komponenty typu Model. Větev master ke dni 20.1.2018

Dále se také stará o přidávání tříd typu `InvestigationVm` do kolekce, která je nadále používána třídou `InvestigationManagerVm`. Toto si nemyslím, že je vhodné. Vhodnější považuji aby tuto kolekci obsahovala přímo třída `InvestigationManagerVm`.

3.1.3 Třída WorkspacesManagerService

Za nešťastné řešení dále považuji využití i třídy `WorkspaceVmService`. Tato třída slouží jako vrstva mezi třídou `WorkspaceManagerVm` a třídou `WorkspaceVm`. Spoustu věcí tedy deleguje na ostatní třídy.

Můžeme zde nalézt ale i metody, které se starají o serializaci seznamu právě nahraných Workspace. Toto je dosaženo pomocí metod `void FromString(string source)` a `string ToString()`.

Ačkoliv asi největší prohřešek této implementace nesouvisí přímo s návrhovým vzorem, považuji za důležité aby názvy tříd definovali k čemu slouží. Jestli tedy třída má v názvu řetězec Vm či ViewModel, měla by pak také jako ViewModel sloužit. Dále zanoření modelů (`WorkspceVm` obsahuje `Workspace`) mi přijde zbytečně složité a špatně udržovatelné.

3.2 Messenger

V aplikaci Netfox se již využívá tohoto vzoru z externí knihovny MVVM Light Toolkit[1]. Jedná se o třídu `GalaSoft.MvvmLight.Messaging.Messenger`.

Komunikace probíhá díky tzv. *zprávám*. Klient se u Messenger registruje k odběru určitých zpráv. V Netfox regitrace probíhá přes metodu `Register`. V této metodě se předá informace o typu zprávy kterou chceme odebírat `WorkspaceMessage`, o adresátu, a metodě, která se má vykonat po přijetí zprávy `WorkspaceMessageHandler`.

[1] MVVM Light Toolkit `http://www.galasoft.ch/mvvm`

```
Messenger.Default.Register<WorkspaceMessage>(this,
    this.WorkspaceMessageHandler);
```

Listing 3.2: Registrace k odběru zpráv ve třídě `InvestigationManagerVm`. Větev master ke dni 23.1.2018

K odesílání zpráv Messenger implementovaný v knihovně MVVM Light Toolkit nabízí metodu `Messenger.Send<TMessage>(TMessage)`. Netfox Detective pro posílání zpráv nepoužívá přímo knihovní funkci jako v případě registrace k odběru zpráv, ale vlastní metody. V aplikaci můžeme najít třídu `DetectiveMessage`, která slouží jako bázová třída pro další typy zpráv. I přestože, tato třída má reprezentovat zprávu k odeslání, implementuje i posílání daných zpráv.

Na tomto způsobů využití návrhového vzoru Messenger v aplikaci Netfox Detective se mi nelíbí nedodržování úrovní abstrakcí. Zatímco k odesílání zpráv je využito přímo funkce z externí knihovny (výpis 3.2), k registraci odběru zpráv používáme metody, které jsou implementované v příslušných Messenger třídách (výpis 3.3).

```
WorkspaceMessage.SendWorkspaceMessage(workspaceVm,
    WorkspaceMessage.Type.Created);
```

Listing 3.3: Odesílání zprávy typu `WokspaceMessage` ve třídě `WorkspaceManagerService`. Větev master ke dni 23.1.2018

3.2.1 Typy zpráv

Další nedostatek vidím v implementaci metod, které jsou vykonány po přijetí zprávy.

```
private void WorkspaceMessageHandler(WorkspaceMessage message)
{
        if(message == null) { return; }
        switch(message.MessageType)
        {
                case WorkspaceMessage.Type.Created:
                        // ...
                case WorkspaceMessage.Type.Opened:
                        // ...
                case WorkspaceMessage.Type.Closed:
                        // ...
        }
}
```

Listing 3.4: Metoda `WorkspaceMessageHandler` ve třídě `WorkspaceManagerService`, která se vykoná po přijetí zprávy. Větev master ke dni 23.1.2018

Jak lze vidět ve výpisu 3.4, po přijetí zprávy se vykoná kus kódu podle užšího typu zprávy. Vhodnější považují vytvořit třídy i pro tyto typy zpráv a používat je pak jako parametry při registraci a odběru, jako je vidět například u typu `WorkspaceMessage`. Kód tak nebude porušovat Open-closed princip, popsaný níže v podkapitole 2.5.2.

3.3 Singleton

Ve výpisu 3.5 lze vidět využití tohoto návrhového vzoru v aplikaci Netfox Detective. Využívá se zde i jedné z vlastností .NET Framework. Díky tomu, že `defautInstance` je statická,

proběhne její inicializace ihned jakmile se přistoupí k tomuto členu. Proto není potřeba řešit v logice vlastnosti `Default` vytváření instance či jedinečný přístup [11].

V aplikaci Netfox je Singleton využito pro přístup k nastavení aplikace.

```
public sealed partial class NetfoxSettings // ...
{
      private static NetfoxSettings defaultInstance =
          ((NetfoxSettings)(global::System.Configuration
      .ApplicationSettingsBase.Synchronized(new NetfoxSettings())));

      public static NetfoxSettings Default {
                get
                {
                    return defaultInstance;
                }
      }

      // ...
}
```

Listing 3.5: Implementace návrhového vzoru Singleton v aplikaci Netfox. Větev master ke dni 20.1.2018

3.4 Command

Jelikož v aplikaci Netfox Detective se využívá návrhového vzoru MVVM, je jisté že se zde bude využívat i návrhového vzoru Command. Jeho použití můžeme nalézt například ve třídě `WorkspaceManagerVm`. Ukázka použití je zobrazena ve výpisu č. A.1.

```
public RelayCommand CCreateWorksCommand =>
  new RelayCommand(() => this.NavigationService.Show(...));
```

Listing 3.6: Ukázka použití návrhového vzoru Command v aplikaci Netfox Detective. Větev develop ke dni 5.5.2018

Lze si všimnout, že implementací tohoto návrhového vzoru je třída `RelayCommand` [2]. Tuto třídu nenalezneme přímo v aplikaci Netfox Detective, ani v .NET Framework, ale opět v knihovně MVVM Light Toolkit.

V tomto případě na pozici `Receiver`, jako bylo ukázáno v podkapitole 2.4 není třída, ale pouze delegát [3]. Dalším parametrem je také delegát, ale určující, zda lze objekt typu Command spustit. Jednoduchou implementaci `RelayCommand` lze vidět v příloze A.

3.5 Dependency injection

V aplikaci Netfox se můžeme setkat se všemi třemi způsoby vkládání závislostí. Bohužel u předávání závislostí přes vlastnost instance není dodrženo doporučení, že by měla existovat defaultní implementace. Není zde ani kontrola při přístupu k vlastnosti, zda není `null`. Z tohoto důvodu se i domnívám, že většina z těchto závislostí by mohla být součástí vkládání závislostí přes konstruktor.

[2] RelayCommand `https://bit.ly/2jX8OqZ`

[3] Delegát představuje referenci na metodu.

3.5.1 SOLID

Single responsibility principle Nedodržení tohoto principu lze ukázat na třídě `Workspace` v aplikaci Netfox. Záměrem této třídy je uchování informací o workspace za účelem serializace a deserializace. Kromě dat k serializaci, zde najdeme ale i metodu, která vykonává samotnou deserializaci `void Load()`, či jiné metody které se starají o vytváření dalších členů této třídy:

- `Task CreateAndAddNewInvestigation(InvestigationInfo investigationInfo)`
- `Task CreateAndLoadInvestigation(InvestigationInfo investigationInfo)`

3.5.2 Castle Windsor Framework

Pro usnadnění používání DI principů jsou k dispozici DI frameworky. Můžeme se setkat i s označením: Inversion of Control framework, Inversion of Control container či Dependency Injection container.

K aplikaci DI principů sice není potřeba DI framework, ovšem jeho využití značně usnadňuje vývoj aplikace. V Netfox Detective aplikaci je využíváno Castle Windsor [4]. Cílem tohoto frameworku, je správa instancí aplikace. To znamená, že za uživatele řeší jejich vytváření, destrukci, životnost, nastavení a závislosti na ostatních instancích [49] .

Pro správné používání tohoto frameworku by mělo být dodržováno, že aplikace o tomto frameworku skoro vůbec neví. Je tím myšleno, že by se k instanci kontejneru nemělo vůbec přistupovat, tedy kromě místa v aplikaci, kde kontejner vytváříme, což je v případě Netfox Detective systému třída `App.xaml.cs`.

```
protected override void OnStartup(StartupEventArgs e)
{
            // ...
    this.ApplicationWindsorContainer =
    new WindsorContainer("DetectiveApp", new DefaultKernel(),
        new DefaultComponentInstaller());
            // ...
}
```

Listing 3.7: Ukázka vytvoření kontejneru v aplikaci Netfox Detective. Jediné místo v celé aplikaci kde by se mělo přímo pracovat s intancí frameworku Windsor Container. Větev master ke dni 24.1.2018

Toto bohužel není dodržováno. Místo toho kontejner je využíván spíše jako anti návrhový vzor *Service Locator*. Toto má za následky nejasné užívání závislostí (závislosti jsou získávány uvnitř metod, nejsou tedy definovány v API třídy), obtížné testování a může přinést i problémy s údržbou [51].

Použití můžeme vidět například ve třídě `DetectiveApplicationWindsorInstaller`, jejíž velice zjednodušena implementace je ve výpisu č. 3.9. Můžeme zde vidět implementaci rozhraní `IWindsorInstaller`, které definuje pouze jednu metodu, jak lze vidět ve výpisu č. 3.8 . V této metodě registrujeme naše závislosti.

Jak již bylo uvedeno dříve, jednou ze zodpovědností Castle Windsor frameworku je řízení životnosti instancí. Zřejmě nejběžnější typy životnosti jsou `LifeStyleSingleton`, jediná instance bude existovat po celou existence kontejneru, a `LifestyleTransient`, nová instance třídy bude vytvořena kdykoliv si o ní nějaký objekt zažádá.

[4] Castle Windsor `https://github.com/castleproject/Windsor`

Ve výpisu č. 3.9 si dále můžeme všimnout registrace pouze třídy `NetfoxFileAppender` a registrace třídy `InvestigationInfo`, která implementuje rozhraní `IInvestigationInfo`. To znamená, že kdykoliv například v konstruktoru třídy budeme mít parametr typu `IInvestigationInfo`, dostaneme instanci třídy `InvestigationInfo`.

Castle Windsor framework nám taky usnadňuje práci při implementaci některých návrhových vzorů jako jsou Factory, či Decorator. Pro využití prvně zmíněného návrhového vzoru je potřeba Castle Windsor předat rozhraní, které definuje metody poskytované třídou reprezentující návrhový vzor Factory. Toto předání lze také vidět ve výpisu č. 3.9.

```
namespace Castle.MicroKernel.Registration
{
        using Castle.MicroKernel.SubSystems.Configuration;
        using Castle.Windsor;

        public interface IWindsorInstaller
        {
                void Install(IWindsorContainer container,
                   IConfigurationStore store);
        }
}
```

Listing 3.8: Rozhraní `IWindsorInstaller`. Převzato z Castle Windsor.

```
public class DetectiveApplicationWindsorInstaller :
    IWindsorInstaller
{
    public void Install(IWindsorContainer container,
        IConfigurationStore store)
    {
        container.Register(Component.For<NetfoxFileAppender>()
            .LifestyleSingleton());
        container.Register(Component.For<IInvestigationInfo,
            InvestigationInfo>()
            .LifestyleTransient());
        container.Register(Component.For<IWorkspaceFactory>()
            .AsFactory());
    }
}
```

Listing 3.9: Třída `DetectiveApplicationWindsorInstaller`.

Kapitola 4

Refaktorizace

V kapitole č. 3 jsem popsala využití návrhových vzorů v aplikaci Netfox Detective a případně i problémy v jejich implementaci. Nyní v této kapitole popíši mnou implementované řešení k některým z těchto problémů.

4.1 Model-View-ViewModel

Problémy, které byly součástí implementace tohoto vzoru jsem popsala v podkapitole č. 3.1. Jednalo se především o porušování Single responsibility principu (viz 2.5.2). Tudíž k většině problémů se kterými jsem se setkala bylo řešení použití refaktorizačního přístupu *extract method* [36, p. 115].

Při refaktorizaci tříd jsem často používala tedy tento postup:

1. Identifikace metod, které chceme z této třídy vyjmout.
2. Identifikace tříd, které tyto metody používají.
3. Pokud je metoda používána pouze jednou třídou, přesuneme do ní danou metodu.
4. Pokud je metoda používaná více nežli jednou třídou, vytvoříme novou třídu, která bude obsahovat tuto metodu. Tuto novou metodu dále používá třída, dříve používající metodu v refaktorované třídě.
5. Původní metodu smažeme.

Obrázek 4.1: Zjednodušený diagram tříd zobrazující vazby mezi třídami sloužící k implementaci Workspace. Stav po refaktorizaci. Zdroj vlastní.

4.1.1 Třída Workspace

Z této metody jsem přesunula všechny vlastnosti a metody, které nebyly nezbytné pro serializi. Zůstali zde tedy pouze vlastnosti, které byly označeny atributem `[DataMember]`. Funkcionalita ukládání a nahrávání objektů typu `Investigation` jsem přesunula do nově

vytvořené třídy `InvestigationSerializationPersistor` implementující rozhraní `ISerializationPersistor<Investigation>`. Výsledek této refaktorizace lze vidět v příloze C.

4.1.2 Třída WorkspaceVm

Jelikož název této třídy byl zavádějící, a třída pouze delegovala většinu funkcionality na třídu `Workspace`, rozhodla jsem se tuto třídu zcela odstranit. Kolekci obsahující instance typu `InvestigationVm` jsem přesunula do třídy `InvestigationManagerVm`.

Při refaktorizaci této třídy jsem také vytvořila implementaci `WorkspaceSerializationPersistor` rozhraní `ISerializationPersistor<Workspace>`, která má za úkol ukládat a načítat data definované ve `Workspace`.

4.1.3 Třída WorkspacesManagerService

Tuto třídu již nadále v kódu aplikace Netfox Detective také nenajdeme. Byla odstraněna ze stejného důvodu jako třída `WorkspaceVm`, pouze delegovala funkcionalitu. Ukládání seznamu načtených Workspace jsem přesunula do další nově vytvořené třídy `WorkspacePathSerializationPersistor`, která opět implementuje generické rozhraní `ISerializationPersistor<WorkspacePath>`.

4.1.4 Rozhraní ISerializationPersistor<TItem>

Ačkoliv se tento typ rozhraní nepoužívá návrhovým vzorem MVVM, považuji za důležité zde toto rozhraní uvést, neboť bylo několikrát zmíněno v předešlých sekcích.

```
public interface ISerializationPersistor<TItem>
{
    TItem Load(string path);
    void Save(TItem item);
}
```

Listing 4.1: Rozhraní `ISerializationPersistor<TItem>`.

Definici tohoto rozhraní můžeme vidět ve výpisu č. 4.1. Obsahuje pouze dvě metody, a to pro načtení a uložení instance daného typu. Toto rozhraní bylo vytvořeno se záměrem možnosti sjednocení způsobu serializace, a jeho změny.

4.2 Messenger

Při popisu implementace tohoto návrhového vzoru jsem popsala problém porušování úrovní abstrakce (viz 3.2). Tento problém jsem vyřešila implementací nové třídy `DetectiveMvvmLightMessenger` (výpis č. 4.3) implementující rozhraní `IDetectiveMessenger` (výpis č. B.1).

```
public interface IDetectiveMessenger
{
    void Register<TMessage>(object recipient, Action<TMessage>
        action);

    void Send<TMessage>(TMessage message);
```

```
    void AsyncSend<TMessage>(TMessage message);
}
```

Listing 4.2: Rozhraní `IDetectiveMessenger`.

Implementace i nadále využívá k posílání zpráv implementaci návrhového vzoru Messenger z knihovny MVVM Light Toolkit. Nyní je ale jednoduché tuto implementaci nahradit jinou implementací, která používá například jinou knihovnu, bez zásahu do kódu používající výše uvedené rozhraní. Takto tedy v případě změny implementace nedojde k porušení pravidel SOLID (viz 2.5.2). V případě použití přímo knihovních funkcí by toto nebylo možné, neboť implementace návrhového vzoru Messenger je uskutečněna pomocí statických metod.

Také při použití této třídy nebude docházet k porušování abstrakce, tedy k registraci se nebude používat přímo metoda implementována knihovnou MVVM Light Toolkit a k odesílání zpráv metoda, která je implementovaná v dané zprávě, jak to bylo doposud.

```
public class DetectiveMvvmLightMessenger: IDetectiveMessenger
{
    private readonly IMessenger _messenger = Messenger.Default;
    public void Register<TMessage>(object recipient,
        Action<TMessage> action)
    {
        this._messenger.Register<TMessage>(recipient, action);
    }

    public void Send<TMessage>(TMessage message)
    {
        _messenger.Send(message);
    }

    public void AsyncSend<TMessage>(TMessage message)
    {
        DispatcherHelper.UIDispatcher?.BeginInvoke(new
            ThreadStart(() => this._messenger.Send(message)),
            DispatcherPriority.Send);
    }
}
```

Listing 4.3: Třída `DetectiveMvvmLightMessenger` implementující rozhraní `IDetectiveMessenger`.

4.2.1 Typy zpráv

Dalším problémem původní implementace byla identifikace užšího typu zprávy. Toto bylo uskutečňováno pomocí konstrukce `switch` jak bylo ukázáno ve výpisu č. 3.4. Tuto konstrukci musel obsahovat každý objekt, který chtěl nějaký typ zpracovávat.

Řešením, jak jsem již dříve navrhla, bylo vytvoření tříd i pro tyto užší typy zpráv. Pro uvedený případ se jedná o vytvoření těchto tříd:

- `CreatedWorkspaceMessage`
- `OpenedWorkspaceMessage`
- `ClosedWorkspaceMessage`

Jednotlivé třídy pak pouze obsahují data, která chtějí předat objektu, který tuto zprávu obdrží. Neobsahují tedy metodu pro odesílání dané zprávy, nebo jakoukoliv jinou metodu, jak bylo implementováno dříve. Takto jedinou zodpovědností dané třídy je pouze uchování informace o zasílané zprávě.

```
class CreatedWorkspaceMessage
{
    public Models.WorkspacesAndSessions.Workspace Workspace
    { get; set; }
}
```

Listing 4.4: Ukázka implementace zprávy typu `CreatedWorkspaceMessage`. Daná třída, která chce přijímat zprávu určitého typu, se registruje přímo pro daný typ zprávy.

4.2.2 Registrace ve Castle Windsor

Aby tento kód mohl být využíván, bylo potřeba jej zaregistrovat ve Castle Windsor. Tohoto jsem dosáhla přidáním jednoho řádku kódu do třídy `DetectiveApplicationWindsorInstaller`, jak lze vidět ve výpisu č. 4.5.

```
container.Register(Component
    .For<IDetectiveMessenger,DetectiveMvvmLightMessenger>());
```

Listing 4.5: Třída `DetectiveApplicationWindsorInstaller` registrující rozhraní `IDetectiveMessenger` a jeho implementaci.

Nutno poznamenat, že jelikož není specifikována životnost instance explicitně, bude instance existovat po celou dobu existence samotného kontejneru. Tato vlastnost, že bude vytvořena pouze jediná instance třídy `DetectiveMvvmLightMessenger`, je nezbytná pro správnou funkcionalitu návrhového vzoru Messenger, neboť zaslaná zpráva se doručí všem odběratelům, kteří jsou zaregistrování u instance rozesílající tuto zprávu.

4.3 Shrnutí

Vybraná oblast Workspace k refaktorizaci lze vidět v repositáři, zde její současnou implementaci neuvádím, neboť se jedná o velké množství kódu. Během této modifikace jsem bohužel musela porušit některé principy, které jsem v této práci popsala. Důvodem k těmto porušením byla skutečnost, že kvůli zachování funkcionality jsem byla nucena zasahovat i do tříd, které neměli s oblastí mnou primárně refaktorovanou téměř nic společného, nebo jsem byla nucena pracovat s třídami z jiných modulů, které nejsou plně funkční a jejich oprava by mi vyčerpala čas určený na téma této práce. Neboť kromě samotného dopsání správné funkcionality, by bylo potřeba napsat i sadu unit testů, zajistit automatické testování, aby opět nedošlo k regresi.

I přes problémy se kterými jsem se setkala, si dovoluji tvrdit, že jsem přispěla ke kvalitnějšímu a přehlednějšímu kódu, neboť typy tříd návrhového vzoru MVVM nyní reprezentují data a funkcionalitu, kterou by měly.

Všechny problémy vytknuté v podkapitole č. 3.2 se mi podařilo odstranit. Nyní při použití uvedeného rozhraní `IDetetiveWorkspace` nedochází k porušení abstrakce. Díky implementací zpráv pomocí tříd jsem dokázala odstranit konstrukce `switch` ze všech tříd, které se registrovali k některé ze zpráv.

Kapitola 5

Testování

Při refaktorizaci se může snadno stát, že narušíme současnou funkcionalitu systému. Abychom se tomuto vyvarovali, je potřeba mít dostupnou sadu testů, která nám pomůže verifikovat, že provedené změny neprovedly nežádoucí modifikace v systému. Jedním z nejdůležitějších testů k tomuto účelu slouží tzv. Unit testy, které dávají téměř okamžitou zpětnou vazbu vývojáři, zda k nežádoucí změně nedošlo, a může tak pokračovat v refaktorizaci s mnohem větší jistotou. V této kapitole tedy představuji především tento typ testů, jakožto jeden z nejdůležitějších nástrojů při refaktorizaci, ale mimo to, i jiné typy testů, díky kterým je možno sledovat regresi funkcionality testované aplikace.

5.1 Unit test

Pojem *unit test* (viz 5.1), je v softwarovém vývoji již dlouhodobě známým pojmem. Za ta léta se používání unit testů ukázalo jako jeden z nejlepších způsobů jak může vývojář zvýšit kvalitu kódu [44]. V této kapitole bych chtěla tedy čtenáři přiblížit tento důležitý pojem a ukázat jeho využití v aplikaci Netfox Detective.

Unit test je automatizovaný kus kódu, který spouští část produkčního kódu, otestuje jej, a vyhodnotí výsledek tohoto spuštění [44].

Při unit testování se také používá pojem *SUT = System Under Test.* Jedná se o komponentu, kterou testujeme.

Unit test by měl mít tzv. *F.I.R.S.T* vlastnosti: [45]

Fast Unit testy by měly být rychlé. Pokud testy budou pomalé, nikdo je nebude chtít spouštět.

Isolated Napsané testy by měly mít pouze jeden důvod k neúspěchu, proto je potřeba je izolovat od ostatních komponent, které by při běhu testu mohly nějakým způsobem selhat. Například při unit testování, by test neměl komunikovat se souborovým systémem. Je zde možnost, že by například čtení či zápis do složky nemusel být oprávněným přístupem, pak by test selhal, a to přesto, že test SUT by za jiných okolností prošel. Testy by ale neměly být jen nezávislé na prostředí kde běží, ale i vůči ostatním testům. To znamená, že by nemělo záležet v jakém pořadí se testy spouští, či v jakých intervalech. Je tedy potřeba zajistit, aby každý test zanechával po sobě prostředí takové, v jakém byl spuštěn.

Repeatable Test by měl poskytovat stejné výsledky kdykoliv jej spustíme. Měl by být tedy stabilní. Na tuto vlastnost má vliv předešlá vlastnost, kdy pokud zvýšíme izolovanost testu, pak je skoro nemožné aby test skončil pokaždé s jiným výsledkem.

Self-verifying Nemělo by se provádět manuální hodnocení úspěšnosti testu. Pokud jsou tu nějaké pochybnosti o výsledku testu, jejich hodnocení zbytečně zabere vývojářům čas. **Timely** Unit testy by měly být psány před psaním produkčního kódů. Toto je vlastností tzv. *Test Driven Development*, česky testy řízeného vývoje. Pokud se neaplikujeme tento způsob vývoje tak toto pravidlo není podstatné.

Při unit testing se používá tzv. *3A pattern*. Jedná se o rozdělení unit testu na tři části:

- **Arrange** Zde se vykonávají potřebné předpoklady pro běh testu, jako například nastavení mockovaných služeb.
- **Act** Výkonání akce nad SUT.
- **Assert** Ověření výsledků. V této části by se mělo objevit pouze a právě jedno ověření.

5.1.1 Mocking

Mockování umožňuje vytvářet falešné objekty, tedy objekty, které se tváří jako skutečné, ovšem nemají jejich funkcionalitu. Chování tohoto mockovaného objektu si můžeme nastavit podle vlastních potřeb, tedy lze nastavit například jakou honotu bude metoda vracet při zavolání s určitými parametry.

Pro mockování využívám frameworku Moq[1]. Tento framework dokáže mockovat rozhraní, tak i třídy.

5.1.2 Vytvoření sady

Pro oblast refaktorizace v systému Netfox Detective jsem si vybrala backend správy objektů workspace. K interakci s těmito komponentami se uživatel dostane ihned po načtení aplikace. Představují pracovní prostředí, kde je možno spravovat tvorbu, mazání, modifikací investigací.

Pokud chce vývojář začít s refaktorizací, je nezbytné mít napsané testy. Bohužel, né vždy platí, že k produkčnímu kódu, který chceme refaktorovat, existuje sada testů. Dalším problém se může vyskytnout v momentě, kdy testy chceme psát - produkční kód je netestovatelný. V této situaci, vývojáří nezbývá nic jiného, než opatrně provádět refaktorizaci bez testů. Toto je problém, se kterým jsem se setkala i já při psaní unit testů.

Netestovatelný kód se kterým se bylo potřeba vypořádat, obsahoval tyto problémy:

1. **Vytváření závislostí uvnitř metod** - tento přístup neumožňuje závislosti při testování odstínit, tedy je potřeba počítat s logikou těchto závislostí, což dělá psaní testů náročnější.
2. **Závislost na konkrétní implementaci** - tento způsob neumožňuje vytvářeních vlastních instancí, vhodných pro testování (mock objektů).
3. **Statické metody** nejsou přepisovatelné.
4. **Law of Demeter** (česky Zákon Deméter) říká, s kým objekt může interagovat. Říká, že je vhodné komunikovat pouze s takovými objekty, kde ke komunikaci není potřeba prostředníka, tedy jiného objektu.

[1] Moq `https://github.com/moq/moq4`

```
workspace.Workspace.WorkspaceDirectoryInfo.FullName;
```

Listing 5.1: Ukázka porušení Law of Demeter v aplikaci Netfox Detective. Zde se přistupuje k proměnné FullName až přes dva objekty.

5. **Logika v konstruktorech** dělá vytváření objektů náročné. Další problém spočívá, když v konstruktoru jsou vytvářený objekty, které mohou také ve svých kontruktorech vytvářet jiné komponenty.

6. **Mnoho podmínek** znesnadňuje testování tím, že je naročné nastavit mockované objekty tak, aby běh testu prošel úseky kódu, které potřebujeme. Také vytváří kód nepřehledným.

Pro problém č. 1 je řešením dependency injection (viz. 2.5). Jak již bylo řečeno, unit testy by měli být co nejvíce izolované, tudíž by neměli ani komunikovat se souborovým systémem. Abych tohoto docílila, využívám knihovny `System.IO.Abstractions`, ktera umožňuje nahrazení tříd z .NET Framework jako je `DirectoryInfo`, která je `sealed`, tedy nemockovatelná, třídami, které poskytují stejnou funkcionalitu jako třídy z .NET Framework ale navíc jsou mockovatelné.

Problém popsaný v bodě č. 2 byl vyřešen následovně - pro každý objekt, který bylo potřeba při testech namockovat, jsem vytvořila rozhraní, které tento objekt implementoval.

Jak již bylo řečeno v bodě č. 3, statické metody nemohou byt dědičností ani mockovacím frameworkem přepsány. Pro snažší testování jsem tedy tyto metody extrahovala do samotné instance, která je po té vkládána do objektů jako závislost.

Pro zbylé body jsem nenalezla jednoduché řešení pro usnadnění testování. Řešením považuji již samotnou refaktorizaci daných tříd. Lepším návrhem se zbavíme porušování pravidla Law of Demeter, což považuji za nedodržování úrovní abstrakce návrhu. Stejně tak se zbavíme složité konstrukce objektů a spousty podmínek v metodách. Toto vnímám jako upozornění, že daná třída má více zodpovědností, nežli pouze jednu.

5.2 Další typy funkcionálních testů

Smoke testy slouží k rychlému ověření, že hlavní funkce aplikace správně fungují a je tedy připravena k důkladnějšímu testování. Pro tento typ testů se používají i další názvy: Sanity Check Test, Build Acceptance Test, Build Verification Test [21].

End-To-End testy (dále jen E2E) ověřují aplikaci jako celek. Jedná se o testy spouštěné na úrovní uživatelského rozhraní, a interakcí s uživatelským rozhraním simulují uživatele.

Integrační testy ověřují komunikace mezi jednotlivýma komponentami. Tím může být myšlena například interakce mezi dvěma třídami, knihovnami či mezi námi vytvořenou komponentou a souborovým systémem operačního systému, hardwarem, kde naše aplikace bude spouštěna [12]. V případě těchto testů se také může využít mockování k odstínění částí prostředí, které nechceme v integračních testech zahrnout.

Dalším rozdílem oproti unit testům je správa integračních testů. Správa a spouštění integračních testů je zodpovědností QA inženýrů.

Regresní testy mají za úkol odhalit narušení funkcionality již implementovaných a otestovaných vlastností a funkcí, které byly vydány v dřívějších verzích aplikace [18].

Akceptační testy představují sadu testů, kterou vykonává zákazník při přebírání produktu. V týmu využívajícího Scrum metodologii, se akceptační testování vykonává průběžně v rámci tzv. Sprintů, tedy na konci vydání verze k akceptační fázi nemusí docházet [2].

5.3 Další oblasti testování aplikace

Kvalitou aplikace nemusíme chápat pouze její funkcionalitu. V některých aplikacích je potřeba zaručit kvalitu jejího výkonu, použitelnosti, spolehlivosti či podpory na platformách. K ověřování kvality těchto typů slouží zase jiné způsoby testování.

Například u testování spolehlivosti aplikace se lze setkat s tzv. *Stress testy*, kdy ověřujeme stabilitu aplikace, tím, že ji dostaneme do neobvyklé situace (nedostatek procesorového výkonu, omezení paměti, ...) a sledujeme, jak se za těchto podmínek chová. Cílem může být identifikovat problémy v chování aplikace, které se objeví pouze právě v extrémních podmínkách [42], nebo nalézt tzv. zlomový bod aplikace, tedy moment, kdy aplikace není schopna nadále fungovat, jak je požadováno, tedy dojde k rozbití aplikace [43, p. 50].

Při testování použitelnosti může být objektem testování grafické rozhraní. Jedním ze způsobů testování je pozorovat uživatele při interakci s testovanou aplikací. Cílem těchto testů je identifikovat, problémy s použitím grafického designu aplikace [27].

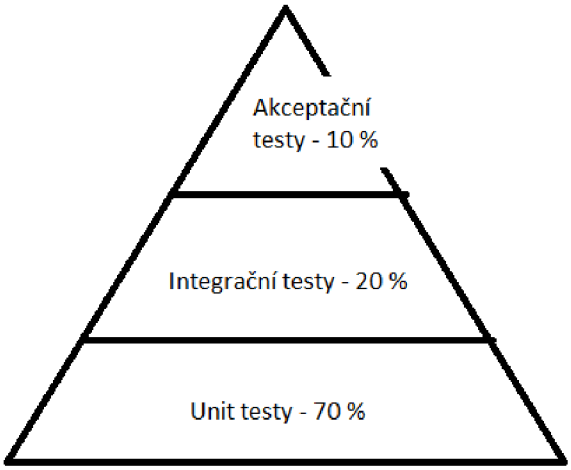
Výkonnost aplikace lze sledovat pomocí *Load testů*, kdy na aplikaci vynaložíme postupně vysokou zátěž ať už ve formě mnoha požadavků na webový server nebo požadavků načtení příliš objemného souboru a sledujeme, jak se za těchto podmínek chová. Například zvyšujeme zátěž a sledujeme aplikaci, dokud využití procesoru nedosáhne 75 %. Tímto získáme informace o chování aplikace, pokud je pod očekávanou zátěží [41].

Subjektem testování podpory u aplikací může být, zda systém bude schopen správně fungovat i v jiných softwarových a hardwarových prostředích, než jaké jsou použita v současném testovacím prostředí.

5.4 Automatizace testů

Jakýkoliv typ testů lze provádět buď manuálně nebo automaticky za pomocí vlastního kódu či využití testovacích frameworků. Pokud vývojový tým by měl prostředky zautomatizovat veškeré potřebné testy a dostatek zdrojů pro jejich častý běh, od unit testů až po akceptační testy, jednalo by se o ideální situaci, kdy veškeré testy by šlo spouštět při každé změně kódu, a tak získat informaci, zda do kódu nebyla zanesena chyba právě přidaným kódem. Psaní některých druhů testů může být implementačně, a tedy časově náročné, nemluvě o investici do dostatečně výkonného a stabilního prostředí, kde by všechny testy běžely a byly schopny být vyhodnoceny v řádech několika minut. Proto je potřeba rozhodnout jaké testy je vhodné psát a v jakém množství. Při tomto rozhodování můžeme využít konceptu *Test pyramid*.

Test pyramid reaguje na složitost implementace a údržbu jednotlivých typů testů. Doporučuje psát mnoho malých a rychlých unit testů, o něco méně integračních testů a nejméně akceptačních testů [53].



Obrázek 5.1: Koncept Test pyramid nám říká, granularitu unit testů, integračních testů a akceptačních testů. Říká, že v ideálním případě by unit testy měli zastupovat 70 % všech napsaných testů, integrační testy 20 % a testy akceptační pouze 10 %. Převzato z článku [24]

Kapitola 6

Rozšiřování kódu

Proces tvorby nové vlastnosti produktu se netýká pouze psaní kódu. Součástí vývojového procesu můžeme vnímat i posouzení napsaného kódu, způsob jeho sdílení či testování. V této kapitole se zabývám způsoby správy kódu v systému pro správu verzí Git, možnostmi posuzování kódu a vysvětluji kontinuální integraci, jako prostředek k zautomatizování testování.

6.1 Správa kódu

Pro správu a sdílení vytvořeného kódu lze používat různé verzovací systémy (správy verzí) jako například SVN[1], Perforce[2], Mercurial[3], Git[4] a další. Každý z nich nabízí jiný přístup ke správě kódu. V projektu Netfox Detective se využívá poslední zmíněný systém, proto se tato kapitola zabývá strategiemi pro správu obsahu v tomto verzovacím systému.

6.1.1 Správa kódu v Git

Při práci s Git se používají k ukládání změn *revize* (tzv. commit). Po vytvoření revize, Git vytvoří snímek současného stavu všech sledovaných souborů. Revize obsahuje informace o změnách provedených v souborech, autora, čas těchto změn a další metadata [30]. Každá revize je identifikovatelná pomocí svého hashe. Nově vytvořená revize odkazuje na revizi předešlou v dané *větvi*. Větev (tzv. branch) představuje způsob organizace změn v systému Git. Jedná se o ukazatel na nějakou revizi. Pokud jsme na nějaké větvi a vytvoříme novou revizi, ukazatel této větvě se přesune na právě námi vytvořenou revizi [33, p. 59]. Pokud se vrátíme o revizi zpět, vytvoříme zde novou větev a vytvoříme další revizi, opět se ukazatel této nové větve přesune na tuto novou revizi. Nyní máme dvě různé revize odkazující na stejnou předcházející revizi. Tímto dochází v systému k tzv. *větvení*. Toto nám umožňuje například odložit práci na nové komponentě a věnovat se tak opravě kritické chyby, bez ztráty provedených změn. Po dokončení opravy kritické chyby se provedené změny sjednotí do příslušné větve, kde jsou změny potřeba, přepne se do větve s rozpracovanou komponentou a lze tak dále bez obtíží pracovat.

[1] SVN `https://subversion.apache.org/`
[2] Perforce `https://www.perforce.com/`
[3] Mercurial `https://www.mercurial-scm.org/`
[4] Git `https://git-scm.com/`

Pokud pracujeme s tímto verzovacím systémem, můžeme se setkat s tzv. *merge konfliktem*. Jedná se o problém, který vznikne pokud chceme sloučit větve, ve kterých proběhly modifikace toho samého souboru [33, p. 70].

Git sám nedefinuje pravidla kdy při vývoji definovat nové větvě či jak je pojmenovávat. Proto vznikly tzv. *strategie větvení*, které zavádějí tyto pravidla a zajišťují tak správu projektu v jednotném stylu. Dodržování těchto pravidel přináší například přehlednější historii revizí.

- **GitFlow** představil v roce 2010 Vincent Driessen na svém blogu [34]. Tato strategie pracuje celkem s pěti typy větví: `Master`, `Release`, `Develop`, `Feature` a `HotFix`.

  `Master` představuje hlavní větev vývoje, nejnovější revizi této větve by měl vždy představovat kód, připravený k vydání. Z `Master` větve vychází větev `Develop`, která reflektuje současný stav vývoje. Jak větev `Master`, tak větev `Develop`, mají neomezenou životnost, tedy existují po celou dobu vývoje.

  Pro vytváření nové komponenty se vytváří nová větev `Feature` z větve `Develop`. Na této nově vytvořené větvi se provádí změny kódu nutné pro implementaci dané komponenty. Po dokončení těchto změn, se větev `Feature` sloučí opět do větve `Develop`.

  Až jsou všechny komponenty pro vydání nové verze hotové, vytvoří se z větve `Develop` větev `Release`. Je důležité, aby všechny komponenty již byly hotovy, a tak se ve větvi `Release` prováděla jen úprava metadat jako například nastavení čísla verze. Při vydání nové verze produktu se větev `Release` sjednotí s větví `Develop`, ale také s větví `Master`. Ve větvi `Master`, se pro poslední revizi představující změny z větve `Release`, vytvoří tag označující danou verzi.

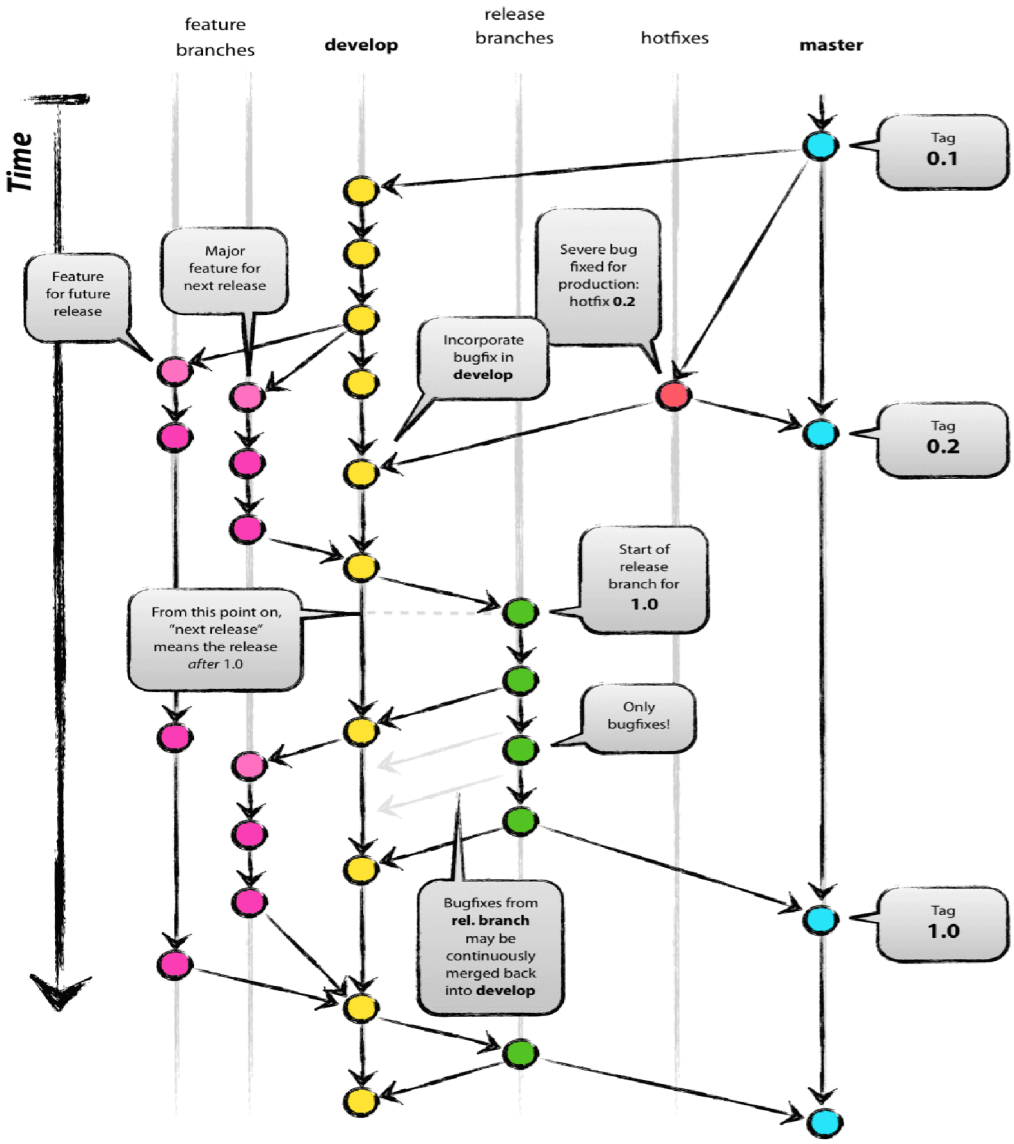
  Poslední typ větve je `Hotfix`, která slouží pro opravu kritických chyb již vydaného kódu. Vytváření tedy probíhá z větve `Master`. `HotFix` se sloučí do větve `Master` i `Develop`. Následně v Master větvi se udělá tag s novou verzí.

  Strategie GitFlow také popisuje formáty názvů větví. Každá větev, kromě `Master` a `Develop`, by měla splňovat následující formát: `typ větve/název větve`, tedy například `feature/novyDialog`.

  V GitFlow se pro sloučení větví používá příkaz `git merge --no-ff`. Použití příkazu `git merge` může mít za následek vytvoření revize navíc, která obsahuje všechny změny, které jsou přidány do slévané větve [7]. Díky této revizi se také uchová informace, že existovala větev, ze které sléváme změny, a to i po jejím odstranění. Parametr `--no-ff` zaručuje, že k vytvoření výše zmíněné revize dojde vždy. Přehled této strategie lze vidět v obrázku č. 6.1.

- **OneFlow** je totožný s GitFlow až na pár drobností. V této strategii větvení se nevyužívají dvě větve, které mají neomezenou životnost, ale pouze jedna, a to `Master`. Tento způsob přináší přehlednější historii uprav [50]. Další změnou je způsob sjednocování větví, zatímco v GitFlow se používá příkaz `git merge` s parametrem `--no-ff`, OneFlow nespecifikuje, jak by se větve měly sjednocovat. Můžeme zde nalézt pouze doporučení, které upřednostňuje používat příkaz `git rebase` místo `git merge`. Při použití příkazu `git rebase` dochazi k přehlednější historii, ovšem i tento přístup přináší určité problémy [16]. Po sjednocení větví se větev, ze které se slévá, smaže.

- **GitHub Flow** je asi nejjednodušší strategií, kterou zde zmiňuji. Pracuje pouze s dvěma typy větví - `Master` a `Feature` [26]. Role mají totožné jako ve strategiích



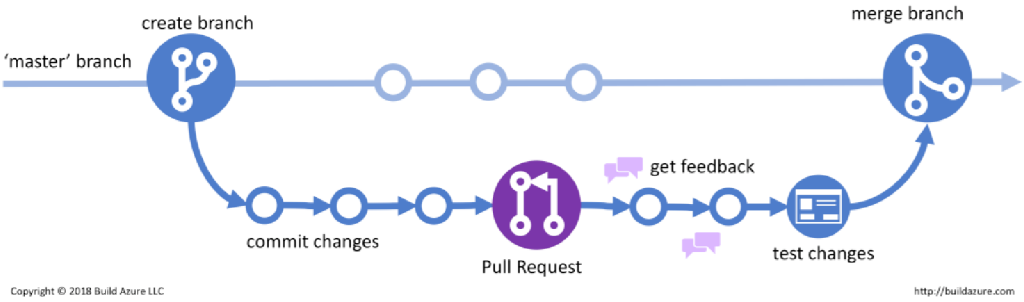
Obrázek 6.1: Přehled GitFlow strategie větvení. Zdroj: [34].

GitFlow nebo OneFlow. GitHub Flow navíc ale zavádí do celého procesu tzv.*Pull request* (viz 6.2). Po schválení pull requestu se větev nasadí do produkčního prostředí, kde proběhne testování. Pokud verifikace je úspěšná, větev `Feature` se sleje do větve `Master`.

- **GitLab Flow** rozšiřuje GitHub Flow o další větve: `Pre-production`, `Production`.

  GitHub Flow předpokládá, že jsme schopni nasadit vytvořený kód do produkce kdykoliv jej slijeme z `Feature` větve do větve `Master`. Tento způsob nemusí však vyhovovat každému týmu.

  Proto GitLab Flow zavádí výše zmíněné environmentální větve. Tyto větvě slouží k nasazování kódu na různá prostředí a k následnému spouštění potřebných dalších procesů nad nimi [8]. Může se jednat například o instalaci kódu na stroj určený k akceptačnímu testování (viz 5.2)



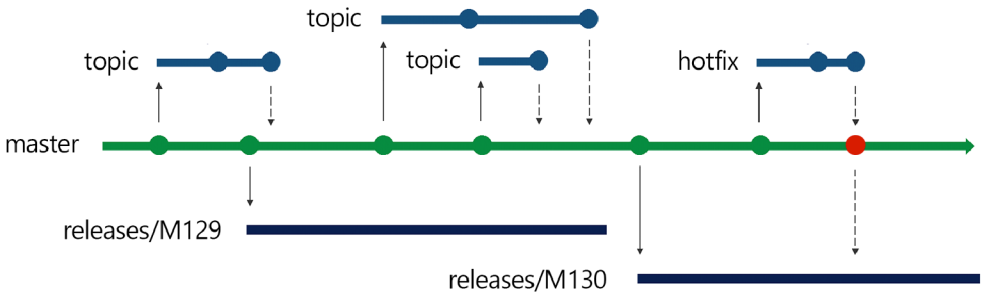
Obrázek 6.2: Přehled GitHub Flow strategie větvení. Převzato z článku: [13].

Slučování pak probíhá například takto: z `Master` větvě, až je implementace nové komponenty hotová, se kód sleje do větve `Pre-production`, kde dochází například k již zmíněnému akceptačnímu testování, po úspěšném akceptačním testování se větev `Pre-production` sleje do větve `Production`, kde se nachází kód připravený k vydání.

- Dalším zajímavým přístupem k problému větvení je **Release Flow**. Tuto strategii využívá VSTS tým Microsoftu. I zde se využívá větve `master`, která obsahuje kód připravený k produkci. V této strategii se k vydání kódu také používají větve typu `release`. Vytvoření větve `release`, tedy k vydání produktu, nedochází kontinuálně jako například u GitHub Flow, ale vždy na konci sprintu [5] [19].

  K vývoji nových vlastností produktu používají větve typu `topic`, účel mají stejný jako větve typu `feature` v předcházejících strategiích, jsou pouze jinak nazvané.

  Větve typu `hotfix` jsou prvně slučovány do větvě `master`, po té do nejnovější větve `release`. Tento přístup zaručuje, že nedojde k regresi kódu, pokud se zapomene sloučit větev `master` s opravou kritické chyby do současné větve `release`, oprava bude součástí další větve `release` na konci současného sprintu.



Obrázek 6.3: Přehled Release Flow strategie větvení. Převzato z článku: [19].

[5] Sprint představuje jednu iteraci v tzv. Scrum agilní metodologii vývoje softwaru.

6.1.2 Nástroje pro správu kódu v Git

Pro práci s verzovacím nástrojem Git lze využít příkazovou řádku či jiné aplikace s grafickým prostředím. Některé z nich rozšiřují sadu příkazů Git o příkazy, které usnadňují aplikaci strategií větvení zmíněných výše.

Do této skupiny aplikací můžeme zařadit rozšíření GitFlow[6] do vývojového prostředí Visual Studio 2017[7]. Toto rozšíření přináší grafické uživatelské rozhraní, ve kterém lze snadno vytvořit například větev `Feature`, provést potřebné modifikace kódu, a danou větev sloučit do větve `Develop` například příkazem `git merge`.

Jednou z klasických aplikací s grafickýcm uživatelským rozhraním pro práci s Git je SourceTree[8]. Kromě podpory GitFlow umožňuje i práci s Git přes příkazovou řádku.

6.2 Code review

Code review, peer code review, (česky *posouzení kódu*) je jednou z technik k zajištění kvality kódu. Jedná se o proces, kdy jeden či více členů vývojového týmu analyzují kód [17]. Tento proces přináší řadu výhod, například pomáhá novým členům v týmu poznat kód aplikace, zabránit vytváření duplicit v kódu, či přinést jednodušší implementaci řešení [48].

Code review můžeme rozdělit na tři typy, podle toho zda nově přidávaný kód je již součástí produkčního kódu na serveru:

- **Post commit review** označujeme posuzování kódu, který je již součástí produkčního kódu uloženého na straně serveru [29]. Subjektem posuzování může být více revizí, celé větve nebo jen vybrané soubory napříč celým kódem.

- Při **pre commit review** posouzení kódu proběhne ještě před tím, než je odeslán na server [29]. Používá se například u projektů, kde kontributor nemá práva přímo přispívat do kódu uloženého na serveru. Jeden ze způsobu praktikování pre commit review je pomocí emailu[9]. Kontributor si stáhne kód ze serveru, provede potřebné změny a vytvoří tzv. Patch, tedy soubor, který popisuje provedené změny. Tento soubor odešle v emailu všem ostatním kontributorům, kteří se starají o kód projektu. Kód se tedy posuzuje pomocí emailové komunikace. Jakmile jsou vývojáří posuzující kód spokojeni, správce projektu tento patch odešle na server, odkud si ostatní mohou stáhnout verzi kódu s patchem.

- **Pull request** umožňuje upozornit ostatní vývojáře na změny, které byly právě odeslány na server v separátní větvi od produkčního kódu. Jakmile je pull request otevřen, přidané změny lze diskutovat s ostatními vývojáři a následně reagovat dalšími revizemi, které odstraňují objevené nedostatky kódu, než přidané změny se stanou součástí produkčního kódu [1]. Subjektem posouzení u tohoto typu code review je tedy celá větev, nikoliv pouze soubor, či revize.

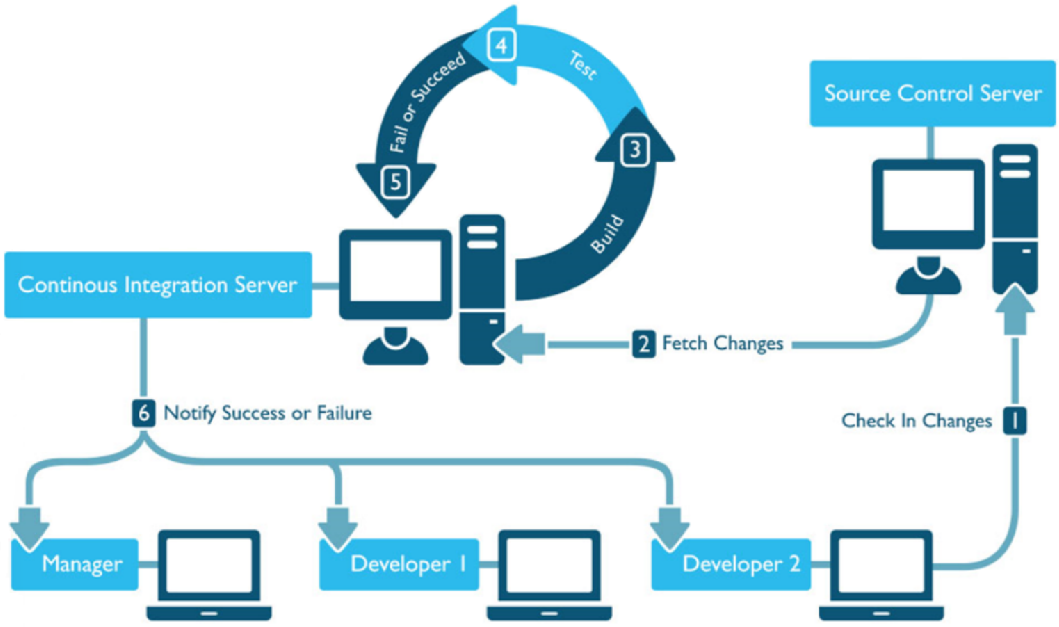
  Průběh vytvoření pull request lze popsat následovně: Kontributor si vytvoří větev, na které provede potřebné modifikace kódu. Až je hotov, odešle tyto změny na server, a pro danou větev otevře pull request. Při vytváření pull requestu se uvádějí osoby, které budou přizvány k posouzení, dále cílová větev, kam se změny přidají, pokud

[6] GitFlow pro Visual Studio 2017 `https://github.com/jakobehn/GitFlow.VS`
[7] Visual Studio 2017 `https://www.visualstudio.com/cs/`
[8] SourceTree `https://www.sourcetreeapp.com/`
[9] Code Reviews vs. Pull Requests - Atlassian Summit 2016 `https://youtu.be/6qKpbWyb6tg`

pull request bude schválen. Také lze přidat popisek provedených změn, či okomentovat jednotlivé řádky kódu a tak více ujasnit účel provedených změn. Pozvaní programátoři do pull requestu mohou psát komentáře k jednotlivým řádkům kódu a mohou daný pull request přijmout nebo odmítnout. Pokud pull request dostane dostatečný počet schválení a splní ostatní požadavky definované správcem projektu, lze větev slít do cílové větve.

6.3 Kontinuální integrace

Kontinuální integrace (anglicky *continuous integration*, dále jen CI) je proces automatizace sestavení a testování kódu pokaždé, když člen týmu potvrdí změny ve správě verzí [38]. Potvrzením si můžeme představit moment, kdy člen týmu odešle změny na server a vytvoří pro větev, kde jsou změny evidovány, pull request. Potvrzením kódu se aktivuje proces, který právě potvrzený kód sestaví, otestuje a ověří, že přidáním vytvořených změn nedochází k regresi produktu.



Obrázek 6.4: Vývojář odešle upravený kód na server. Server, kde běží CI získá modifikovaný kód, sestaví jej a spustí testy. Po té, pokud je tak nastaveno, odešle například vývojáři email o výsledku celého procesu. Převzato z článku [35]

Jak je zobrazeno výše v obrázku č. 6.4, kromě sestavení a otestování kódu může být součástí CI procesu i oznámení o výsledku celého procesu. Toto ovšem není jediný typ rozšíření. Při úspěšném sestavení a otestování kódu, může být dalším krokem nasazení kódu na jiná testovací prostředí (viz GitLab Flow 6.1.1) či publikování NuGet[10] balíčku. Publikování NuGet balíčků můžeme považovat jako součástí procesu Continuous Deployment, což je vlastně rozšíření procesu integrace a testování o další krok, vydání produktu [46].

[10]NuGet `https://www.nuget.org/`

Součástí CI může být i statická analýza a tak informovat uživatele o možném zhoršení kvality kódu. Lze tak detekovat tzv. pachy v kódu, chyby či bezpečnostní rizika. Jedním z nástrojů který nám k tomuto může pomoci je například SonarQube [11].

6.3.1 Nástroje pro kontinuální integraci

Existuje několik nástrojů pro CI, jako například TeamCity[12], Jenkins[13], Circle CI[14]. Rozdíly mezi nimy jsou například v ceně za poskytované služby jako například podpora integrací s verzovacími systémy, modifikovatelnost a rozšiřitelnost jednotlivých kroků procesu či počet poskytovaných agentů. Aplikace Netfox Detective využívá Team Foundation Server[15].

Team Foundation Server (dále jen TFS) je produkt od firmy Microsoft[16], který poskytuje služby pro řízení a vývoj softwarového projektu. Nejedná se tedy pouze o produkt zabývající se pouze CI, ale poskytuje i jiné nástroje jako například konfigurovatelné karty Kanban, interaktivní backlogy, jednoduché plánovací nástroje a předem připravenou podporu pro Scrum. Dále obsahuje i uložiště pro sdílení balíčků, možnost evidace manuálních testů či plánování spouštění testů [23].

Jednou z výhod oproti ostatním zmíněným produktům jsou připravené kroky pro proces CI. Uživatel tedy standardní kroky jako, získání kódu, sestavení a otestování kódu, nemusí psát sám ale využít již těchto kroků, kde je potřeba pouze nastavit adresu serveru kde kód je uložen a přihlašovací údaje k tomuto serveru. Pokud kroky nabízené systémem TFS uživateli nestačí, může si kroky definovat sám například pomocí skriptovacího jazyka PowerShell[17].

CI proces skládající se z několika těchto kroků se v TS nazývá `Build Definition`. Pro každý `Build` dále lze definovat agenty, počítače na kterých celý proces proběhne, či podmínky, které musí být splněny, aby `Build` mohl být považován za úspěšný. Příkladem takové podmínky může být například, že pokrytí kódu unit testami musí být vyšší nebo rovno 80 %.

Pro zrychlení celého procesu CI lze využít možnosti spouštění tohoto procesu již při modifikaci větve, která je subjektem některého z otevřených pull requestů. Toto lze uskutečnit v tzv. `Policies` pro větev, kam chceme slučovat provedené změny. Zde je mimo to možno nastavit další podmínky pro schválení pull requestu, do větve kam sléváme. Jedná se například o minimální počet potřebných schválení od uživatelů k tomu, aby posuzované změny mohli být přidány k produkčnímu kódu, či vynucení způsobu slučování.

6.4 Shrnutí

V této kapitole jsem uvedla několik strategií větvení pro systém správy kódu Git. Z popisu je zřejmé, že jedna strategie větvení se nemusí hodit pro každý typ projektu. Projekty, které vydávají produkt každý den zřejmě nezvolí strategii GitFlow, neboť provádět celý proces vydání (vytvoření větve `Release`, vytvoření metadat, přidání větve `Release` do větve `Develop` a `Master`) může být časově náročný. Na jiných projektech může pracovat pouze jeden vývojářský tým, pak použití strategie GitFlow může přinášet zbytečnou práci navíc, a zvolí se tedy strategie GitHub Flow. Další možností je se těmito strategiemi pouze

[11] SonarQube `https://www.sonarqube.org/`
[12] TeamCity `https://www.jetbrains.com/teamcity/`
[13] Jenkins `https://jenkins.io/`
[14] Circle CI `https://circleci.com/`
[15] Team Foundation Server `https://www.visualstudio.com/cs/tfs/`
[16] Microsoft `https://www.microsoft.com`
[17] PowerShell `https://docs.microsoft.com/en-us/powershell/`

inspirovat a vytvořit vlastní. Například v týmu se vyskytnou požadavky na uchování plné historie úprav, pak příkaz `merge` bude použit pro slučování větví. Členové týmu se dále dohodnou na využívání nástroje SourceTree, které rozděluje větve podle prefixu do složek, tedy navrhne se používání více typu větví: `master`, která bude představovat produkční kód, větev `bug` pro opravu nalezených chyb, větev `feature`, pro implementaci nových vlastností produktu.

Jak jsem dále uvedla, existuje několik přístupů k posuzování kódu. Každý z nich přináší výhody a nevýhody. Post commit nám dovoluje posuzovat kód již začleněný do produkčního kódu. Vývojář tak může kontinuálně přidávat změny a nebýt tak zdržován. Nevýhodou může být začlenění kódu, který řeší problém neefektivní cestou, například spoustou přístupu do databáze, které by se dali omezit tzv. cachováním. Na tento problém testy nemusí přijít, ale zkušený vývojář při kontrole kódu by mohl na tento problém upozornit.

Pre commit review ve formě emailu jak byl v této kapitole č. 6 uvedeno, se může zdát jako zastaralý způsob kontribuce, ale lze se s ním dnes setkat. Například projekty Linux [31] či Git [9] jej stále používají. Tento přístup přináší nevýhodu, pokud je ke kódu spousta připomínek, může se stát,že tato emailová komunikace se stane nepřehledná, neboť komentář ke kousku kódu se vytváří jeho citací.

Podle mě nejpohodlnější způsob revize kódu je pomoci pull requestů. Tento způsob odstraňuje nevýhody post commit review, posuzovaný kód stále není v produkční větvi, i nevýhodu pre commit review ve formě emailu, připomínky k pull requestu se netvoří citací emailové konverzace, ale pomocí komentářů. Pull request je tedy možno vytvářet přes webové rozhraní serveru, který hostuje repositář projektu. V pull requestu se také okamžitě dozvíme , zda dochází ke konfliktu. Dokončení celého procesu kontribuce je také velice jednoduché, a to pomocí tlačítka pro sjednocení změn či jejich zamítnutí.

Uvedený CI proces, který se skládá pouze ze sestavení kódu a spuštění testů je poměrně jednoduchý, a tak se může zdát že vytvoření celého CI procesu je zbytečné. Přeci jen, sestavit kód a spustit nad ním Unit testy dokáže každý vývojář sám na svém počítači. Projekt, který ale také vyžaduje například testování výkonu, či práci se spoustou jiných komponent, může být pro vývojáře zbytečně časově náročné. Toto řeší CI proces, který je například spuštěn odesláním změn kódu na server. Po té vývojář se může věnovat dalším činnostem, mezitím agent ověří místo něj integraci jeho změn do produktu.

Kapitola 7

Rozšiřování kódu pro aplikaci Netfox

Kapitole č. 6 jsem uvedla způsoby správy kódu v systému Git, nástroje pro práci s tímto systémem, možnosti způsobů posouzení kódů a také popsala proces kontinuální integrace. V této kapitole uvedu jak jsou tyto nástroje využívány při vývoji aplikace Netfox a popíši celý proces od získání kódu, jeho modifikace až po zařazení změn do produkčního kódu.

7.1 Správa kódu

Jak již bylo zmíněno v úvodu kapitoly č. 6, při vývoji aplikace Netfox se pro správu kódu využívá systému pro správu verzí Git. Ze jmenovaných strategií větvení se využívá GitFlow. Integrační větví je tedy větev `Develop`, pro implementaci nových vlastností aplikace se využívají větve `Feature`. Jelikož GitFlow nespecifikuje jak postupovat při řešení chyb, v projektu Netfox Detective se chyby opravují buď přímo do větve `Develop`, a nebo v případě složitější opravy se modifikace kódu provádějí do větve `Feature`.

Jelikož se v projektu Netfox Detective využívají i jiné projekty, je důležité nezapomenou při získávání zdrojových kódu na stažení i těchto dalších projektů. Správné stažení zaručí například příkaz `git clone` s přepínačem `--recursive`. Využívané projekty lze nalézt v repozitáři `lib`.

7.2 Kontinuální integrace

Pro kontinuální integraci je vytvořena Build Definition Netfox - Unit Tests, která využívá předem definované agenty, na které stáhne potřebný kód, získá nejnovější NuGet, získá potřebné NuGet balíčky, kód sestaví a spustí nad sestaveným kódem sadu Unit testů. Přehled těchto kroků může být viděn v obrázku č.7.1. Nakonec se spustí vyčištění prostředí. Po skončení celého procesu může výsledek uživatel vidět na tzv. *dashboardu* (viz příloha 7.1)

Použité kroky jsou již definovány systémem TFS, jejich použití je tedy jednoduché. Přídání do `Build Definition` probíhá pouze tzv. *drag and drop*, a poté nastavením informací jako cesta k souboru `.solution` či nastavení platformy a konfigurace.

Ačkoliv to není přímo specifikováno jako samostatný krok, během procesu CI dochází i k statické analýze. Výsledek této analýzy lze vidět níže na obrázku č. 7.2. Jedná se o část zvanou *Issues*, kde jsou zobrazeny záznamy s vykřičníky v oranžových trojúhelnících. U některých záznamů můžeme vidět i kód upozornění, například `CS1584`. Pokud tento

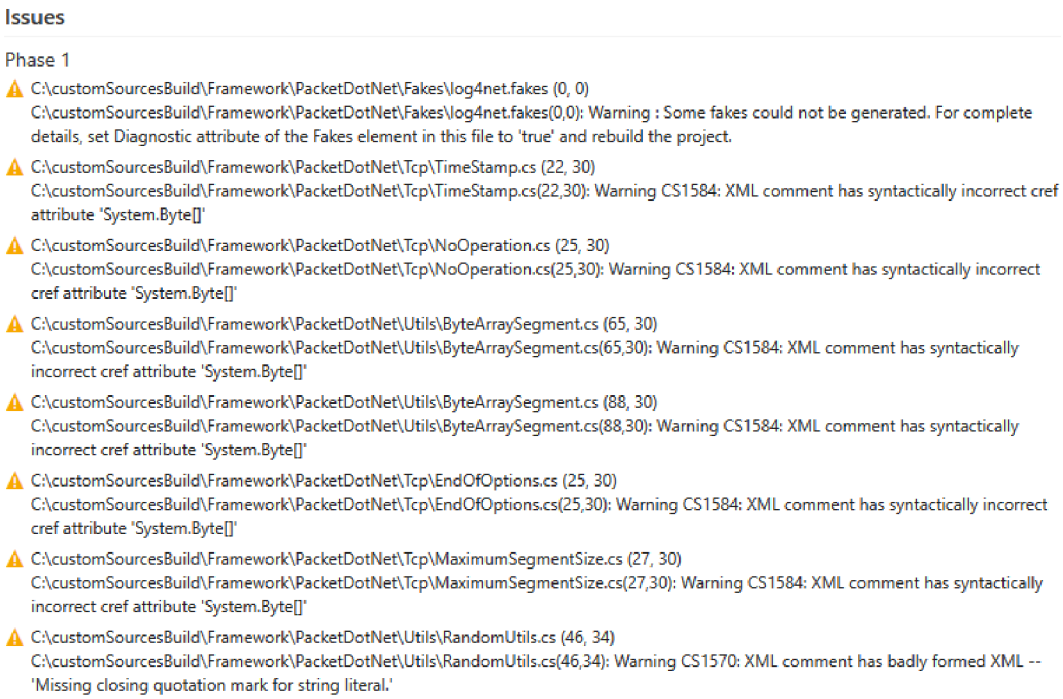
Obrázek 7.1: Agent si stáhne potřebné zdrojové kódy. Dále stáhne aplikaci NuGet verze 4.6.0, která je potřebná pro stažení NuGet balíčků aplikací Netfox Detective. Po získání všech potřebných závislostí, se spustí program MSBuild , který aplikaci sestaví. Jako předposlední krok proběhne spuštění Unit testů. Nakonec dojde k vyčištění prostředí, což představuje smazání stažených souborů v předešlých krocích. Zdroj vlastní.

kód vyhledáme v dokumentaci[1], můžeme zde nalézt více informací o tomto problému. V některých případech i doporučené řešení.

Podmínkou pro úspěšný proces CI je úspěšné provedení všech zmíněných kroků. Spuštění se provede pokaždé pokud jsou detekovány změny ve větvi `develop` nebo větvi typu `feature`. Je zde možnost spouštět celý proces v pravidelných intervalech, této možnosti ale není využito.

Kromě automatického spuštění lze proces CI spouštět i manuálně. V tomto případě je pouze potřeba zadat název větve, pro kterou se proces má spustit, případně specifikovat přímo revizi jejím hash kódem.

[1]Compiler Warning (level 1) CS1570 `https://docs.microsoft.com/cs-cz/dotnet/csharp/misc/cs1570`



Obrázek 7.2: Zdroj vlastní.

7.3 Posouzení kódu

V projektu Netfox Detective se využívá pull requestů. Tvorbu pull requestů umožňuje sám systém TFS. Při odeslaní změn na server, TFS sám nabídne vytvoření pull requestu po navigaci na stráknu Pull Requests jak je ukázáno v obrázku č. 7.3. pokud využijeme nabídnutého tlačítka, TFS za nás předvyplní formulář: nastaví cílovou větev kam změny po schválení slít, předvyplní popisek pull requestu názvy revizí obsažených ve větvi a přidá recenzenty.

Nastavení pull requestu se váže k jednotlivým větvím, do kterých se změny budou slévat. Například pro integrační větev `develop` jsou momentálně nastavení taková, že podmínkou pro schválení pull requestu je potřeba alespoň jedno schválení recenzenta. Recenzenti jsou defaultně nastaveni na všechny členy skupiny Netfox team.

Při posuzování kódu je vhodné vědět, zda navržené modifikace nerozbíjí již implementované komponenty. Toto lze zajistit spuštěním CI procesu, který obsahuje testy ověřující funkcionalitu již naimplementovaných vlastností produktu. Proto je v nastavení pull requestů pro větev `develop` nastavena podmínka pro spouštění CI procesu `Netfox - Unit tests` při aktualizaci větve kterou se chystáme slít do větve `develop`. CI proces se ale nespustí přímo pro větev, kterou se chystáme slit do větve `develop`, ale pro větev `merge`, která představuje již větev `develop` rozšířenou o revizované změny. K automatickému spuštění CI procesu dojde tedy pouze tehdy, pokud nedochází k merge konfliktům.

Obrázek 7.3: Zdroj vlastní.

7.4 Shrnutí

V teto kapitole jsem uvedla jak se používá verzovací systém Git, popsala proces CI a jaké posouzení kódu se provádí v aplikaci Netfox Detective.

Celý proces přispívání kódu do aplikace Netfox Detective bychom mohli popsat následovně. Přispěvovatel si stáhne ze systému TFS zdrojové kódy aplikace. Vytvoří si odpovídající typ větvě, například při přidávání nové vlastnosti produktu `feature/název větvě`. Při vývoji je vhodné si pravidelně spouštět Unit testy, tak se dozvíme, zda naše modifikace nerozbíjí současnou implementaci. Po té co provedeme potřebné modifikace a napíšeme k novým vlastnostem produktu Unit testy, odešleme změny na server, a otevřeme pro naši větev pull request. Je vhodné po otevření pull requestu okomentovat části kódu, které nemusí být na první pohled srozumitelné. Po vytvoření pull requestu se spustí CI proces. Jeho průběh lze vidět v levé horní oblasti pull requestu. Po té, co recenzenti jsou spokojeni a proces CI proběhl úspěšně, je možno provedené změny sloučit do integrační větve.

Jak bylo zmíněno v podkapitole č. 6.3, systém TFS nabízí interaktivní backlogy, které obsahují tzv. itemy. Tohoto lze využít při vytváření větví a zahrnout tak identifikační číslo itemu do názvu větve, například `feature/123 - Create dialog window for Workspace`. Tímto získáme přehlednější historii revizí. Dále pokud vytvoříme větev v dialogu pro vybraný item, tak při vytvoření pull requestu se na tento item objeví reference. Pak tedy daný item není nutno dohledávat, postačí jedno kliknutí.

V podkapitole č. 7.1 jsem uvedla, že v současnosti při řešení chyby v produktu, se buď opravy provádí přímo do větve `develop`, nebo při větších modifikacích kódu se vytváří větev `feature`. Toto nepovažuji za vhodné z několika důvodu:

- Větev typu `feature` neslouží k opravě chyb, ale k přidávání nových vlastností produktu.
- Přidáváním přímo do větvě `develop` obcházíme celý proces popsaný výše, nedojde tedy k žádnému posouzení kódu. Toto skýtá riziko zanesení chyby, která by se dala odhalit při pull requestu nebo při proběhnutí určitého Unit testu. Tato chyba se pak propaguje k jiným kontributorům a může je tak zpomalit ve vývoji.

Řešením pro problém výše je neobcházet posouzení kódu. Navrhuji tedy, pro nalezenou chybu vytvořit item v systému TFS, kde se tato chyba popíše. Popis by měl obsahovat aktuální chování, očekávané chování a pokud je to možné, vložit výpis zásobníku či jiné další relevantní informace. Při řešení chyby se v dialogu itemu vytvoří větev, ve formátu `bug/čísloItemu-názevItemu`. Po odeslání změn se otevře pull request. Postup je vlastně totožný jako u vytváření nové vlastnosti produktu, s tím rozdílem, že typ větve není `feature` ale `bug`.

Kapitola 8

Závěr

Cílem této práce bylo seznámit se s nástrojem Netfox Detective a provést jeho analýzu zaměřenou na správné použití návrhových vzorů. Následně vytvořit sadu unit testů ověřující aktuální chování nástroje, identifikovat nesprávně použité návrhové vzory a provést jejich refaktorizaci. V poslední řadě bylo potřeba zajistit automatické testování aplikace.

Při popisu návrhových vzorů jsem uvedla konkrétní problémy jejich aplikace v systému a v některých případech i nabídla postup řešení těchto problémů. Také jsem se věnovala vysvětlení tzv. SOLID principů, které usnadňují testovatelnost kódu. V další kapitole jsem popsala implementaci zmíněných návrhových vzorů v aplikaci Netfox Detective a uvedla jejich případné nedostatky. V následující kapitole jsem se věnovala popisu mého řešení refaktorizace pro návrhové vzory Mediator a MVVM. Uvedla jsem, co provedené modifikace kódu přináší oproti původní implementaci. Dále jsem představila pojem unit testů, jejich správné vlastnosti, které by měly mít i jejich strukturu. Popsala jsem problémy, se kterými jsem se při psaní testů setkala, a jak jsem přistupovala k jejich řešení. Především se jednalo o problémy, které vycházeli z nedodržení již zmíněných principů. V posledních dvou kapitolách jsem se věnovala správě kódu, posuzování kódu a procesu integrace změn do produktu. Představila jsem automatický proces testování, který probíhá při odeslání změn do repositáře.

K refaktorizaci jsem si vybrala pouze část kódu aplikace Netfox Detective, tudíž je zde spoustu dalších částí, které by mohly být napsané lépe tak, aby kód byl čitelnější a neporušoval principy SOLID, jejichž dodržování přináší další výhody jako testovatelnost či rozšiřitelnost kódu. Při refaktorizaci není potřeba zůstat pouze u tříd, ale můžeme se přenést na úroveň jednotlivých knihoven. Jelikož o automatickém sestavení a testovaní jsem se zmínila krátce, lze toto téma dále zpracovat. Například refaktorizaci modulů považuji jako velmi důležité téma u aplikací, kde CI proces trvá příliš dlouho. Pak se můžeme ptát, zda produkt nelze rozdělit na menší části, a tyto části sestavovat a testovat paralelně, čímž bychom celý proces urychlili.

Literatura

[1] *About pull requests*. [Online; navštíveno 17.03.2018].
URL https://help.github.com/articles/about-pull-requests/

[2] *Acceptance Testing*. [Online; navštíveno 23.02.2018].
URL https://www.scrum.nl/blog/user-acceptance-test-scrum/

[3] *Code refactoring*. [Online; navštíveno 22.01.2018].
URL https://en.wikipedia.org/wiki/Code_refactoring

[4] *CommandPattern*. [Online; navštíveno 14.05.2018].
URL https://is.mendelu.cz/eknihovna/opory/index.pl?cast=32230

[5] *Data Binding (WPF)*. [Online; navštíveno 23.01.2018].
URL https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-wpf

[6] *Double-checked lock is not thread-safe*. [Online; navštíveno 8.05.2018].
URL https://help.semmle.com/wiki/display/CSHARP/Double-checked+lock+is+not+thread-safe

[7] *Git - git-merge Documentation*. [Online; navštíveno 17.03.2018].
URL https://git-scm.com/docs/git-merge

[8] *GitLab Flow*. [Online; navštíveno 17.03.2018].
URL https://about.gitlab.com/2014/09/29/gitlab-flow/

[9] *Git: Submitting patches*. [Online; navštíveno 1.05.2018].
URL https://github.com/git/git/blob/master/Documentation/SubmittingPatches

[10] *How do I use the singleton pattern in C#?* [Online; navštíveno 8.05.2018].
URL https://www.oreilly.com/learning/how-do-i-use-the-singleton-pattern-in-c

[11] *Implementing Singleton in C#*. [Online; navštíveno 21.01.2018].
URL https://msdn.microsoft.com/en-us/library/ff650316.aspx

[12] *Integračni testování*. [Online; navštíveno 23.02.2018].
URL http://testovanisoftwaru.cz/tag/integracni-testovani/

[13] *Introduction to Git Version Control Workflow*. [Online; navštíveno 6.05.2018].
URL https://buildazure.com/2018/02/21/introduction-to-git-version-control-workflow/

[14] *Mediator Design Pattern*. [Online; navštíveno 23.01.2018].
URL https://sourcemaking.com/design_patterns/mediator

[15] *Mediator pattern*. [Online; navštíveno 7.05.2018].
URL https://en.wikipedia.org/wiki/Mediator_pattern

[16] *Merging vs. Rebasing*. [Online; navštíveno 17.03.2018].
URL https://www.atlassian.com/git/tutorials/merging-vs-rebasing

[17] *Peer Review*. [Online; navštíveno 17.03.2018].
URL https://en.wikipedia.org/wiki/Software_peer_review

[18] *Regression Testing*. [Online; navštíveno 23.02.2018].
URL https://en.wikipedia.org/wiki/Regression_testing

[19] *Release Flow: How We Do Branching on the VSTS Team*. [Online; navštíveno 6.05.2018].
URL https://blogs.msdn.microsoft.com/devops/2018/04/19/release-flow-how-we-do-branching-on-the-vsts-team/

[20] *Singleton pattern*. [Online; navštíveno 7.05.2018].
URL https://commons.wikimedia.org/wiki/File:Singleton_pattern_uml.png

[21] *Smoke Testing*. [Online; navštíveno 23.02.2018].
URL http://softwaretestingfundamentals.com/smoke-testing/

[22] *Swis Army Knife*. [Online; navštíveno 10.02.2018].
URL https://sourcemaking.com/antipatterns/swiss-army-knife

[23] *Team Foundation server*. [Online; navštíveno 29.04.2018].
URL https://www.visualstudio.com/cs/tfs/

[24] *Test Automation Basics – Levels, Pyramids & Quadrants*. [Online; navštíveno 21.02.2018].
URL http://www.duncannisbet.co.uk/test-automation-basics-levels-pyramids-quadrants

[25] *The MVVM Pattern*. [Online; navštíveno 23.01.2018].
URL https://msdn.microsoft.com/en-us/library/hh848246.aspx

[26] *Understanding the GitHub Flow*. [Online; navštíveno 17.03.2018].
URL https://guides.github.com/introduction/flow/

[27] *Usability Testing*. [Online; navštíveno 22.02.2018].
URL
https://www.interaction-design.org/literature/topics/usability-testing

[28] *Vkládání závislostí*. [Online; navštíveno 15.01.2018].
URL https:
//cs.wikipedia.org/wiki/Vkl%C3%A1d%C3%A1n%C3%AD_z%C3%A1vislost%C3%AD

[29] *What is Post-Commit and Pre-Commit Review?* [Online; navštíveno 17.03.2018].
URL https://www.devart.com/review-assistant/learnmore/pre-commit-vs-post-commit.html

[30] *Úvod - Základy systému Git.* [Online; navštíveno 17.03.2018].
URL
https://git-scm.com/book/cs/v1/%C3%9Avod-Z%C3%A1klady-syst%C3%A9mu-Git

[31] *First Kernel patch.* 2018, [Online; navštíveno 1.05.2018].
URL https://kernelnewbies.org/FirstKernelPatch

[32] Brown, W. J.; Malveau, R. C.; III, H. W. M.; aj.: *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis.* Robert Ipsen, 1998, ISBN 0-471-19713-0.

[33] Chacon, S.: *Pro Git.* CZ.NIC, z. s. p. o., 2009, ISBN 978-80-904248-1-4.
URL https://knihy.nic.cz/files/nic/edice/scott_chacon_pro_git.pdf

[34] Driessen, V.: *A successful Git branching model.* Jan 2010, [Online; navštíveno 27.02.2018].
URL http://nvie.com/posts/a-successful-git-branching-model/

[35] Foo, D.: *Continuous integration and continuous delivery with nuget.* [Online; navštíveno 29.04.2018].
URL http://danielcoding.net/continuous-integration-and-continuous-delivery-with-nuget/

[36] Fowler, M.: *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN 0-201-48567-2.

[37] Gamma, E.; Heim, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994, ISBN 978-0201633610.

[38] Guckenheimer, S.: *Co je kontinuální integrace?* [Online; navštíveno 29.04.2018].
URL
https://www.visualstudio.com/cs/learn/what-is-continuous-integration/

[39] Hordějčuk, V.: *Návrhové vzory.* [Online; navštíveno 5.12.2017].
URL http://voho.eu/wiki/navrhovy-vzor/

[40] Martin, R. C.: *Agile principles, patterns, and practices in C#.* Prentice Hall, Jul 2006, ISBN 978-0-13-185725-4.

[41] Meier, J.; Farre, C.; Bansode, P.; aj.: *Load Testing Web Applications.* Sep 2007, [Online; navštíveno 21.02.2018].
URL https://msdn.microsoft.com/en-us/library/bb924372.aspx

[42] Meier, J.; Farre, C.; Bansode, P.; aj.: *Stress Testing Web Applications.* Sep 2007, [Online; navštíveno 21.02.2018].
URL https://msdn.microsoft.com/en-us/library/bb924374.aspx

[43] Molyneaux, I.: *The Art of Application Performance Testing.* O'Reilly Media, Inc.„ Dec 2014, ISBN 978-1-491-90054-3.

[44] Osherove, R.: *The Art of Unit Testing.* Manning Publications Co., 2014, ISBN 978-1-61-729089-3.

[45] Ottinger, T.; Langr, J.: *Unit Tests Are FIRST*. Leden 2012, [Online; navštíveno 25.01.2018].
URL https://pragprog.com/magazines/2012-01/unit-tests-are-first

[46] Pittet, S.: *Continuous integration vs. continuous delivery vs. continuous deployment*. [Online; navštíveno 29.04.2018].
URL https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd

[47] Radford, M.: Singleton - the anit-pattern! *Overload*, ročník 57, Říjen 2003.
URL https://accu.org/index.php/journals/337

[48] Reidinger, J.: *Code reviews v praxi*. Sep 2014, [Online; navštíveno 10.03.2018].
URL https://www.zdrojak.cz/clanky/code-reviews-praxi/

[49] Rossi, J.: *Inversion of Control*. Červen 2015, [Online; navštíveno 15.01.2018].
URL https://github.com/castleproject/Windsor/blob/master/docs/ioc.md

[50] Ruka, A.: *OneFlow - a Git branching model and workflow*. [Online; navštíveno 17.03.2018].
URL
http://endoflineblog.com/oneflow-a-git-branching-model-and-workflow

[51] Seemann, M.: *Service Locator is an Anti-Pattern*. Únor 2010, [Online; navštíveno 24.01.2018].
URL http://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

[52] Seemann, M.: *Dependency Injection in .NET*. Manning Publications Co., 2012, ISBN 978-0-19-3518250-4.

[53] Vocke, H.: *The Practical Test Pyramid*. [Online; navštíveno 21.02.2018].
URL https://martinfowler.com/articles/practical-test-pyramid.html

Přílohy

Seznam příloh

Příloha A

Relay command

```
public class RelayCommand : ICommand
{
    private readonly Action executeAction;
    private readonly Func<ool> canExecuteAction;

    public RelayCommand(Action executeAction)
    :this(p => executeAction(), p => true) {}

    public RelayCommand(Action executeAction, Func<bool>
        canExecuteAction)
    {
        this.executeAction = executeAction
            ?? throw new
                ArgumentNullException(nameof(executeAction));
        this.canExecuteAction = canExecuteAction;
    }

    public bool CanExecute(object parameter)
    {
        return canExecuteAction.Invoke(parameter) ?? true;
    }

    public void Execute(object parameter)
    {
        executeAction.Invoke(parameter);
    }

    public event EventHandler CanExecuteChanged;

}
```

Listing A.1: Jednoduchá implementace `RelayCommand`. Zdroj vlastní.

Příloha B

Workspace

```
[KnownType(typeof(DirectoryInfoWrapper))]
    [DataContract(Name = "Workspace", Namespace =
       "Netfox.Detective.Models.WorkspacesAndSessions")]
    public class Workspace
    {
        public Workspace(string workspaceName, string
           workspacesStoragePath, string connectionString)
        {
            this.Name = workspaceName;
            this.ConnectionString = connectionString;
            this.Guid = Guid.NewGuid();
            this.Created = DateTime.Now;
            this.LastRecentlyUsed = this.Created;
        }
        public Workspace() { }

        [DataMember]
        public DirectoryInfoBase WorkspaceDirectoryInfo { get; set;
           }

        [DataMember]
        public string Name { get; private set; }

        [DataMember]
        public string ConnectionString { get; set; }

        [DataMember]
        public Guid Guid { get; private set; }

        [DataMember]
        public DateTime Created { get; private set; }

        [DataMember]
        public DateTime LastRecentlyUsed { get; set; }

        [DataMember]
        public List<String> InvestigationsFilePaths
        {
```

```
            get { return this._investigationFilePaths ??
                (this._investigationFilePaths = new List<string>());
                }
            set { this._investigationFilePaths = value; }
        }
        private List<String> _investigationFilePaths;

        public override string ToString() => this.Name;
    }
```

Listing B.1: Třída `Workspace` po refaktorizaci.

Příloha C

Aplikace Netfox

V přiloženém DVD lze nalézt obsah repositáře Netfox Detective. Pro funkčnost je potřeba stažení submodulů příkazem `git submodule update --recursive`.