VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MIGRACE A REFAKTORIZACE NETFOX DETECTIVE NA .NET 5

MIGRATION AND REFACTORIZATION OF NETFOX DETECTIVE FOR .NET 5

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. ŠIMON POKORNÝ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN PLUSKAL**
SUPERVISOR

BRNO 2021

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Pokorný Šimon, Bc.**
Program: Informační technologie Obor: Informační systémy a databáze
Název: **Migrace a refaktorizace Netfox Detective na .NET 5**
Migration and Refactorization of Netfox Detective for .NET 5
Kategorie: Softwarové inženýrství
Zadání:

1. Seznamte se s nástrojem Netfox Detective, který je implementovaný s použitím .NET Framework. Analyzujte a zdokumentujte jeho architekturu. Nastudujte refaktorizační techniky, zvolte vhodné z nich a odůvodněte, proč bylo bezpečné je použít.
2. Zhodnoťte existující testy tohoto nástroje a rozšiřte je, aby bylo možné provést bezpečně bod 3. Vytvořte vlastní dataset obsahující vhodná data a konzultujte s vedoucím.
3. Proveďte migraci tohoto nástroje na platformu .NET 5. Hledejte části kódu, které jsou těžkopádné, znesnadňují testování a tyto refaktorujte. Zdokumentujte postup, který jste pro migraci a refaktorizaci použili.
4. Konkrétní řešení refaktorizace a nalezených chyb konzultujte s vedoucím.
5. Zhodnoťte úspěšnost refaktorizace nástroje s využitím jak existujících testů tak vámi implementovaných s využitím datasetu z bodu 2.

Literatura:

1. Osherove, R., 2009. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co..
2. Fowler, M., 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
3. Martin, R.C., 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Při obhajobě semestrální části projektu je požadováno:

- Hotové body 1, 2 a rozpracované zbývající.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 26. října 2020

Abstrakt

V internetu probíhá každou sekundu obrovské množství pokusů o útoky na různé subjekty. Aby bylo možné zpětně jednoduše analyzovat síťovou komunikaci, je potřeba kvalitních, rychlých a aktualizovaných nástrojů. Netfox Detective je jedna z aplikací sloužící k forenzní analýze síťové komunikace. Cílem práce je migrace tohoto produktu na moderní platformu .NET 5, včetně refaktorizace s ohledem na uživatelskou zkušenost a správné využití návrhových vzorů. Práce se zabývá nejen samotnou migrací, ale uvádí i sadu častých programátorských faulů a způsoby jejich eliminace. Postupně se v kapitolách nachází záznamy jednotlivých rozhodnutí, které mohou pomoci ostatním vývojářům při řešení dalších nástrah. V závěru se práce zabývá analýzou a tvorbou testů a správním využitím nástrojů pro CI/CD. Výstupem je pak nejen kompletní migrovaný projekt, ale také připravené prostředí v systému GitLab.

Abstract

Every second, there are many attempts to attack various entities on the Internet. This is why high-quality, fast, and up-to-date tools are needed to easily analyze network traffic. Netfox Detective is one of such tools. Specifically, it is used for forensic analysis of network communication. The aim of this work is to migrate Netfox Detective to the newest version of .NET platform (.NET 5), including refactoring with respect to user experience and correct use of software design patterns. This thesis deals not only with the migration itself, but is listing common mistakes programmers make along with possible solutions to these mistakes. The chapters contain a detailed decision log that can help guide other developers to better solutions. Furthermore, the work deals with analysis and creation of unit tests and with correct use of tools for CI/CD. Fully migrated project is not the only output of this thesis. A development environment for the project has been prepared in GitLab and it is ready to be used.

Klíčová slova

Refaktorizace, Netfox Detective, migrace kódu, síťová analýza, čistý kód, testování

Keywords

Refactoring, Netfox Detective, code migration, network analysis, clean code, testing

Citace

POKORNÝ, Šimon. *Migrace a refaktorizace Netfox Detective na .NET 5.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

Migrace a refaktorizace Netfox Detective na .NET 5

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana inženýra Jana Pluskala. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Šimon Pokorný
17. května 2021

Poděkování

Rád bych touto formou poděkoval mému vedoucímu práce, inženýrovi Janu Pluskalovi, za projevenou důvěru a volnou ruku v rámci zpracování této diplomové práce. Veškerá naše komunikace byla vždy ve velmi příjemném duchu a prostředí, ať už jsme řešili jakoukoliv situaci. Jsem velmi vděčný za to, že jsem se mohl zpracovávat zajímavý projekt, a to plně ve vlastní režii. V neposlední řadě si také velmi vážím všech odborných rad a podnětných připomínek, které mi pomáhaly posouvat jak praktickou část, tak tyto řádky správným směrem.

Obrovské poděkování patří také mé mamince a tatíkovi, kteří se o mě za celý můj život starali a vždy stavěli náš společný vztah na důvěře, ve které jsem se nikdy nezklamal. Nejen v rámci dětského života, ale i toho studentského jsem měl jejich plnou podporu, kde bylo třeba, i když jsem to někdy sám ani nevěděl.

Moc také děkuji své přítelkyni, která mě po celou dobu této práce, i před ní, plnila láskou, příjemným prostředím v bytě a zároveň spoustou jídla, které bylo vždy výborné. Zároveň vždy pracovala nad své síly tak, aby mi pomohla získat co nejvíce času, který jsem pak mohl věnovat škole a především této práci.

V žádném případě nesmím zapomenou také na mého bratra Matyáše, který si i přes všechno své vlastní vytížení na vysoké škole na mě našel dostatek času, při řešení některých zapeklitostí v rámci zpracování této práce a nikdy mě neodmítl vysvětlit cokoliv, čemu jsem nerozuměl.

Obsah

1 Úvod **2**

2 Čistý kód **4**
2.1 Základní pravidla čistého kódu . . . 4
2.2 Měření kvality kódu . . . 6
2.3 Udržování kvalitního kódu . . . 11

3 Refaktorizace a návrhové vzory **13**
3.1 Cíle refaktorizace . . . 13
3.2 Bezpečná úprava kódu . . . 14
3.3 Návrhové vzory . . . 15

4 Netfox Detective **19**
4.1 Důležité technologie . . . 19
4.2 Struktura aplikace . . . 21
4.3 Testování . . . 25
4.4 Kritické části projektu . . . 29
4.5 Příprava pro implementační část . . . 33

5 Příprava prostředí a migrace **37**
5.1 Agilní metodika vývoje . . . 37
5.2 Podpůrné nástroje . . . 38
5.3 Příprava migrace . . . 43
5.4 Migrace projektu . . . 45

6 Testování **59**
6.1 Testy kódu . . . 59
6.2 Testování v Netfox . . . 63
6.3 Popis manuálních testů . . . 68
6.4 Zhodnocení refaktorizace s využitím testů . . . 70

7 Závěr **72**
7.1 Možná další rozšíření . . . 73

Literatura **74**

A Obsah přiloženého paměťového média **79**

B Testovací sady pro manuální testy **80**

Kapitola 1

Úvod

Internetem každý den proudí obrovské množství informací. I přes to, že se může zdát, že jsou pro nás některé informace naprosto nepodstatné, můžeme být na velkém omylu. Jedním z příkladů, který potvrzuje, že udržování veškerého toku v rámci sítě může být užitečné, je forenzní analýza síťového provozu, která je nezbytnou součástí vyšetřování, při napadení sítě. Abychom mohli rychle analyzovat síťový provoz, je nutné vlastnit kvalitní a aktualizované nástroje. Právě slovo *aktualizované* je zde velmi důležité, neboť informační technologie se stále vyvíjejí a je třeba, abychom udržovali tyto nástroje aktuální a v moderní podobě.

Udržování nástrojů v moderní podobě a jejich aktualizace může zahrnovat několik různých věcí. Mezi první, které jsou vidět na první pohled, patří například moderní grafické uživatelské rozhraní, zvyšování rychlosti, nebo třeba nové funkce. Jednou a neméně důležitou disciplínou je také refaktorování, tedy změna kódu programu takovým způsobem, že není ovlivněné chování. Ve chvíli, kdy je potřeba kód lépe sjednotit a v ideálním případě i převést na novější platformu, je také nutná migrace celého zdrojového kódu. Vysvětlení pojmů refaktorizace a migrace, seznámení se s důvody proč jsou důležité a jejich aplikace jsou popsány v kapitole 3.

Abychom mohli s kódem dobře pracovat, je potřeba také znát doporučení, které se při vývoji kódu používají. Většina z nich vede k dobré udržitelnosti, jednoduché orientaci a bezproblémovému testování. Všechny tyto vypsané principy jsou popsány v kapitole 2.

Jedním z podpůrných nástrojů forenzní analýzy je Netfox Detecitve, který je vyvíjen v rámci výzkumné skupiny NES@FIT na Fakultě informačních technologií Vysokého učení technického v Brně. V kapitole číslo 4 je popsáno, k čemu tato aplikace slouží, a zároveň proč je potřeba kód aplikace migrovat. V další části této kapitoly je popsána struktura kódu, závislosti jednotlivých částí a nakonec návrh jednotlivých opatření pro zvýšení čistoty a přehlednosti v rámci kódu.

Implementační část, která je rozepsaná v kapitole 5, se zabývá metodikami vývoje, podpůrnými nástroji a především intenzivně probírá všechny zajímavé změny, přístupy a řešení, které bylo v rámci této migrace potřeba projít. Tato kapitola se také podrobněji zabývá přípravou repositářů Git a nastavením automatizovaných procesu pro CI/CD. V závěru kapitoly je pak shrnuto, co všechno se ve výsledku podařilo v této diplomové práci udělat a jaký je vlastně výsledek.

Testování je nejen nedílnou součástí každého nového kusu kódu, ale zároveň velmi důležitým prvkem ve chvíli, kdy je potřeba provést v rámci programu nějaké změny. Proto je v kapitole 6 popsáno, jakým způsobem byly provedené změny testovány a zároveň rozebráno, jaké všechny metodiky v rámci testů byly použity. Zajímavou částí jsou také zásahy

do testovacího prostředí, které zavádí některé nové principy a zjednodušují tak v různých ohledech další vývoj v rámci tohoto nástroje.

Závěr této práce je pak v kapitole 7. Zde se nachází shrnutí toho, jak se všechny práce podařily, osobní názor na celý tento produkt a také celkové zhodnocení této diplomové práce. Na konci samotného závěru jsou ještě doplněny některé nápady, které by mohly být implementovány v rámci dalšího vývoje.

Kapitola 2

Čistý kód

Dříve, než bude rozebrána základní myšlenka a toho, co čistý kód je, proč a jak jej udržovat a jaké jsou možné způsoby, si dovolím krátkou citaci Michaela Featherse z knihy *Čistý kód: návrhové vzory, refaktorování, testování a další techniky agilního programování* [38]:

> Čistý kód vypadá vždy tak, jako by ho psal někdo pečlivý. Neexistuje nic samozřejmého, co můžete udělat pro jeho zlepšení. O všech těchto záležitostech již autor kódu přemýšlel, a jestliže se pokusíte uvažovat o nějakých vylepšeních, dovede vás to zpět tam, kde právě jste.

Z citace vyplývá, že čistý kód je právě takový kód, který je nejlepší. V kontextu významu *Čistý kód*, anglicky uváděný jako „Clean code“, není myšlen kód nejrychlejší, případně nejoptimálnější vzhledem k paměťové náročnosti, nýbrž kód takový, že se v něm jiný vývojář bez problému orientuje a chápe jeho význam již při prvním pohledu.

2.1 Základní pravidla čistého kódu

Dle definice výše vyplývá, že k čistotě kódu přispívá nejvíce správná organizace a struktura. Možnému čtenáři můžeme zjednodušit práci tak, že budeme psát kód, který podléhá následující skupině vlastností [24].

- **Jednoduchost**, která odkazuje už na Leonarda da Vinciho, který řekl, že v jednoduchosti je krása. Jednoduchý kód je možné rychle pochopit, tedy i upravit bez zanesení chyb ve spojení s nepřímočarou návazností na jinou část programu.
- **Přímočarost** ve zdrojovém kódu rozděluje dlouhé části na kratší tak, aby každá posloupnost operací dohromady prováděla jedinou jasně popsanou věc.
- **Absence opakování** kód nejen zkracuje, ale také zajišťuje, že není nutné opravovat případnou vzniklou chybu na více místech v rámci celého projektu.
- **Dobrá testovatelnost** je výsledkem přímočarosti a absence opakování. Umožňuje programátorovi jednoduše otestovat jednotlivé části kódu samostatně.

K zajištění všech bodů výše popsaných nám slouží nejen následující pravidla, ale také návrhové vzory popsané v kapitole 3.3. Níže popsané body jsou obecně doporučená pravidla, pro psaní a udržování čistého kódu. Konkrétní implementace těchto pravidel se může v rámci projektů lišit. Programátoři by měli vždy nastavená pravidla respektovat a dodržovat.

Formátování kódu

Způsob, jakým je kód formátován je úplně první věc, které si vývojář všimne již při prvním pohledu. Jedná se především o odsazení jednotlivých řádků, nelogicky umístěné závorky ohraničující blok kódu, více příkazů v rámci jednoho řádku, nebo třeba struktura, která je aplikována při tvorbě tříd [34]. Zpravidla se využívá nový řádek pro každý příkaz, hlubší odsazení pro každý vnořený blok kódu a atributy třídy píšeme zejména nad všechny funkce a metody.

Pojmenování prvků

Jména proměnných, metod, funkcí, tříd a rozhraní si mohou, mimo klíčová slova, vždy vývojáři určit podle sebe. Ve většině případů je nejlepší volit jména takovým způsobem, aby to bylo pro jiného vývojáře samovysvětlující. Do pojmenování nepatří pouze samotné slovo, nebo slovní spojení, ale také forma. Mezi vývojáři je rozlišováno několik základních forem. *Pascal case*, která říká, že všechna počáteční písmenka budou velká, například `void DoSomeStuff()`, *Camel case*, který říká, že budou všechna počáteční písmenka slov velká až na to první, například `int someVariableName` [15, 43], nebo také *Snake case*, který odděluje jednotlivá slova podtržítkem, například `int give_me_random_number()`. Nejdůležitější je především to, že ať už se vývojáři domluví na jakýchkoliv konvencích, je třeba je vždy dodržovat.

Funkce, metody

Nejmenší částí kódu a zároveň nejmenší zodpovědnost mají jednotlivé funkce a metody. Nejlepší praktikou je psát metody krátké a jednoznačné. Rozdělování složitějšího kódu na menší části, které se vzájemně volají napomáhá také snazšímu testování. Jednoduchým ukazatelem toho, zda není metoda příliš složitá je pojmenování. Pokud nejsme schopni vymyslet výstižný a jednoznačný název, nejspíš je možné rozdělit metodu či funkci na více částí. Druhým jednoznačným ukazatelem je počet parametrů. Pokud jich funkce, nebo metoda, přijímá hodně a není možné je předávat jako jeden společný objekt, z největší pravděpodobnosti není metoda atomická, tedy neplní pouze jednu funkci [43].

Třídy

Třídy jsou skvělou možností, jak zapouzdřit a oddělit jednotlivé logické celky v rámci projektu. Jejím cílem je ukrytí implementace jednotlivých funkcí a metod a zároveň přidání možnosti implementovat odraz toho, co se děje v reálném světě. V případě vývoje informačního systému pro správu známek ve škole, by se v projektu měly určitě nacházet třídy pro studenta, učitele, předmět, známku, případně i třídu. Každá z těchto tříd pak reprezentuje atributy, kterými se daná třída identifikuje a funkce a metody, pomocí kterých komunikuje s okolním světem - s okolními třídami.

Komentáře

Možnost doplnit kód o komentář je pro vývojáře možnost, jak vysvětlit důvod a funkci nějaké části programu. V ideálním případě by měl být kód za sebe vypovídající a nemělo by být potřeba doplnění žádných komentářů. Ve chvíli, kdy je pro plné pochopení významu komentář potřeba, neměl by přesahovat dva řádky a měl by sloužit pouze pro pochopení celkového kontextu [10].

Komentáře mohou také sloužit pro takzvané „anotace“, které vývojáři pomáhají při psaní kódu zjistit, co dané třídy, či metody dělají a k čemu slouží. Podrobněji jsou použití a forma anotací rozepsány v podsekci 4.4.3.

Duplicita kódu

Kód, který se opakuje, si zpravidla zaslouží extrakci do nové funkce, nebo metody. Duplicitní kód komplikuje případnou změnu kódu a zvyšuje riziko, že při úpravě bude do projektu zaneseno více chyb. Toto pravidlo je také v anglické literatuře označováno jako *DRY*[1] z anglického *Don't Repeat Yourself*, tedy *Neopakuj se* [26].

Zpracování chyb

Ač se může zdát tento poslední bod jako nedůležitý, může být dalším ze zdrojů nepřehledného a komplikovaného kódu. V některých jazycích vývojáři neměli, a dodnes nemají, k dispozici nástroje pro zpracování výjimek v podobě `try{...} catch {...}` a využívají zastaralých praktik, jako je navrácení chybového kódu z funkce, nebo metody. Dnes by se měly funkce zaměřovat především na návrat požadované hodnoty, nikoliv chybového kódu, a v případě potřeby vyvolat výjimku kdekoliv v kódu obsahující co nastalo za problém. Vyhnutí se zpracování chyb také pomáhá oddělení funkce kódu od zpracování chyb a případného zotavení. Nesmí se však zapomínat ani na to, že případné vyvolané výjimky je třeba řešit, jako to popisuje Pete Goodliffe v knize [10]. Pokud k tomu není dobrý důvod, nemělo by se stávat, že některé výjimky budeme prostě ignorovat (viz 2.1) bez dalšího zpracování (třeba v podobě informování uživatele). V ideálním případě bychom jim však měli předcházet.

```
1  double Vydel(int a, int b) {
2      double vysledek = 0;
3
4      // Osetreni deleni nulou
5      try {
6          vysledek = a / b;
7      }
8      catch (Exception ex) { }
9
10     return vysledek;
11 }
```

Výpis 2.1: Nevhodné použití konstrukce try-catch

2.2 Měření kvality kódu

Pokud chceme měřit kvalitu čehokoliv, je nejdříve nutné definovat atribut, který na dané věci hodnotíme. Může se totiž stát, že pod pojmem *kvalita* si každý představí něco jiného. Jednoduchým příkladem může být pracovní židle. Pro někoho může znamenat kvalita materiály, ze kterých je vyrobena, pro jiného pak pohodlnost posedu. Stejná myšlenka se dá

[1] Setkat se můžeme také s pojmem *Shy code*, tedy volně přeloženo jako *Ostýchavý kód*, který poukazuje na to, že by část programu měla komunikovat pouze s nejbližšími sousedy a nemělo by docházet například ke konstrukcím jako `getClass().getStudent().getGrade().getScore()`.

také přenést na zdrojový kód. V následujících odstavcích je představeno, jakými způsoby je možné kvalitu kódu počítat a udržovat.

2.2.1 Počet řádků

Počítání počtu řádků (anglicky označováno jako *LOC*, tedy „Lines of code") v je jedna z nejjednodušších metrik, kterou má každý k dispozici. Její hodnota určuje, jak moc je daný subjekt (soubor, třída, metoda) rozsáhlý. V rámci této metriky můžeme měřit několik různých hodnot, které dělíme na následující typy [50]:

- Celkový fyzický součet řádků kódu.
- Celkový logický součet řádků kódu.
- Součet všech prázdných řádků.
- Součet všech řádků s komentáři.

Rozdíl mezi prvními dvěma hodnotami (fyzickým a logickým počtem řádků) je ve způsobu měření. Počet fyzických řádků je jednoduché sečtení znaků pro oddělení řádku[2]. Logický součet řádků je se pak skládá analýzy jednotlivých příkazů. Příkladem může být konstrukce ve výpisu 2.2, kde v případě fyzické analýzy dostaneme hodnotu 1, v případě logické analýzy pak hodnotu 2. Počet volných řádků a řádků s komentáři v tomto případně vždy rovno jedné.

```
1 // Nasledujici konstrukce je pouze na jednom radku
2
3 for (int i = 0; i < 10; i++) Console.WriteLine("Ahoj");
```

Výpis 2.2: Vyhodnocení počtu řádků

2.2.2 První ukazatele kvality kódu

Už v sedmdesátých letech dvacátého století se začalo přicházet na to, že by bylo potřeba nějakým způsobem porovnat různé přístupy k řešení konkrétních problémů, tedy umět změřit, který přístup je lepší. Mezi jednu z prvních metodik porovnávání a měření kvality kódu se řadí *Efektivnost detekce chyb*[3] [31]. Na rovnici 2.1 je pak znázorněna o 10 let mladší definice, která staví na Faganově myšlence a lehce jí upravuje. Nepočítá již efektivnost pouze testovací fáze programu, ale věnuje se samotnému poměru nalezených chyb před a po vydání produktu.

$$Removal\ efficiency = \frac{Defects\ found}{Defects\ found + Defects\ found\ later} \times 100\% \qquad (2.1)$$

Hodnota *Removal efficiency* tedy přímo ukazuje procentuální počet chyb, které byly odhaleny v průběhu vývoje (tedy před samotným vydáním produktu), z celkového počtu nalezených chyb v průběhu celého životního cyklu daného softwaru. Jedná se o zajímavý atribut z hlediska kvality kódu, nicméně nemusí být kvůli možným nenahlášeným chybám

[2] `\n` v případě Linuxu, `\r\n` v případě Windows.

[3] Z anglického *Error Detection Efficiency* představena Michaelem Faganem jako proces obsahující několik fází, pro detekci chyb nejen ve zdrojovém kódu, ale dokumentech obecně.

přesný a zároveň není možné hodnotu vypočítat ještě před tím, než je dané dílo vydáno. Jednalo se tedy především o zpětnou vazbu pro samotné vývojáře, kteří se mohou v dalším vývoji z chyb poučit.

2.2.3 Cyklomatická složitost

Předchozí metrika ukazuje to, jak dobře byl daný produkt vytvořen. Zaměřuje se totiž především na výsledek jako takový, nikoliv přímo na kód programu. Cyklomatická složitost je na rozdíl od počítání nalezených chyb ve výsledném produktu odlišná. Zaměřuje se totiž přímo na zdrojový kód.

Představena byla v roce 1976 Thomasem J. McCabem starším a zjednodušeně nám říká, kolika různými cestami může daný kód procházet. V případě, že zdrojový kód neobsahuje žádné podmínky `if`, případně cykly, je složitost rovna jedné. V případě, že se v kódu bude vyskytovat jedna podmínka `if`, bude cyklomatická složitost rovna 2 [50]. Složitost můžeme počítat na funkcích, modulech, nebo třídách.

Výpočet složitosti se provádí především pomocí grafu, který představuje možné toky a větvení programu. Na obrázku 2.1 je graf představující cesty, které je možné procházet. Jednotlivé uzly představují neoddělitelné části kódu, orientované hrany pak všechny možné případy, kam se může program přesouvat. Červený uzel představuje vstupní bod, modrý uzel pak bod výstupní. Na obrázku je znázorněn kód z výpisu 2.3.

```
1  double spocitejCenu(List<Produkt> produkty, bool sleva) {
2      double cena = 0;
3      for (int i = 0; i < produkty.count; i++)
4          cena += p[i].cena;
5
6      if(sleva)
7          cena *= 0.8;
8
9      return cena;
10 }
```

Výpis 2.3: Výpočet jednotek představující skutečnou práci

Dle příkladu je možné vidět, že se cesty dělí v rámci cyklu `for` na řádku 3. Tomuto dělení odpovídá sekce A v grafu 2.1, kde jednotlivé hrany tvoří cyklus. K druhému větvení kódu dochází v podmínce `if` na řádku 6, kterému odpovídá sekce B v grafu.

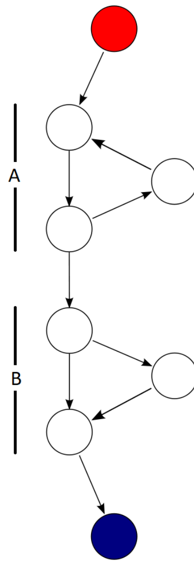
Samotná cyklomatická složitost se pak vypočítá pomocí rovnice 2.2, kde

- MCC značí cyklometrickou složitost (z anglického „McCabe cyclomatic complexity“),
- E značí počet hran grafu,
- N počet uzlů[4].

$$MCC = E - N + 2 \tag{2.2}$$

V zobrazeném příkladu je použita jedna podmínka a jeden cyklus. Jakmile je graf zkonstruován, je možné vyčtené hodnoty vložit do rovnice a zjistit výslednou hodnotu. Pro

[4] Setkat se můžeme také s obecnější verzí, kde místo dvojky na konci rovnice je 2P, kde P značí počet připojených komponent. Tato práce pracuje pouze se zjednodušeným výpočtem.



Obrázek 2.1: Graf zobrazující alternativní toky programu, převzato z [58]

zjištění počtu hran a uzlů se vychází z celkového počtu daných prvků v grafu, včetně vstupních a výstupních uzlů. Samotný výpočet je pak triviální a v rámci rovnice 2.3 je výsledná hodnota 3. Byť by bylo možné samotnou metodu ještě zjednodušit (například nahrazení atributu `bool velkaSleva` přímo hodnotou slevy, tedy zbavením se podmínky `if`), můžeme být na základě následujících odstavců spokojení.

$$
\begin{gathered}
MCC = E - N + 2 \\
MCC = 9 - 8 + 2 \\
MCC = 3
\end{gathered}
\tag{2.3}
$$

Maximální hodnota cyklomatické složitosti

V roce 1976, kdy byl tento výpočet představen bylo stále využívané procedurálního paradigmatu, v rámci kterého se kód třídil do jednotlivých modulů. McCabovým cílem bylo, aby si každý programátor hlídal složitost jednotlivých modulů, kde by maximální hodnota neměla přesahovat číslovku 10. V dnešní době se masivně využívá především objektově orientované paradigma. Je nutné přemýšlet nad hranicí „velkého kódu“ trošku jinak.

Na základě výzkumu fakulty informačních technologií univerzity v Pensylvánii [36] bylo zjištěno, že 95% funkcí, které analyzovali bylo dosaženo maximální cyklomatické složitosti 6. Z toho vyplývá, že nastavená hranice na hodnotu 10 je pro dnešní dobu nastavena příliš vysoko a můžeme ji považovat za nadhodnocenou. V dnešní době se nejčastěji programátoři snaží dostat maximálně do hodnoty 4 u jednoduchých funkcí. Hodnoty, které se blíží číslu 10 mohou být stále považovány za stabilní[5], nicméně složitost vyšších hodnot je pro vývojáře již alarmující. V tak vysokých hodnotách je velká šance výskytu skrytých chyb, zároveň jsou tyto části kódu velmi často špatně udržovatelné a čitelné [1].

[5]Stabilní je v tomto kontextu, a za podmínky dobré struktury, myšleno dobře udržovatelné.

2.2.4 Halsteadův objem

Z výše uvedených metrik je možné zjistit, jak moc byl kód chybový, nebo kolik obsahuje různých větvení. Metrika, která byla v roce 1977 pojmenovaná po svém představiteli, Howardovi Halsteadovi, se zabývá tím, z kolika různých programových jednotek se daný kód, nebo jeho část, skládá [29]. Halstead tak zavádí právě nový pojem *Programová jednotka*, která představuje úsek kódu, který od sebe rozlišuje překladač v syntaktické části překladu. Samotný výpočet je pak definován rovnicí 2.4, kde:

- N představuje celkový počet operátorů a operandů,
- n říká, kolik z nich je unikátních [56].

$$V = N * \log_2(n) \tag{2.4}$$

Halsteadův objem nám tedy říká to, jak moc je kód rozsáhlý z hlediska počtu prováděných operací. V případě krátkých funkcí, které jsou bez parametru, by hodnota měla nabývat minimálně čísla 20[6]. Pokud má funkce objem menší, měl by programátor zvážit, zda se opravdu vyplatí aby tato funkce existovala. Maximální hodnotou je pak číslo 1000, kde při překročení této hodnoty z největší pravděpodobnosti dochází k porušení SOLID principů, popsaných v podkapitole 3.3.1 a funkce tak dělá více věcí a není jednoúčelová. Samotný soubor by pak neměl přesahovat hodnotu 8000 [56].

2.2.5 Index udržovatelnosti

Kombinací výše uvedených metrik můžeme vypočítat takzvaný *Index udržovatelnosti*, v literatuře označován anglicky jako *Maintainability index*. Tato veličina zjednodušeně říká, jak jednoduše bude možné provádět zásahy do kódu a jak složitá bude jeho údržba. Vzorec pro výpočet indexu v rovnici 2.5 se skládá z následujících složek:

$$max(0, 100\frac{171 - 5.2lnV - 0.23G - 16.2lnL + 50sin(\sqrt{2.4C})}{171}) \tag{2.5}$$

- **V** - hodnota Halsteadova objemu,
- **G** - celková cyklomatická složitost,
- **L** - počet řádků kódu,
- **C** - počet řádků obsahující komentář.

Většina integrovaných vývojových prostředí (IDE) mají nějakým způsobem výpočet a prezentaci těchto metrik zabudovanou a vývojář tak může mít přístup k jejich hodnotám v reálném čase, což může vývojáři pomoci jako kontrola, jak dobře a čistě svůj kód napsal.

I přes to, že se může zdát, že je *Maintainability index* nejlepším ukazatelem, protože slučuje více různých aspektů, má i některé zásadní nevýhody. Protože index je reprezentován pouze jednou hodnotou, která je složena dohromady ze čtyř složek, může být obtížné hledat důvod, proč je jeho hodnota nízká a jakým způsobem jí zvýšit. Je proto důležité se vždy dívat na tuto vlastnost v kontextu s atributy ze kterých vychází [33]. Hlavní důvod, proč se tento index využívá pramení z nutnosti komunikace a kontroly například netechnicky

[6]Takto nízkou hodnotu mají například jednořádkové funkce.

založeným managementem. Hodnota indexu je pro něj lépe pochopitelná a především nabízí jednodušší možnost srovnání s jinými projekty.

Celkově bohužel není možné nastavit přesné hranice výše popsaných atributů na konkrétní hodnoty. Vždy je potřeba práh přizpůsobit tak, aby byl vývoj pohodlný a zároveň efektivní a funkční. Nutné je také pochopení, že ani jedna z těchto hodnot neurčuje, zda je kód rychlý a správně napsaný. Zmíněné techniky měření poukazují pouze na to, jak dobře je kód čitelný a udržovatelný pro ostatní.

2.3 Udržování kvalitního kódu

Jednou částí životního cyklu kódu je jeho tvorba, tedy začátek cyklu samotného. V průběhu času ale může docházet k jeho úpravě a je stále třeba, aby byl kód udržován dle stejně nastavených pravidel. K tomu nám pomáhají různé typy analyzátorů, které jsou popsány v dalších podsekcích.

2.3.1 Statická analýza kódu

Zvyšování kvality kódu, jak stran chybovosti, tak i například čitelnosti je možné dosáhnout pomocí různých nástrojů. Základní metodiky k udržování čistého kódu jsou popsány v následujících bodech.

- Samostatná kontrola vlastního kódu. K odhalení nejvíce chyb je dobré se ke kontrole vracet s odstupem času, tedy nikoliv hned po napsání.
- Vzájemná kontrola s jiným vývojářem (dnes označovaná jako „Code review“[7]). Je dobré, aby se na kód podíval někdo, kdo k danému kódu bude přistupovat nestranně.
- Kontrola v rámci týmu. Využívá se v případě komplikovaných, nebo problematických případech důležitých, nebo složitých částí zdrojového kódu.
- Nezávislý audit. Audit aplikace dnes slouží především k potvrzení, případně vyvrácení správnosti funkcí. Využívá se ale také k prověření, že nedošlo k porušení bezpečnostních standardů[8].
- Kontrola kódu prováděná automatizovanými nástroji.

Tyto body jsou v různých formách využívány téměř každou firmou, která je zaměřena na vývoj softwaru. Velmi kvalitně jsou také podporovány systémy pro správu verzí kódu. Jedná se o takzvanou statickou analýzu kódu, která má za úkol kontrolu zdrojového kódu jako takového. Druhým protipólem je pak dynamická analýza, která se věnuje především správné funkčnosti kódu. Při dynamické analýze je kód, nebo jeho části, spouštěn s různými parametry a kontrolovány očekávané výsledky [20].

[7] Celá myšlenka a jednotlivé kroky Code review jsou velmi dobře popsány univerzitou MIT na `https://web.mit.edu/6.005/www/fa15/general/code-review.html`.

[8] Například potvrzení o tom, že aplikace neodesílá soukromá data uživatelů třetí straně, případně že není náchylná k určitému typu útoku. Příkladem je prohlášení ČVUT (`https://erouska.cz/downloads/cvut4.pdf`) ohledně aplikace eRouška (`https://erouska.cz/`)

2.3.2 Automatizace statické analýzy

Většina bodů popsaných v předešlé podkapitole 2.3.1 se zaměřuje na statickou analýzu člověkem. Ta je ale bohužel velmi časově nákladná a nepodaří se vždy odhalit všechny chyby. Proto postupem času začaly vznikat automatizované analyzátory, které pomáhají vývojářům odhalit možné skryté nedokonalosti v kódu. V dnešní době se aktivně využívají například v rámci programu Visual Studio[9], které vývojáři kontroluje kód už ve chvíli, kdy jej píše a v reálném čase doporučuje využití jiných, mnohdy i lépe pro člověka čitelných konstrukcí. Ve srovnání s analýzou pomocí lidí mají programové analyzátory implementována přesná pravidla, tedy i dostatečnou znalost problematiky, jejíž rozsah se u lidí liší. V dnešní době umí nástroje pro statickou analýzu také odhalit možné bezpečnostní chyby v programu a navrhnout, jakým způsobem je ošetřit. Jedná se však ale o určitou formu hledání vzorů v kódu. Ve srovnání s člověkem nemusí vždy analyzátor správně vidět kód v širším kontextu, tedy nemusí vždy odhalit všechny možné chyby [9]. I přes to, že se tyto nástroje stále vyvíjí, nesmí být jejich výstupy brané jako dogma. Může se stát, že analyzátor nezahlásí nějakou chybu, nebo naopak indikuje chybu, která neexistuje.

[9]Integrované vývojové prostředí vyvíjené společností Microsoft `https://visualstudio.microsoft.com/cs/`.

Kapitola 3

Refaktorizace a návrhové vzory

Při požádání jakéhokoliv vývojáře z oboru informačních technologií o vysvětlení pojmu *refaktorizace*, dostaneme z největší pravděpodobností obdobnou formu následující věty: *Refaktorizace je proces, kdy měníme či vylepšujeme kód takovým způsobem, že změny nemají vliv na vnější chování programu.*[1] Tato definice není špatná, ale nevysvětluje vše, co je třeba k úplnému pochopení toho, proč je třeba a jakým způsobem se aplikuje.

3.1 Cíle refaktorizace

Samotný proces refaktorizace není pouze o vylepšování kódu ve smyslu zrychlení běhu programu. V rámci vývoje existuje celá řada cílů a důvodů, proč se úprava kódu provádí. Mezi nejčastější příčiny patří především odstranění takzvaných pachů, které často odkazují na špatné návyky vývojářů. Refaktorizace se tedy nejčastěji zabývá odstraněním následujících problémů:

- odstranění duplicitního kódu,
- zkrácení dlouhých funkcí, metod a tříd,
- redukce potřebných parametrů,
- sloučení datově závislých atributů,
- odstranění přebytečných komentářů.

Výše zmíněné body mají jeden velký společný cíl, **zvýšení přehlednosti v kódu**. V dnešní době má každý uživatel k dispozici řadu internetových služeb pro sdílení kódu. Tyto služby pomáhají především při kooperativním vývoji. Vývojáři tak mají rychlý a jasný přehled nad tím, kdo právě na které části programu pracoval a k jakých změnám v kódu došlo. Aby se ale skupina vývojářů navzájem vyznala v cizím kódu, je potřeba udržovat kód čistý, jednoduchý a jasně strukturovaný. Pravidla, která pomáhají udržovat kód čitelný a nejlepší praktiky jsou popsány v předchozí kapitole 2.

Důvod použití refaktorizace nemusí nutně znamenat vždy pouze to, že kód není nepřehledný. Často se také může jednat o pozdější nutnost rozšíření funkce programu. Často jsou rozsáhlé projekty nasazeny v rámci korporací i několik desítek let, proto je potřeba psát kód takovým způsobem, aby byl snadno rozšiřitelný. V ideálním případě by měly být

[1] Převzato z obecného popisu https://cs.wikipedia.org/wiki/Refaktorování

části programu rozšiřitelné pouhým přidáním nové funkcionality bez úpravy již existujícího kódu. Nejčastěji nám k tomuto dopomáhají návrhové vzory, popsané v podkapitole 3.3, podpořené technikami, které se při refaktorizaci používají, popsané v kapitole 3.2.

I v případech, kdy je kód dobře napsaný a udržovaný může stále nastat chvíle, kdy je potřeba jej rozsáhle upravit. Případem může být přechod na jinou platformu, nebo třeba nutnost podpory novějších technologií. Jedním z příkladů může být nutnost rozšíření programu pro další platformy. Ukázkou může být tato diplomová práce, jejímž cílem je právě migrace kódu z .NET Framework, který je podporovaný pouze v rámci ekosystému Windows na .NET 5, které přidává podporu pro další platformy.

3.2 Bezpečná úprava kódu

Úprava kódu, ať už z důvodu rozšíření funkce, nebo oprav určitých chyb, může být v rámci více rozsáhlých projektů komplikované. Proto je velmi důležité, aby se každý vývojář zodpovědně držel doporučených zásad a praktik, které jsou v rámci vývoje moderních aplikací využívány. Jedná se především o udržování čistého kódu, které je popsané v kapitole 2, ale také správnou implementaci a dodržování dostupných návrhových vzorů, které jsou popsány v následující podsekci 3.3.

Pod úpravou kódu se může schovávat širší škála různých aktivit. Může se jednat pouze o doplnění funkce, opravu určité chyby, refaktorizaci s ohledem na vyčištění kódu, nebo samotnou migraci na novou platformu. Celý proces se skládá z několika základních kroků, které je vždy doporučeno řádně dodržovat.

1. **Identifikace kódu**, který je potřeba upravit, nebo opravit.

2. **Tvorba testů**, potvrzujících funkčnost, případně nefunkčnost dané části kódu. Testy jsou jedním ze základních stavebních kamenů nejen úpravy jako takové, ale i celého projektu.

3. Samotné **provedení úpravy** kódu. V tomto kroku jsou připraveny všechny testy a je jasně definováno, jakou část a proč je třeba upravit.

4. **Spuštění testů**. Nejedná se pouze o testy, které přímo souvisí s danou úpravou, ale v ideálním případě o všechny testy, které testují kód, který využívá upravenou část. Bude tak potvrzeno, že oprava chyby v jedné části kód, nenaruší fungování jiné části programu.

5. Podle výsledků testů z předchozího kroku jsou provedeny **dodatečné úpravy** kódu, které přímo souvisí s tím, co bylo upraveno. V tomto kroku můžou být také doplněny ještě nové testy, nicméně nemělo by docházet k úpravě testů stávajících [38].

Jednou z modifikací bodů výše může být metoda, kdy je potřeba přepsat více než jednu funkci. Je tak možné, po vytvoření dostatečného počtu testů, vytvořit část kódu úplně od začátku [35]. Vývojář tak nejdříve napíše novou část kódu, zkontroluje, že všechny testy skončí úspěšně a teprve až poté zkusí všechna stará volání přepsat na nově vytvořený kód. Následuje opět spuštění testů a případné ladění nového kódu. Ve chvíli, kdy je potvrzené, že změna nezpůsobila chyby jinde, může být původní kód bezpečně odstraněn.

Obecně je ale doporučeno, aby jakékoliv úpravy probíhaly pouze na menších částech kódu. Je tak zajištěno, že je vidět určitý postup oprav a zároveň je snížené riziko vnesení dalších chyb.

3.3 Návrhové vzory

Každé softwarové dílo vzniká proto, aby řešilo, respektive zjednodušovalo, nějaký problém. Každý vývojář, který se podílí na tvorbě zejména středně velkého, až velmi rozsáhlého softwarového produktu se s pojmem *Návrhový vzor* určitě setkal. Je to skupina nástrojů, které nám pomáhají především v následujících dvou situacích [25].

- Vyřešit nějaký problém v rámci tvorby programu, nejvíce efektivním a především již prověřeným způsobem.
- Pomoci pochopit se s ostatními vývojáři, o jaký problém se jedná a jak jej řešit. Je jednodušší říci, že se jedná o problém, který řeší například `observer`, než složitě demonstrovat na schůzi danou implementaci v rámci našeho projektu.

Návrhové vzory jsou určité myšlenky, které představují již hotová a ověřená řešení nějakého problému. Za jedny z nejstarších návrhových vzorů můžeme považovat funkce a metody v rámci kódu - tedy možnost vytvoření znovupoužitelného kódu bez nutnosti jej celý opakovat. Následující skladba návrhového vzoru je převzata z bakalářské práce Hany Slámové, čerpající z knihy *Design Patterns: Elements of Reusable Object-Oriented Software* [51, 19]:

- **Název** je velmi důležitou součástí, jenž většinou ve dvou slovech definuje problém, který řeší.
- **Problém, který daný návrhový vzor řeší** vysvětluje, jaký problém je tímto vzorem přesně řešen. Může se jednat o problémy behaviorální, tedy řešení komunikace, mezi objekty, nebo problém strukturální, tedy jak mají být dané třídy strukturovány a uspořádány.
- **Řešení problému** detailně popisuje způsob, jak je možné problém v rámci kódu vyřešit. Nejedná se o část kódu, kterou je možno přímo vložit do projektu, nýbrž o šablonu, která může být aplikována v různých situacích.
- **Výhody a nevýhody** daného vzoru. Není vždy pravdou, že použití určitého návrhového vzoru je nutné a správné. Často s sebou mohou vzory nést nevýhody v podobě komplikovanější implementace, zvýšení časové, nebo paměťové náročnosti. Mezi výhody se často řadí zvýšení čitelnosti kódu, lepší testovatelnosti, jednodušší rozšiřitelnosti a především udržovatelnosti.

Zda implementovat návrhový vzor je ideální přemýšlet ve chvíli, kdy vzniká čistě nový kód. V případě úpravy již existujícího kódu se návrhové vzory zavádějí složitěji, protože může být třeba upravit chování více částí programu najednou. V případě špatného pochopení již existujícího kódu, respektive řešeného problému v případě kódu nového, se může stát, že dojde k špatnému použití daného návrhového vzoru, což může zanést pouze negativní důsledky daného vzoru do fungování programu.

Speciálním případem jsou takzvané *Anti-patterns*, česky *nevhodné návrhové vzory*. Ty jsou složeny ze stejných částí, jako klasické návrhové vzory a stejně tak popisují určité myšlenkové pochody, jak se vypořádat s určitým problémem. Zásadním rozdílem je však fakt, že anti-návrhové vzory vnášejí do kódu pouze negativa - nepřehlednost, komplikace při rozšiřitelnosti a to často včetně vyšší časové a paměťové náročnosti. Jedním z nejhorších možných představitelů je *Big Ball of Mud*, česky přeloženo jako *Velká hrouda bahna*, který

byl popularizován již v roce 1997 Brianem Footem a Josephem Yoderem. Jedná se zpravidla o výsledek vývoje, který je ve velké časové tísni a může tak vyústit až v globalizaci všech důležitých proměnných, tvorbu špagetového kódu, ztrátu zapouzdření a jakéhokoliv oddělení díla na více logických celků [52, 16].

Níže je obecně popsáno několik návrhových vzorů, které jsou stěžejní pro tuto práci. Vybrány jsou především ty, které se již v rámci celého projektu nachází včetně těch které budou případně doplněny a využity. Detailní popis toho, jak jsou v rámci projektu implementovány a proč byly vybrány se nachází v podkapitole 4.2.1.

3.3.1 SOLID

SOLID jako takový nepředstavuje jeden konkrétní návrhový vzor. Jedná se o sadu principů a pravidel, které by měl v ideálním případě každý vývojář dodržovat a snažit se o jejich uplatnění v jakékoliv části vývoje. Název SOLID pochází ze spojení počátečních písmen anglických slov:

1. Single Responsibility Principe (SRP) - Princip jedné odpovědnosti,
2. Open-Closed Principle (OCP) - Princip otevřenosti a uzavřenosti,
3. Liskov Substitution Principle (LSP) - Liskovové princip zaměnitelnosti,
4. Interface Segregation Principle (ISP) - Princip oddělení rozhraní,
5. Dependency Inversion Principle (DIP) - Princip obrácené závislosti [30].

První z výše vypsaných principů tvrdí, že by každá třída měla mít zodpovědnost vždy za jednu konkrétní činnost. Přímo tak vyvrací použití anti-návrhového vzoru, který je pojmenovaný jako švýcarský nůž [55], tedy implementace nesouvisejících funkcí a metod do jedné třídy. Někteří vývojáři tvrdí, že přísné dodržování tohoto pravidla může vést k složitější implementaci. Pravdou je, že výsledné rozdělené třídy se mohou zprvu jevit jako složitější návrh, nicméně v průběhu vývoje mohou značně zjednodušit doplňování nových funkcí, zvlášť ve spojení například s využitím rozhraní, která jsou popsána ve čtvrtém bodě [30].

Druhý princip otevřenosti a uzavřenosti odkazuje především na možnosti abstrakce a polymorfismu. Tedy cílem je psát třídy takové, že je abstrahována společná funkcionalita a menší změny jsou definování až v děděných třídách. Díky dodržování tohoto pravidla, je splněn jeho název, tedy otevřenost vůči rozšíření, uzavřenost proti změnám. Sám Robert C. Martin označuje tento princip za nejdůležitější [38], neboť právě dodržení tohoto doporučení sníží zásahy do existujícího kódu na minimum.

V roce 1987 představila Barbara Liskovová třetí z částí, které jsou součástí principů SOLID. Jedná se o princip zastoupení, kdy každý potomek nějaké třídy musí plně umět nahradit třídu rodičovskou. Jedná se tedy pouze o správné využití polymorfizmu v kódu. Správnou implementaci tohoto principu zajišťuje především shoda případných podmínek před a po vykonání dané funkce[2], které musí podtřída respektovat v plném rozsahu [59, 30].

Předposledním bodem je princip využívání různých rozhraní v rámci vývoje. Ač se opět může zdát, že pro vývojáře to znamená více kódu, z dlouhodobého hlediska je vývojářům ulehčeno. Využívání rozhraní totiž omezuje to, jakým způsobem k jakým instancím můžeme přistupovat. Zároveň můžou různé třídy implementovat různá rozhraní zároveň, a tak se nemůže stát, že bychom museli některé třídě implementovat metody, které vůbec nemá mít.

[2] V anglické literatuře označované jako *precondition* a *postcondition*.

Posledním bodem je princip obrácené závislosti, který poukazuje na to, že by třídy vyšší úrovně abstrakce neměly záviset na těch, které implementují nižší úroveň abstrakce. Zároveň by všechny závislosti měly vést přímo na rozhraní, případně abstraktní třídy, nikoliv pak na implementované podtřídy. Implementace tohoto principu přináší především snížený počet závislostí napříč kódem a zároveň zabraňuje přepisování kódu na více místech, kvůli jedné úpravě třídy [30].

3.3.2 Dependency injection

Větou, která je uvedena na následujícím řádku shrnul zdroj [42] celou podstatu tohoto celého návrhového vzoru.

> „Nic nesháněj, ať se postará někdo jiný."

Cílem tohoto návrhového vzoru je právě to, aby třídy nemusely řešit životní cyklus dalších instancí tříd, které ke své práci potřebují. V praxi je injekce závislosti implementována kontejnerem, který má právě vytváření a ukončování života dalších tříd na starosti. Dependency injection také umožňuje sestavení programu bez toho, aby na sebe měly různé třídy přímé závislosti.

V rámci tohoto vzoru pak identifikujeme 3 různé subjekty. **Konzument**, který očekává při svém vytvoření určitou **službu**, kterou dodá právě zmíněný **kontejner**. Konzument tedy při jeho vytváření dostane od kontejneru všechny požadované služby, které ke svému fungování potřebuje. Předávání služeb může probíhat buď pomocí vkládání přes konstruktor, kdy jsou v konstruktoru konzumenta definované potřebné závislosti, prostřednictvím vkládání referencí na služby do vlastností tříd, nebo pomocí vkládání závislostí skrze metody, které umožňuje podtřídám v případě potřeby požadované závislosti měnit.

Jedním z nejdůležitějších bodů je pak samotné testování aplikace. Právě díky injekci závislostí je možné testovat samostatně všechny součásti bez toho, aby vždy existoval celý řetězec instancí daných závislostí [8].

3.3.3 Model-View-ViewModel

Tento návrhový vzor slouží jako třívrstvá architektura pro jednoduché oddělení klientské aplikace od byznys logiky. Jedná se o návrhový vzor vytvořený přímo pro technologii Windows Presentation Foundation, zkráceně WPF. Na rozdíl od uživatelského rozhraní řízeného událostmi je zde využíván binding (volně přeloženo jako „napojení") a command (přeloženo jako „příkaz").

Základní stavební jednotkou je část ViewModel, která má za úkol udržovat stav aplikace. Na základě ViewModelu vykresluje uživatelské rozhraní (view) ovládací prvky. Při interakci uživatele, kdy například zadá text do textového pole, je díky napojení (binding) zajištěna propagace zpět do ViewModelu. Poslední částí je pak model, který je reprezentován třídami, se kterými byznys logika pracuje. Jedná se tak například o databázové objekty [14].

3.3.4 Fixture

Návrhový vzor fixture (česky přeloženo jako „přípravek"), je jedním z velmi důležitých nástrojů při testování. Existuje totiž kategorie testovacích jednotek, kterým nestačí pouze vstup formou parametrů a jedna návratová hodnota, ale potřebují určité prostředí ke svému správném fungování. Tento vzor nám umožňuje takové prostředí simulovat a předložit tak testované funkci vše, co potřebuje. Příkladem může být tvorba dočasné databáze uložené

v paměti RAM, která se na začátku testů vytvoří, před každým testem inicializuje na požadovaná data a po skončení testu zas vyčistí.

Příprava prostředí nemusí probíhat pouze pro zvolený test, ale například pro sadu testů, nebo celé třídy. Různé přípravy prostředí na sobě mohou být nezávislé, tedy určitá část přípravy může proběhnout na začátku celého testování, později pak před každou skupinou testů a nakonec také před každým testem. Metody, které připravují testovací prostředí se nazývají *SetUp*, naopak metody čistící, co testy napáchaly, se nazývají *TearDown* [54].

Nejčastěji se tak připravují již zmíněné stavy databáze, testovací vzorová data, příprava vstupních souborů, které mají testy zpracovávat, nebo nastavování konkrétního požadovaného stavu závisejících instancí tříd.

Kapitola 4

Netfox Detective

Když jsem poprvé četl slovní spojení *Síťová forenzní analýza*, respektive v anglické podobě *Network forensics analysis*, věděl jsem, že se jedná o něco, co je úzce spojeno se síťovou bezpečností. Význam tohoto pojmu se v průběhu let měnil a různí autoři vysvětlovali tento pojem odlišnými způsoby. Kolem nového tisíciletí, na začátku masivního růstu internetu, byl význam vysvětlován jako nástroj, který chrání vnitřní sítě společností před síťovými útoky. Dnes byl význam upraven a adaptován s ohledem na aktuální zákony v rámci internetového světa. Síťová forenzní analýza dnes představuje nástroje a metodiky, které umožňují identifikaci, extrakci a interpretaci síťových dat. Jedná se tedy o přirozený vývoj tohoto odvětví počítačové bezpečnosti, který zahrnuje zachytávání, sběr dat a analýzu síťových událostí za účelem nalézt a usvědčit narušitele systému [44].

Analýza síťového provozu může být chápána v rámci dvou různých skupin, které jsou odděleny především časem, kdy jsou data zkoumána. První skupinou je NFAT (Network Forensic Analysis Tool), která zahrnuje nástroje určené k monitorování a analýze shromážděných dat. Největší uplatnění tyto nástroje nacházejí při vyšetřování nelegální činnosti v rámci počítačových sítí. Druhou skupinu tvoří NSM (Network Security and Monitoring), jenž se zaměřuje spíše na monitorování a správu počítačových sítí v reálném čase.

Netfox Detective se řadí do první skupiny. Cílem tohoto nástroje je extrakce a rekonstrukce dat z uložené síťové komunikace. Zaměřuje se především na aplikační vrstvu TCP/IP modelu[1], zejména pak na HTTP (Hyper Text Trasport Protocol), VoIP (Voice over Internet Protocol), FTP (File Transfer Protocol) či například emailové protokoly POP (Post Office Protocol), SMTP (Simple Mail Transfer Protocol), nebo IMAP (Internet Message Access Protocol). Podporovány jsou také technologie instantních komunikátorů, jako například ICQ, Google Hangouts, nebo Facebook Messenger. Mezi více atypické případy lze pak zařadit rekonstrukce zpráv v počítačových hrách Minecraft a Warcraft, případně extrakce informací v rámci Bitcoin sítě [47, 46].

4.1 Důležité technologie

Velká spousta dnešních nástrojů již není tvořena na zelené louce. Není třeba psát nové knihovny plné funkcí, případně celé frameworky, které lze později v kódu uplatňovat. Zbytečně bychom si tak zanášeli možné chyby a bezpečnostní rizika do našeho programu a zároveň bychom se tak snažili „znovu vynalézt kolo“. Dnes oblíbené a často používané knihovny jsou vyvíjeny již řadu let a prošly různými testy a způsoby použití u tisíců uživatelů. Dá se

[1] https://en.wikipedia.org/wiki/Internet_protocol_suite

tedy předpokládat, že u rozsáhlých knihoven bude drtivá většina chyb opravena a případné problémy napraveny. Zároveň také můžeme u aktivních projektů počítat s tím, že nalezené chyby budou opraveny, nebo bude zvýšena rychlost některých částí dané knihovny.

V následujících podsekcích jsou představeny nejdůležitější technologie, bez kterých by se tento produkt neobešel. Každý z těchto frameworků a knihoven má v projektu své důležité místo a v této práci s nimi úzce přichází do styku.

4.1.1 .NET technologie

Pojem .NET v dnešní době nepředstavuje přímo jeden konkrétní framework, nebo nástroj, ale jde o celou rodinu různých frameworků zaměřených na odlišná odvětví. Nejčastěji se setkáváme s pojmem .NET Framework (popsaný v podsekci 4.1.2), případně dnes s modernějším a novějším aplikačním rámcem .NET Core, který je popsaný v podsekci 4.1.3.

Mezi zajímavé výhody patří jazyková kompatibilita, což znamená, že je možné psát různé části kódu v různých jazycích podporovaných jednotlivými frameworky. Zdrojové kódy se pak přeloží do mezikódu nazvaný CIL[2]. Tento mezikód není přímo spustitelný procesorem, ale interpretuje se pomocí aplikačního virtuálního zařízení CLR[3]. Výhodou tohoto přístupu je zpracování virtuálním strojem, který se za nás stará například o správu paměti, nebo odchytávání výjimek.

Druhým stavebním kamenem je pak knihovna nazvaná FCL[4], která obsahuje již připravené funkcionality pro usnadnění vývoje. Jedná se především o knihovny pro vývoj grafického uživatelského rozhraní, přístupu k databázím, kryptografické metody, nebo třeba knihovny pro síťovou komunikaci a tvorbu webových aplikací. Programátor při vývoji nevyužívá přímo zdrojové soubory těchto knihoven, ale pouze jejich rozhraní [7, 28].

4.1.2 .NET Framework

Aplikační rámec pojmenovaný jako .NET Framework je dnes základní komponentou celé rodiny. Jedná se o framework vyvíjení primárně pro systémy Windows. První verze byla vydána pro systémy Windows NT a aktivně je dnes vyvíjen i pro moderní operační systémy. Pro samotné vývojáře je dnes pohodlný díky tomu, že je předinstalován i v samotném operačním systému Windows. Pokud tedy vývojář zvolí správnou cílovou verzi, pro kterou bude vyvíjet, nemusí po uživatelích ani požadovat instalaci dodatečného softwaru.

Důvody k jeho zvolení oproti novějšímu .NET Core zobrazuje Microsoft přímo ve své vlastní dokumentaci [41]. Uvádí zde, že je vhodné zvolit tento framework, pokud doplňuje jinou aplikaci, psanou v tomto frameworku, nebo využívá technologií, které nejsou dostupné pro novou platformu.

4.1.3 .NET Core a .NET 5

V dnešním světě velmi rozličných systémů a technologií je velmi těžké udržovat kód vždy aktualizovaný s ohledem na současné trendy. V tomto ohledu mají většinou tzv. Open-source[5] systémy převahu, protože se často umí rychleji přizpůsobit aktuální poptávce na

[2]Common Intermediate Language, případně označení jako MSIL, tedy Microsoft Intermediate Language

[3]Common Language Runtime, virtuální stroj, který interpretuje mezikód.

[4]Framework Class Library

[5]Open-source, česky jako Otevřený software, je produkt, jehož zdrojový kód je k dispozici zdarma pod určitou licencí, která definuje, jakým způsobem je možno s daným produktem nakládat.

trhu. Platforma .NET Core, dnes označovaná jako .NET 5, je právě jednou z takových technologií, která otevřený přístup ke zdrojovému kódu nabízí[6] [7].

První verze .NET Core byla představena v listopadu 2014 a ukázala tak nový směr, kterým se chce firma Microsoft posouvat. Zjednodušeně se jedná o takzvaného cross-platform[7] následníka .NET Framework. Pro vývojáře je tak velmi zjednodušen vývoj produktů tvořených v rámci rodiny .NET. Byl znovu vytvořen interpret nazvaný CoreCLR obsahující základní sadu knihoven CoreFX. Společně s tímto základem bylo přepsáno i několik důležitých rozšiřujících knihoven, jako napířklad Entity Framework, který se stará o objektově-relační mapování do databáze. Některé části z původního .NET Framework byly přesunuty do balíčkovacího systému NuGet.

Pro samotné vývojáře je pak velmi důležitá právě možnost psát jeden kód, který bude možné spustit jak na Windows, nebo Linux zařízeních, tak i například Android telefonech. V této práci představuje .NET Core, respektive .NET 5 právě cílový framework, do kterého bude celá aplikace migrována.

4.1.4 DotVVM

Vývoj webových aplikací je dnes velmi rozsáhlý. Často se využívá odděleného aplikačního rozhraní (například formou REST API) a samotné webové aplikace, napsané v některém z moderních frameworků. V rámci technologií .NET si může vývojář zvolit z připravených ASP.NET Core MVC, nebo třeba APS.NET Core API s použitím samostatné webové aplikace. Toto řešení ale nemusí být pohodlné pro vývoj rozsáhlých projektů, například kvůli nutnosti udržovat souběžný vývoj dvou projektů v odlišných jazycích.

DotVVM nabízí jednoduchou implementaci návrhového vzoru MVVM (viz 3.3.3) přímo pro webové aplikace. Po vývojáři je požadována pouze základní znalost HTML a CSS, včetně jazyka C#. Práce se samotným frameworkem pak není složitá. Stránky jsou tvořeny HTML soubory doplněnými pouze o značky DOTHTML[8]. DotVVM nezajišťuje pouze možnost separace kódu na jednotlivé znovupoužitelné komponenty, ale také možnost provázání HTML stránek se samotnou aplikací (anglicky nazývané jako „data binding“). Veškeré chování stránek je pak definované pomocí C# kódu v samostatných třídách a komunikaci mezi HTML stránkou a kódem aplikace zajišťuje na pozadí právě DotVVM za použití volání AJAX [4].

4.2 Struktura aplikace

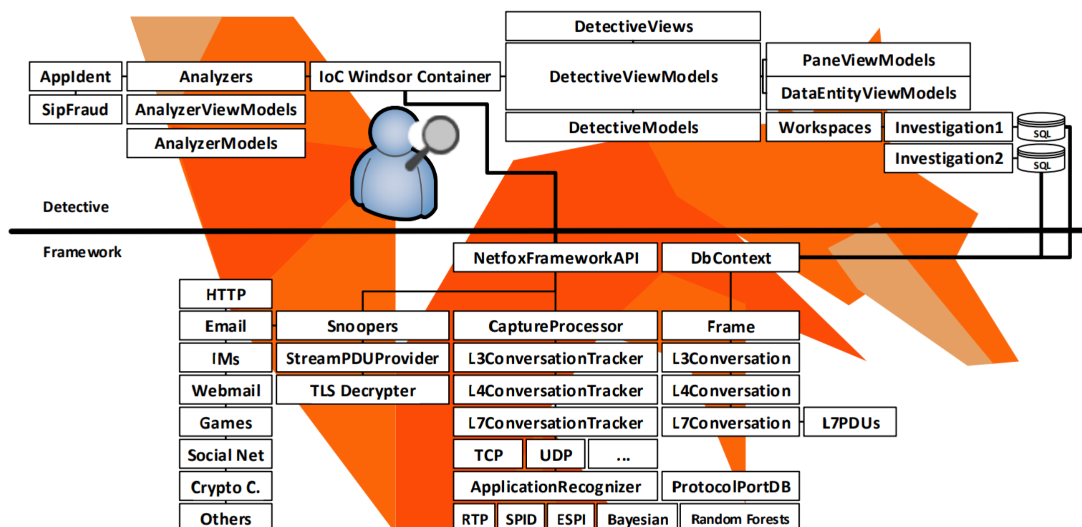
Samotná aplikace se skládá z několika základních částí. Na nejvyšší úrovni je možné identifikovat dvě části, které spolu navzájem komunikují. Na obrázku 4.1 je vidět rozdělení na část Framework a Detective.

Spodní část obrázku představuje Netfox Framework a stará se především o zpracování a analýzu síťové komunikace ze vstupních souborů. Vrchní část pak představuje Netfox Detective, který se stará o komunikaci mezi uživatelem a Netfox Frameworkem. Z uživatelského rozhraní je voláno aplikační rozhraní s názvem `NetfoxFrameworkAPI`, které se dále stará o zpracování požadavků a zadání dílčích částí jednotlivým komponentám.

[6] https://source.dot.net/

[7] Česky jako multiplatformní software. Představuje dílo, které může být spuštěno na různých počítačových platformách (například Windows, Linux, MacOS).

[8] Souhrnný název pro přidané značky z DotVVM do jazyka HTML.



Obrázek 4.1: Architektura Netfox Detective, převzato z [45]

Z hlediska této práce je ale důležitější skutečné rozdělení jednotlivých částí kódu. Na obrázku 4.2 je znázorněno rozdělení části kódu na nejvyšší úrovni do jednotlivých složek. Níže je popsáno, o jakou část programu se jednotlivé složky starají. Pro zajištění jednodušší správy kódu a oddělení jednotlivých logických celků je vytvořen pro každou část samostatný projekt. I přes to, že se v celém projektu nachází zhruba 140 dílčích projektů, skutečně spustitelné jsou pouze některé, mezi které patří,

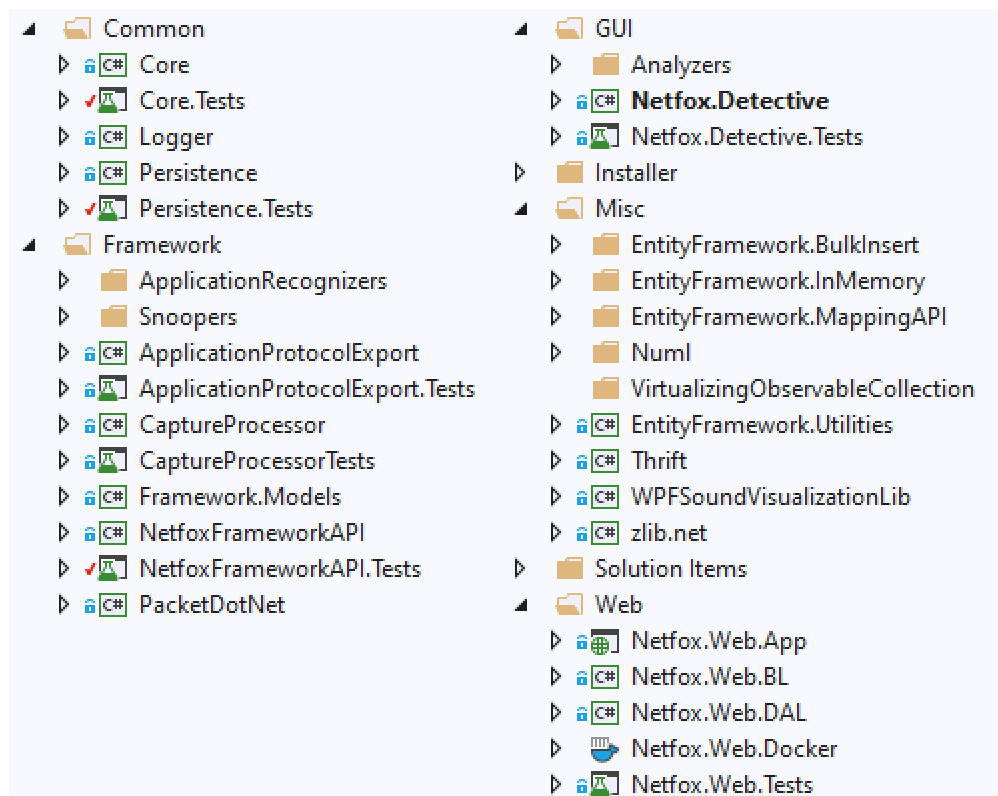
- grafické uživatelské rozhraní aplikace Netfox Detective jako aplikace pro operační systém Windows,
- webová aplikace, kterou je možno spustit v jakémkoliv operačním systému za použití webového prohlížeče,
- některé podpůrné části, které slouží pro rozpoznání komunikující aplikace.

Common

Modul Common obsahuje základní třídy používané napříč celou aplikací. Nacházejí se zde například výčty, bázové typy některých tříd a rozhraní, která je možné implementovat napříč aplikací. Nejedná se tedy o kód, který je přímo spuštěn, ale o připravenou rozsáhlou knihovnu funkcí, rozhraní a metod, které zaručuje jednotný přístup a především omezení duplicitního kódu. Všechny tyto třídy se nachází v modulu `Core`.

Dále se zde nachází modul `Logger`, který se stará o zpracování a bezpečné ukládání všech záznamů a událostí, které v systému nastanou. Jednotlivé události se skládají z úrovně závažnosti, času a popisu. Ukládají se v souborech pojmenovanými časovým razítkem.

Poslední částí je modul `Persistence`, jehož úkolem je komunikace s databázovým strojem na nejnižší úrovni. Třídy tohoto modulu představují jednotlivé databázové kontexty, které lze chápat jako předpisy pro komunikaci s danou databází. V tuto chvíli jsou implementovány přístupy pro *Microsoft SQL* a takzvaný „InMemory“ databázový přístup, který data po ukončení aplikace smaže.



Obrázek 4.2: Struktura projektu v programu Visual Studio 2019

Framework

V této složce se nachází samotné jádro celé aplikace. Mezi nejdůležitější komponenty se řadí:

- NetfoxFrameworkAPI, jenž se stará o komunikaci s grafický uživatelským rozhraním a je vstupní komponentou pro komunikaci s celým frameworkem,
- CaptureProcessor, který analyzuje vstupní datový soubor a rozkládá jej na jednotlivé části, které ukládá do databáze,
- ApplicationProtocolExport, který je na konci řetězce a zpracovává výstupní zprávu o tom, co bylo v rámci analýzy zjištěno,
- ApplicationRecognizer se stará o detekci aplikace, která komunikaci vyvolala.

Poslední zmíněný modul je nejdůležitější částí celého produktu. Na jeho základě se vybírá další modul, který z dané komunikace již extrahuje data. Pokud by tedy ApplicationRecognizer zvolil špatný aplikační protokol, výstup aplikace bude nepravdivý, nebo žádný.

Mezi další podpůrné moduly se řadí PacketDotNet, využívaný pro zpracování jednotlivých rámců, a velká řada podmodulů ve složce `Snoopers`. Každý z těchto malých modulů se stará o zpracování jednotlivých aplikačních protokolů, které jsou vypsány v úvodu této kapitoly. Každý ze „snooperů“ obsahuje vlastní modely, které využívá, implementaci rozhraní, skrze které s nimi komunikuje framework a také předpisy pro zobrazení v klientské a

webové aplikaci. Každý z nich je oddělen v samostatném projektu, což zjednodušuje přidání nových aplikačních protokolů jejich pouhým přeložením a vložením do složky s aplikací.

GUI

Složka s grafickým uživatelským rozhraním obsahuje jediný projekt, který zpracovává komunikaci mezi uživatelem a frameworkem. Implementuje celou uživatelskou aplikaci napsanou za použití WPF (Windows Presentation Foundation) a na obrázku 4.1 představuje horní část, tedy Netfox Detective. Uživatel skrze toto rozhraní může vkládat nové soubory, spouštět analýzu a číst výstupy zpracování.

Misc

Tento modul, který zkráceně přestavuje výraz „Miscellaneous“ (neboli česky „smíšený“), obsahuje pomocné funkcionality pro přístup k databázi. Nacházejí se zde například pomocné třídy pro hromadné vkládání dat do databáze, a některá rozšíření systému Entity Framework. Dále se zde nachází modul pro analýzu pomocí různých matematických modelů využívaných pro rozpoznání jednotlivých aplikací.

Web

Posledním důležitým modulem, který se v rámci tohoto projektu nachází je samotná webová aplikace, která přináší na rozdíl od desktopové verze například možnost oddělení více uživatelů. Zároveň je zde možné mluvit o zavedení multiplatformního prostředí, protože ke spuštění stačí pouze webový prohlížeč, který je dostupný na všech používaných operačních systémech. V rámci tohoto modulu je webová aplikace rozdělena do tří logických celků, přístup k datům (DAL, z anglického „Data Access Layer“), byznys vrstva (BL, z anglického „Business logic“) a aplikační vrstva obsahující samotnou webovou aplikaci napsanou za použiší DotVVM (popdáno v podsekci 4.1.4).

4.2.1 Využití návrhových vzorů

Všechny vzory, které byly popsány v kapitole 3.3 jsou již v rámci kódu implementovány. Součástí migrace kódu bude kontrola, zda je jejich využití správné a případně bude jejich využití rozšířeno.

Dependency injection

Vkládání závislostí, jak je česky tento vzor označován, je v tomto projektu implementováno využitím frameworku *Castle Windsor Framework*[9]. Využití tohoto frameworku značně usnadňuje vývoj samotné aplikace, protože je možné jej také využít pro implementaci návrhových vzorů factory, či singleton.

Při startu aplikace je nutné nejdříve inicializovat celý kontejner vytvořením nové instance `new WindsorContainer(<Název kontejneru>, <další atributy>)`. Dále do něj můžeme pomocí metody `Register` vkládat nové komponenty. Ve chvíli, kdy nějaký kód žádá o určitou komponentu, může tak učinit pomocí metody `Resolve`, které je předán požadovaný datový typ.

[9] http://www.castleproject.org/projects/windsor/

V rámci tohoto projektu je využíván na všech úrovních Netfox Detective. Všechny závislosti, které mají být prostřednictvím kontejneru nalezeny, mají připravené metody, které implementují rozhraní `IWindsorInstaller`. Díky tomuto rozhraní musejí implementovat právě metodu `Register`, která je do daného kontejneru zavede.

Model-View-ViewModel

Tato architektura, která je popsána v podsekci 3.3.3, je využita k tvorbě grafického uživatelského rozhraní. Jednotlivé view zde představují soubory s příponou `.xaml` a definují vzhled samotného okna, nebo jeho části. Napojení dat na nižší vrstvu (ViewModel), je definováno pomocí klíčového slova `Binding`, jako je to naznačeno ve výpisu 4.1. Na jakou konkrétní třídu se dané mapování provádí je pak definováno v parametru `DataContext`. Výhodným pomocníkem pak může být využití takzvaných `Converter` tříd, kterými lze definovat například dodatečný styl, jak bude výsledný text vypadat.

```
1  <views:DetectiveDataEntityPaneViewBase
2   d:DataContext="{d:DesignInstance conversations:ReassembledStreamDetailVm}"
3   ... >
4      ...
5      <TextBox IsReadOnly="True" BorderThickness="0"
6        Text="{Binding Path=HexValue, Mode=OneWay}"
7        Foreground="{Binding Path=FlowDirection,
8        Converter={StaticResource PDUDirectionToColorConverter}}"
9        FontFamily="Courier new" />
10      ...
11 </views:DetectiveDataEntityPaneViewBase>
```

Výpis 4.1: Příklad mapování

Samotný ViewModel pak představuje třídu uchovávající samotný stav daného okna. Pro správné fungování automatické propagace nových dat je nutné využívat místo klasických kolekcí kolekce doplněné o možnost informování o změnách. Příkladem může být využití `ConcurrentObservableCollection<Datový typ>`. Dalším článkem jsou pak samotné modely, které zde představují jednotlivé datové jednotky (například konkrétní úkol, který se zpracovává na pozadí).

4.3 Testování

Testování je velmi důležitou součástí jakéhokoliv produktu. Tato sekce se zaměřuje na zhodnocení aktuálního stavu takzvaných jednotkových testů (v anglické literatuře jako „Unit tests“). Součástí této sekce bude také navrženo, jaké testy je třeba doplnit, případně upravit.

4.3.1 Rozložení testů

Celá aplikace se v současné chvíli skládá ze 140 projektů, které jsou rozděleny na nejvyšší úrovni do skupin, které jsou popsány v sekci 4.2. Dohromady je implementováno napříč

projektem 1010 testů[10]. K testování bylo využito modulu *NUnit*, který patří mezi nejpoužívanější prostředí pro testování.

Rozdělení testů do jednotlivých modulů je rozepsáno v následujících odrážkách.

- Common obsahuje dohromady 11 testů,
  - Core (6 testů)
  - Logger (0 testů)
  - Persistence (5 testů)
- Framework implementuje celkem 596 testů,
  - ApplicationProtocolExport (20 testů)
  - CaptureProcessor (25 testů)
  - NetfoxFrameworkAPI (1 test)
  - ApplicationRecognizers (196 testů)
  - Snoopers (354 testů)
- GUI zpracovává 80 testů,
  - Netfox.Detective (34 testů)
  - Analyzers (46 testů)
- Misc obsahuje celkem 304 testů,
  - EntityFrameowrk.BulkInsert (44 testů)
  - EntityFramework.InMemory (8 testů)
  - EntityFramework.MappingAPI (22 testů)
  - Numl (230 testů)
- Web implementuje dohromady 19 testů,
  - Netfox.Web (19 testů)

Ze seznamu výše je velmi patrné, že největší část testů připadá na matematický modul Numl a identifikační mechanizmus ApplicationRecognizer. I přes to, že složka Snoopers obsahuje dohromady 354 testů, v průměru se jedná o zhruba 16 testů na jeden modul. Naopak při prvním pohledu chybí testy především v rámci modulu Logger a NetfoxFrameworkAPI.

Nachází se zde ale i několik modulů, které testy vůbec nemají. Příkladem může být Framework.Models, který obsahuje v drtivé většině pouze sdílené modely, které se používají napříč celou aplikací. Protože se jedná pouze o definice tříd často bez aktivního kódu, není ani možné testy pro tyto třídy psát. Naopak některé části projektu mají velké množství testů, příkladem je modul Numl, který se využívá ke strojovému učení.

V rámci tohoto projektu se nachází také několik testů, které slouží pouze jako generátory konfiguračních souborů. Tyto soubory jsou pak mapovány do projektu a představují cesty k jednotlivým testovým souborům, které testy využívají. I přes to, že generování souborů nebrání použití, nejedná se o čistou praktiku pro tvorbu testů. Neověřují totiž žádnou funkčnost.

[10] Všechna čísla jsou vázána ke commitu `4cdbd0d160edf7fb35706d0858e4cf4892383601` větve `develop` z 18.10.2019.

4.3.2 Zhodnocení testů

Hodnocení kvality testů z hlediska pokrytí kódu může být velmi zavádějící. I přes to, že můžeme mít pokrytí kódu testy na 100%, nemusí to znamenat, že jsou testy správně napsané a testují všechny skutečnosti, které mohou v reálném světě nastat. Testovat není třeba pouze hodnoty, které mají bez problému systémem projít, ale také ty, které má systém umět vyloučit.

V rámci tohoto projektu je napsáno velké množství různých testů, některé nejsou funkční z různých důvodu. První překážkou, která znemožňuje spustit některé testy je závislost na konkrétních verzích balíčků kódu, které nejsou k dispozici na žádném ze standardně dostupných repositářů. To může nastat například ve chvíli, kdy se k vývoji využívá vlastní modifikace některého ze standardně dostupných balíčků, která je uložena v soukromém repositáři.

Druhou komplikací, která v některých testech nastává je využívání explicitních cest k testovacím souborům přímo v kódu. Tyto testy jsou ale označeny příznakem `Explicit`, který zajišťuje, že nebude test spuštěn se všemi ostatními a je třeba jej spustit samostatně. Nedochází tak k falešně negativním vyhodnocení, který by mohl zkreslovat celkový výsledek. Naopak se také v projektu nachází testy, které jsou označeny tímto příznakem, nicméně při spouštění těchto testů se ukázalo, že tento příznak zde být nemusí. Obdobně tak je v některých testech využívána pro testování přímá url adresa. Příkladem tak může být kód 4.2.

```
1 [Test]
2 public async Task GetJsonMessageTest()
3 {
4     using (var proxy = new NemeaProxy())
5     {
6         await proxy.Connect(new Uri("telnet://172.16.0.1:9999"));
7         var msg = proxy.GetJsonMessage();
8     }
9 }
```

Výpis 4.2: Test, který využívá konkrétní URL adresu

Před refaktorizací je nutné všechny nefunkční testy opravit, v případě chybějících podkladových souborů, odstranit. Mělo by tak před migrací dojít ke stoprocentní úspěšnosti testů. Zároveň budou všechny testy znovu použity také v rámci nové verze celé aplikace Netfox Detective.

4.3.3 Pokrytí kódu testy

Samotné hodnoty pokrytí kódu nemohou být v tuto chvíli brány jako dogma, které platí, neboť je velká část testů, které z různých důvodů neprochází. Mezi tyto důvody také patří fakt, že některé testy nevyužívají přímé reference na dané části kódu, ale nechávají si instance potřebných tříd dodávat pomocí injekce závislostí. Určitý odhad se dá ale udělat i z testů, které jsou funkční a vracejí kladný výstup. Níže jsou rozepsány jednotlivé složky včetně projektů, které obsahují, a jejich současné hodnoty pokrytí testy[11]. Zajímavostí také

[11] K vyhodnocování pokrytí testy bylo využito výchozích nástrojů v aplikaci JetBrains Rider ve verzi 2020.3.2.

je, že se v hodnotách pokrytí testů nachází samotné testovací moduly. Dochází k tomu z důvodu, že jsou napříč testy využívány jiné části testovacích tříd, proto budou v následujícím rozpisu vynechány.

- **Common** obsahuje především moduly `Core` a `Persistence`, kde `Core` vykazuje pokrytí zhruba 20% (při rozptylu[12] 13% - 55%) a `Persistence` má hodnotu 0%. Nulová hodnota je zde z důvodu, že se jedná o speciální modul, který se stará o ukládání dat. Části kódu, které se těmito testy kontrolují, jsou testu předávány jako již existující reference na danou instanci třídy. Z tohoto pohledu se jedná o nepřímé testovaní ve smyslu, že si každý test nevytváří přímo svou instanci. Ve skutečnosti jej ale pokrývá 5 testů, které všechny skončí bez chyby.

- Složka **Framework** obsahuje především moduly `Appident`, který identifikuje dané aplikační protokoly, `NetfoxFrameworkAPI`, jakožto rozhraní mezi uživatelskou aplikací a byznys logikou, `CaptureProcessor` pro zpracování balíku zachycené komunikace a jednotlivé analyzátory protokolů (Snoopers). Až na poslední zmíněný hlásí všechny moduly zhruba 50% pokrytí. Jednotlivé Snooper třídy pak mají chybně vypočtené pokrytí (aplikace zobrazuje 0%), nicméně u drtivé většiny procházejí všechny testy.

- **GUI** se skládá z modulů `NetfoxDetective` a `Analyzers`. Společně tak tvoří kompletní grafické uživatelské rozhraní. Testovány jsou zde některé funkce, které vykonávají práci (například práce se složkami na disku), nebo analyzují určitý aplikační protokol. Samotné hodnoty pokrytí se proto pohybují v rozmezí 1% - 3%. Testy grafického uživatelského rozhraní jako takového zavedeny nejsou.

- Poslední složka **Misc** pak obsahuje rozšíření modulu `EntityFramework` a další moduly, které se využívají pro identifikaci správného protokolu. Zmíněný matematický modul `Numl` je pokryt dle analýzy z 19%, rozšiřující moduly `EntityFramework` pak z 20% a 57%. Opět se ale díky využití v jiných částech produktu může jednat o zkreslené hodnoty, protože dané funkce mohou být využity i v zcela nesouvisejících testech.

Z analýzy výše vyplývá, že průměrné pokrytí testy se pohybuje mezi 10% až 40% kódu. Bohužel díky neoddělenému testování jednotlivých částí tak dochází k „míchání“ více částí a testů dohromady a samotné výsledky jsou tak silně zkreslené. Zabránění tak vysokému provázání je možné, pokud nebudou testy jednotlivých částí kódu využívat pro svůj běh moduly jiné. Tento fakt je také zohledněn v návrhu v podkapitole 4.5.4.

4.3.4 Oddělení jednotlivých testů

Pro většinu testů je využito návrhového vzoru Fixture, který je popsaný v podsekci 3.3.4. Pro každý test, který této metody využívá je vygenerován náhodný identifikátor, který odděluje jednotlivé prostředí všech testů. Každý z testů má při spuštění všechny svá potřebná data na disku `C` ve složce `NetfoxTemp` pod svým vlastním identifikátorem. Tento zvolený návrh je funkční především z hlediska oddělení jednotlivých testů a možnosti, aby byly testy spuštěny paralelně bez toho, aby se navzájem ovlivňovali. Jeden průchod testy zabírá až několik stovek MB, je tedy na to třeba brát zřetel a po každém použití dočasnou složku vyčistit, v ideálním případě zařadit do testů retenční mechanizmus, který bude ponechávat pouze poslední 2 běhy testů a starší smaže.

[12] Rozptyl říká, v jakých hodnotách se pohybuje pokrytí jednotlivých souborů v daném modulu.

4.4 Kritické části projektu

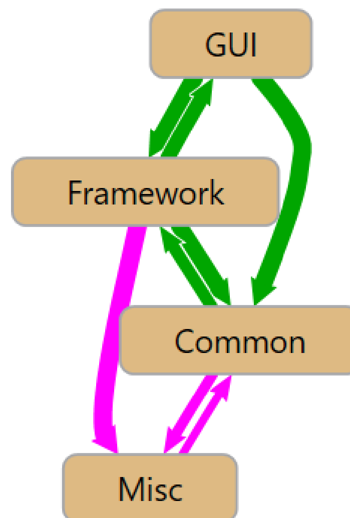
V rámci procházení jednotlivých částí projektu se vyskytují různé zvláštnosti, které by v dokončeném dílu určitě měly být vyřešeny. V následujících podkapitolách jsou tyto zvláštnosti sepsány včetně možného řešení.

4.4.1 Závislosti v rámci projektu

Při tvorbě rozsáhlých projektů se velmi často využívá vrstevnatého modelu, který odděluje jednotlivé logické celky kódu. Nejčastěji se volí následující model.

- Kód se samotnou uživatelskou vrstvou. Řadí se sem především část s webovou aplikací, případně spustitelná nativní aplikace pro daný operační systém. Jejím cílem je získávání požadavků od uživatele a prezentace výsledků zpět.
- Datově zaměřená vrstva (z anglického DAL, „Data Access Layer“) má za úkol ukládání dat do perzistentní paměti a jejich čtení. Často se v rámci této vrstvy implementuje i takzvané objektově-relační mapování, které zajišťuje konverzi dat právě mezi perzistentní pamětí a kódem programovacího jazyka.
- Byznys vrstva (anglicky označovaná jako „Business logic“) se stará o aplikační logiku. Jejím úkolem je abstrahování fungování reálného světa do kódu a stará se tak o správný chod a řízení napříč celou aplikací. V rámci celého modelu se nachází uprostřed mezi aplikační a datově orientovanou vrstvou a navzájem je propojuje [40].

Netfox Detective tohoto přístupu také určitým způsobem využívá. Na obrázku 4.3 jsou vidět jednotlivé složky, kde složka GUI obsahuje vrstvu uživatelské aplikace a složka Framework byznys logiku. Poslední dvě složky Misc a Common z části tvoří přímo datovou vrstvu ve spojení s podpůrnými funkcemi, které se využívají napříč projektem[13].



Obrázek 4.3: Závislosti v kódu na úrovni jednotlivých složek

[13]Pro zjednodušení byla vypuštěna složka obsahující webovou aplikaci, která by v grafu byla zahrnuta do části GUI.

Na obrázku 4.3 je možné vidět závislosti kódu mezi různými moduly. Růžové šipky značí především volání funkcí a metod, zelené pak doplňují volání o implementaci rozhraní, případně dědění z rodičovských tříd. V grafu lze vidět, že mezi moduly `GUI` a `Framework` dochází k obousměrné závislosti. Je tomu tak z důvodu, že jednotlivé Snoopery, tedy moduly, které analyzují a zpracovávají přenosy jednotlivých aplikačních protokolů, obsahují testy, které které využívají testovacího Fixture modulu `Netfox.Detective.Test`.

Odstranění obousměrné závislosti je možné rozdělením jednotlivých Snooper modulů do dvou části (uživatelské rozhraní a modul zpracovávající data), oddělení testovacího prostředí, které je využíváno v GUI, nebo oddělení samotných Snooper modulů ze složky `Framework`. Protože je jedním z požadavků možnost rozšiřitelnosti funkcí nezávisle na samotné aplikaci, nejlepší volbou je poslední možnost. Podrobnější návrh nového rozdělení jednotlivých modulů je popsán v sekci o přípravě k implementaci 4.5.1.

4.4.2 Chybějící kód

Při vývoji jakéhokoliv softwarového díla se můžeme setkat s využíváním rozhraní (z anglického „interface“). Rozhraní nám určuje, jakým způsobem s daným objektem můžeme pracovat, tedy jaké metody jsou pro danou třídu veřejné. Při použití rozhraní, které obsahuje více metod je třeba vždy implementovat všechny metody, aby bylo možné projekt celý přeložit a případně spustit, nebo otestovat. Abychom nemuseli vždy kompletně implementovat celé rozhraní, pokud chceme například otestovat pouze část, můžeme každou metodu nechat s prázdným tělem. Protože by tak ale mohlo docházet k opomenutí některých implementací, využívá se zpravidla vyvolání výjimky, které způsobí pád celé aplikace a vývojáři tak připomene, že někde nechal část kódu nevyplněnou.

V rámci celého .NET Framework i .NET Core je na to připraven přímo speciální typ výjimky, `NotImplementedException`, která se využívá právě v těchto případech. Může se ale stát, že vývojář na danou výjimku, tedy i implementaci samotné metody, zapomene a dostane se do produkční verze. V rámci celé aplikace Netfox Detective, včetně části Netfox Framework je možné nalézt zhruba 60 případů použití této výjimky. Je ale nutno podotknout, že v některých případech se jedná o části kódu, které nejsou nikde využívány, jedná se tak o „mrtvý kód“, který existuje například z důvodu implementace určitého rozhraní. Podle principů SOLID, které jsou popsány v podsekci 3.3.1, se tak jedná o možné porušení 4. principu, oddělení rozhraní. I přes možné porušení tohoto pravidla může být výše uvedené z různých důvodů účelné.

Příkladem takového použití může být soubor `EFDataReader.cs`, který se nachází v modulu `EntityFramework.Utilities`, kde je řada těchto metod obalena klauzulí `Region` a další vývojáři tak vědí, že se jedná o záměr. Naopak tomu je například v souboru `YahooGetListedMessages.cs` nacházejícím se v modulu `SnooperWebmails`, kde jsou některé metody implementovány a jiné pouze tvoří výjimku bez podrobnějšího popisu, z jakého důvodu se tam takový kód nachází.

Další možností chybějícího kódu je prázdný blok kódu, tedy použití složených závorek bez obsahu `{ }`[14]. Prázdného bloku kódu se může využívat například při nutnosti implementace konstruktoru, který však nedělá nic jiného, než volání konstruktoru nadtřídy. V tomto projektu se pak vyskytuje takový kód například v souboru `PmFramePcap.cs`, jehož část je ve výpisu 4.3. V tomto případě se nejdříve vykoná konstruktor rodičovské třídy, ihned poté pak kód samotné podtřídy. Další možností je využití v rámci návrhového vzoru Fixture (viz 3.3.4), kdy v testu očekáváme určitou metodu, ale v rámci testů nepotřebujeme, aby cokoliv

[14]Pomocí regulárního výrazu je možné takový kód najít pomocí `\{ \s \}`.

vykonávala. Takový kód se nachází například v testech `SnooperTwitter` a je vyobrazený ve výpisu 4.4.

```
1  public class PmFramePcap : PmFrameBase
2  {
3      public PmFramePcap() : base() { }
4
5      // rest of this class...
6  }
```

Výpis 4.3: Volání rodičovského konstruktoru bez dalších akcí

```
1  [TestFixture]
2  public class SnooperTwitterWindsorInstallerTests :
3      WindsorInvestigationInstallerTestsBase<SnooperTwitterWindsorInstaller>
4  {
5      protected override void AssertIEntityViewModels() { }
6      protected override void AssertServices() { }
7      protected override void AssertViews() { }
8
9      // rest of this classs...
10 }
```

Výpis 4.4: Příklad prázdné metody z důvodu jejich volání v testech

Oba výše zobrazené případy jsou v souladu se zmíněnými metodikami a není třeba je upravovat. V rámci refaktorizace se však budu snažit co nejvíce tyto části eliminovat a pokud možno, doplnit funkčním kódem, případně upravit tak, aby nebylo v testech výše zmíněné omezením. Ostatní problematické části budou odstraněny, doplněny funkčním kódem, případně komentářem, který bude vysvětlovat, proč zde není implementován kód. Je tak rozhodnuto z důvodu, aby za sebe byl sám kód včetně komentářů jasně vypovídající pro další vývojáře a nedocházelo tak ke zmatení, případně k pozdějšímu volání těchto metod.

4.4.3 Komentáře v kódu

Komentáře a jejich využívání v kódu nemusí být vždy přímočaré. Podrobněji je rozebráno použití komentářů v sekci 2.1. V ideálním případě by měla být anotována každá funkce každé třídy, nebo alespoň všechny veřejně přístupné funkce. Každá anotace by měla obsahovat popis toho, co daná část kódu dělá a definici všech vstupních parametrů včetně návratové hodnoty [39]. Za úplnou anotaci se dá považovat anotace ze souboru `NetworkAddress.cs` v modulu `PacketDotNet.LLDP` zobrazenou ve výpisu 4.5. V rámci této práce bude kladen důraz na to, aby bylo v ideálním případě vše řádně anotováno, protože i samotné vytváření anotací funkcí a tříd může poukazovat na to, že je porušován některý z principů[15], typicky první SOLID princip, popsaný v podsekci 3.3.1.

[15]Typicky tak může poukazovat název funkce, nejčastěji tak spojka „a“ nasvědčuje tomu, že se daná třída/metoda stará o více věcí najednou.

```
1  /// <summary>
2  /// Equals override
3  /// </summary>
4  /// <param name="obj">
5  /// A <see cref="System.Object" />
6  /// </param>
7  /// <returns>
8  /// A <see cref="System.Boolean" />
9  /// </returns>
10 public override bool Equals(object obj)
11 { ... }
```

Výpis 4.5: Příklad dostatečně vyplněné anotace funkce

V rámci dlouhodobějšího vývoje kódu nastávají chvíle, kdy je kód třeba upravit, nebo některou z částí kompletně přepsat. Pokud tak nastane, můžeme si zakomentovat část kódu, kterou je třeba změnit, například z důvodu zachování původní myšlenky. Ve chvíli kdy máme nový kód sepsaný a řádně otestovaný, měli bychom původní kód v komentáři smazat. Příkladem může být soubor `SnooperXMPP.cs` v modulu `SnooperSMPP` ve výpisu 4.6. Při využívání funkcí programu Git tak o něj nepřijdeme, protože bude stále k nalezení v předchozích verzích daného souboru. V případě, že předchozí kód potřebujeme ponechat, aby se jej nikdo nepokoušel přepsat do původní podoby, měli bychom jej doplnit komentářem s tím, proč tomu tak je.

```
1  protected override void RunBody()
2  {
3      Debug.WriteLine(@"SnooperXMPP.RunBody() called");
4      base.ProcessAssignedConversations();
5      //this.SelectedConversations.LockSelectedConversations();
6      // ... and more than 30 lines of commented code
7  }
```

Výpis 4.6: Příklad dlouhé části kódu, který je v komentáři

Posledním typem komentářů, který se v tomto celém projektu nachází souvisí využívání speciální výjimky typu `NotImplementedException`. Moderní integrovaná vývojová prostředí nám nabízejí mimo napovídání, syntaktické a sémantické kontroly, také menší doplňky, které mohou vývojářům zjednodušit a zpřehlednit práci. Jedním z takových vylepšení je analýza zdrojových souborů s ohledem na využívání klíčových slov `TODO`, případně `BUG <bug_id>` (kde hodnota `bug_id` odkazuje na identifikátor chyby v systému pro řízení vývoje). Do popisu se také přidává jméno autora a datum přidání. Příklad takového použití přímo v Netfox Detective je vidět ve výpisu 4.7, které odkazuje na opravení nevhodně použitého kódu.

Použití těchto klíčových slov v komentářích lze jednoduše vyhledávat. V případě, že by vývojář zapomněl nějakou část kódu, která není dokončená, ale je takto označena, může jí před odevzdáním doplnit. Zamezí tak případným zbytečným chybám dále v kódu. S tímto bodem také úzce souvisí jedna z částí podkapitoly o statické analýze kódu (2.3.1), která doporučuje před začleněním nové části kódu, využití více lidí ke kontrole (v praxi označováno jako „Code review“).

```
public int IndexOf(T item)
{
    //TODO fix .AsEnumerable() workaround
    return this.ConcurrentLock(() => this.Query.AsEnumerable()
       .TakeWhile(i => i.Id != item.Id).Count());
}
```

Výpis 4.7: Využití klíčového slova TODO v poznámce

4.4.4 Uživatelská zkušenost

V rámci spuštěné aplikace Netfox Detective jsem se setkal s momenty, kdy aplikace prováděla nějakou činnost, ale její celé rozhraní bylo zaseknuté. Na základě diplomové práce [37], která se věnovala tvorbě samotného grafického uživatelského rozhraní, mají všechny úkoly být vykonávány na pozadí za využití modulu `TaskManager`. Tento modul má v projektu také své vlastní okno, které ukazuje všechny právě běžící úkoly a je zde možné vidět celou historii jednotlivých úkolů, případně vybrané běžící úkoly zastavit.

V migrované aplikaci bude nutné dbát na to, aby každý vývojář, který bude přispívat svým kódem, tohoto zavedeného mechanizmu využíval. Z hlediska uživatelské zkušenosti je zamrzání aplikace neakceptovatelné, neboť uživatel nabývá pocitu ztracené kontroly nad tím, co se přesně děje.

4.4.5 Nesouvisející moduly

V celém projektu se nachází několik různých modulů, které přímo nesouvisí s danou implementací celé aplikace. Mezi takové patří například `Numl`, `VirtualizingCollection`, `Thrift`, `zlib.net`, případně vlastní rozšíření pro `EntityFramework`. I přes to, že jsou tyto moduly nutné ke správnému běhu programu, není nutné, aby byl jejich zdrojový kód přímo součástí tohoto projektu. Postup včetně návrhů začlenění je popsán v podsekci 4.5.2.

4.5 Příprava pro implementační část

V této sekci rozeberu na základě předchozích podkapitol důvody a způsoby, jak bude pokračovat tato práce. Zhodnotím jednotlivé možné přístupy a odůvodním, proč byly dané postupy zvoleny.

Vzhledem k robustnosti a provázanosti jednotlivých částí projektu je rozhodnuto, že dojde k vytvoření nového repositáře, do kterého bude projekt postupně přepisován. Mezi hlavní důvody patří především jednoduchost v odstranění nepotřebných modulů a nepoužívaného kódu. Zvyšuje se také možnost, že budou odhaleny jiné problematické úseky kódu, které bude snazší upravit, nebo odstranit. V následujících podsekcích jsou rozepsány nejdůležitější body, na které bude v implementaci brán zřetel, včetně samotného návrhu nového rozdělení modulů.

Při migraci nebude zasahováno do základních myšlenek, na kterých je projekt postavený, ani nebude změněna žádná podstatná část stěžejních funkcí projektu.

4.5.1 Návrh jednotlivých modulů

V rámci nového řešení by se tedy měly objevit následující moduly, které respektují původní architekturu, která je definována v původní diplomové práci *Nástroj pro analýzu obsahu síťové komunikace* [37].

- Nejnižší vrstvou je jádro aplikace, které implementuje univerzální rozhraní pro komunikaci mezi jednotlivými komponentami. Tato vrstva také již z původního návrhu implementuje jednotný prostor pro ukládání a načítání všech aplikačních nastavení. Společně vedle tohoto modulu leží také vrstva *DataPersistence*, která zajišťuje komunikaci s jednotlivými typy databázových strojů. Tyto části se nachází v modulu `Common`. Z této části mohou čerpat všechny vyšší vrstvy.

- Další vrstvou je samotná aplikační logika, kterou v rozdělení na obrázku 4.1 představuje dolní část, Netfox Framework. Zde probíhá zpracování zadaných souborů, jejich konverze a postupné předávání Snooper třídám. Tyto třídy mají na úkol analýzu protokolu na aplikační úrovni a zpětné předání výsledků prostřednictvím API do grafického uživatelského rozhraní.

- Nepostradatelnou částí je pak již zmíněný balík všech Snooper tříd. Zde se budou nacházet všechny implementované třídy dodávané jako součást tohoto produktu. Každá taková třída by měla implementovat rozhraní pro WPF aplikaci, webovou aplikaci, testy potvrzující její funkčnost a také její samotnou implementaci.

- Nejvyšší vrstvou je grafické uživatelské rozhraní, které se bude skládat ze dvou částí. Tyto části budou s Netfox Framework komunikovat prostřednictvím již existujícího připraveného `NetfoxFameworkAPI`.

  1. Uživatelská aplikace, napsaná pomocí knihovny WPF, pro možnost spustit aplikaci nativně v rámci operačního systému Windows.
  2. Webová aplikace, která je v současné chvíli oddělená.

4.5.2 Odstranění nesouvisejících balíčků

V celém řešení tohoto produktu se nachází několik modulů, které přímo nesouvisí s implementací Netfox Detective. Jedná se tak například o matematický balíček `Numl`, využívaný ke strojovému učení. Tento balíček, jako některé další, nejsou vyvíjeny v rámci tohoto nástroje a začlenění jejich zdrojových kódu zde nedává smysl. Všechny nalezené části budou buď,

- pouze odstraněny a případné závislosti na těchto balíčkách budou doplněny například z výchozího repositáře NuGet,

- nebo přesunuty do nového repositáře, který je připraven pro tento projekt a popsán v implementační části této práce.

V případě, že některý z těchto balíčků je pro potřeby tohoto projektu rozšiřován, bude veškerá jeho dodatečná implementace v samostatném projektu rozšířena například za využití návrhového vzoru *facade*[16], nebo může být využito rozšíření kódu pomocí *Extension*

[16] `https://refactoring.guru/design-patterns/facade`

method[17], případě jinou vhodnou formou. Cílem tohoto bodu je odstínění implementace jednotlivých částí tak, aby v případě menších zásahů do jedné části, nebylo nutné zasahovat také do jiných částí. Zároveň tak bude možné balíčky jednoduše aktualizovat, v případě oprav některých chyb původními vývojáři.

4.5.3 Práce s výkonným kódem

V sekci *Kritické části projektu* (4.4), jsou rozepsány nejdůležitější body, které bude třeba v rámci nového kódu zohlednit, případně doplnit, nebo opravit. V následujících bodech je jejich shrnutí.

- Odstranění provázání závislostí mezi jednotlivými částmi celého projektu.
- Doplnění, respektive odstranění částí kódu, který není v rámci projektu implementován, respektive využíván.
- Dodržování daných návrhových vzorů. Zejména pak zaměření na principy SOLID, které jsou popsány v podsekci 3.3.1.
- Důkladné popisování funkcí formou anotace, která bude vysvětlovat, k čemu dané metody slouží. V případě potřeby může být doplněno komentářem přímo v kódu, pro úplné a jednoduché pochopení cíle funkce.
- Řádné využívání mechanizmů pro zaručení odezvy aplikace v jakémkoliv stavu (například využití vestavěného správce úloh popsaného v podsekci 4.4.4.
- Dle doporučení, bude maximální snaha o dodržování správných metrik kódu tak, jak je popsáno v sekci 2.2.

Obecně pak kód vyvíjen s ohledem na jednoduchou rozšiřitelnost a testovatelnost. Dodržování výše popsaných bodů bude zajištěno především pomocí metod pro kontrolu kódu, jako je *Code review*, které je popsané v sekci 2.3.

4.5.4 Práce s testy

Z výše popsaných podsekcí, které se věnovaly testům, vyplývá několik důležitých bodů, které je třeba po refaktorizaci zohlednit.

- Je nutné testovat jak hodnoty platné, tak také hodnoty potenciálně nevhodné (neexistující vstupní hodnota, hodnota mimo povolený rozsah).
- Všechna testovací data budou dostupná přímo v rámci samotného repositáře projektu. V rámci testů (ani jiného kódu) se nesmí nacházet absolutní cesta nikam na disk, webovou stránku či jiný zdroj.
- Musí být patrné, co daný test ověřuje a jaký má být výsledek. V případě potřeby by měl být test doplněn komentářem.
- Testovací procedury by neměly vyžadovat žádné další části projektu. Všechny potřebné závislosti pro průběh testů by měly být simulovány testovacími třídami.

[17] https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

- Vlastní testovací prostředí pro každý test, jak je popsáno v podsekci 4.3.4, bude implementováno s definovaným retenčním modelem mazání starých dat testů.

Všechny testy, které budou vyhovovat výše popsaným pravidlům, budou postupně přesouvány v původní podobě do nově vzniklého projektu. Stejně bude naloženo s testy, které podléhají pouze drobným úpravám. Zbylé testovací metody, které zásadně porušují výše popsané principy a pravidla, by měly být buď kompletně přepsány, nebo nahrazeny novými testy, kontrolující stejnou část kódu.

Celý projekt bude také nutné doplnit řadou dalších testů, které kontrolují dosud nepokryté části kódu. Všechny testy tak budou vytvářeny až ve chvíli tvorby kódu v novém projektu postupně s tím, jak bude vznikat.

Kapitola 5

Příprava prostředí a migrace

Základem jakéhokoliv nového projektu je dobře připravené prostředí a navržené metodiky. V následujících sekcích jsou popsány základní principy, na kterých praktická část této diplomové práce stojí, jaké jsou důvody k jejich použití a jejich výhody a případné nevýhody.

V minulých dobách byl software vyvíjen jako monolitický systém tak, že zadavatel předal zhotoviteli zadání a po několika měsících, až letech, dostal zpět hotový produkt. Tomuto modelu se říká vodopádový. Často tak docházelo k tomu, že zadavatel nebyl s výsledkem spokojen z různých důvodů. Mezi nejčastější příčiny patří rozdílná představa o vzhledu a funkčnosti základních komponent zhotoveného systému. Bohužel tak docházelo často k zbytečnému plýtvání penězi i času vývojářů a výsledný produkt nemusel být plně nasazen.

5.1 Agilní metodika vývoje

Největší změnou oproti výše popsanému případu bylo zavedení dnes velmi využívané agilní metodiky vývoje. Jedná se o soubor několika různých pravidel a doporučení, která mají pomoci celému vývojovému týmu, aby dodal produkt přesně takový, jaký si zadavatel představuje. Velké zaměření je především na rychlou reakci na změnu požadavků. Hlavním důvodem, proč jsou dnes tyto metodiky využívány je uspokojení zadavatele průběžnými dodávkami produktu. V průběhu vývoje si tak zákazník ověřuje, že se na produktu pracuje a má zároveň možnost případné neshody s jeho představou konzultovat.

Základním stavebním kamenem je vodopádový model, který se v průběhu vývoje opakuje stále dokola. V nejjednodušší implementaci skládá z pěti částí.

1. **Sběr požadavků** probíhá na samotném začátku každého cyklu. Vývojový tým, či pouze jeho vedoucí, na základě požadavků od zákazníka připravuje plán toho, na čem se bude v následujícím cyklu pracovat a co bude na jeho konci zákazníkovi prezentováno. V této části se také začíná tvořit dokumentace.

2. **Návrh** je postupně seskládáván vývojovým týmem. Cílem je vytvoření modelů a schémat toho, jak budou dané komponenty pracovat.

3. **Implementace** již aplikuje plánované změny do samotného produktu.

4. **Verifikace**, jinak také nazvaná jako testování, má na cíl ověřit, že žádná ze změn nepoškodila již existující kód.

5. **Údržba**, v anglické literatuře označována jako „operations“, sdružuje operace jako: nasazení, migrace dat a případná další operativa kolem již vytvořeného produktu [3].

Tento jednoduchý model se pak na základě domluvy stále opakuje a postupně se tak tvořené dílo přibližuje finální podobě. Nejčastější dobou, po kterou trvá jeden cyklus je 14 dní až tři týdny. Doba však může být přizpůsobena konkrétní metodice. Mezi nejznámější metodiky se řadí tyto.

- Extrémní programování, se zaměřuje na velmi rychlé dodávání kódu. Programuje se pouze to, co vede k přímému naplnění požadavků. Samotný vývoj je také často spojován s párovým programováním. Cílem je především zkrátit vývojový cyklus na co nejnižší možnou dobu [49].
- „Lean software development“ je zaměřen především na odstranění všeho, co není v danou chvíli důležité. Cílem je především snížení nákladů, s čímž souvisí také potlačení některých aktivit, jako je tvorba dokumentace, nebo samotné plánování. Oproti ostatním metodám se zpětná vazba na nové funkcionality získává od zákazníka na základě prototypů, designů, nebo dokonce pouze z drátových modelů[1] [22].
- „Scrum“ je oproti výše popsaným rozdílný především v plánování. Stavebními kameny této metodiky je tvorba několika rolí v rámci týmu. Hlavní dělení spočívá ve zvolení osoby, která bude rozhodovat o dalším dění projektu (product owner), testovací a připomínkovací tým od zákazníka (stakeholders), vývojáře a případně i další role [61].
- „Test driven development“ se v české literatuře objevuje jako testy řízený vývoj. Jedná se o metodiku, která klade velmi velký důraz na tvorbu testů. V praxi se tato metoda aplikuje následujícím způsobem. Nejdříve je vymyšlena nová funkcionalita, poté je vytvořena sada testů, které jsou spuštěny (očekávaný výsledek je, že žádný z nich neprojde), až na závěr je pak implementována funkce jako taková. Protože je hlavním cílem uspokojit úspěšnost testů, je po implementaci dané funkce ještě velmi často její kód refaktorizován [62].

5.2 Podpůrné nástroje

Samotná volba vývojové metodiky nestačí. Na trhu proto existuje několik nástrojů, které vývojářům pomáhají s vývojem, kolaborací s ostatními a s vedením týmu.

5.2.1 Systém udržování verzí

Aby mohl tým vývojářů současně pracovat na určitém dílu produktu najednou, je potřeba aby měli pohodlný způsob jak společný kód sdílet a dále jej zdokonalovat. Zároveň je žádoucí, aby se vývojáři mohli vracet k předešlým verzím kódu. Zachování historie kódu je důležitá součást dokumentace, která pomáhá vývojářům pochopit kontext funkcionalit. Nejčastěji používaným nástrojem je systém Git.

Historie verzí kódu

Jak už bylo zmíněno výše, Git umožňuje vývojářům nahlédnout do historie kódu, kdykoliv potřebují. Historie je tvořena takzvanými „commity“, tedy body, kdy byl uložen daný otisk kódu. Mezi těmito body je možné se přesouvat, v případě potřeba je i upravovat.

[1] Drátový model je nástroj, který je využíván k definici toho, jak budou na stránce jednotlivé funkční prvky umístěny. Naopak neslouží jako nástroj pro prezentaci grafické části, zpravidla se nevyužívá široké palety barev.

Sdílení kódu

Důležitá je také možnost sdílení kódu. Každý vývojář v průběhu času tvoří vlastní otisky. V případě, že dva vývojáři chtějí uložit nový otisk kódu, kde byly upraveny stejné soubory, dochází ke konfliktu, který je třeba vyřešit sloučením, nebo přepsáním jednoho z otisků. Sdílení kódu samotný vývoj signifikantně zjednodušuje.

Větve

V anglické literatuře označované jako „branches“, tedy větve, umožňují oddělovat jednotlivé části otisků do separátních sekcí a tím přímo navádí k ucelené implementaci jednotlivých funkcionalit. Postupně tak dochází k větvení, které následně při slučování větví umožňuje řešit vzniklé konflikty na jednom místě a to až v momentě sloučení těchto větví [11]. Více o nastavení jednotlivých větví je popsáno v podsekci 5.2.4.

5.2.2 Úkoly a problémy

Mimo samotné udržování kódu nabízí většina systémů také další podpůrné nástroje pro vývoj a řešení problémů. Nejčastěji je tak implementován formou lístkového systému, kdy jednotlivé lístečky představují úkoly[2]. Ty se pak dále třídí do aktuálních stavů na tabuli (anglicky nazývané „board“), jenž je rozdělena do několika částí. Nejčastěji se využívají následující stavy.

- **Nový**, tedy nově vložený úkol, který ještě nebyl zpracován.
- **Plánovaný** úkol již někdo analyzoval, a zařadil do fronty k řešení, kde čeká na vývojáře, až na něm začne pracovat.
- Úkol označený jako **v řešení** je právě zpracováván členem týmu. Je tak označen proto, aby na něm nezačalo pracovat více vývojářů najednou.
- V další fázi je kód **revidován** a čeká na zhodnocení od jiného vývojáře.
- V předposlední fázi je funkcionalita **v testu**, kde je ověřena správnost implementace.
- Jakmile kód prošel kontrolou a testem, je zapojen do hlavní vývojářské větve a je tak považován za **dokončený**.

Tvorba jednotlivých kategorií závisí vždy na daném týmu. Pro jednoho samotného vývojáře mohou postačovat pouze dva až tři stavy, větší týmy mohou mít stavů i více [27].

Mimo samotné stavy, jenž definují životní cyklus daného úkolu, jsou také úkoly děleny do kategorií. Nejčastěji se využívá forma označení (anglicky „tag“), které mezi sebou lze kombinovat. Nejčastěji se označení využívá k definování následujících faktů.

- Označení části kódu, do které daný problém patří - backend, frontend, dokumentace, testování.
- Zda se jedná o nalezenou chybu, nebo vývoj nové vlastnosti.
- Označení je možné využít k definování priority.

[2] Jedná se zpravidla o nové požadavky, nahlášené chyby, testy, případně další úlohy spojené s vývojem.

- Případně je možné zavést další metadata, jako například označení duplicitní chyby, nebo vznesení dotazu.

V poslední fázi jsou jednotlivé úkoly přiřazovány vývojáři, nebo skupině vývojářů, aby za daný úkol byl vždy někdo zodpovědný.

Při volbě kolaboračního systému je nutné znát potřeby týmu a projektu. Na trhu je velké množství nástrojů. Každý z nich poskytuje různé funkcionality a jinou úroveň přizpůsobení. Mezi často využívané nástroje patří například Atlassian Jira, GitHub, GitLab, Azure DevOps, které se zaměřují na celou problematiku vývoje a řízení projektu a poskytují propojení se systémem Git. Existují však i nástroje, které nesvazují řízení a kód dohromady. Mezi takové patří například TrelloBoard, nebo Asana.

5.2.3 CI/CD

Zkratka CI/CD v sobě sdružuje dva relativně blízké pojmy. Prvním z jich je **kontinuální integrace** (z anglického „Continuous Integration“), tedy průběžné sestavování a testování aplikace. Druhá část představuje **kontinuální dodávání kódu** (z anglického „Continous Delivery“), tedy nasazení nového kódu do produkčního prostředí zákazníka.

Kontinuální integrace

Kontinuální, nebo také průběžná integrace je proces, v rámci kterého vývojáři vyvíjí kód na svém vlastním zařízení a postupně jej pak spojují až v rámci integrovaného úložiště, například v sytému Git. Samotná integrace nesouvisí pouze se sdílením kódu, ale zabývá se také jeho testováním, nebo případným sloučením do výsledného produktu. Samotný postup se pak skládá z následujících kroků.

1. Vývojář pracuje na svém úkolu, jenž průběžně (případně přednostně) doplní testy, jenž ověří správnou funkčnost jeho výtvoru.
2. Jakmile je úkol splněn, spustí všechny dostupné testy na svém vlastním zařízení, aby ověřil, že svým zásahem nepoškodil žádnou další část aplikace.
3. Pokud některé testy selžou, vývojář napraví případné chyby a po úspěšném dokončení všech testů vloží svůj kód do sdíleného repositáře.
4. Server, který se stará o přeložení a sestavení aplikace pak spustí opět všechny testy s případnými rozšířeními a informuje vývojáře o výsledku.

V rámci čtvrtého bodu seznamu výše se může vyskytovat také nasazení aplikace v rámci testovacího prostředí. Je to z toho důvodu, že aplikace jako taková může podléhat náročnějším testům uživatelského rozhraní, které nelze jednoduše provádět jednotkovými testy bez existujícího prostředí [48].

Kontinuální dodávání kódu

V drtivé většině případů je na integraci také navázáno průběžné dodávání kódu do všech prostředí. Obvykle se v praxi využívá několik typů prostředí, kde každé prostředí slouží k jiným účelům.

- **Vývojové prostředí** obsahuje všechny služby, které vývojáři mohou využívat v rámci vývoje. Vývojové prostředí může mít buď vývojář přímo na svém zařízení, nebo může být sdílené pro celý tým.
- **Testovací prostředí** slouží nejčastěji pro provádění testů. V testovacím prostředí ověřují testeři funkčnost, případně hledají chyby, které pak nahlašují zpět vývojářům v podobě úkolů.
- **Pre-produkční prostředí** je velmi podobné produkčnímu. Je složeno z identických zdrojů jako produkční a slouží pro ověření se zákazníkem, že je vše připraveno k nasazení do produkce.
- **Produkční prostředí** je pak přímo instance daného produktu, kterou využívají uživatelé zákazníka.

V rámci průběžného nasazení nového kódu jsou vždy definována pravidla, která určují, za jakých okolností bude do daného prostředí nasazena nová verze aplikace. Tento proces je plně nastavitelný a záleží na vývojovém týmu a produktu, za jakých podmínek se spustí. Jednotlivé kroky procesu mohou být různé, nejčastěji však obsahují samotné přeložení aplikace, spuštění automatizovaných testů, nebo nasazení do daného prostředí [48].

Běhové prostředí

Zpracování procesů uvedených výše může probíhat na stejném zařízení, na kterém běží systém pro správu verzí, zpravidla se ale využívá sada oddělených zařízení. Výhodou oddělení je možná paralelizace a specializace zařízení pro jednotlivé kroky procesu a jeho celkové zrychlení.

5.2.4 Efektivní využívání větvení

Větve v rámci nástroje Git slouží k oddělení jednotlivých verzí kódu. Při vytvoření nového repositáře je zpravidla vytvořena jediná výchozí větev `Master`, se kterou může vývojář pracovat. Pokud pracuje jeden vývojář sám na malém projektu, může být jediná větev dostačující. V případě, že na jednom projektu pracuje více vývojářů, může docházet ke kolizím se změnami jiného vývojáře. Z tohoto důvodu se zpravidla využívá pro každý úkol nová větev, která je po dokončení změn zapojena zpět do hlavní vývojové větve.

V rámci vývoje většího projektu se vyplatí integrovat důmyslnější hierarchii větví na oddělení rozdílných kontextů. Za tímto účelem byla přímo autory nástroje Git implementována metodika pro tento účel. **Git flow** se skládá z několika předem připravených větví, které jsou průběžně aktualizovány vývojáři a správci repositáře.

Níže popsané rozložení vychází z doporučeného nastavení [6], nicméně může být libovolně přizpůsobeno pro daný projekt.

Větev `master`

Výchozí větev každého repositáře představuje hlavní produkční verzi kódu. Jednotlivé verze kódu v této větvi jsou označené tagem, a s každým novým commitem do této větve vzniká nová produkční verze aplikace. Do této větve je zapojován kód z `release-<verze>` a zároveň je z této větve možné zapojit kód do `hotfix-<verze>`.

Větev `develop`

Jedná se o vývojářskou větev, ve které se nachází všechny dokončené úkoly vývojářů. Z poslední verze kódu v této větvi si vývojáři tvoří nové větve `feature/<ukol>`. Z této větve také některé firmy vydávají takzvané noční sestavení, tedy nejnovější verze programu pro testovací účely dalších stran.

Větev `feature/<úkol>`

Úkolové větve Vyhází zpravidla z poslední verze kódu v `develop` větvi a mají za úkol oddělit postupný vývoj nové funkcionality. Jejich úloha skončí ve chvíli, když je úkol splněn a revidován a kód je zapojen do `develop` větve. `Feature` větev tímto zaniká. Zpravidla se pro každý nový úkol vytváří nová větev.

Větev `release-<verze>`

Před tím, než je vydána nová verze aplikace, je potřeba kód připravit (doplnit číslo verze, připravit datum přeložení a další dílčí úkoly). Do této větve se ve zvolený okamžik zapojí kód z vývojářské větve a v případě potřeby jsou v něm opravovány pouze nalezené chyby, žádné nové funkcionality od této chvíle přidávány do připravované verze nejsou. Jakmile je vše připraveno, tak se kód přesune do `master` větve, ze které je následně automatizovaně kód přeložen a připraven k nasazení.

Větev `hotfix-<verze>`

Poslední používaná větev slouží k velmi podobnému účelu, jako `release` větev. Hlavním rozdílem je, že vzniká z `master` větve a slouží pouze pro opravy nalezených produkčních chyb po vydání nové verze. Jakmile je chyba odstraněna, tak je kód opět zapojen zpět do `master` větve a zároveň aktualizován v `develop` větvi.

5.2.5 Balíčky NuGet

Jak už bylo popsáno v kapitole 4.5.2, v rámci různých projektů může být využíváno různých částí kódu od jiných vývojářů. Aby nebylo nutné importovat celý kód z repositáře cizího vývojáře, v rámci .NET byl společností Microsoft vyvinut balíčkovací systém NuGet. Tento systém se skládá z repositářů, které mohou, ale nemusí, být zabezpečeny heslem. V rámci každého repositáře se pak nacházejí samotné balíčky, které zpravidla reprezentují samostatné projekty, nebo jejich části. Hlavním repositářem, který je volně k dispozici je galerie NuGet na adrese `https://www.nuget.org/`, která obsahuje přes čtvrt milionu různých knihoven.

Pokud vývojáři chtějí určitou část kódu nebo projektu znovu použít, mohou kód zabalit a umístit jej do některého z dostupných repositářů. Tím bude implementace existovat pouze na jednom místě a bude veřejně dostupná. V případě, že je žádoucí, aby přístup nebyl veřejný, je možné si nainstalovat vlastní instanci repositáře a kód umístit tam [2].

Mezi hlavní výhody dělení kódu na menší logické celky a jeho recyklace v dalších projektech je především centralizovaná správa tohoto sdíleného kódu. V případě, že v určitém modulu vývojář narazí na chybu, stačí ji opravit na jednom místě a postupně pak všechny závislé projekty aktualizovat. Mezi nevýhody se pak řadí udržování univerzálního rozhraní v případě, že si vývojář udržuje vlastní balíček. V případě využívání veřejně dostupných modulů pak může být riziková jakákoliv aktualizace, kdy hrozí, že nová verze dané knihovny

bude mít změněné, nebo dokonce kompletně přepracované rozhraní. Nic ovšem nikomu nebrání v tom si buď pro danou knihovnu napsat vlastní adaptér, který lze kdykoliv upravit, případně využívat starší verzi, nad kterou byl kód vytvořen.

5.3 Příprava migrace

Před tím, než byla spuštěna samotná migrace celého projektu, bylo potřeba si velmi dobře navrhnout a připravit prostředí, které bylo pak později využito. Mezi nejdůležitější body, které bylo třeba vyřešit patří tvorba repositářů pro nový kód, příprava repositáře pro odstranění závislostí a v neposlední řadě také návrh a rozběhnutí automatizovaných procesů pro překlad a nasazení kódu.

5.3.1 Tvorba Git repositářů

Z důvodu velkých zásahů do samotného repositáře projektu, byl vytvořen nový repositář `NetfoxCore`, který obsahuje migrovanou kopii nástroje Netfox. Mimo tento hlavní repositář byly vytvořeny také prostory pro moduly, které jsou v projektu využívány. I přes to, že většina modulů je k dispozici od původních vývojářů, některé jejich části byly upraveny pro potřeby Netfox aplikace. Z toho důvodu nejsou využita přímo zdrojová data původních vývojářů, ale udržuje se upravená kopie jejich projektů.

V rámci této diplomové práce se pro udržování verzí kódu a přípravě automatizace překládání, využívá nástroje GitLab[3]. Privátní instance tohoto programu je spuštěna v rámci výzkumné skupiny NES@FIT[4]. Mimo samotné udržování verzí kódu také umožňuje sdružování repositářů do skupin. Je tak umožněno oddělovat jednotlivé části aplikace do samostatných repositářů, ale zároveň neztratit kontext, ke kterému projektu patří.

V rámci této práce byla vytvořena skupina `Netfox`, která obsahuje nejen původní repositář `Netfox`, ale také příslušné Git moduly (`lib`, `TestingData`). Ve stejné skupině byly připraveny také prostory pro migrovaný kód aplikace, `NetfoxCore` a další podmoduly, jako například `PacketDotNet`, `Numl` a další.

Protože podmoduly byly odděleny až v závěru migrace (viz podsekce 5.4.5, postup migrace), příprava větví ve formě GitFlow zde neměla praktický význam. V repositáři obsahující migrovaný projekt se naopak tohoto modelu využívá, aby z pohledu historie bylo vidět, kdy byl funkční kód integrován ho hlavní vývojářské větve a jaké změny v něm byly prováděny.

5.3.2 Příprava CI/CD

Aby nebylo nutné zabalené balíčky vkládat do privátního NuGet repositáře ručně, jsou v rámci GitLab CI/CD připraveny automatizované procesy. Ty samostatný projekt přeloží, zabalí a uloží do repositáře. Definice takového procesu se nejčastěji dělá pomocí popisného souboru `.yaml`, jehož příklad je možné vidět ve výpisu 5.1 a 5.2.

První část souboru znázorněná ve výpisu 5.1 definuje obraz prostředí, který bude v tomto procesu využíván. Dále je v atributu `stages` výpis všech akcí, které jsou spouštěny postupně za sebou. Atribut `variables` zavádí proměnné, které budou v rámci tohoto zpracování používány. Konečně `.get_version` představuje samostatnou funkci, jenž nastavuje vnitřní proměnnou `$VERSION`, která je dále použita v druhé části.

[3] https://about.gitlab.com/
[4] https://gitlab.nesad.fit.vutbr.cz/netfox/netfox

```
image: mcr.microsoft.com/dotnet/sdk:5.0

stages:
    - pack
variables:
    DOCKER_DRIVER: overlay2
    NUGET_PACKAGES: $CI_PROJECT_DIR/nuget

.get_version: &get_version
  - git describe --tags
  - $VERSION=$(.\get-version.ps1)
  - Write-Host $VERSION
```

Výpis 5.1: Zjednodušený příklad popisu přípravy automatizovaného procesu

```
nuget-publish:
    stage: pack
    timeout: 5m
    before_script:
        - *get_version
    script:
        - dotnet build -p:Version=$VERSION -c release projekt.csproj
        - dotnet pack -p:Version=$VERSION -c release projekt.csproj
        - dotnet nuget push -k $NUGET_API_KEY -s $NUGET_URL \
          bin/release/projekt.$($VERSION).nupkg
```

Výpis 5.2: Zjednodušený příklad popisu akce automatizovaného procesu

Druhá část výpisu 5.2 už implementuje samotný proces přeložení a zabalení projektu. Funkce obsahuje několik argumentů, které postupně definují, do jaké etapy daný krok patří, jaká je maximální doba běhu a které další funkce mají být spuštěny před samotným zpracováním. V atributu `script`, který definuje posloupnost jednotlivých příkazů, je možné vidět použití získané hodnoty `$VERSION` z dříve definované funkce, také použití globálních proměnných (`$NUGET_API_KEY`, `$NUGET_URL`).

Globální proměnné

Globální proměnné se využívají při automatizovaném zpracování především za účelem znovupoužití a zároveň také z důvodu ochrany citlivých údajů. Velmi často je potřeba v samotném zpracování používat hesla, nebo přístupové kódy, které zajišťují autorizaci k dalším službám. Protože občas bývají zdrojové kódy přístupné nejen samotným vývojářům, ale také široké veřejnosti, je potřeba mít tyto klíče odstíněné od dostupného kódu. Tyto klíče jsou pak zpravidla v systémech pro sdílení kódu uloženy bez možnosti je znovu zobrazit právě za účelem zajištění integrity.

5.3.3 Příprava NuGet repositáře

Pro účely projektu Netfox byla rozběhnuta instance produktu ProGet[5]. V rámci tohoto privátního repositáře jsou uchovávány všechny moduly, které jsou v projektu využity, ale nemají přímou návaznost na samotný projekt. Pro správné fungování musí být tento privátní repositář zaveden do seznamu poskytovatelů ve vývojářském prostředí. Konfigurace těchto zdrojů se nachází ve výchozím nastavení v souboru `%appdata%\NuGet\NuGet.Config` a jeho podoba je zobrazena ve výpisu 5.3.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <packageSources>
4      <add key="nuget.telerik.com"
5        value="https://nuget.telerik.com/nuget" />
6      <add key="nuget.org"
7        value="https://api.nuget.org/v3/index.json" protocolVersion="3" />
8      <add key="nesad"
9        value="https://nuget.nesad.fit.vutbr.cz/nuget/nuget/v3/index.json" />
10   </packageSources>
11 </configuration>
```

Výpis 5.3: Příklad zkráceného výpisu souboru `NuGet.config`

5.4 Migrace projektu

Jakmile je prostředí připraveno, je možné začít se samotnou migrací kódu. V rámci migrace byl kladen důraz na to, aby byl na všechen upravený kód v souladu s pravidly, které byly definovány v sekci 2.1. V následujících podsekcích jsou popsány všechny překážky a problémy, které bylo nutné vyřešit.

5.4.1 Kompatibilita .NET

Historie repositáře, ze které tato práce čerpá, sahá až do roku 2015. Avšak první verze byla představena ve školním roce 2013/2014 v rámci diplomové práce, pana inženýra Jana Pluskala [44]. Určité části kódu tedy mohou být staré až sedm let. Právě v této době bylo možné využívat nejnovější verze jazyka C# 5.0 v rámci .NET Framework 4.5. V současné době vydala firma Microsoft na konci roku 2020 poslední verzi C# 9.0 společně s prostředím .NET 5, do kterého je právě tento projekt migrován.

Aktuálně je projekt udržován v rámci .NET Framework 4.7 za použití verze jazyka C# 7.0. Díky tomu, že se firma Microsoft snaží udržovat zpětnou kompatibilitu, nebylo nutné řešit kompletní přepis celého kódu. Zároveň je velmi zajímavým faktem, že knihovny pro .NET Framework fungují i v rámci projektu .NET 5. Díky tomu byl velmi zjednodušen začátek migrace, kdy nebylo nutné hledat a případně i přepisovat některé části kvůli nový verzím různých knihoven. Ve chvíli, kdy je použita externí knihovna určená pro .NET Framework v rámci .NET 5 projektu, není samozřejmé, že bude bezchybně fungovat. Stejně tak není možné překládat a spouštět projekt na jiném operačním systému, než Microsoft Windows.

[5] https://nuget.nesad.fit.vutbr.cz/

5.4.2 Nekompatibilita .NET

I přes to, že byla valná většina kódu bez zásadnějších úprav po migraci funkční, bylo potřeba řešit i případy, kdy původní kód nebyl v rámci .NET 5 kompatibilní. Velmi často se jednalo o nepodporované knihovny, nebo odlišný přístup k systémovým knihovnám.

Jedním z příkladů je webová verze Netfox Detective, která v původním projektu využívá knihovnu `System.Web.Hosting`, jenž není přímo v .NET 5 přístupná v dané podobě a neobsahuje tak některé dříve využívané funkce a proměnné. Ve výpisu 5.4 je znázorněná jedna z několika dalších změn, které musely být provedeny z důvodu nekompatibility v rámci verzí .NET knihoven.

```
1 // Puvodni kod
2 var applicationPhysicalPath = HostingEnvironment.ApplicationPhysicalPath;
3
4 // Novy kod
5 var applicationPhysicalPath = Directory.GetCurrentDirectory();
```

Výpis 5.4: Změna v rámci souboru `Netfox.Web.AppStartup.cs`

Dalším příkladem nekompatibility je přístup k lokálnímu nastavení aplikace. V projektech vytvořených v rámci .NET Framework je možné tvořit, přistupovat a měnit různé proměnné na uživatelské, nebo systémové úrovni. V čistém .NET 5 projektu již toto možné není a bylo potřeba přijít s jinou metodikou, jak nastavení ukládat a měnit. Více je o tomto postupu popsáno v jedné z částí podsekce 5.4.4.

Nekompatibilní projekt lze také vnímat z jiného úhlu pohledu. Migrace tohoto projektu na platformu .NET 5 by měla sice na první pohled přinést rozšíření podpory o další operační systémy, nicméně díky využití grafického subsystému WPF je samotná aplikace Netfox Detective dostupná opět pouze pro operační systémy Microsoft Windows. V případě že by bylo požadavkem také fungování v rámci systému Linux, nebo MacOS, muselo by být využito jiných prostředí pro tvorbu grafického uživatelského rozhraní. Na toto omezení naráží i zaváděcí soubor projektu (`.csproj`), který jako cílový framework určuje právě Windows, jak je ukázáno ve výpisu 5.5.

```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3      <PropertyGroup>
4          <OutputType>Exe</OutputType>
5          <TargetFramework>net5.0-windows</TargetFramework>
6          <UseWPF>true</UseWPF>
7          <PlatformTarget>x64</PlatformTarget>
8          <ApplicationIcon>Views\Resources\Netfox128x128.ico</ApplicationIcon>
9      </PropertyGroup>
10     ...
11 </Project>
```

Výpis 5.5: Část výpisu souboru `Netfox.Detective.csproj`

5.4.3 Knihovny pro .NET Framework

Celý projekt Netfox není složen pouze z vlastního kódu, ale využívá i několik dalších knihoven, které doplňují jeho funkčnost. Velká část knihoven se využívá proto, aby nebylo třeba

vymýšlet řešení problému, když už jej vyřešil někdo za nás. V podsekci 5.3.2 bylo pojednáváno o přípravě CI/CD procesů, které měly za úkol ukládat poslední verzi kódu do repositáře NuGet. Na rozdíl od těchto knihoven, je v projektu využito i několika již přeložených knihoven, které nemají volně dostupné zdrojové soubory. Tyto knihovny jsou k projektu přiloženy jako soubory `.dll` a při kompilaci jsou pouze kopírovány do cílového adresáře.

Díky aktivnímu vývoji těchto knihoven bylo ale možné odstranit všechny `.dll` soubory, které byly takto přibaleny. Jedná se především o komponenty společnosti Telerik[6] Infragistics[7], které obsahují rozšíření pro uživatelské rozhraní klientské aplikace WPF. Obě společnosti mají k dispozici časově omezenou verzi pro vývoj, nad kterou lze sestavovat aplikace, a zároveň jsou cílené na .NET Standard, který je plně kompatibilní s .NET 5. S jejich integrací do projektu proto nebyly velké problémy.

5.4.4 Úprava kódu

V průběhu migrace se také podařilo odstranit několik těžkopádných konstrukcí, které se v kódu nacházely. Označení těžkopádné je na místě z různých důvodů. V určitých částech kódu se vyskytoval zbytečně komplikovaný kód náchylný na chyby, v jiných částech bylo možné podobu kódu zjednodušit a zkrátit na méně řádků.

```
1  public NBARProtocolPortDatabase()
2  {
3    string taxonomyFilePath = null;
4    if(File.Exists(NBARprotocols.Default.NBARTaxonomy))
5    {
6      taxonomyFilePath = NBARprotocols.Default.NBARTaxonomy;
7    }
8    else if (File.Exists(HostingEnvironment.ApplicationPhysicalPath + "/bin/"
9      + NBARprotocols.Default.NBARTaxonomy))
10   {
11     taxonomyFilePath = HostingEnvironment.ApplicationPhysicalPath + "/bin/"
12       + NBARprotocols.Default.NBARTaxonomy;
13   }
14   else
15   {
16     var cwd = Directory.GetCurrentDirectory();
17     Console.WriteLine(cwd);
18     throw new ArgumentException
19       ("NBAR protocol port database do not exists!");
20   }
21   // the rest of the constructor
22 }
```

Výpis 5.6: Původní začátek konstruktoru třídy `NBARProtocolPortDatabase`

Příkladem je konstruktor třídy `NBARProtocolPortDatabase`, jenž má za úkol při vytvoření nové instance najít potřebný soubor na disku. V původní implementaci se cesta k tomuto souboru postupně skládá. Tím, že je celý projekt vždy překládán do stejného adresáře, je vždy jisté, že se bude po překladu nacházet na stejném místě. Samotné hledání

[6] https://www.telerik.com/
[7] https://www.infragistics.com/

tohoto souboru bylo tedy možné odstranit a zároveň se tak i zbavit závislosti na nedostupné knihovně `System.Web.Hosting`. Zhruba patnáct řádků dlouhý kód (zobrazen ve výpisu 5.6) bylo tak možné zkrátit jednoduše na třetinu bez změny funkčnosti. Zkrácený kód je ve výpisu 5.7.

```
1  public NBARProtocolPortDatabase()
2  {
3    Assembly self = typeof(NBARProtocolPortDatabase).Assembly;
4    string dir = Path.GetDirectoryName(self.Location);
5    string taxonomyFilePath = Path.Join(dir, NBARFiles.TAXONOMY_FILE);
6    if (!File.Exists(taxonomyFilePath))
7      throw new ArgumentException($"Database file for NBAR protocol " +
8        "ports not found in '{taxonomyFilePath}'!");
9    // the rest of the constructor
10 }
```

Výpis 5.7: Nový začátek konstruktoru třídy `NBARProtocolPortDatabase`

Současně je v opraveném výpisu vidět další ze změn, které se v migrovaném projektu nacházejí. Chybová hláška z původní aplikace pouze říká, že daný port neexistuje. Vývojář se tak v případě této chyby ani nedozví, kde se mělo co nacházet a je nucen tak procházet celý kód, aby zjistil, v čem je problém. V tomto případě, i řadě dalších, byla upravena i samotná chybová hláška, která uživatele přímo upozorní na to, který konkrétní soubor nebyl nalezen a kde se má nacházet.

Dalším zajímavým příkladem je pak refaktorizace v rámci třídy `EmailBodyContent`, jenž se stará o zobrazení obsahu emailové zprávy v několika různých formátech. Mimo samotnou refaktorizaci kódu došlo také k jeho zkrácení a zpřehlednění.

```
1  void EmailBodyContent_DataContextChanged(object sender,
2    DependencyPropertyChangedEventArgs e)
3  {
4    var mimePartVm = e.NewValue as MimePartVm;
5    if(mimePartVm != null)
6    {
7      RadDocument content = null;
8      var plain = true;
9        if(!String.IsNullOrEmpty(mimePartVm.MIMEpart.ContentSubtype))
10       {
11         if(mimePartVm.MIMEpart.ContentSubtype.Equals("html"))
12         {
13           content = new HtmlFormatProvider().Import(mimePartVm.RawContent);
14           plain = false;
15         }
16         /* else if (mimePartVm.Data.ContentSubtype.Equals("pdf"))
17         {
18            content = new PdfFormatProvider().Import(mimePartVm.RawContent);
19         }*/
20         else
21         {
22           content = new TxtFormatProvider().Import(mimePartVm.RawContent);
```

```
23          //content = new XamlFormatProvider().Import(mimePartVm.RawContent)
24        }
25      }
26      else
27      {
28        // content = new XamlFormatProvider().Import(mimePartVm.RawContent);
29        //TxtFormatProvider formatProvider = new TxtFormatProvider();
30        content = new TxtFormatProvider().Import(mimePartVm.RawContent);
31        //content = new TxtDataProvider(mimePartVm.RawContent);
32      }
33    // rest of function
34    }
35  }
```

Výpis 5.8: Původní funkce třídy `EmailBodyContent`

```
1   void EmailBodyContent_DataContextChanged(object sender,
2     DependencyPropertyChangedEventArgs e)
3   {
4     var mimePartVm = e.NewValue as MimePartVm;
5     if (mimePartVm != null)
6     {
7       var radDoc = CreateFlowDoc(mimePartVm.MIMEpart, mimePartVm.RawContent);
8
9       if (IsPlain(mimePartVm.MIMEpart))
10        foreach (var span in radDoc.EnumerateChildrenOfType<Span>())
11          span.FontSize = 12;
12      // rest of function
13    }
14  }
15
16  private static RadDocument CreateFlowDoc(MIMEpart mime, string data)
17  {
18    return mime.ContentSubtype switch
19    {
20      "html" => new HtmlFormatProvider().Import(data),
21      "rtf" => new RtfFormatProvider().Import(data),
22      "pdf" => new PdfFormatProvider().Import(Encoding.ASCII.GetBytes(data)),
23      _ => new TxtFormatProvider().Import(data)
24    };
25  }
26
27  private static bool IsPlain(MIMEpart mime) =>
28    mime.ContentSubtype is not "html" or "rtf" or "pdf";
```

Výpis 5.9: Refaktorovaná funkce třídy `EmailBodyContent`

Ve zjednodušeném výpisu 5.8, obsahující část původního kódu, si lze všimnout nepřehledností v podobě trojnásobného zanoření podmínky `if`, nebo zakomentovaného kódu bez jakéhokoliv komentáře. Ve výsledku se pak jedná o funkci delší, než 45 řádků. Díky tomu, že byla podmínková část z kódu extrahována a zakomentovaný kód opraven a doplněn, došlo k výraznému zjednodušení, které pomůže v orientaci dalším vývojářům (viz podsekce 2.2.3). Nová zjednodušená podoba se pak nachází ve výpisu 5.9, kde si lze všimnout extrahované funkce `CreateFlowDoc`, nebo extrakce podmínky `IsPlain` zjednodušující přehlednost a dodávající možnost případného znovupoužití.

Dlouhé komentáře

Jak už bylo popsáno v podsekci 4.4.3, která se zabývala analýzou komentářů v kódu, bylo třeba odstranit velkou řadu těchto případů. Velmi často se tak jednalo o již nepoužívaný kód, který by v případě potřeby bylo možné nalézt v historii systému pro udržování verzí. V číslech se pak v projektu nacházely části, kde bylo zakomentováno více, než 230 souvislých řádků kódu z celkových 320. Toto bylo nalezeno ve třídě `ExportContentExplorerVm`, z níž se po vyčištění i dalších zakomentovaných částí podařilo vytvořit soubor o celkové délce 60 řádků. Obdobný výsledek bylo možné získat také z třídy `ExportContentExplorerView`, kde z původních 350 řádků kódu došlo ke zkrácení na 56 řádků. Velká řada kratších komentářů pak byla odstraněna z více, než třiceti různých souborů.

NuGet balíčky

V rámci celého projektu se nacházela velká řada závislostí, které zde byly pouze z historických důvodů a nebyly dále využívány. Díky tomu, že migrace probíhala postupně projekt po projektu a jednotlivé závislosti byly taktéž přidávány postupně, bylo možné kontrolovaně redukovat jejich počet. Příkladem může být WPF uživatelská aplikace Netfox Detective, která v původním projektu obsahovala 46 závislostí v rámci NuGet balíčků a 24 přímých závislostí na dynamické knihovny `.dll`. Po migraci došlo k přesunu závislostí z dynamických knihoven na NuGet balíčky. I přes to, se podařilo celkový počet NuGet balíčků redukovat na 31. Knihovny `.dll` nejsou využívány žádné.

Obdobný trend lze sledovat například také u `Framework/CaptureProcesor`, kde se podařilo redukovat závislosti na čtvrtinu, tedy z původních 40 na pouhých 10 NuGet balíčků. Toto čištění pomohlo především při redukci velikosti výsledného projektu.

Zjednodušené implementace

Tím, že se vyvíjí nejen samotný jazyk, ale také knihovny, bylo umožněno odstranění některých komplikovaných konstrukcí.

Ve výpisu 5.10 je původní kód entity, který definuje proces uložení IP adresy do databáze. Na řádcích 60, 71 a 74 se nachází volání rozšiřující metody `SetPrivateFieldValue`. Tato metoda je importovaná z projektu `Netfox.Core` a zajišťuje přístup k privátním vlastnostem třídy. I přes to, že v některých případech je nutné z důvodu mutace třídy zapouzdření[8] nějakým způsobem obejít, po migraci to již není potřeba. Jedná se také o porušení principů SOLID, které jsou podrobněji rozepsány v podsekci 3.3.1.

Díky odstranění dědění z třídy `IPAddress` a jednoduché transformaci do formátu `Json`, je možné pohodlně s adresou pracovat. Stejně tak je možné entitu uložit, nebo načíst,

[8]Zapouzdření umožňuje skrýt vlastnosti, nebo metody třídy a zabraňuje tak nechtěným změnám.

z databáze. Ve výpisu 5.11 je zjednodušená ukázka nové podoby této třídy. Celá úprava podléhala procesu bezpečné úpravy kódu, jak se popsáno v sekci 3.2.

```
1  ...
2  using Netfox.Core.Extensions;
3
4  namespace Netfox.Core.Database.Wrappers
5  {
6    [ComplexType]
7    public class IPAddressEF : IPAddress
8    {
9      private byte[] _addressData;
10
11     internal const int IPv4AddressBytes = 4;
12     internal const int IPv6AddressBytes = 16;
13     internal const string dns_bad_ip_address = "dns_bad_ip_address";
14     internal const int NumberOfLabels = IPv6AddressBytes / 2;
15
16     /* Constructors */
17
18     [MaxLength(16)]
19     public byte[] AddressData
20     {
21       get { return this._addressData; }
22       private set
23       {
24         this._addressData = value;
25
26         AddressFamily m_Family;
27         Int64 m_Address = 0;
28         var m_Numbers = new ushort[NumberOfLabels];
29         if (this._addressData == null)
30         {
31           throw new ArgumentNullException(nameof(value));
32         }
33         if (this._addressData.Length != IPv4AddressBytes &&
34           this._addressData.Length != IPv6AddressBytes)
35         {
36           throw new ArgumentException(dns_bad_ip_address, nameof(value));
37         }
38
39         if (this._addressData.Length == IPv4AddressBytes)
40         {
41           m_Family = AddressFamily.InterNetwork;
42           m_Address = ((this._addressData[3] << 24 |
43                         this._addressData[2] << 16 |
44                         this._addressData[1] << 8 |
45                         this._addressData[0]) & 0x0FFFFFFFF);
```

```
46          this.SetPrivateFieldValue("m_Address", m_Address);
47          this.ScopeId = (long) ProtocolFamily.InterNetwork;
48        }
49        else
50        {
51          m_Family = AddressFamily.InterNetworkV6;
52          for (var i = 0; i < NumberOfLabels; i++)
53          {
54            m_Numbers[i] = (ushort)(this._addressData[i * 2] * 256 +
55              this._addressData[i * 2 + 1]);
56          }
57          this.SetPrivateFieldValue("m_Numbers", m_Numbers);
58          this.ScopeId = (long)ProtocolFamily.InterNetworkV6;
59        }
60        this.SetPrivateFieldValue("m_Family", m_Family);
61      }
62    }
63
64    [NotMapped]
65    public new long ScopeId { get; set; }
66    [NotMapped]
67    public new long Address { get; set; }
68
69  }
70 }
```

Výpis 5.10: Původní třída pro entitu IP adresy

```
1  using ...
2
3  namespace Netfox.Core.Database.Wrappers
4  {
5    [ComplexType]
6    [JsonConverter(typeof(IpEfConverter))]
7    public class IPAddressEF
8    {
9      internal const int IPv4AddressBytes = 4;
10     internal const int IPv6AddressBytes = 16;
11     internal const string dns_bad_ip_address = "dns_bad_ip_address";
12     internal const int NumberOfLabels = IPv6AddressBytes / 2;
13
14     public IPAddress Address { get; set; }
15
16     /* Constructors */
17   }
18 }
```

Výpis 5.11: Nová třída pro entitu IP adresy

Složitější implementace

Nová verze .NET 5 s sebou nepřináší pouze zjednodušení, ale také komplikace.

Jednou z komplikovanějších záležitostí, byla nutnost přepsání zaváděcí třídy pro webovou aplikaci Netfox Detective. Projekt `Detective/Web/Netfox.Web.App` staví mimojiné na těchto dvou knihovnách.

1. `Owin`, která se stará o to, aby bylo možné rozběhnout webovou aplikaci pod více různými servery pro webové aplikace. Tento modul tak zavádí abstrakci mezi webovým serverem a projektem, který je napsaný v rámci rodiny .NET. Prakticky je tak možné za nižších nákladů na úpravu kódu migrovat .NET projekt na jinou webovou technologii.

2. `Hangfire`, jenž má na starosti plánování úkolů na pozadí. Tato knihovna poskytuje komplexní prostředí pro tvorbu a monitoring jednotlivých úloh a vývojáři tak výrazně zjednodušuje práci ohledně správy úloh náročných na délku zpracování. V neposlední řadě také knihovna poskytuje intuitivní dashboard pro rychlý přehled stavu jednotlivých úloh.

Právě tyto dvě knihovny spolu nejsou v rámci protředí .NET 5 kompatibilní a bylo tedy nutné přepracovat spouštěcí soubory za použití standardních knihoven pro čisté ASP.NET. Tato úprava má za následek redukci možných typů webových serverů, na kterých aplikace může běžet.

Mimo takto složitý úkol bylo třeba také opravit několik menších případů spojených s přechodem na platformu .NET 5. Exemplárním příkladem je úprava některých rozhraní pro systémová volání. Z uživatelské aplikace WPF je v určitých místech možné otevřít prohlížeč souborů v určité cestě. Ve výpisu 5.12 je vidět, že před migrací bylo možné požádat operační systém o spuštění procesu a předat mu pouze cestu k adresáři, který jsme chtěli otevřít. Operační systém si sám odvodil, jaký program má pro danou cestu spustit a vše si vyřešil sám.

```
1  // old approach
2  Process.Start(this.Workspace?.WorkspaceDirectoryInfo.FullName);
3
4  // new approach
5  var explorer = Path.Combine(Environment.GetEnvironmentVariable("WINDIR"),
6    "explorer.exe");
7  Process.Start(explorer, this.Workspace?.WorkspaceDirectoryInfo.FullName);
```

Výpis 5.12: Upravený přístup ke spouštění nových procesů

Nově je však z důvodu některých úprav knihoven .NET 5 potřeba předávat i konkrétní program, který jako programátor chceme spustit. Argumentem pro nově spuštěný proces je pak druhý parametr metody `Process.Start(...)`, ve kterém se nachází přímo cesta, kterou je třeba v prohlížeči souborů zobrazit.

Odstranění nepotřebných souborů

Jedním z důvodů, proč bylo zvoleno vytvoření nového repositáře, byla možnost zachovat strukturu a historii původního repositáře, dále snazší a rychlejší restrukturalizace a především organizovaná filtrace nepotřebných souborů.

Postupně během času se ve starém repositáři hromadily soubory, které již nebyly využívány. Podařilo se tak odstranit velké množství kódu, který byl pouze obalen v komentáři. Využívání komentářů k „bezpečnému“ odstranění některých metod, nebo dokonce celých tříd je velmi nečistou praktikou. V dnešní době se k udržování historie jednotlivých souborů využívají Git repositáře, které tuto úlohu stoprocentně zastávají.

Zároveň se v původním projektu nacházelo několik souborů, které nebyly namapované v definičním souboru projektu `.csproj`. V rámci migrace bylo takto odstraněno zhruba dvacet tříd s příponou `.cs` a zhruba deset předpisů pro WPF s příponou `.yaml`. I přes to, že se může zdát, že odstranění několika textových souborů nebude znamenat žádný zásadní rozdíl, opak je pravdou. V rámci projektových souborů `.csproj` již nejsou definované konkrétní zdrojové soubory nutné ke kompilaci. V nové implementaci se kompilátor pokusí přeložit vše, co v projektové složce najde. Velká řada těchto souborů využívala starých a nefunkčních rozhraní, což znemožňovalo překlad aplikace. Je tak ve výsledku ušetřen nejen čas při spouštění vývojového prostředí, ale také čas strávený při kompilaci. Ušetřen je také prostor Git repositáře.

Nastavení

Velmi zajímavou oblastí k vyřešení bylo nastavení aplikace. V původním projektu bylo využíváno nastavení, které je možné dělit na uživatelská a globální. Uživatelská nastavení jsou ukládána ve složce `%userprofile%\appdata\local\<nazev aplikace>`, globální jsou pak uložena společně se zbytkem projektu. Protože je aplikace migrována na platformu .NET 5 za využití `WPF`, tak pro zpřístupnění těchto nastavení by bylo potřeba navíc přidat závislost na grafickém uživatelském rozhraní `WinForms`. Přidávání celého dalšího sytému (vedle `WPF`) pouze kvůli samotnému nastavení by bylo zbytečné zavádění dalších závislostí do celého projektu. Byla proto vytvořena samostatná třída, která bude nastavení udržovat sama.

Rozhraní `INetfoxSettings` představuje soubor atributů, které lze v aplikaci nastavovat a jejich úpravou tak modifikovat její chování. Tato rozhraní pak implementuje třída `IAppSettings`, jenž sdružuje rozdílná nastavení pro `Common/Core` projekt a `Common/Logger`. Konečně se pak přímo v projektu `Detective/App/Netfox.Detective` nachází již samotná implementace nastavení ve třídě `NetfoxJsonSettings`. Tento způsob hierarchie umožňuje vytvořit další, odlišný systém ukládání nastavení v budoucnosti. Zjednodušený příklad výše popsaných tříd je vyobrazen ve výpisu 5.13.

```
1  // INetfoxSettings.cs
2  namespace Netfox.Core.Infrastructure {
3      public interface INetfoxSettings {
4          string WorkspaceFileExtension { get; set; }
5          string DatabaseFileExtension { get; set; }
6          void Save();
7      }
8  }
9
10 // IAppSettings.cs
11
12 namespace Netfox.Detective.Infrastructure {
13   public interface IAppSettings : INetfoxSettings, ILoggerSettings { }
14 }
```

```
15
16 // NetfoxJsonSettings.cs
17 namespace Netfox.Detective.Infrastructure
18 {
19   public class NetfoxJsonSettings : IAppSettings
20   {
21     public string WorkspaceFileExtension
22         { get => _model.WorkspaceFileExtension; }
23     public string DatabaseFileExtension
24         { get => _model.DatabaseFileExtension; }
25
26     public NetfoxJsonSettings() : this("settings.json") { }
27
28     public NetfoxJsonSettings(string file) {
29       _file = file;
30       if (File.Exists(file))
31         _model = JsonConvert
32                   .DeserializeObject<Model>(File.ReadAllText(file));
33       else
34         _model = new Model();
35     }
36
37     public void Save() {
38       string dir = Path.GetDirectoryName(_file);
39       if (!string.IsNullOrWhiteSpace(dir) && !Directory.Exists(dir))
40         Directory.CreateDirectory(dir);
41       File.WriteAllText(_file, JsonConvert.SerializeObject(_model));
42     }
43
44     private sealed class Model {
45       public string WorkspaceFileExtension { get; set; } = "nfw";
46       public string DatabaseFileExtension { get; set; } = "mdf";
47     }
48   }
49 }
```

Výpis 5.13: Nová implementace nastavení aplikace

Téměř totožný přístup byl pak zvolen také u nastavení webové aplikace. Všechna nastavení jsou ukládána přímo v adresáři se spustitelným projektem a pokud vývojář neurčí jinak, tak se nastavení uloží do souboru `settings.json`. Díky tomuto způsobu je nastavení přístupné z jednoho místa, v lidsky čitelném formátu. Zároveň je využito systému `Castle Windsor`, který zajišťuje vkládání závislostí, a je tak možno si nastavení vyžádat prakticky kdekoliv v aplikaci. Principy vkládání závislostí jsou rozepsány v podsekci 3.3.2.

5.4.5 Postup migrace

Mimo některé zásadní změny, které byly popsány výše, se samotná migrace držela velmi obdobného scénáře. Ještě před tím, než se ale začala celá aplikace postupně migrovat, bylo

třeba zjistit, jak má správně fungovat a vypadat rozběhnutý projekt. Po stažení poslední verze kódu `4cdbd0d1` ve větvi `develop`, bylo zjištěno, že aplikace sama o sobě správně nefunguje a je potřeba nejdříve opravit chybějící návaznosti. Bylo potřeba opravit správné cesty k přiloženým `.dll` knihovnám, cestu a mapování projektu `VirtualizingCollection` a některé další minoritní chyby.

Po úspěšném spuštění projektu začaly být tvořeny první verze testovacích scénářů, které jsou popsané v sekci věnující se popisu manuálních testů (6.3). Tímto krokem bylo zajištěno, že se aplikace bude chovat před i po migraci identicky a uživatele tak nepřekvapí zásadní odlišnosti. Z procházení původního projektu se také začal utvářet obraz toho, jak uživatel aplikaci používá, jaké jsou její kvality a ve kterých částech se nacházejí případné nedokonalosti.

Jakmile byl původní projekt funkční a bylo možné jej pohodlně používat, bylo třeba připravit hlavní repositář pro nový kód. Z důvodu nepřímé myšlenkové návaznosti frameworku .NET Core na .NET 5, byl nový repositář pojmenován `NetfoxCore`.

V rámci tohoto repositáře byly postupně tvořeny jednotlivé části projektu od začátku a do nich byl postupně migrován původní kód. Aby bylo možné vždy přeložit a spustit testy nad migrovanou částí, bylo třeba začít přesouvat kód od nejméně závislých částí. V první řadě tedy byly přesunuty všechny podprojekty, které přímo nesouvisely se samotnou aplikací Netfox, jako například rozšíření pro Entity Framework, knihovny Numl, nebo PacketDotNet. Důvodem jejich přesunu byla pohodlnější oprava nalezených chyb a ujasnění jejich místa v rámci celého projektu.

Po přesunutí kódu do nového projektu, vývojářské rozhraní zpravidla nalezlo několik desítek, až stovek chyb. Velká část z nich byla způsobena chybějícími závislostmi. Tyto závislosti byly postupně doplňovány s tím, že byly vybírány jejich poslední verze, kompatibilní s .NET 5. Nezřídka se stávalo, že některé nové balíčky, nad kterými byl daný kód postaven, měly upravená rozhraní, které daný kód využíval a bylo třeba problematickou část upravit. Valnou většinu takovýchto případů se podařilo odbavit relativně rychle, a to díky tomu, že většina knihoven je dodávána s relativně obsáhlou dokumentací.

V rámci některých části celého projektu se využívalo direktiv preprocesoru. Ty mají za úkol ovlivnit chování kompilátoru v průběhu překladu kódu. Na některých částech se tak vyskytovalo například rozdílné chování metod v závislosti na použité verzi .NET Framework, případně knihovny Entity Framework. Ukázkou použití těchto direktiv je možné vidět ve výpisu 5.14. Velká řada z nalezených direktiv mohla být odstraněna, neboť se nepočítá s tím, že by měl být kód zpětně kompatibilní s původní verzí .NET Framework.

```
1  #if NET40
2      value = tmp.GetType().GetProperty(segments[i]).GetValue(tmp, null);
3  #else
4      value = tmp.GetType().GetProperty(segments[i]).GetValue(tmp);
5  #endif
```

Výpis 5.14: Ukázka použití direktiv

Protože se projekt skládá z velké řady menších částí, které jsou do projektu zapojeny pouze jako dynamické knihovny `.dll`, nebylo možné od začátku projekt spouštět a sledovat to, jak se postupně zvětšuje. Fakticky se v projektu nachází pouze dvě spustitelné aplikace. První z nich je uživatelská aplikace WPF, která funguje pouze v rámci operačního systému Microsoft Windows, druhou z nich je pak webová aplikace, kterou je možné ovládat prakticky z jakéhokoliv operačního systému, který obsahuje internetový prohlížeč.

Ve chvíli, kdy byla dokončena migrace uživatelské aplikace ve frameworku WPF, bylo možné poprvé projekt spustit a zjistit, co všechno se v rámci migrace podařilo a které části bylo třeba opravit. Postupně s úspěšným spuštěním aplikace mohla také přijít první část manuálních testů, jenž byly připraveny ještě před migrací samotnou a jsou popsané v podsekci 6.3. Ve chvíli, kdy některý z připravených testů selhal, bylo třeba najít příčinu a problém opravit. Zpravidla se tak využívalo ladících nástrojů obsažených přímo v integrovaném vývojovém prostředí.

Jakmile byla ověřena funkčnost WPF aplikace, začala migrace aplikace webové. Zde je samotný webový projekt rozdělen do tří částí: datová vrstva, byznys logika, a webový server. Toto rozdělení velmi usnadnilo analýzu nalezených chyb a problémů, neboť je v každé úrovni jasné, co má na starosti a jak pracuje. Datová vrstva (v anglické literatuře „data access layer“, nebo-li „DAL“), obstarává komunikaci s databází a převod databázových entit na instance jednotlivých tříd. O vrstvu výše se nachází byznys logika, která řeší všechny logické záležitosti a implementuje fasádu mezi datovou vrstvou a klientskou aplikací. V případě potřeby je tak možné nahradit i celou klientskou aplikaci. Na nejvyšší vrstvě se pak nachází již zmíněná klientská aplikace, která tvoří webový server a obsahuje všechny předpisy stránek, které uživatel vidí ve svém prohlížeči. Zde bylo také třeba řešit několik problémů, z nichž některé jsou popsány v podsekci 5.4.4.

V závěru, když byla dokončena migrace všech původních součástí a zároveň úspěšně procházely všechny připravené testy, bylo možné oddělit nesouvisející moduly do vlastních repositářů (podrobněji rozepsáno v podsekci 4.5.2). Díky tomu, že v rámci těchto přidružených repositářů (popsaných v podsekci 5.3.1) byly již připraveny procesy pro automatické přeložení, zabalení a uložení do privátního NuGet repositáře (popsaného v podsekci 5.3.2). Díky tomu byl jeden z posledních kroků bezproblémový a nebylo třeba zde řešit zásadnějších komplikací. V rámci hlavního projektu `NetfoxCore` bylo pouze třeba zaměnit závislosti z interních odkazů na privátní NuGet.

5.4.6 Dosažené cíle

Cílem této diplomové práce nebylo pouze přesunutí samotného projektu na novou platformu, ale také velká řada dílčích činností kolem samotných repositářů, testů, opravy a aktualizace kódu, nebo samotného vyčištění nepotřebných částí. V následujících bodech jsou rozepsány všechny hlavní, i vedlejší body, které se podařilo vyřešit a mohou tak pomoci k lepší orientaci budoucím vývojářům tohoto projektu.

Migrace na .NET 5

Nedůležitějším bodem je migrace celého projektu na platformu .NET 5, ale také samotná aktualizace kódu. Díky této migraci se podařilo otevřít dveře dalším vývojářům, kteří by chtěli na tomto projektu pokračovat a dále jej rozvíjet. Změna platformy právě na .NET 5 umožňuje migraci grafického uživatelského rozhraní, tedy i rozšíření podpory na další operační systémy.

Vyčištění repositáře

Protože je projekt vyvíjen již velmi dlouho a na samotném vývoji se podílela řada programátorů s rozdílnými zkušenostmi, docházelo v průběhu času k zanášení nečistot. Tyto nečistoty se projevovaly především v chaotickém umístění některých projektových složek, kdy samotné rozložení projektu, jak ukazují nástroje pro vývoj, neodpovídá tomu, jak jsou

soubory a složky řazeny na disku. Díky migraci se podařilo tyto cesty sjednotit a pomoci tak budoucím vývojářům v lepší orientaci. Z hlediska velikosti zabraného prostoru na disku se pak podařilo ušetřit více, jak polovinu místa. Z původní velikosti 67 MB, se podařilo redukovat data na 32 MB. Obě tyto hodnoty jsou bez testovacích dat.

Vyčištění kódu

V průběhu řešení této práce se také podařilo odstranit celou řadu aplikovaných neduhů, jako jsou dlouhé komentáře, ponechané soubory, které se nepřekládají, nebo například mrtvý kód, který není nikde využitý. Zároveň bylo možné pročistit kód stran direktiv pro preprocesor, díky tomu, že není třeba udržovat zpětnou kompatibilitu.

Lepší práce s balíčky

Díky migraci bylo také možné aktualizovat řadu závislostí na poslední vydané verze, které obsahují opravy určitých chyb, případně zjednodušují přístup k jejich funkcím. Zároveň se podařilo odstranit z kódu závislosti, které již nebyly využívány. V neposlední řadě došlo také k oddělení nepřímo souvisejících knihoven do vlastních projektů. Zároveň se také podařilo odstranit všechny dynamické knihovny `.dll`, které byly pro potřeby tohoto projektu zakoupeny, ale již nebyly aktualizovány.

Rozběhnutí původního projektu

Jak už vyplývá ze začátku podsekce 5.4.5, dokonce ani původní projekt se nepodařilo po stažení bezproblémově spustit. V rámci této práce bylo třeba také opravit některé části a znovu tak uvést projekt v jeho funkčnost. Do budoucna může být tato oprava využita například k porovnání rychlosti, nebo z důvodu zachování určité zpětné kompatibility.

Oprava překlepů

Relativně malým, ale užitečným bodem je pak také oprava některých překlepů, které byly v aplikaci nalezeny. Nejednalo se pouze o názvy proměnných, nebo funkcí, ale dokonce bylo opraveno i několik názvů souborů a tříd. Ač se jedná o relativně nenápadné změny, budoucím vývojářům to pak nekazí celkový dojem při pohledu do kódu. Při úvodním spuštění analýzy typografických chyb, bylo nalezeno necelých osm tisíc problémů. Po bližší analýze se podařilo velkou část odfiltrovat, neboť se jednalo o použití jmen, českých komentářů, nebo některých zkrácených slov. Opraveny byly například případy jako „propery“, „evnt“, „investigat“, nebo například „commang“, či „parrent“.

Kapitola 6

Testování

Testování je ve světe softwarového inženýrství velmi širokým pojmem. Jako takové začíná ještě před prvním napsaným řádkem kódu a končí až ve chvíli, kdy je ukončena podpora daného produktu. V rámci této kapitoly budou nastíněny nejdůležitější typy testování a v další části pak jejich podrobnější rozebránív rámci projektu Netfox.

V předchozí kapitole v sekci 5.1, která se zabývala agilní metodikou vývoje, bylo čtvrtým bodem opakujícího se cyklu pojem „Verifikace“. Lze si všimnout, že se procesně provádí neprodleně po implementaci. Poukazuje to na fakt, že by měl sám vývojář tvořit testy pro svůj vlastní kód a tím se tak snažit konfrontovat svůj dojem, že je jeho kód bezchybný. Jedním z cílů testování je právě odhalování chyb v kódu. V rámci testů se ale můžeme setkat i s dalšími pojmy, které se zaměřují na lehce odlišné typy testů, které se v praxi využívají.

6.1 Testy kódu

I přes to, že z předchozího textu vyplývá, že je řada odlišných typů testů a způsobů testování, jsou jisté faktory, které mají všechny testy společné. Mezi ty nejdůležitější patří účel testování.

Hlavním cílem testování jakéhokoliv programu, je odhalení a oprava možných chyb co nejdříve. Je tak možné předejít tomu, aby nad chybným kódem, nebo nepochopenou funkcionalitou, nebyla tvořena větší část systému. Důsledkem toho je snížení nákladu v delším časovém horizontu. Včas objevená chyba se velmi často snadněji izoluje a odstraňuje. Vedlejším jevem je zákazník, který byl ušetřen případné frustrace.

6.1.1 Obecná podoba testů

Ve své nejobecnější podobě se testy drží stále stejného scénáře, který se skládá z několika základních kroků.

1. **Příprava**, neboli anglicky „arrange“, je proces při kterém se tvoří prostředí, ve kterém bude daný test probíhat. Příprava může být v rámci jednoduchých testů takřka ignorována. Můžou se ale naskytnou případy, kdy samotná příprava testů může obsahovat vložení dat do databázové tabulky, nebo dokonce celou tvorbu databáze a dalších podpůrných systémů. V případě takto velkých příprav se využívá návrhového vzoru „Fixture“, který má za úkol připravit prostředí právě pro danou sadu testů. Více je tento návrhový vzor rozepsán v podsekci 3.3.4.

2. **Akt** již spočívá v samotném spuštění daných testů. Zpravidla se testy spouštějí sériově za sebou, aby nedocházelo k vzájemnému ovlivňování. Paralelně jsou pak spouštěny testy napříč jednotlivými projekty.

3. **Potvrzení**, neboli anglicky „assert“, je poslední krok celého testovacího procesu. Zde jsou vyhodnocovány výsledky testu. V praxi se pak jedná o podmínky, jenž musí být splněny, aby mohl být test prohlášen za úspěšný [21].

Je velmi důležité, aby na sobě byly testy nezávislé, v žádném případě by neměl výsledek (ani průběh) jednoho testu ovlivnit testy ostatní.

6.1.2 Testy z pohledu přístupnosti

Jakékoliv testování lze také rozdělit na základě přístupu k tomu, co je testováno. Obecně se v rámci testování rozlišují následující tři pojmy, které oddělují to, jak hluboko je pozorovatelné chování testované funkce.

Black box

Nejjednodušším pojmem na vysvětlení je takzvaná černá skříňka. Tester nemá tušení, jak kód uvnitř skříňky funguje. Jeho jediným úkolem je na vstup testované části programu vložit nějaká data a na výstupu zkontrolovat shodu s očekávaným výsledkem. Zpravidla je u takovýchto systémů velmi obtížné hledat krajní hodnoty, které by mohly správné fungování narušit. Odhalování chyb je tedy velmi složité.

White box

V českém překladu za bílou skříňku považujeme systém, nebo součást, ve které máme kompletní přístup ke zdrojovému kódu. Víme přesně jak daný kód funguje, kde jsou jeho slabiny a snáze se hledají případy, kdy takový kód selže, tedy jaké jeho části je třeba ještě ošetřit. Zpravidla se zde lépe testují průchody všemi částmi kódu, než v předchozím případě.

Grey box

Posledním pojmem, se kterých se lze setkat je takzvaná šedá skříňka. Jedná se o kombinaci dvou předchozích pojmů. V tomto případě má tester k systému zpravidla omezený přístup a je schopný plně kontrolovat jen některé jeho součásti. Příkladem může být využívání uživatelského rozhraní k samotnému testování (tedy část, kdy tester nezná přesnou implementaci), ale má plnou kontrolu nad databázovým strojem pro ověření správné komunikace s databází (tedy část, kterou má tester plně pod kontrolou) [12, 23].

6.1.3 Životní cyklus testů

Životní cyklus (anglicky označován jako „life cycle“) je posloupnost stavů, ve kterých se daná entita v průběhu své existence nachází. U samotných testů se skládá z několika kroků, které jsou shrnuty níže.

1. **Plánovací fáze** zahrnuje analýzu požadavků a zjištění, zda je danou komponentu vůbec možné testovat. Zjišťují se okolnosti, za jakých musí být testováno a seskupují se možné situace, které je potřeba kontrolovat. Zároveň je potřeba určit plánovací a vyhodnocovací strategii. Cílem je vytvořit podklady pro další fázi.

2. **Analytická fáze** již přímo pracuje s projektem samotným. Určuje se, co přesně bude testováno, do jaké hloubky a definují se přesné podmínky všech testů. Cílem této fáze je dokumentace všech testů.
3. **Fáze návrhu** má za úkol rozpadnutí výstupních podmínek testu na podobu, kterou lze implementovat do kódu. Zároveň jsou seskupována vstupní data a je navrženo samotné prostředí. V případě, že se očekává sbírání metrik, tak je specifikováno, jakým způsobem budou sbírány.
4. **Fáze implementace** už spočívá v samotném přepsání zadání do kódu dle specifikace. V případě, že je v rámci prostředí vše připraveno, pro vývojáře to znamená pouze přepsání lidského textu na ten programový.
5. **Fáze spuštění** spočívá v tom, že jsou postupně všechny testy spouštěny a vyhodnocovány. Výstupem je potom seznam úspěšných a neúspěšných testů a hodnoty všech metrik, které byly měřeny.
6. **Fáze zhodnocení** nastává jakmile je předchozí fáze dokončena. Cílem je zhodnocení všech výsledků a jejich reflektování zpět do vývoje.
7. **Fáze ukončení** je poslední fází, která nastává před předáním produktu zákazníkovi. Zde by měly být všechny testy úspěšně dokončeny a metriky by měly dosahovat předem určených hodnot. Zároveň je možné zhodnotit v rámci týmu jak se vše dařilo a nedařilo, aby bylo možné předejít případným problémům v příštím projektu.

Těmito kroky by měly projít všechny připravované testy. V rámci tohoto procesu je možné, že se některé testy vrátí do předchozích kroků, neboť například nedošlo k úplné definici podmínek, nebo všech možných stavů dané komponenty [53].

6.1.4 Typy testů

Při testování kódu se neřeší pouze to, zda určitá část funguje korektně, ale také zda funguje optimálně vzhledem k časové a prostorové náročnosti. Nejdůležitější typy testů, které se v praxi používají jsou vypsány v následujících podsekcích.

Jednotkové testy

Jednotkové testy (v anglické literatuře jako „unit tests") ze své podstaty testují pouze jednotky (tedy malé části) kódu. Zpravidla se jedná o kontrolu správné funkčnosti jedné konkrétní metody, nebo funkce. V případě, že daná funkce potřebuje komunikovat s okolím, jsou tvořeny náhražky, které pouze simulují potřebnou funkci. Je tak zajištěno, že testy různých komponent na sobě nejsou vzájemně závislé. Zároveň je tak snazší vyhodnotit pokrytí kódu testy.

Jednotkové testy jsou zpravidla velmi rychlé na vyhodnocení, neboť jsou všechny závislosti pouze simulovány a všechny výsledky, které komponenta potřebuje má téměř okamžitě. Příkladem může být simulace databázového stroje, kdy je možné testovat všechny možné stavy bez toho, aniž by bylo s nějakou instancí databázového systému přímo manipulováno. Díky této simulaci je možné vyvíjet komponenty nezávisle na sobě [63].

Integrační testy

Oproti jednotkovým testům se integrační testy nacházejí o úroveň výše. Vstupem integračních testů jsou otestované komponenty a jejich cílem je ověření, že i komponenty mezi sebou komunikují tak, jak definuje specifikace. Zpravidla se tak ověřuje komunikace mezi různými moduly daného softwaru, případně také komunikace mezi softwarem a operačním systémem. Nezřídka může být také vyžadováno, aby pro účely testů již existovalo určité prostředí (databázový server, dostupný síťový zdroj). Testy jako takové mohou být jak automatizované, tak manuální prováděné testerem [18].

Zátěžové testy

Zátěžové testy slouží zjednodušeně pro měření výkonu a jejich hranic v rámci daného systému. Jejich cílem je odhalení neoptimalizovaných částí kódu a zároveň zjištění limitů daného systému ve spojení s hardwarovou výbavou, na které je systém provozován. Aby mohl být výsledek porovnatelný a dával smysl v kontextu daného použití, provádí se tyto testy na identickém zdroji, na kterém funguje i produkční verze systému. Samotné testy se dělí na několik dalších podkategorií.

- Zátěžové testy zjišťují, zda je prostředí správně dimenzováno na očekávané vytížení daného systému. Cílem je odhalení chyb spojených s únikem paměti, případně správným návrhem běhové architektury.
- Testy hraniční zátěže se zabývají tím, jak vysokou zátěž je systém schopen obsloužit, než se některá z jeho částí zhroutí.
- Test záložního řešení ověřuje, zda je dobře připravený proces přepnutí na záložní systém v případě chyby v hlavním prostředí.
- Testy odolnosti vychází ze standardních zátěžových testů s tím rozdílem, že běží zpravidla několik dní. Hledají se tak především hodnoty dostupnosti a střední doby mezi selháním[1].
- Benchmark je způsob testování částí systému, kdy se snažíme odhalit část, která je nejpomalejší, takzvané úzké hrdlo.
- Test objemu dat se využívá k ověření, že systém zvládne fungovat i v případě, že dostává ke zpracování vysoké objemy dat. Zpravidla se tak ověřuje že má databáze dostatek kapacit.
- Test citlivosti sítě má za úkol prověřit, jak moc je spolehlivá infrastruktura, na které daný systém funguje [57].

Další typy testů

Mezi další důležité typy testů se řadí například takzvané „smoke“ testy, které mají za úkol ověřit funkčnost všech nejdůležitějších částí projektu. Nejsou tak testovány všechny krajní případy, ale je kladen důraz na všechny majoritní akce, které systém poskytuje.

[1] Anglicky označováno jako MTTF (Mean Time to Failure), představuje statistickou veličinu určující spolehlivost dané součásti. Jednotkou této veličiny je čas, který říká, jak dlouho musí být v průměru zařízení v provozu, než se rozbije

Důležitou součástí nejen akceptačních testů jsou „end-to-end“ testy, které mají za úkol prověřit, zda lze pomocí dané aplikace docílit předem daných úkonů. Testování probíhá podle uživatelských scénářů. Příkladem může být úkol „nákup mobilního telefonu“ v daném internetovém obchodu [32].

Mezi velmi známé druhy testů se řadí alpha a beta testy. V případě alpha testů je produkt distribuován uzavřené skupině testerů, kteří mají za úkol produkt vyzkoušet a reportovat zpět nalezené chyby. Beta testy jsou pak velmi často otevřené široké veřejnosti, případně jsou podmíněny přihlášením se do testovacího programu. Cílem těchto testů je nalezení co nejvíce chyb, které by se mohly objevit v produkční verzi.

Důležitým způsobem testování je také ověřování kompatibility s daným běhovým prostředím. V případě webové aplikace je třeba zkontrolovat, že daná stránka bude fungovat ve všech požadovaných prohlížečích.

Nedílnou součástí každého předání projektu jsou také akceptační testy. Tyto testy jsou definovány přímo zákazníkem a určují kritéria, za kterých je možné považovat systém za dokončený. Do akceptačních testů mohou být zahrnuty nejen hranice daného informačního systému, ale mohou být doplněny o další kritéria, jako jsou například požadavky na výkon, nebo velikost.

6.2 Testování v Netfox

Jak už bylo posáno v předchozích sekcích, cílem testování je ověřit, že aplikace neobsahuje žádné chyby. Tvůrci testů tak snaží odhalit všechny možné případy, kdy by aplikace mohla selhat, nebo vracet nesprávné výsledky. Správný tester se tak snaží ze všech svých sil aplikaci rozbít.

Aby mohla být zaručena stejná funkčnost, jako před migrací, byly výsledky testů porovnávány mezi původní a migrovanou aplikací. Velkou výhodou bylo to, že mohla být využita totožná testovací data, neboť způsob jejich zpracování nebyl migrací narušen. Nebylo tedy nutné tvořit kompletní sadu testovacích dat. Zároveň bylo možné plně využít testy z původního projektu, kdy některé z nich vyžadovaly pouze drobné úpravy.

6.2.1 Přístup k testovým souborům

Protože jsou v rámci testů využívaná zdrojová data ve formě `.pcap` souborů obsahující odchycenou síťovou komunikaci, je potřeba v rámci každého testu tuto cestu specifikovat. Zároveň ale může více testů využívat stejný vstupní soubor. Z tohoto důvodu bylo před migrací v projektu využíváno aplikačního nastavení, které obsahovalo všechny potřebné cesty. Protože za účelem snížení závislostí byla odebrána platforma `WinForms`, není již možné použít výhozí aplikační nastavení k udržování seznamu cest.

Za tímto účelem byl přístup k cestám kompletně přepracován za využití nové třídy, která poskytuje kompletní seznam všech vstupních souborů. Třída `PcapPath` obsahuje výčet všech názvů testových souborů, které jsou k dispozici. Prostřednictvím hodnot z tohoto seznamu pak vývojář volá funkci `GetPcap(...)`, která mu už vrátí nalezenou cestu. Použitím výčtu také byla zachována možnost nápovědy v rámci vývojářských nástrojů.

Protože testové soubory mohou být uložené v různých relativních vzdálenostech, není možné udržovat přesné cesty k daným souborům. Tato funkce tedy nejdříve nalezne složku obsahující testovací data a až poté vytvoří kompletní cestu ke vstupnímu souboru. Ve výpisu 6.1 je pak vidět částečná implementace této třídy.

```
public static class PcapPath
{
  private const string TESTING_DATA_DIR = "TestingData";

  public enum Pcaps
  {
    appIdent_test, ...
  }

  public static string GetPcap(Pcaps pcap)
  {
    if (!_pcapPaths.ContainsKey(pcap))
      throw new KeyNotFoundException(
        $"Pcap \"{pcap}\" not found in PCAP dictionary");

    return Path.GetFullPath(Path.Combine(GetTestingDir(),
      _pcapPaths[pcap]));
  }

  private static string GetTestingDir()
  {
    string path = Directory.GetCurrentDirectory();
    while (!Directory.Exists(Path.Combine(path, TESTING_DATA_DIR)))
      path = Path.GetDirectoryName(path);

    return Path.Combine(path, TESTING_DATA_DIR);
  }
}
```

Výpis 6.1: Třída obsluhující sadu testových souborů

Dříve byl soubor definující všechny testy generován speciálně připraveným testem, což porušovalo význam použití testů v rámci projektu. Podařilo se tak odstranit tuto zvláštní praktiku, která byla identifikována v rámci analýzy v podsekci 4.3.2. V případě, že by měla být zachována metoda generátoru tohoto souboru, neměl by být zdrojový kód upravován některým z testů, ale měl by být vytvořen samostatný skript, který seznam všech souborů připraví.

Připravením nové cesty využívání souborů se také podařilo odstranit některé cesty a vstupní soubory, které v testech nebyly vůbec využívány. I přes to, že se nejedná o velké množství dat, podařilo se tak opět vyčistit další část tohoto projektu a ulehčit tak údržbu testových souborů.

6.2.2 Integrační testy testových souborů

Protože již nejsou samotné cesty k testovým souborům tvořené automatizovaně, ale je nutné je udržovat v rámci třídy `PcapPath`, bylo nutné také doplnit speciální testy, které májí za úkol kontrolovat existenci těchto souborů. Aby mohla být zajištěna alespoň částečná integrita těchto testů, kontroluje se také samotná signatura těchto vstupních souborů.

Operační systémy na bázi Unixu určují podle signatury formát souboru. Jedná se tak o několik prvních bytů, které daný soubor obsahuje. V případě vstupních souborů obsahujících odchycenou komunikaci se kontrolují hodnoty pro knihovnu `Libpcap` obsahující signaturu `a1 b2 c3 d4`, případně `d4 c3 b2 a1` a formát `PCAP NG` uvozený signaturou `0a 0d 0d 0a`.

6.2.3 Retenční mechanismus

Pro testovaní frameworku celého projektu a snooper modulů je připravena třída, která zajišťuje tvorbu odděleného prostředí pro každý test. `UnitTestBaseSetupFixture` obsahuje především metody, které se mají provést na začátku testů a po jejich skončení. Tyto metody (`void SetUp()` a `void TearDown()`) mají za úkol před spuštěním testu vytvořit databázi, která bude pro potřeby testu použita, zavést vstupní instance pro injekci závislostí (anglicky „dependency injection“), které zajišťuje `Castle.Windsor` a spustit časovač pro měření délky běhu testu. Po ukončení testu je pak časovač zastaven a jsou vypsány hodnoty na výstup terminálu.

Tato třída implementuje metody pro daný test, které nakopírují vstupní soubory do předem připravené složky. Před migrací byly tyto soubory kopírovány na systémový disk a po skončení testu zde byly ponechány. Nově byl zaveden retenční mechanismus odstraňování starých výstupů testů. Metoda `void OneTimeTearDown()` je spuštěna na konci všech proběhlých testů a stará se o smazání všech připravených složek. Všechny složky, které mají atribut posledního přístupu starší, než je nastavená konstanta `MAX_TEST_AGE_DAYS`, jenž je v současné době nastavena na dva dny, budou smazány.

Zavedení tohoto mechanismu bylo důležité, neboť spuštění všech testů v rámci projektu způsobilo kopírování několika stovek MB až jednotek GB dat. Z dlouhodobějšího horizontu tak může docházet ke zbytečnému úniku dat v řádu desítek, až stovek GB v závislosti na četnosti spuštění. Standardně by měl vývojář spouštět testy před každým uložením dat do sdíleného repositáře.

6.2.4 Jednotkové testy

Počet jednotkových testů v rámci celého projektu se z důvodu odstranění řady nesouvisejících částí poměrně snížil. V původním projektu se před migrací nacházelo zhruba 1010 testů. Ve chvíli, kdy celý projekt prošel migrací, snížil se celkový počet testů na 705. Odstraněných 305 testů náležely především projektům `Numl`, který byl přesunut do vlastního repositáře a `SIPFraud` balíčku, který v repositáři sice byl, nebyl ale nikde využíván.

V rámci úpravy přístupu k testovým souborům, která je dále rozepsána v podsekci 6.2.1, se také podařilo opravit zhruba 40 testů, které na své vstupní soubory odkazovali přímou cestou v souborovém systému. Ukázka takové opravy je vidět ve výpisu 6.2. I přes to, že v sobě test obsahuje přímou cestu ke vstupnímu souboru, je správně zařazen do kategorie „Explicit“, která zajistí, že test nebude spuštěn společně s ostatními.

```
1 [Test][Explicit][Category("Explicit")]
2 public void EPItestipluskal_appIdent_test1()
3 {
4   this.EpiTestCliBase(@"D:\pcaps\pcpluskal2\appIdent_test1.cap", 0.9);
5 }
```

Výpis 6.2: Explicitní test s přímou cestou k souboru

Oprava těchto testů spočívala v nahrazení explicitně vypsané cesty za připravenou funkci PcapPath.GetPcap(...), která testu na základě zadaného parametru dohledá správnou cestu k testovému souboru. Díky této opravě mohly být testy označené příznakem „Explicit“ uvedeny zpět do standardního provozu. Opravený příklad je možné vidět ve výpisu 6.3.

```
1 [Test]
2 public void EPItestipluskal_appIdent_test1()
3 {
4  this.EpiTestCliBase(PcapPath.GetPcap(PcapPath.Pcaps.appIdent_test1), 0.9);
5 }
```

Výpis 6.3: Opravený test bez přímé cesty k souboru a atributu Explicit

V případě, že jsou spuštěny všechny testy, úspěšně jich projde více, než 510. Ze zbylých zhruba 200 testů pak čtvrtina neprochází kvůli chybám, které se v průběhu migrace nepodařilo opravit. Tyto testy bohužel nebyly úspěšné ani v původním projektu před migrací. Některé z těchto testů se podařilo opravit, nicméně řada z nich selhává z důvodů různých změn, které v průběhu času probíhaly v architektuře sytému, přičemž pak samotné testy nebyly opraveny. Ostatní testy u sebe mají atribut „Explicit“, ať už z důvodu potřeby vysokého výkonu, nebo protože obsahují různé chyby.

Stran pokrytí kódu testy se pak pohybuje zpravidla kolem třiceti procent u projektových složek Common, Framework a Misc. Pokrytí uživatelské aplikace je pak odhadováno na pět procent, neboť testy grafického uživatelského rozhraní se v projektu prakticky nenacházejí, byť je jejich úspěšnost stoprocentní.

Protože zásadní část tohoto projektu staví na modulu VirtualizingCollection a ve zdrojovém repositáři[2] se nenachází prakticky žádné pokrytí testy, bylo nezbytné vytvořit alespoň minimální sadu ověřující funkčnost tohoto modulu. Protože práce s tímto modulem vyžaduje určitý počet tříd alespoň pro základní funkčnost, podařilo se implementovat několik testů ověřující správné chování některých funkcí, které jsou modulem poskytovány. Celkově se v modulu nachází více jak 1800 výrazů (anglicky označených jako „statement“), přičemž se podařilo pokrýt více, než 300 z nich, tedy zhruba 16 procent. I takto relativně malá sada testů ale objevila několik minoritních chyb, které se zde vyskytovaly. Všechny nalezené chyby se podařilo odstranit a zlepšit tak nejen správné fungování tohoto modulu, ale také podložit formou testů jeho správnou funkčnost.

6.2.5 Testovací datové sady

Jak už bylo zmíněno v předchozí podsekci, v rámci celého projektu se nachází více jak 700 jednotkových testů. Samotný projekt je možné testovat pouze těmito testy, ale pro plné ověření funkčnosti kódu je potřeba mít připravenou také dostatečnou sadu testovacích souborů. Jak vychází z předchozích sekcí, za tímto účelem se využívají soubory obsahující odchycenou komunikaci ve formátu .cap, .pcap, případně .pcapng. V rámci celého projektu se před migrací v repositáři TestingData nacházelo 240 těchto souborů. Celková velikost těchto souborů pak dosahovala zhruba 2,10 GB. Jedná se tak o poměrně rozsáhlou databázi vstupních souborů.

Pro kvalitní testování není důležitý pouze počet, nebo velikost zabraného místa. Velmi záleží na rozprostření těchto dat napříč samotnými částmi celé aplikace Netfox a všech jejich přidružených částí. V následujících bodech jsou rozepsány jednotlivé části, ze kterých

[2] https://github.com/evgenekov/VirtualizingObservableCollection

se tento projekt skládá a u každého bodu pak i stručné zhodnocení toho, jak dobré je pokrytí z hlediska vstupních souborů.

- Základní stavební kámen celého projektu, složka `Common`, obsahuje především sadu modelů, rozhraní a různých podpůrných funkcí. Zároveň se zde nachází modul pro zpracování zpráv, které aplikace generuje, nebo modul pro práci s databázovým strojem. Protože tyto součásti slouží jako podpora celého projektu, nenachází se zde nic, co by aktivně zpracovávalo odchycenou síťovou komunikaci. Nedává tedy smysl, aby se pro tuto část vůbec nějaká vstupní data generovala.

- O třídu výše se nachází část `Framework`, jenž zastřešuje nejen výpočetní jádro a byznys logiku celé aplikace, ale také se zde nachází řada analyzátorů jednotlivých síťových protokolů. Modul `CaptureProcessor`, který se stará o předzpracování vstupních souborů, je testován z hlediska jeho správné činnosti. V rámci testů se tedy ověřuje, že předzpracování proběhlo v pořádku a výstup tak odpovídá očekávaným hodnotám. Tento modul je testován za pomocí devíti různých datových souborů, které se svou charakteristikou liší z hlediska toho, zda obsahují fragmentovaná data, různě velká data, nebo například přetečení sekvenčního čísla v rámci komunikace. Jedná se tak o poměrně rozmanitou sadu případů, se kterými se musí tato část aplikace vypořádat.

  Velmi podobně je na tom také modul `ApplicationProtocolExport`, jenž je ověřován především z hlediska správné funkčnosti při dešifrování dat za pomocí klíče. Využívá se zde dohromady sedmi různých vstupních souborů, které se skládají z odlišných aplikačních protokolů.

- Samotné analyzátory jednotlivých aplikačních protokolů, neboli jednotlivé `snoopery`, pak celkově využívají ke svému testování více, než 150 souborů s odchycenou síťovou komunikací. Rozptyl využití těchto souborů je pak v rozsahu jeden, až 53 testů. Medián počtu souborů na jednotlivé analyzátory je pak přesně na hodnotě tři. Vzhledem k tomu, že samotné analyzátory nejsou velmi rozsáhlými projekty, neboť se jedná o zpětnou rekonstrukci daného aplikačního protokolu, lze považovat tuto hodnotu za přiměřenou a vyhovující. Rozložení počtu vstupních souborů je pak adekvátně přiměřené složitosti daného protokolu. Například analyzátor `SnooperTwitter` je malý modul o několika krátkých třídách a proto v testech využívá pouze dvou vstupních souborů. Protipólem je pak `SnooperSIP`, jehož implementace je poměrně složitá a obsahuje proto také velkou řadu testů. Celkově se pro tento modul v projektu nachází 61 testů, a prakticky pro každý test je vytvořen vlastní vstupní soubor obsahující speciálně připravená data.

- `Netfox.Detective` a `Netfox.Web`, obsahující uživatelské aplikace pro práci s tímto nástrojem, neslouží k samotné analýze dat, ale obstarávají komunikaci mezi uživatelem a jádrem aplikace. Je tedy zcela logické že pro tyto součásti už opět vstupní data nejsou přímo potřeba a ověření správné funkčnosti je zde nutné řešit jinak. Tento fakt také vyplývá z podsekce 6.1.4, která říká, že pokud určitá testovaná komponenta vyžaduje ke své činnosti komponentu jinou, je potřeba její funkčnost pouze simulovat.

Na začátku této sekce bylo řečeno, že se v rámci repositáře nacházelo 240 testovacích souborů. Tato sada byla ještě doplněna o zhruba 10 dalších souborů, které v repositáři chyběly. Protože se jednalo o poměrně velké soubory, celková velikost této sady vyrostla na více, než 14 GB. Z tohoto důvodu bylo také rozhodnuto, že složka obsahující tuto

sadu se nebude nacházet společně se zbytkem projektu, ale bude do projektu připojena jako podmodul systému Git. Je tak zaručena bezproblémová práce s projektem v případě jakékoliv manipulace se samotným repositářem. V případě, že by testové soubory nebyly odděleny, některé akce systému Git by mohly způsobovat komplikace z důvodu časově náročného zpracování.

Celkově se tak jedná o velmi masivní sadu vstupních dat, kde velikost některých z nich přesahuje i jednotky GB. Samotné rozložení vstupních souborů napříč projektem je velmi vyvážené a to i s ohledem na rozmanitost aplikačních analyzátorů. Celkově se tak jedná o velmi kvalitní a rozmanitý zdroj dat, který neobsahuje vždy pouze konkrétní aplikační protokol, ale nacházejí se zde i soubory obsahující více různých protokolů naráz. Z těchto důvodů bylo rozhodnuto, že v aktuálním stavu není potřeba tvořit žádné další obdobné soubory pro tento projekt a ušetřený čas byl investován do jiných částí, které zjednodušují budoucí práci na tomto projektu.

6.3 Popis manuálních testů

V rámci celé migrace bylo pro ověření funkčnosti využíváno také manuálního testování. Zdrojem správných výsledků byla aplikace před migrací celého projektu. Na základě těchto testů pak byly opravovány nalezené chyby. Jakmile byla chyba opraveny, v testu se již dále nepokračovalo, ale celá testovací sada byla spuštěna znovu od začátku.

6.3.1 Smoke testy

Jak už bylo popsáno v podsekci 6.1.4, takzvané „smoke" testy slouží k ověření funkčnosti hlavních částí aplikace. Zpravidla se v rámci testů ověřuje správné načtení všech dostupných obrazovek[3] a splnění jejich hlavních účelů. Testy jako takové se nezaměřují na kontext dané obrazovky, tedy není důležitá posloupnost procházení jednotlivými částmi aplikace. Cílem testů je tedy ověření, že daná aplikace splňuje svůj účel.

Na základě požadavků daného projektu může být granularita testů v různých částech aplikace odlišná. Například aplikace zaměřena na zobrazení dat v tabulkách bude podrobněji testovat filtrování, nebo řazení zobrazovaných dat. Menší důraz pak může být kladen například na změnu barevného tématu aplikace.

Odlišné testování se bude také projevovat na základě toho, pro koho je daná aplikace určena. Pokud je cílovou skupinou široká veřejnost, bude kladen větší důraz na to, aby byly otestované krajní případy. V opačném případě může být kladen velký důraz na správnost dat.

Samotné provedení testů pak může být jak automatizované, tak manuální. Protože tento typ testů je zaměřen na celkovou funkčnost aplikace, snaží se testeři tvořit testy automatizované. Největší výhodou automatizovaných testů je jejich celková rychlost, která bývá zpravidla mnohonásobně vyšší, než u manuálního testování. Zároveň je možné testy spouštět paralelně a častěji. U rozsáhlejších projektů je tendence spouštět tyto testy co nejčastěji, aby byla případná chyba odhalena co nejdříve. Nejčastěji se testy spouštějí po vyvinutí větší nové funkcionality, nebo například před vydáním nové verze do produkce a zároveň i po jejím vydání, pro ověření funkčnosti v produkčním prostředí. Nevýhodou automatizace je fakt, že je třeba testy v případě zásadnějších změn chování aplikace aktualizovat. Při manu-

[3] Za obrazovku je považována každé okno, nebo podokno aplikace, se kterým může uživatel jakýkoliv způsobem interagovat.

álním testování tester prochází postupně zadanými kroky a kontroluje, zda výstup aplikace odpovídá očekávaným výsledkům.

V kontextu této práce se pak jednalo o sadu kroků a úkonů, které jsou popsané v tabulce v příloze B.1. Každý z testů se skládá z jeho názvu, sady kroků, které je potřeba vykonat a očekávaného výsledků. V rámci rozsáhlejších testů se v názvu mohou využívat testovací kódy, které slouží pro vývojáře, aby mohli snáze identifikovat test, který selhal. Aby bylo zaručeno, že základní funkcionalita aplikace nebyla porušena, před migrací bylo třeba připravit si základní testovací scénáře, podle kterých bylo následně ověřováno zda migrace nezpůsobila poškození některých ze základních součástí aplikace.

I přes to, že tato testovací sada vypadá na první pohled relativně triviálně, jako minimální základ testů svůj účel splnila. Podařilo se díky ní odhalit některé velmi důležité chyby, které se po migraci projektu objevily. První test, který po migraci projektu selhal, byl test číslo 2 — Vytvoření nového pracovního prostoru. I přes to, že byl pracovní prostor vytvořen, nebyla v seznamu investigací žádná položka a zároveň nebyl splněn ani druhý bod očekávaného výsledku, tedy seznam dostupných analyzátorů byl prázdný. Testovací proces bylo tedy nutné přerušit, zjistit, co danou chybu zapříčinilo a chybu pak odstranit. Jakmile byla odstraněna, celá testovací sada byla procházena opět od prvního testu, aby bylo zajištěno, že oprava nezpůsobí neúspěch některého z předchozích testů.

V průběhu migrace, testování i opravy chyb byly testovací scénáře rozšiřovány o další, podrobnější případy. Zajímavým příkladem je třetí test — Vložení nového zdrojového souboru a spuštění analýzy. V tabulce B.1 je test popsán v původní podobě, tedy přidání jednoho testovacího souboru a k němu jednoho patřičného analyzátoru. Kroky tohoto testu byly rozšířeny o přidání více, než jednoho testovacího souboru a analyzátoru. Po provedení testu bylo zjištěno, že se ve výsledcích zobrazuje výstup pouze z jednoho analyzátoru a ostatní zvolené analyzátory vykazovaly prázdné výsledky. Tuto chybu způsobila absence knihovny `DotVVM`, která se v tomto případě starala o správné mapování tříd do grafického uživatelského rozhraní.

6.3.2 End-to-end testy

Tento typ testů ověřuje komplexnost životních cyklů jednotlivých uživatelů napříč celou aplikací. Testování sestává ze sady úkonů, které má být schopen uživatel udělat. Testuje se, zda dané komponenty do sebe logicky zapadají a zda jednotlivé posloupnosti akcí dávají uživateli smysl. Cílem je ověřit, že je uživatel schopen pomocí dané aplikace vybraný úkon splnit. Na nejvyšší úrovni pohledu se testuje, zda daný produkt skutečně naplňuje podstatu své existence.

End-to-end testy se provádějí zpravidla na konci projektu a mohou být součástí akceptačních testů. Testovacími sadami s úkoly procházejí jak testeři, tak i vybraná sada koncových uživatelů. Zjišťuje se tak, že je zákazník schopen pohodlně aplikaci ovládat a dokončit tak úspěšně dané úkoly. Jednotlivé úkoly reflektují uživatelské příběhy, které obsahují různé aplikační procesy. Příkladem může být test v rámci internetového obchodu — nákup mobilní telefonu. Jednotlivé kroky tohoto testu mohou odpovídat přihlášení do aplikace, výběru mobilního telefonu, platbě a odhlášení z aplikace. Testovací subjekt může po ukončení testu podávat zpětnou vazbu, která je považována za velmi kvalitní a důležitou, neboť pochází přímo od koncového zákazníka.

V příloze B.2 je vidět příklad testovacích scénářů, který byl aplikován v rámci tohoto projektu. Třetí a čtvrtý z těchto scénářů pomohl odhalit další nedokonalosti, které se v aplikaci nacházely. V prvním případě se nepodařilo dokončit úspěšně test z důvodu, že se v pří-

praveném okně správce úloh nenacházely žádné položky i přes to, že tam měly být. Tento problém byl způsoben chybným mapováním tříd v migrovaném projektu.

Druhý z problémů odhalil špatné ukládání nastavení. Změna nastavení sice proběhla, ale z důvodu odlišného přístupu k ukládání nastavení právě k uložení nedošlo. V průběhu času byly tyto testy postupně rozšiřovány, což pomohlo k odhalení dalších chyb v kódu. Všechny takto nalezené chyby byly opraveny a uloženy do repositáře.

6.3.3 Další manuální testy

V průběhu jakéhokoliv vývoje se zpravidla každý vývojář setká s ladícími nástroji pro odstraňování chyb. Procesu, při kterém se tyto nástroje využívají, se říká „debugging“. V průběhu celé migrace se tak stávalo, že bylo potřeba tyto chyby odstranit, a protože velmi často nebylo na první pohled jasné, kde se chyba nachází, nezbývalo než využít ladících nástrojů. Velmi často bylo potřeba dohledat, proč v určitých místech například chybí instance některé třídy, nebo proč se zobrazuje hodnota jiná, než která by na daném místě měla být.

Jedním z velmi kvalitních ladících nástrojů je možnost využití takzvaných bodů zastavení (v angličtině „breakpoint“), který daný kód v nastaveném okamžiku zastaví a umožní tak vývojáři zkontrolovat obsahy jednotlivých proměnných, prohlédnout si zásobník volání (anglicky „call stack“), nebo dokonce i určité proměnné pozměnit. Jedná se tak částečně o další způsob testování, které provádí přímo vývojář sám a to pouze v okamžiku, když na nějaký problém přijde.

6.4 Zhodnocení refaktorizace s využitím testů

Samotná migrace kódu na novou platformu nebyla jednoduchá a bylo třeba řešit poměrně velkou část problémů, které byly spojené především se špatnou kompatibilitou s aktualizovanými balíčky, nebo samotnou nekompatibilitou mezi .NET 5 a .NET Framework. Díky tomu, že bylo potřeba celý kód projít, bylo snazší narazit na některé části, které bylo třeba upravit.

Refaktorizace jako taková nezahrnuje pouze úpravu čistého kódu, ale zahrnuje také změny, které byly provedeny v rámci větších aplikačních celků. Příkladem může být změna přístupu k nastavení, nebo testovým souborů. Jednalo se tak o úpravu relativně důležité části projektu, jenž určuje jeho chování. Všechny takto upravené části se podařilo refaktorovat úspěšně. Zároveň se tak otevřely cesty k rozšíření testů, které nově kontrolují i správnou funkci těchto součástí.

Určité části projektu, které byly upraveny, napomohli snazšímu testování. Příkladem je úprava třídy `IPAddressEF`, jenž je popsána ve výpisu 5.11. Zde došlo k odstranění velké části kódu, který by musel být testován. Protože se ale využívá základní třídy z knihovny `System.Net`, lze počítat s tím, že si autoři dají dostatečný pozor na to, aby třída fungovala bezchybně. I tak byly ale pro jistotu doplněny i některé testy pro ověření správné funkčnosti této třídy.

6.4.1 Pokrytí kódu testy

V samotném projektu se nachází více, než 70 tisíc výrazů[4], které nejsou přímo pokryty žádnými testy. Pokud by byl kladen důraz pouze na jádro aplikace, tedy složky `Framework`

[4]Všechny hodnoty jsou měřeny pomocí vestavěných nástrojů aplikace Jetbrains Rider.

a `Common`, které provádějí hlavní činnost, jedná se zhruba o 53 tisíc výrazů, které nemají přímé pokrytí testy. V procentuálních hodnotách je pak celkové pokrytí kódu na hodnotě 18 procent. Složka `Common` je pokryta z 35 procent, `Framework` z 20 procent a samotná aplikace je pokryta z pěti procent. Z pohledu standardů, které jsou shrnuty Svevem Cornettem [13] se pak cílí zpravidla na hodnoty pokrytí 100 procent u kritických částí systému. Zároveň je ale patrné, že maximální pokrytí nemusí být vždy efektivní jak z hlediska nákladů na vývoj, tak z hlediska testování. Samotná hodnota totiž neurčuje kvalitu pokrytí, ale pouze jeho rozsah. Při vyčíslování například není zohledňována kontrola krajních případů [17].

I přes to, že se v průběhu migrace podařilo několik stovek výrazů pokrýt, samotné doplnění testů na určitý standard by vyžadovalo nejen obrovské úsilí ale především hlubokou znalost samotného systému, i celé problematiky. Jako standardní hodnotu z praxe lze považovat pokrytí zhruba 70 procent, které doporučuje přímo firma Google [5]. V úvahu je ale také možné brát Paretův princip, který říká, že 80 procent důsledků pramení z 20 procent příčin [60]. V přeneseném kontextu lze tak brát v úvahu, že pouze 20 procent vytvořeného kódu bude využíváno v 80 procentech času. Na základě této úvahy by pak maximální pokrytí kódu mělo směřovat právě k 20 procentům nejčastěji spouštěného kódu.

Kapitola 7

Závěr

V první části této diplomové práce jsem se zaměřil nejdříve na jednotlivé kroky, které je potřeba podstoupit, pokud se snažím refaktorovat nějaký kód. Zároveň jsem se zaměřil na různé aspekty udržování čistého kódu, které byly v rámci samotného procesu refaktorizace zohledňovány. Příkladem bylo dostatečné anotování funkcí, udržování krátkých funkcí, a především dodržení návrhových vzorů.

V druhé polovině první části této práce jsem pronikal do útrob samotného nástroje Netfox Detective a jeho základu Netfox Framework, abych pochopil, jakým způsobem pracuje a proč je navržen tak, jak autoři zvolili. Výše uvedené spočívalo v prostudování několika dalších bakalářských a diplomových prací, které se zabývaly právě vývojem tohoto nástroje. V průběhu studia jsem narazil na problémy týkající se rozběhnutí nejen celého projektu, ale i samotných testů. Část těchto testů využívala nefungující metody a stávalo se, že testy odkazovaly na vstupní data, která ke svému běhu potřebují, přičemž daná data nebyla součástí repositáře.

V celém projektu jsem identifikoval několik implementačních problémů, které bylo třeba v druhé části této práce vyřešit. Velmi často se v projektu vyskytoval nečinný kód, nebo kód, který byl zakomentovaný. Narazil jsem také na zdánlivý problém související s obousměrnou závislostí. Ukázalo se ale, že veškeré závislosti jsou opodstatněné. Podařilo se extrahovat velkou řadu částí do samostatných balíčků, které jsou pak projektem importovány.

V rámci testovací části se podařilo opravit řadu testů. Zároveň bylo několik testů doplněno. Testování v rámci této práce nezahrnovalo pouze jednotkové testy, ale zaměřil jsem se také na manuální testy ve formě end-to-end, nebo smoke testů. Určité části testovacího prostředí byly upraveny a doplněny. Jednalo se především o změnu metodiky přístupu k testovým souborům. Došlo ale i na několik dalších ošetření, jako například automatické mazání dočasných složek vytvořených pro jednotlivé testy.

Z retrospektivního pohledu byla práce na tomto projektu velmi zajímavá. Jedná se totiž o dílo, které má již svou historii a uplatnění. Mám tedy jistotu, že má odvedená práce je aktivně zapojena do fungujícího systému a zároveň může sloužit jako inspirace pro další projekty, které se budou setkávat s obdobnou tématikou. V další části této závěrečné kapitoly se mi podařilo identifikovat některé části projektu, které by bylo možné implementovat v rámci případných budoucích závěrečných prací.

Na úplný závěr tohoto textu bych ještě rád doplnil citaci Martina Flowera, který je autorem jedné z nejznámějších knih o refaktorování, *Refactoring: improving the design of existing code.*. Přímá citace zní následovně: „Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“

7.1 Možná další rozšíření

Ve srovnání s dalšími produkty na trhu jsem vytvořil krátký seznam poznatků, které by bylo možné v rámci Netfox Detective aplikace doplnit.

Dokumentace kódu

Každý kód by měl být sám za sebe vysvětlující. Slouží k tomu různá pravidla, jako je tvorba krátkých funkcí, používání krátkých komentářů, nebo správná volba názvu jednotlivých funkcí a tříd. Dodržování těchto pravidel může pomoci lépe pochopit, co daná funkce dělá, nicméně nezaručuje, že člověk pochopí, jak kód pracuje jako celek. Velké projekty, jako je například právě Netfox, se skládají z mnoha dalších částí a proto není vždy jednoduché pochopit celý princip pouze z kódu samotného.

Za tímto účelem jsou pro různé aplikace tvořeny obsáhlé dokumentace, které vysvětlují vnitřní mechanismy fungování a dodávají vývojáři celkový pohled na principy projektu. Zároveň by tak byla usnadněna práce vývojářům, kteří si pro tento projekt chtějí vytvořit vlastní aplikační analyzátor.

Jazyková mutace

Protože je rozhraní Netfox Detective napsané v angličtině, není tento bod nezbytně nutný k implementaci. Nicméně se stále najdou mocnosti, kde převládá jiný jazyk, než angličtina. Případnému rozšíření tohoto produktu do jiných zemí, by pomohla možnost změnit, či dokonce doplnit, vlastní jazykovou mutaci.

Podpora více operačních systémů

Aby bylo možné rozšířit aplikaci na více operačních systémů, je potřeba nahradit `WPF` framework za systém, který je podporován více operačními systémy. Příkladem tak může být přepsání uživatelské aplikace do některého z JavaScript frameworků pro tvorbu desktopových rozhraní.

Tmavé téma

V dnešní době je velmi rozšířená podpora různých barevných témat. Zpravidla se používá světlá a tmavá varianta grafického uživatelského rozhraní. Současný trend udává, že se změna barevného tématu stává standardem.

Literatura

[1] Aivosto developpers. *Complexity metrics* [online]. [cit. 2020-12-07]. Dostupné z: https://www.aivosto.com/project/help/pm-complexity.html.

[2] Akbaş, B. *NuGet 101: Introduction to NuGet* [online]. Leden 2020 [cit. 2021-05-01]. Dostupné z: https://medium.com/@burcakbas/nuget-101-introduction-to-nuget-800c8fa86573.

[3] Allan, M. *What Is Iterative Development – An Easy Guide For Beginners* [online]. Říjen 2019 [cit. 2021-05-13]. Dostupné z: https://www.goodcore.co.uk/blog/iterative-development.

[4] Ambrož, T. *Analytické webové prostředí pro zpracování síťové komunikace.* Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22049.

[5] Arguelles, C., Ivanković, M. a Bender, A. *Code Coverage Best Practices* [online]. Červenec 2020 [cit. 2021-05-07]. Dostupné z: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

[6] Atlassian. *Gitflow Workflow* [online]. Leden 2020 [cit. 2021-05-01]. Dostupné z: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[7] Chand, M. *Difference Between .NET and .NET Core* [online]. Květen 2020 [cit. 2021-01-18]. Dostupné z: https://www.c-sharpcorner.com/article/difference-between-net-framework-and-net-core.

[8] Chauhan, S. *Implementation of Dependency Injection Pattern in C#* [online]. Duben 2013 [cit. 2021-01-05]. Dostupné z: https://www.dotnettricks.com/learn/dependencyinjection/implementation-of-dependency-injection-pattern-in-csharp.

[9] Chess, B. a Mcgraw, G. *Static analysis for security* [online]. IEEE, prosinec 2004 [cit. 2020-11-22]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1366126.

[10] Collective Wisdom from the Experts. *97 Thing Every Programmer Should Know.* O'Reilly Media, Inc., únor 2010. ISBN 978-0-596-80948-5.

[11] Computing for the Social Sciences. *Why Git?* [online]. Leden 2021 [cit. 2021-04-30]. Dostupné z: https://cfss.uchicago.edu/setup/what-is-git.

[12] Copper M.. *Difference among Black Box, White Box and Grey Box Testing* [online]. Prosinec 2018 [cit. 2021-05-03]. Dostupné z: https://medium.com/@mcccccc/difference-among-white-box-black-box-and-grey-box-testing-35482292473f.

[13] Cornett, S. *Minimum Acceptable Code Coverage* [online]. [cit. 2021-05-13]. Dostupné z: https://www.bullseye.com/minimum.html.

[14] Dajbych, V. *MVVM: model-view-viewmodel* [online]. Duben 2009 [cit. 2021-01-05]. Dostupné z: https://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel.

[15] Dofactory. *C# Coding Standards and Naming Conventions* [online]. [cit. 2021-01-05]. Dostupné z: https://www.dofactory.com/reference/csharp-coding-standards.

[16] Foote, B. a Yoder, J. *Big Ball of Mud* [online]. Září 2000 [cit. 2020-11-18]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.6449&rep=rep1&type=pdf.

[17] Fowler, M. *TestCoverage* [online]. Duben 2012 [cit. 2021-05-13]. Dostupné z: https://martinfowler.com/bliki/TestCoverage.html.

[18] Fowler, M. *IntegrationTest* [online]. Leden 2018 [cit. 2021-05-08]. Dostupné z: https://martinfowler.com/bliki/IntegrationTest.html.

[19] Gamma, E., Helm, R., Johnson, R. E. a Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Addison-Wesley Professional Computing Series. ISBN 978-0-201-63361-0.

[20] Gomes, I., Morgado, P., Gomes, T. a Moreira, R. *An overview on the Static Code Analysis approach in Software Development* [online]. Únor 2020 [cit. 2020-11-21]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.734.6506&rep=rep1&type=pdf.

[21] Gomes, P. *Unit Testing and the Arrange, Act and Assert (AAA) Pattern* [online]. Září 2017 [cit. 2021-05-03]. Dostupné z: https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80.

[22] Grossmann, L. *Lean Software Development a Kanban* [online]. Srpen 2020 [cit. 2021-04-28]. Dostupné z: https://www.itnetwork.cz/navrh/metodiky/lean-software-development-a-kanban.

[23] Hlava, T. *Testování bílé a černé skříňky* [online]. Srpen 2011 [cit. 2021-05-03]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/testovani-bile-a-cerne-skrinky-white-box-black-box-grey-box.

[24] HOLOTA, L. *Čistý kód v jazyce C#* [online]. 2015 [cit. 2020-11-18]. Bakalářská práce. Vysoká škola ekonomická v Praze, Praha. Vedoucí práce Buchalcevová, A. Dostupné z: https://theses.cz/id/llvlxc.

[25] Horděјčuk, V. *Návrhové vzory* [online]. [cit. 2020-11-18]. Dostupné z: http://voho.eu/wiki/navrhovy-vzor.

[26] Hunt, A. a Thomas, D. *Keep It DRY, Shy, and Tell the Other Guy* [online]. Duben 2004 [cit. 2020-11-23]. Dostupné z: http://cs.wlu.edu/~sprenkle/cs209/readings/dry_shy.pdf.

[27] Inflectra. *What are Scrum Task Boards?* [online]. Leden 2020 [cit. 2021-04-30]. Dostupné z: https://www.inflectra.com/ideas/topic/using-a-task-board.aspx.

[28] Jeelani, K. *What is .NET Technology?* [online]. Únor 2020 [cit. 2021-01-18]. Dostupné z: https://www.c-sharpcorner.com/blogs/what-is-net-technology.

[29] Jeffrey, N. *Visual Programming* [online]. 1994 [cit. 2020-12-08]. Dostupné z: https://web.stevens.edu/jnickerson/ch7.pdf.

[30] Jonáš, M. *Návrhové principy: SOLID* [online]. Květen 2012 [cit. 2021-01-05]. Dostupné z: https://www.zdrojak.cz/clanky/navrhove-principy-solid.

[31] Kan, S. H. *Metrics and Models in Software Quality Engineering*. 2. vyd. 2001. ISBN 0-201-72915-6.

[32] Katalon. *What is End-to-End (E2E) Testing? All You Need to Know* [online]. [cit. 2021-05-08]. Dostupné z: https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing.

[33] Kuipers, T. a Visser, J. *Maintainability Index Revisited* [online]. 2007 [cit. 2020-12-08]. Dostupné z: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.568.4007&rep=rep1&type=pdf.

[34] Kumar, A. *Best Practices Of Writing C# Code* [online]. Květen 2018 [cit. 2021-01-05]. Dostupné z: https://www.c-sharpcorner.com/article/best-practice-of-write-c-sharp-code.

[35] Lippert, M. a Roock, S. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. 1. vyd. 2006. ISBN 978-0-470-85893-6.

[36] Lopez, M. a Habra, N. *Relevance of the Cyclomatic Complexity Threshold for the Java Programming Language* [online]. 2005 [cit. 2020-12-07]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.1789&rep=rep1&type=pdf.

[37] Mareš, M. *Nástroj pro analýzu obsahu síťové komunikace*. Brno, CZ, 2014. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/16832.

[38] Martin, R. C. *Čistý kód: návrhové vzory, refaktorování, testování a další techniky agilního programování*. 1. vyd. Computer Press, 2009. ISBN 978-80-251-2285-3.

[39] Microsoft documentation. *Documentation comments* [online]. Leden 2017 [cit. 2021-01-18]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments.

[40] Microsoft documentation. *Common web application architectures* [online]. Leden 2020 [cit. 2021-01-18]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.

[41] Microsoft documentation. *.NET 5 vs. .NET Framework pro serverové aplikace* [online]. Duben 2021 [cit. 2021-05-03]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/standard/choosing-core-framework-server.

[42] Nette. *Dependency Injection* [online]. [cit. 2021-01-05]. Dostupné z: https://doc.nette.org/cs/3.0/dependency-injection.

[43] Nguyen, L. *Lessons learnt from "The Clean Code" — Robert C. Martin* [online]. Červenec 2018 [cit. 2021-01-05]. Dostupné z: https://medium.com/@huytrongnguyen1985/lessons-learnt-from-the-clean-code-robert-c-martin-cecbe2b09139.

[44] Pluskal, J. *Framework for Captured Network Communication Processing*. Brno, CZ, 2014. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/16748.

[45] Pluskal, J. *Netfox Detective 2.0 - Nástroj pro síťovou forenzní analýzu*. 2017. 16 s. Dostupné z: https://www.fit.vut.cz/research/publication/11567.

[46] Pluskal, J., Breitinger, F. a Ryšavý, O. Netfox detective: A novel open-source network forensics analysis tool. *Forensic Science International: Digital Investigation*. 2020. ISSN 2666-2817. Dostupné z: https://www.sciencedirect.com/science/article/pii/S2666281720300871.

[47] Pluskal, J., Matoušek, P., Ryšavý, O., Kmeť, M., Veselý, V. et al. Netfox Detective: A tool for advanced network forensics analysis. In: *Proceedings of Security and Protection of Information (SPI) 2015*. University of Defence in Brno, 2015, s. 147–163. ISBN 978-80-7231-997-8. Dostupné z: https://www.fit.vut.cz/research/publication/10863.

[48] Sacolick, I. *What is CI/CD? Continuous integration and continuous delivery explained* [online]. Leden 2020 [cit. 2021-04-30]. Dostupné z: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html.

[49] Sambasivam, G. *Extreme programming* [online]. Září 2011 [cit. 2021-04-28]. Dostupné z: https://www.unf.edu/~broggio/cen6016/ExtremeProgramming(XP)Article.pdf.

[50] Sherstobitov, V. *Source Code Metrics for Quality in Java*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21650.

[51] Slámová, H. *Refaktorizace síťového forenzního nástroje Netfox Detective*. Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20380.

[52] Smith, C. U. a Williams, L. G. *Software Performance AntiPatterns* [online]. Leden 2001 [cit. 2020-11-18]. Dostupné z: https://www.researchgate.net/publication/221445933_Software_Performance_AntiPatterns_Common_Performance_Problems_and_their_Solutions

[53] Software Testing Help. *What Is Software Testing Life Cycle (STLC)?* [online]. Srpen 2020 [cit. 2021-05-07]. Dostupné z: https://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc.

[54] Software Testing Help. *What Is A JUnit Test Fixture* [online]. Duben 2021 [cit. 2021-05-01]. Dostupné z: https://www.softwaretestinghelp.com/junit-test-fixture-with-examples.

[55] SourceMaking. *Swiss Army Knife* [online]. [cit. 2021-01-05]. Dostupné z: https://sourcemaking.com/antipatterns/swiss-army-knife.

[56] Verifysoft Technology. *Measurement of Halstead Metrics with Testwell CMT++ and CMTJava (Complexity Measures Tool)* [online]. [cit. 2020-12-08]. Dostupné z: https://www.verifysoft.com/en_halstead_metrics.html.

[57] Vodrážka, M. *Způsoby definování požadavků pro výkonnostní testování softwaru.* Prague, CZ, 2015. Diplomová práce. Vysoká škola ekonomická v PrazePraha. Dostupné z: https://vskp.vse.cz/45290.

[58] Wikipedia: The Free Encyclopedia. *Cyclomatic complexity* [online]. Prosinec 2020 [cit. 2020-12-07]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=992651876.

[59] Wikipedia: The Free Encyclopedia. *Liskov substitution principle* [online]. Prosinec 2020 [cit. 2021-01-05]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Liskov_substitution_principle&oldid=994619575.

[60] Wikipedia: The Free Encyclopedia. *Pareto principle* [online]. Duben 2021 [cit. 2021-05-13]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Pareto_principle&oldid=1020345663.

[61] Wikipedia: The Free Encyclopedia. *Scrum (software development)* [online]. Duben 2021 [cit. 2021-04-28]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Scrum_(software_development)&direction=prev&oldid=1019958361.

[62] Wikipedia: The Free Encyclopedia. *Test-driven development* [online]. Duben 2021 [cit. 2021-04-28]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Test-driven_development&direction=prev&oldid=1020619802.

[63] Zemek, P. *Proč rozlišovat jednotkové a integrační testy* [online]. Duben 2015 [cit. 2021-05-08]. Dostupné z: https://cs-blog.petrzemek.net/2015-04-18-proc-rozlisovat-jednotkove-a-integracni-testy.

Příloha A

Obsah přiloženého paměťového média

Na DVD, které je přiloženo k této práci se nachází:

- tato práce ve formátu `.pdf`,
- zdrojové soubory textu této bakalářské práce v jazyce LaTeX,
- složka `netfox` obsahující veškeré zdrojové kódy z repositáře `NetfoxCore` bez testovacích dat,
- instalační soubor ve formátu `.msi`.

Příloha B

Testovací sady pro manuální testy

#	Proces	Výchozí stav	Kroky	Očekávaný výsledek
1	Spuštění a ukončení aplikace		1. Spuštění projektu do uživatelského rozhraní 2. Uzavření aplikace přes tlačítko v menu, nebo křížek.	Aplikace bude bez chyby spuštěna a ukončena
2	Vytvoření nového pracovního prostoru	Spuštěná aplikace	1. Přidání nového pracovního prostoru tlačítkem + 2. Potvrzení tlačítkem pro vytvoření	Vytvořený a otevřený nový pracovní prostor, který bude obsahovat jednu investigaci. V rozhraní investigace je výběr všech dostupných analyzátorů síťového provozu.
3	Vložení nového zdrojového souboru a spuštění analýzy	Otevřený pracovní prostor	1. Po stisknutí tlačítka pro přidání souboru vybrat soubor s daty k analýze 2. Zvolení patřičného analyzátoru podle datového souboru 3. Spuštění analýzy	Analýza bude bez chyby dokončena. Podle zvoleného analyzátoru bude v seznamu výsledků nenulová hodnota. Počet rozpracovaných úkolů na pozadí je roven nule.
4	Otevření existujícího pracovního prostoru	Spuštěná aplikace	1. Spuštění aplikace a poklepání na existující pracovní prostor	Pracovní prostor bude úspěšně načtený. V seznamu zdrojových souborů bude soubor z minulého testu.
5	Zobrazení výstupu analýzy	Dokončená analýza	1. Poklepání na patřičný analyzovaný výstup 2. Přepnutí na záložku Exports 3. Poklepání na některý z exportovaných výsledků	V dokončené analýze se vyskytují nějaké výsledky. Extrahovaný výsledek obsahuje relevantní data.

Tabulka B.1: Část úkonů prováděných v rámci „smoke“ testů

#	Aktivita	Akční kroky	Očekávaný výsledek
1	Extrahování dat z datového souboru	1. Spusť aplikaci Netfox Detective 2. Vytvoř nový pracovní prostor 3. Vlož nový testový soubor (imap_smtp_collector.pcap) 4. Zvol IMAP analyzátor a spusť analýzu 5. Otevři výsledek analýzy a přepni se na záložku Email 6. Rozklikni email, který obsahuje přílohu 7. Otevři přílohu 8. Ukonči aplikaci	Testerovi se podaří otevřít extrahovanou přílohu bez poškození. Tester nenarazí na žádný problém v průběhu žádného z předepsaných kroků.
2	Analýza datového souboru	1. Spusť aplikaci Netfox Detective 2. Otevři pracovní prostor z minulého scénáře 3. Poklepej na vložený datový soubor 4. Zjisti velikost datového souboru 5. Ukonči aplikaci	Tester nenarazí na žádné komplikace v průběhu testu. Tester zjistí, že velikost datového souboru je 10209222 B.
3	Kontrola běžících úkolů na pozadí	1. Spusť aplikaci Netfox Detective 2. Otevři pracovní prostor z minulého scénáře 3. Zvol IMAP analyzátor a spusť analýzu 4. Počkej, až je analýza dokončena 5. Otevři správce úloh (View ->Tasks) 6. Zkontroluj, že jsou všechny úlohy dokončeny 7. Ukonči aplikaci	Tester nenarazí na žádné komplikace v průběhu testu. Tester ověří, že všechny úlohy byly úspěšně dokončeny.
4	Změna nastavení	1. Spusť aplikaci Netfox Detective 2. Otevři nastavení aplikace 3. Uprav adresář pro ukládání logů 4. Ukonči aplikaci 5. Spusť znovu aplikaci Netfox Detective 6. Zkontroluj, že nastavení zůstalo upravené 7. Ukonči aplikaci	Tester nenarazí na žádné komplikace v průběhu testu. Testerovi se podaří změnit nastavení, a ukončit aplikaci

Tabulka B.2: Část úkonů prováděných v rámci „End-to-end“ testů