FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Refaktoring a metodika testování nástroje pro anonymizaci dat
Student:	Bc. Pavel Perner
Vedoucí:	Ing. Jiří Mlejnek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Analyzujte aktuální metodiku testování nástroje pro anonymizaci osobních údajů. Popište nevýhody současného řešení. Navrhněte její rozšíření či přepracování. Při návrhu nové metodiky se zaměřte na preciznost jednotkových testů, využití nástrojů nebo postupů zjednodušujících přípravu testovacího prostředí a testovacích dat, automatické regresní testy a automatické testovací scénáře, které ověří E2E funkčnost nástroje Winch.

Dále analyzujte výsledky bakalářské práce Implementace datové vrstvy pro anonymizační nástroj [1] a navrhněte postup, jak dokončit implementační část práce zabývající se refaktoringem nástroje Winch. Dle navrženého postupu implementaci dokončete.

Na výslednou implementaci aplikujte nově navrženou a implementovanou metodiku testování, zaměřte se především na ověření zpětné kompatibility s existujícími anonymizačními modely.

–

[1] Schuh, Matěj. Implementace datové vrstvy pro anonymizační nástroj. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 13. října 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Refaktoring a metodika testování nástroje pro anonymizaci dat

Bc. Pavel Perner

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Mlejnek

11. února 2022

Poděkování

Rád bych poděkoval vedoucímu diplomové práce Ing. Jiřímu Mlejnkovi za jeho rady, připomínky k řešení a vstřícný přístup při konzultacích. Mé další díky patří rodině, přátelům a partnerce, kteří mne během tvorby práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. února 2022 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2022 Pavel Perner. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Perner, Pavel. *Refaktoring a metodika testování nástroje pro anonymizaci dat.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Diplomová práce se věnuje přípravě a nasazení přepracovaného vývojového prostředí, návrhu rozšíření metodiky testování a implementaci nové datové vrstvy zajišťující multiplatformní využití anonymizačního nástroje Winch.

Analytická část práce se zabývá analýzou aplikace Winch a definicí požadavků na výsledné řešení. V teoretické části se pak nejdříve zkoumají různé varianty tvorby a provozu jednotných vývojových prostředí. Následně se v ní nahlíží na problematiku testování softwaru, jeho vztah s psychologií člověka a popis rozličných kategorií testů včetně krátkého přiblížení možností jejich automatizace. V poslední řadě pak seznamuje čtenáře s různými formáty pro persistenci dat desktopových aplikací.

Výsledek diplomové práce představuje nové a jednotné vývojové prostředí, jež usnadní budoucí rozšiřování nástroje Winch. Dále přináší návrh nové metodiky testování, která obsahuje pravidla a doporučení pro psaní testovacího kódu, včetně příkladu aplikace představené metodiky. Konečným výsledkem je poté přepracovaný datový model pro anonymizaci, jenž nyní podporuje více druhů úložišť, a s ohledem na zvolený formát přidané datové vrstvy lze nástroj Winch testovat a používat na libovolném operačním systému.

Klíčová slova Winch, refaktoring, softwarové testování, vývojové prostředí, persistence dat

Abstract

The master thesis focuses on the preparation and deployment of a redesigned development environment, the design of an extended testing methodology and the implementation of a new data layer ensuring the multiplatform use of the Winch anonymization tool.

The analytical part of the thesis introduces the analysis of the Winch application and the definition of requirements for the resulting solution. The theoretical part then first examines different variants of creating and operating unified development environments. It then looks at the issue of software testing, its relationship with human psychology, and a description of the different categories of tests, including a brief discussion about the possibilities of automating them. Finally, it introduces the reader to various formats for persisting data of a desktop application.

The result of the thesis is a new and unified development environment that will facilitate future extensions of Winch. Furthermore, it presents a proposal for a new testing methodology that includes rules and recommendations for writing test code, including an example application of the presented methodology. The end result is then a redesigned data model for anonymization that now supports multiple storage types, and given the chosen format of the added data layer, Winch can be tested and used on any operating system.

Keywords Winch, refactoring, software testing, development environment, data persistence

Obsah

Úvod **1**

1 Cíle práce **3**

2 Analýza nástroje Winch **5**
2.1 Anonymizační nástroj Winch 5
2.1.1 Winch Actor moduly 6
2.2 Stávající vývojové prostředí 6
2.3 Aktuální testovací metodika 7
2.3.1 Stávající jednotkové testy 7
2.3.2 Stávající integrační testy 8
2.3.3 Využití parametrizovaných testů 8
2.3.4 Ostatní druhy testů 9
2.3.5 Struktura testů 9
2.4 Struktura anonymizačního modelu 10
2.5 Závislosti nástroje 12

3 Požadavky na řešení **13**
3.1 Přepracované vývojové prostředí 13
3.1.1 Snadná přenositelnost (DEV-1) 13
3.1.2 Multiplatformita (DEV-2) 13
3.1.3 Pokrytí databázových závislostí (DEV-3) 14
3.2 Rozšířená testovací metodika 14
3.2.1 Precizní jednotkové testy (TEST-1) 14
3.2.2 Integrační testy (TEST-2) 15
3.2.3 Systémové testy (TEST-3) 15
3.2.4 Automatizace procesu testování (TEST-4) 15
3.3 Datová vrstva pro anonymizační model 15
3.3.1 Multiplatformita (DATA-1) 16

3.3.2 Otevřenost (DATA-2) 16
3.3.3 Verzovatelnost formátu (DATA-3) 16
3.3.4 Zpětná kompatibilita (DATA-4) 16
3.3.5 Konverze formátů (DATA-5) 16

4 Možnosti tvorby vývojového a testovacího prostředí 17
4.1 Virtuální stroje 17
4.1.1 VirtualBox 18
4.1.2 VMware Workstation Player 18
4.2 Softwarové kontejnery 18
4.2.1 Docker 19
4.2.2 Podman 20
4.3 Vagrant 20

5 Testování softwaru 23
5.1 Proces testování 23
5.2 Základní principy testování 24
5.3 Klasifikace testů 25
5.3.1 Znalost vnitřní struktury 25
5.3.2 Úroveň testu 25
5.4 Pokrytí testy 26
5.5 Jednotkové testy 26
5.5.1 Testové alternativy 26
5.5.2 Parametrizované testy 27
5.6 Integrační testy 27
5.7 Funkční a systémové testy 27
5.8 Regresní testování 27
5.9 Automatizace procesu testování 28
5.9.1 Nástroje pro automatické testování 29

6 Datová úložiště desktopových aplikací 31
6.1 XML 31
6.1.1 Validace struktury a obsahu XML 32
6.1.2 Zpracování XML 33
6.2 JSON 33
6.2.1 Validace textu ve formátu JSON 34
6.3 SQLite 34

7 Návrh a realizace 37
7.1 Návrh a příprava nového vývojového prostředí 37
7.1.1 Docker Compose pro databázové stroje 38
7.2 Návrh testovací metodiky 39
7.2.1 Nezávislost jednotkových a integračních testů 40
7.2.2 Refaktoring testů 40

7.2.3 Zaznamenávání průběhu testování 41
7.2.4 Zavedení systémových testů 41
7.3 Návrh a implementace nové datové vrstvy 41
7.3.1 Volba formátu datového úložiště 41
7.3.2 Nová architektura anonymizačního modelu 42
7.3.3 Přenos jednoho druhu úložiště na jiný 47
7.3.4 Proces objevování a persistence úkolů 48

8 Ověření řešení 51
8.1 Vývojové prostředí 51
8.2 Metodika testování 51
8.3 Datová vrstva nástroje Winch 53
8.3.1 Zpětná kompatibilita s nástrojem EA 53
8.3.2 Regresní ověření 55
8.4 Shrnutí ověřovací fáze 55

Závěr 57

Bibliografie 59

A Obsah přiloženého CD 61

Seznam obrázků

2.1 Aktuální struktura anonymizačního modelu 11

7.1 Návrh struktury anonymizačního modelu 43
7.2 Třídy pro čtení modelů z jejich datových úložišť 44
7.3 Třídy pro čtení a přenos informací z úložišť modelů 45
7.4 Třída ModelLoadingUtil 46
7.5 Klientský pohled na konverzi datového úložiště 47
7.6 Konvertory datových úložišť 48
7.7 Abstraktní třída TaskReaderWriter a její implementace 49
7.8 Třída TaskSingleton 50

Seznam tabulek

3.1 Požadavky na vývojové prostředí 13
3.2 Požadavky na testovací metodiku 14
3.3 Požadavky na novou datovou vrstvu 15

7.1 Vyhodnocení požadavků na vývojové prostředí 38
7.2 Vyhodnocení požadavků na formát nového úložiště 42
7.3 Mapování původních tříd anonymizačního modelu na nové 43

Výpisy kódu

2.1 Test konzolové aplikace Winch Actor 7
2.2 Ukázka vlastního parametrizovaného testu 8
2.3 Test využívající nativní aserce 9
2.4 Test využívající aserce z frameworku JUnit 9
2.5 Záznam události v testu na standardní výstup 10
7.1 Soubor `docker-compose.yml` 38
7.2 Testovací případ v třídě `NotNullDecoratorTest` 40
8.1 Parametrizovaný test s využitím JUnit 52
8.2 Skript ověřující zpětnou kompatibilitu 54

Úvod

Diplomová práce se věnuje analýze stávajícího vývojového prostředí, aktuální metodice testování a implementaci nové datové vrstvy v anonymizačním nástroji Winch. Zvláště stávající způsob ukládání dat a závislost na nástroji Enterprise Architect zapřičiňují nemožnost provozu a testování programu na na libovolném operačním systému, což omezuje lokální vývoj i automatické testování na serveru.

Výběr téma byl ovlivněn předchozí kladnou zkušeností s prací na aplikaci Winch a zájmem o téma anonymizace dat a refaktoringu kódu zatíženým zásahy programátorů s rozličnými zkušenostmi s vývojem.

První kapitola seznamuje čtenáře s jednotlivými cíli práce, jenž byly odvozeny primárně ze zadání a následných konzultací. V další části se diplomová práce zabývá analýzou anonymizačního nástroje Winch z pohledu definovaných cílů. Následuje kapitola představující konkrétní požadavky na dílčí cíle práce. Na ni navazují teoretické části diplomové práce. První zkoumá různé způsoby tvorby jednotných vývojových prostředí a možnosti jejich distribuce mezi vývojáři. Druhá přibližuje proces testování softwaru a čtenáři představuje jeho definici, základní principy testování nebo například kategorizaci testů dle jejich účelu. Kapitola také zevrubně popisuje proces automatizace testování. Poslední kapitola teoretické části práce se zabývá vybranými datovými úložišti vhodnými pro použití v desktopových aplikacích. Další kapitola přináší návrh a realizaci jednotlivých cílů – návrh a nasazení nového vývojového prostředí, rozšíření metodiky testování nástroje Winch a implementaci nové datové vrstvy nezávislé na konkrétní platformě. Poslední kapitola stručně ověřuje představené návrhy a realizace popsané v předchozí části práce.

Kapitola 1

Cíle práce

Cílem diplomové práce je analýza aktuální metodiky testování anonymizačního nástroje Winch, návrh jejího rozšíření či přepracování a zapracování samotné metodiky podle vytvořeného návrhu. Nová metodika bude zaměřena na precizní jednotkové testy a automaticky spouštěné integrační, systémové a regresní testy. Stávající metodika testování nástroje Winch obsahuje užití jednotkových a integračních testů, které ale vznikají bez jednotného postupu.

Dílčím cílem návrhu nové metodiky testování je seznámení se s nástroji a postupy umožňující jednoduchou přípravu vývojového prostředí včetně testovacích dat. Na základě požadavků na nové testovací prostředí bude vybrána a použita vhodná technologie nebo metoda. Nově vytvořené testovací prostředí poslouží také jako snadno přenositelné a konzistentní vývojové prostředí, které je aktuálně v nástroji Winch realizováno jako obraz virtuálního stroje, jenž si musí jednotliví vývojáři složitě předávat.

Dalším cílem práce je analýza výsledků bakalářské práce Implementace datové vrstvy pro anonymizační nástroj [1], která se zabývá refaktoringem nástroje Winch. Úprava aplikace spočívala v odstranění závislosti datové vrstvy nástroje Winch na aplikaci Enterprise Architect, jež je dostupná pouze pro operační systém Microsoft Windows. O výsledky analýzy se bude opírat návrh dokončení implementační části práce a podle navrženého postupu bude samotná implementace realizována.

Dílčím cílem implementace je její ověření pomocí nově vytvořené metodiky testování. Nedílnou součástí ověření je kontrola zpětné kompatibility s původní implementací nástroje Winch, protože odstranění závislosti na nástroji Enterprise Architect nesmí ovlivnit stávající funkčnost nástroje u zákazníků.

Výsledkem práce bude upravený anonymizační nástroj Winch, který již nebude závislý na nástroji Enterprise Architect. Nástroj Winch bude disponovat přepracovanou testovací metodikou včetně nového vývojového a testovacího prostředí.

Kapitola 2

Analýza nástroje Winch

Kapitola čtenáři představuje anonymizační nástroj Winch. První podkapitola se věnuje popisu činnosti aplikace, následující se pak zabývají jejím aktuálním vývojovým a testovacím prostředím, strukturou datové vrstvy a závislostmi.

2.1 Anonymizační nástroj Winch

Aplikace Winch umožňuje generování anonymizovaných dat z podporovaných relačních databází, anonymizaci souborů s prostým textem i binárním obsahem a zjišťování možných výskytů citlivých informací v relačních databázích.

Samotný nástroj se skládá z konzolové aplikace *Winch Actor* vykonávající zmiňované a další procesy, rozšíření (*Add-In*) do aplikace Enterprise Architect, skrze který se spouští Winch Actor s různými příkazy a parametry dle vybraných elementů v Enterprise Architect, a volitelnými rozšířeními pro Winch Actor, jež se používají k přidávání dalších funkcionalit do základní aplikace Winch Actor.

Nástroj Winch disponuje různými režimy, ve kterých může generování anonymizovaných dat fungovat. Aplikaci tak lze využít pro různé scénáře – například vytváření validních testovacích dat z produkčních, ukládání historických dat do anonymizovaných archivů nebo anonymizovat data přímo v tabulkách, v nichž jsou uložena. Funkce objevování osobních údajů upozorní administrátora na možnost přítomnosti citlivého údaje (včetně pravděpodobnosti) v databázovém sloupci pomocí úkolů vytvořených v Enterprise Architect. Na základě vzniklých úloh může následně přiřadit adekvátní anonymizační funkce a vygenerovat odpovídající anonymizovaná data.

Konzolová aplikace Winch Actor je vyvíjena v programovacím jazyce Groovy a sestavována nástrojem Gradle, Winch *Add-In* pro nástroj Enterprise Architect je napsán v jazyce C# a volitelná rozšíření pro Winch Actor se opět píší v jazyce Groovy.

2.1.1 Winch Actor moduly

Anonymizační nástroj Winch je rozdělen do následujících Gradle modulů:

- disl-winch-connector,
- disl-winch-db2,
- disl-winch-filesystem,
- disl-winch-generator,
- disl-winch-license,
- disl-winch-mssql,
- disl-winch-oracle,
- disl-winch-postgresql.

disl-winch-connector představuje společný modul, jenž je sdílen s ostatními moduly nástroje Winch. Obsahuje základní datový model pro anonymizaci, výčet jednotlivých operací, jež lze spouštět, anonymizační třídy, tabulkové vzory, objevitele osobních údajů a další společné části aplikace Winch. Zároveň se stará o načítání databázového modelu z nástroje Enterprise Architect a jeho konverzi do společného datového modelu pro anonymizaci. Mimo společného kódu určuje modul i závislosti na knihovnách třetích stran a jejich verze.

Moduly **disl-winch-db2**, **disl-winch-mssql**, **disl-winch-oracle** a **disl-winch-postgresql** nesou každý specifický kód pro konkrétní databázový stroj. Z pohledu kódu se typicky jedná o implementace různých rozhraní a abstraktních tříd ze sdíleného modulu disl-winch-connector, obsahují ale také konkrétní SQL skripty a šablony pro generování kódu pro danou databázi. Veškeré činnosti vykonávají nad společným datovým modelem pro anonymizaci.

Modul **disl-winch-filesystem** se stará o objevování osobních údajů a anonymizaci různých druhů souborů na souborovém systému.

disl-winch-generator slouží jako generátor iniciálních dat na základě zvoleného databázové stroje a anonymizačního modelu.

Modul **disl-winch-license** generuje a validuje licenční soubory, jež jsou distribuovány majitelům licence nástroje Winch.

2.2 Stávající vývojové prostředí

Zdrojový kód nástroje Winch je spravován distribuovaným systémem správy verzí Git. Spolu s kódem jsou verzovány i některé závislosti – knihovny pro práci s nástrojem Enterprise Architect, komunikaci s Microsoft Windows DLL a COM knihovnami a ovladače pro podporované databázové stroje.

Zdroje dostupné ve verzovacím systému jsou dostatečné pro vývoj a spouštění testů v modulu disl-winch-connector za předpokladu, že vývojář disponuje operačním systémem Microsoft Windows a v něm instalací nástroje Enterprise Architect, jenž je ke spuštění testů vyžadován.

K exekuci testů v databázových modulech aplikace Winch Actor jsou vyžadovány běžící instance odpovídajících databázových strojů. Pro vývojáře pracujícím na rozvoji nástroje Winch jsou k dispozici instalace podporovaných databázových strojů pro systém Microsoft Windows. Instalace jsou mezi jednotlivými vývojáři distribuovány fyzicky na přenosném paměťovém úložišti.

Hlavní nevýhodou aktuálního vývojového prostředí je závislost na operačním systému Microsoft Windows a nástroji Enterprise Architect. V obou případech se jedná o proprietární software, k němuž je nutné vlastnit licenci, jež si musí každý vývojář zajistit.

Další problém představuje distribuce instalací databázových strojů, které jsou vyžadovány při integračních testech databázových modulů nástroje Winch Actor. Vývojář si typicky vyžádá fyzické předání instalací databázových strojů (včetně inicializačních skriptů) od jiného programátora podílejícího se na rozvoji aplikace Winch, nebo si může databáze zprovoznit a inicializovat sám, čímž ale stoupá riziko vytvoření nekonzistencí napříč vývojovými prostředími různých vývojářů a zanesení chyby do programu.

2.3 Aktuální testovací metodika

V aplikace Winch Actor jsou aktuálně využity dva druhy testů – jednotkové a integrační.

2.3.1 Stávající jednotkové testy

Jednotkové testování slouží v nástroji k ověření jednoduchých funkcionalit na nejnižší úrovni implementace. Specificky se jedná například o kontrolu textových výstupů tříd a metod, které generují SQL skripty pro konkrétní databázový stroj.

Nástroj Winch obsahuje testy, jejichž význam není na první pohled zcela zjevný, například test uvedený v listingu 2.1.

```
@Test
 public  void testHelpExecutionMode() {
  WinchLauncher.main('-h')
}
```

Listing 2.1: Test konzolové aplikace Winch Actor

Uvedený test neobsahuje žádné aserce a jediným jeho účelem tak je pouze ověření, že volání statické metody `main` třídy `WinchLauncher` neskončí vyho-

zením výjimky. Třída `WinchLauncher` představuje v nástroji Winch nejvyšší úroveň abstrakce a vstupní bod konzolové aplikace – jedná se tedy spíše o nástin E2E testu, v rámci něhož dochází ke spuštění celé aplikace Winch Actor se vstupním parametrem `-h`.

2.3.2 Stávající integrační testy

Integrační testy jsou v aplikaci využity ke kontrole kompatibility s dalšími systémy, na kterých je aplikace Winch Actor závislý. Jmenovitě se jedná o testování načítání databázového modelu z nástroje Enterprise Architect a spouštění SQL skriptů nad běžícími instancemi databázových strojů.

Protože se samotné testy nestarají o správnou inicializaci či čištění testovacích dat, je třeba si databáze nejen instalovat, ale i inicializovat a případně kontrolovat jejich stav po provedení všech testů. Například testování anonymizačních funkcí postrádá přípravu a exekuci potřebného SQL kódu, což vede k situacím, kdy první běh celé testovací sady databázového modulu skončí chybami o chybějících funkcích nebo procedurách v dané relační databázi, ale v dalším běhu již tyto selhávající testy uspějí, neboť v rámci prvního spuštění testovací sady proběhla inicializace databáze z pohledu přípravy kódu pro následnou anonymizaci. Pokud si navíc databázové stroje na lokálním prostředí persistují svá data dlouhodobě, k situaci při prvním běhu testovací sady již nedojde, ale pokud by došlo k jejich vyčištění, situace by se znovu opakovala.

2.3.3 Využití parametrizovaných testů

Všechny integrační testy ověřující funkčnost anonymizačních tříd dědí z abstraktního předka `AbstractFunctionTest` a při svém běhu využívají parametrizované testy. Ukázka se nachází na listingu 2.2.

```
abstract  class AbstractFunctionIcoUniqueTest
   extends AbstractFunctionTest {

 @Test
  public  void fc_icoUniqueTest() {
   List caseList =
     //positive cases
     [[input: "12341231", output: "46132571"],
     [input: "25596641", output: "36197041"],
     [input: "46132571", output: "59821639"],
     [input: "36197041", output: "67625363"],
     //negative cases - invalid inputs
     [input: "aaa123451234512345", output:
        "aaa123451234512345"],
     [input: "NULL", output:  null]]
```

```
    runTest(caseList, false);
  }
}
```

Listing 2.2: Ukázka vlastního parametrizovaného testu

Metoda `runTest` z abstraktního předka `AbstractFunctionTest` pak prochází předaný seznam vstupů a očekávaných výstupů. V rámci tohoto průchodu spouští odpovídající SQL skripty nad testovanou relační databází.

2.3.4 Ostatní druhy testů

Automatické spouštění všech testovacích sad v jednotlivých modulech na serveru lze do jisté míry klasifikovat jako regresní testování. Dále ale nástroj Winch nedisponuje žádnými dalšími automaticky spouštenými druhy testů.

Vlastní skupinu tvoří manuální testování (prováděné typicky za účelem ověření konkrétních požadavků od klientů), jenž by se často dalo specifikovat jako E2E testy, neboť se obvykle provádějí nad finálně distribuovaným balíčkem aplikace spolu s Winch-Addinem v nástroji Enterprise Architect.

2.3.5 Struktura testů

V aplikaci Winch není dodržena jednotná struktura psaní testovacího kódu. Jedním z příkladů je rozdílný způsob asercí v testovacích metodách. V ukázkovém testu obsaženém v listingu 2.3 jsou k aserci výsledných hodnot užity nativní aserce programovacího jazyka Groovy. Listing 2.4 naopak přibližuje test, v němž se užívají aserce definované ve frameworku JUnit.

```
@Test
 public  void testConnection(){
  String realOutput = sqlInstance.firstRow("select * from
      SYSIBM.SYSDUMMY1")[0]
  assert realOutput=='Y';
}
```

Listing 2.3: Test využívající nativní aserce

```
@Test
 void testJsonOutputWithDefaultValue() {
  AnonymizationFunctionsProvider provider =
      new AnonymizationFunctionsProvider()
  String json = provider.getJson()

  assertEquals(JSON_OUTPUT, json)
```

```
}
```

Listing 2.4: Test využívající aserce z frameworku JUnit

Další příklad představuje záznam průběhu testů. V produkčním kódu nástroje Winch Actor se události zaznamenávají skrze dostupnou logovací knihovnu. Naproti tomu se v testovací části aplikace Winch využívá obyčejný zápis informací na standardní výstup. Ukázku záznamu události v testu zachycuje listing 2.5.

```
@Test
 void testOutputDecoratorEvaluate(){
  ScriptHelper helper =  new ScriptHelper()
  WColumn column =  new WColumnMock()
  column.name = 'test'
  String outputSQL = helper.outputDecoratorEvaluate(column,
    "ReplaceDecorator('/','')","'abc'")
  println(outputSQL)
  assert "REPLACE('abc','/','')" == outputSQL
}
```

Listing 2.5: Záznam události v testu na standardní výstup

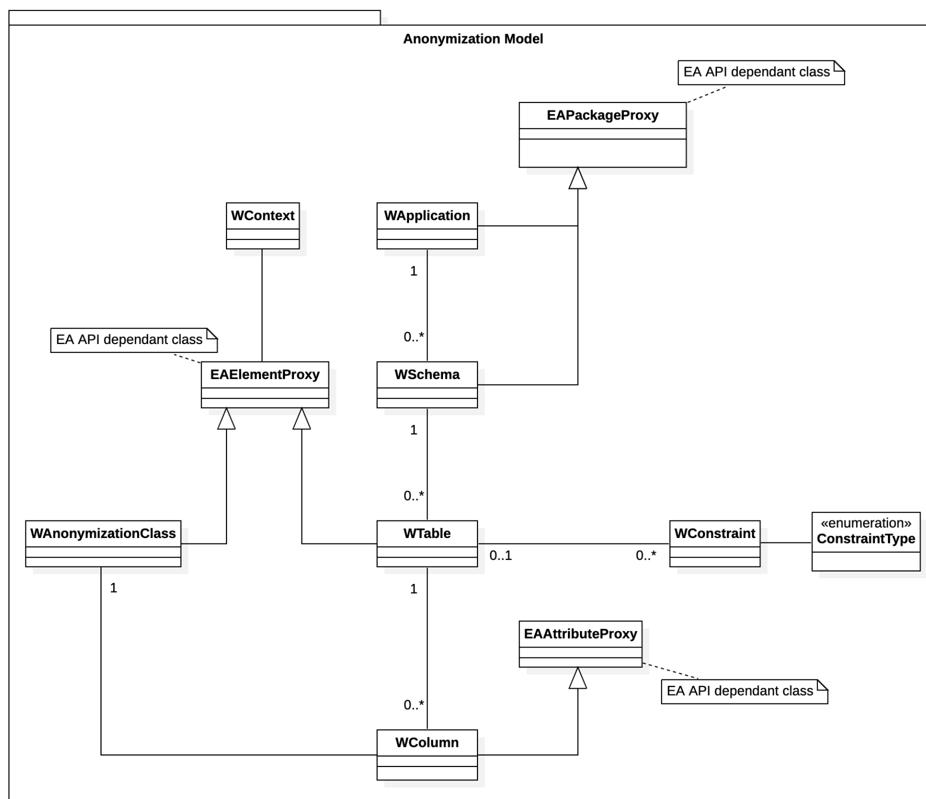
2.4 Struktura anonymizačního modelu

Aktuální struktura anonymizačního modelu je vyobrazena na diagramu tříd 2.1. Třídy anonymizačního modelu a jejich asociace (včetně poznámky o závislosti na externí nástroji) v aplikaci Winch Actor přímo reflektují strukturu anonymizačního modelu v nástroji Enterprise Architect.

Třída `WApplication` odpovídá stereotypu *WApplication* v EA, v němž představuje nejvyšší úroveň anonymizačního modelu. Sdružuje pod sebou schémata anonymizačního modelu a elementy označené stereotypem *WConfiguration*, jež slouží ke konfiguraci běhu aplikace Winch Actor. Konfigurační elementy v sobě schraňují kontexty (element se stereotypem *WContext*) a anonymizační funkce (element nesoucí stereotyp *WAnonymizationClass*).

Element se stereotypem *WContext* je v aplikaci Winch Actor reprezentován třídou `WContext`. Umožňuje přístup ke konfiguracím běhového prostředí (například adresa a port databázové instance). V nástroji Enterprise Architect lze pod balíčkem *WConfiguration* vytvořit libovolné množství kontextů. GUID (*Globally Unique IDentifier*, v překladu globálně unikátní identifikátor) elementu *WContext* je jedním ze vstupních parametrů aplikace Winch Actor.

Třída `WAnonymizationClass` reflektuje element se stereotypem *WAnonymizationClass*.



Obrázek 2.1: Aktuální struktura anonymizačního modelu

Třída `WSchema` představuje EA balíček označený stereotypem *WSchema*. Obě entity v logickém významu odpovídají schématům v databázích. Každá instance `WSchema` by si měla udržovat referenci na instanci třídy `WApplication`, do které patří.

Třída `WTable` odpovídá EA elementu se stereotypem *WTable*. Třída i EA element reprezentují databázovou tabulku. Podobně jako ve vztahu mezi `WApplication` a `WSchema` si i instance `WTable` uchovává odkaz na instanci `WSchema`.

`WColumn` je reflexí EA atributu nesoucí atribut *WColumn*. Logicky představují sloupce databázových tabulek. Instance třídy `WColumn` jsou schraňovány v instancích třídy `WTable`. Každá instance `WColumn` si uchovává referenci na instanci třídy `WTable`, jež je sdružuje.

Třída `WConstraint` představuje EA element označený stereotypem *WConstraint*. Třída i stereotyp odpovídají integritním omezením v relačních databázích. Pokud instance `WConstraint` vyjadřuje cizí klíč (typ integritního omezení udává výčet `ConstraintType`), uchovává si daný objekt referenci na cílovou

tabulku (instanci třídy `WTable`).

Součástí diagramu jsou také třídy `EAPackageProxy`, `EAElementProxy` a `EAAttributeProxy`. Jejich prefix `EA` indikuje, že jsou závislé na knihovně pro přístup k modelu v nástroji Enterprise Architect. Sufix `Proxy` značí, že třídy implementují návrhový vzor *proxy*. Subjektem je vždy odpovídající EA entita (popořadě se tedy jedná o proxy pro balíček, element a atribut).

Proxy třídy pro prvky nástroje Enterprise Architect představují předky pro některé třídy anonymizačního modelu, vždy podle EA prvku, jenž třída modelu reflektuje. Stávající anonymizační model v aplikaci Winch Actor je tak přímo závislý na knihovně pro přístup k Enterprise Architect. [1]

2.5 Závislosti nástroje

Z diagramu tříd a jeho popisu v podkapitole 2.4 vyplývá zásadní závislost aplikace Winch Actor na nástroji Enterprise Architect. Pro přístup k modelům v EA je spolu s kódem verzovaná i knihovna pojmenovaná *eaapi.jar*, jež umožňuje načítání a upravování EA modelu a všech jeho entit.

Produkční kód aplikace Winch Actor pracuje vždy s běžící instancí nástroje Enterprise Architect. Jelikož je aplikace EA dostupná pouze pro operační systém Microsoft Windows, musí Winch Actor umět přistupovat k běžícím procesům tohoto operačního systému. K tomu nástroj Winch využívá knihovnu *jacob*, která slouží k přístupu k DLL a COM knihovnám. Skrze ni aplikace Winch Actor nalezne běžící instanci aplikace Enterprise Architect a napojí se na ni.

Kapitola 3

Požadavky na řešení

V následující kapitole bude čtenář seznámen s požadavky na řešení, jež převážně vychází ze zadání práce a analýzy nástroje Winch.

3.1 Přepracované vývojové prostředí

Analýza anonymizačního nástroje Winch ukázala, že stávající způsob lokálního vývoje a testování je značně nevyhovující pro nově příchozí vývojáře. Vývojovému prostředí chybí přenositelnost a možnost úprav, jež by bylo možné snadno distribuovat mezi ostatní vývojáře. Tabulka 3.1 obsahuje nadefinované požadavky na přepracované vývojové prostředí spolu s jejich identifikátorem, jenž bude ve zbytku práce sloužit jako odkaz na daný požadavek.

Požadavek	Identifikátor
Snadná přenositelnost	DEV-1
Multiplatformita	DEV-2
Pokrytí databázových závislostí	DEV-3

Tabulka 3.1: Požadavky na vývojové prostředí

3.1.1 Snadná přenositelnost (DEV-1)

Nové vývojové prostředí by mělo disponovat snadnou přenositelností mezi jednotlivými vývojáři nástroje Winch bez nutnosti fyzického kontaktu nebo stahování vývojových a testovacích nástrojů z neověřených úložišť.

3.1.2 Multiplatformita (DEV-2)

Implementace nové datové vrstvy pro anonymizační modely nástroje Winch umožní jeho provoz na libovolném operačním systému, proto by ani přepra-

cované vývojové prostředí nemělo záviset na konkrétní platformě.

3.1.3 Pokrytí databázových závislostí (DEV-3)

Přepracované prostředí pro lokální vývoj by mělo obsahovat veškeré databázové instance (včetně konfigurace), pro něž aktuálně existuje v anonymizačním nástroji Winch podpora. Jmenovitě se jedná o relační databáze:

- PostgreSQL ve verzi 9.6.2,
- Oracle Database 12c (verze 12.1.0.2.0),
- Microsoft SQL Server 2016 a 2019,
- IBM Db2 Database v11.1.2020.1393.

V rámci požadavku musí nové vývojové prostředí disponovat databázovými instancemi, jejichž hlavní verze bude shodná s hlavní verzí podporovaných relačních databází.

3.2 Rozšířená testovací metodika

Přímo ze zadání práce vyplývají požadavky na rozšíření stávající metodiky testování nástroje Winch, jež byly dále rozšířeny o poznatky z analytické fáze. Jednotlivé požadavky shrnuje tabulka 3.2.

Požadavek	Identifikátor
Precizní jednotkové testy	TEST-1
Integrační testy	TEST-2
Systémové testy	TEST-3
Automatizace procesu testování	TEST-4

Tabulka 3.2: Požadavky na testovací metodiku

3.2.1 Precizní jednotkové testy (TEST-1)

Ačkoliv anonymizační nástroj Winch obsahuje velké množství jednotkových testů, ne všechny přinášejí očekávanou přidanou hodnotu testování, což je způsobeno především absencí pravidel pro jejich vytváření. Nová testovací metodika by měla dbát na preciznější tvorbu jednotkových testů, jenž se budou řídit nastavenými pravidly.

3.2.2 Integrační testy (TEST-2)

Podobně jako v případě jednotkových testů, ani integrační testy napsané v aplikaci Winch Actor se neřídí žádnou konkrétní specifikací a jejich opakované spouštění končící selháním vede k zatěžování stroje, na němž jsou tyto integrační testy spouštěny. V rámci návrhu a nasazení testovací metodiky budou stejně jako pro jednotkové testy sepsána pravidla a postupy jednotné tvorby integračních testů, jenž se více zaměří zpracování jejich selhání.

3.2.3 Systémové testy (TEST-3)

Nová testovací metodika přinese základní návrh tvorby systémových testů, jenž se zaměří na testování anonymizačního nástroje Winch jako celku. Cíl představuje zavedení scénářů, které budou moci být spuštěny k ověřování E2E fungování aplikace.

3.2.4 Automatizace procesu testování (TEST-4)

Především s požadavkem TEST-3 souvisí zavedení automatického procesu systémového a E2E testování nástroje Winch. V rámci požadavku se navíc zreviduje automatické spouštění existujících jednotkových a integračních testů. Na základě revize se pak v případě potřeby navrhne jiný způsob jejich automatizace.

3.3 Datová vrstva pro anonymizační model

Součást zadání práce tvoří také navržení a implementaci nové datové vrstvy pro anonymizační nástroj Winch. Aktuální datový model aplikace úzce závisí na nástroji Enterprise Architect, což jej činí značně neflexibilním především při testování nebo provozu, neboť nástroj EA přináší do aplikace závislost na operačním systému Microsoft Windows. Souhrn požadavků na novou datovou vrstvu pro anonymizační modely nástroje Winch, jenž adresují zmíněné problémy, se nachází v tabulce 3.3.

Požadavek	Identifikátor
Multiplatformita	DATA-1
Otevřenost	DATA-2
Verzovatelnost formátu	DATA-3
Zpětná kompatibilita	DATA-4
Konverze formátů	DATA-5

Tabulka 3.3: Požadavky na novou datovou vrstvu

3.3.1 Multiplatformita (DATA-1)

Nová datová vrstva by měla být postavena na formátu, jenž do nástroje Winch nebude přinášet závislost na žádné konkrétní platformě nebo softwarovém nástroji, načež bude možné provozovat anonymizační nástroj Winch na libovolném operačním systému v celém jeho rozsahu.

3.3.2 Otevřenost (DATA-2)

Formát zvolený pro refaktorovanou datovou vrstvu by neměl být zatížen žádnou komerční licencí, což zajistí jeho volné použití na všech platformách bez nutnosti zajišťování jakéhokoliv souhlasu.

3.3.3 Verzovatelnost formátu (DATA-3)

Anonymizační modely založené na nové datové vrstvě by mělo být možné evidovat v systému správy verzí s možností náhledu na historicky provedené úpravy.

3.3.4 Zpětná kompatibilita (DATA-4)

Nad anonymizačními modely postavenými na nové datové vrstvě musí být umožněno spouštět veškeré existující procesy nástroje Winch. Taktéž je nutné zajištění zpětné kompatibility s původními anonymizační modely založenými na provázání s nástrojem Enterprise Architect.

3.3.5 Konverze formátů (DATA-5)

Anonymizační nástroj Winch by měl být rozšířen o novou funkcionalitu umožňující převod jednoho formátu na druhý. Konverze musí zachovávat veškeré informace nutné pro proces anonymizace a objevování citlivých údajů, naopak není žádoucí uchovávat informace nesouvisející s anonymizací (například grafické rozložení elementů v EA).

Kapitola **4**

Možnosti tvorby vývojového a testovacího prostředí

Následující kapitola pojednává o možnostech tvorby a distribuce vývojových a testovacích prostředí. Čtenáře seznamuje s moderními způsoby vývoje softwaru a automatizaci při vytváření lokálního prostředí.

4.1 Virtuální stroje

Virtuální stroj je software, jenž simuluje klasický fyzický stroj. Na hostujícím počítači nebo serveru lze provozovat několik virtuálních strojů. Každý virtuální stroj disponuje vlastním operačním systémem a vlastními virtuálními výpočetními zdroji a pracuje zcela izolovaně jak od hostujícího stroje tak ostatních souběžně běžících virtuálních strojů.

Historicky se virtualizace fyzických strojů využívala především na serverech k efektivnějšímu rozdělení výpočetních zdrojů, izolovanému běhu potenciálně nebezpečných úkonů, jenž by bez virtualizace mohly ohrozit celý fyzický stroj, nebo například k možnosti běhu různých operačních systémů na jednom stroji. Dnes nachází virtualiazce využití například i při testování aplikací v bezpečném a předem přesně nastaveném prostředí. [2]

Virtuální stroje jsou spravovány tzv. hypervizorem, jenž se stará o virtuální sdílení výpočetních zdrojů hostujícího stroje, a představuje tak abstraktní vrstvu mezi softwarem a hardwarem počítače nebo serveru. Hypervizory se dělí na typ 1 a typ 2. Hypervizor typu 1 se v angličtině označuje jako *bare metal* a chová se jako samostatný odlehčený operační systém, který běží přímo na hostitelském hardwaru. Typ 2 (známý jako *hosted*) vyžaduje ke svému běhu existující operační systém na hostitelském stroji, na němž následně běží jako další softwarová vrstva. [3]

4.1.1 VirtualBox

VirtualBox je multiplatformní aplikace pro virtualizaci, jenž zastává roli hypervizora druhého typu. Nástroj VirtualBox se na většině podporovaných operačních systémech chová velmi podobně a užívá na nich stejné soubory a formáty obrazů s virtualizovanými operačními systémy, což ve výsledku znamená, že virtuální stroj vytvořený pomocí aplikace VirtualBox na jednom operačním systému bude fungovat i na dalším podporovaném systému. VirtualBox navíc podporuje i *Open Virtualization Format* – standardní formát souborů pro virtualizovaný software.

Nástroj VirtualBox se dělí na základní balíček a dodatečná rozšíření. Základní balíček obsahuje veškeré open source komponenty a sám je také veden jako svobodný software. Dodatečná rozšíření slouží k rozšiřování základního balíčku o další funkcionality. Typicky se jedná přidávání dalších integrací mezi virtualizovaným strojem a hostujícím operačním systémem (například podpora čtení a zápisu pomocí USB verze 2 a 3 nebo přístup k webové kameře).

VirtualBox umožňuje vytváření snapshotů, jenž zachycují celkový stav virtuálního stroje v daném časovém okamžiku. Do takto zachycených stavů se pak lze libovolně vracet bez ohledu na změny, které byly ve virtuálním stroji provedeny. [4]

4.1.2 VMware Workstation Player

VMWare Workstation Player je podobně jako Virtual Box hypervizor typu 2. Nástroj je dostupný pro operační systémy Microsoft Windows a Linux zdarma, pokud nebude využit ke komerčním účelům. Podporuje import a export virtuálních strojů pomocí *Open Virtualization Format* nebo vlastního formátu souborů.

Společnost VMWare nabízí také aplikaci Workstation Pro, pro níž je nutno zakoupit licenci. Narozdíl od Workastation Player umožňuje vytváření snapshotů, klonování virtuálních strojů nebo jejich šifrování. Nástroje Workstation Player i Workstation Pro používají stejný hypervizor.

Pro použití na operačním systému macOS nabízí firma VMWare placený produkt Fusion. [5]

4.2 Softwarové kontejnery

Softwarový kontejner představuje abstrakci na aplikační úrovni, jež může obsahovat aplikace či jiný spustitelný kód včetně jeho závislostí. Kontejnery lze v libovolném množství spouštět na cílovém stroji, kde navzájem sdílí systémové prostředky hostujícího počítače nebo serveru. [6]

4.2.1 Docker

Docker je platforma pro vývoj, distribuci a běh aplikací využívající softwarové kontejnery vyvíjena společností Docker Inc. Docker kontejnery lze využít ke standardizaci lokálního vývojového prostředí nebo pro kontinuální integrace a dodávání. Díky své přenositelnosti lze Docker kontejnery provozovat na lokálních počítačích, serverech nebo například virtualizovaných strojích s libovolným operačním systémem.

Docker využívá modelu klient-server. Klient komunikuje s Docker démonem, jenž se stará o správu kontejnerů (jejich vytváření, běh nebo distribuci) a dalších Docker objektů. Komunikace mezi klientem a démonem může probíhat pomocí REST API, UNIXových soketů nebo obecného síťové rozhraní. Umožněna je také komunikace mezi několika Docker démony. Primárního Docker klienta představuje CLI utilita `docker`, jež využívá Docker API ke komunikaci s démonem. Docker klient může komunikovat s více démony.

Jak bylo v předchozím odstavci zmíněno, Docker démon spravuje kontejnery a další Docker objekty. Mezi tyto objekty se řadí například Docker obrazy (anglicky *images*).

Obraz představuje šablonu obsahující instrukce pro vytvoření kontejneru. Mezi obrazy může existovat dědičnost – jeden Docker obraz může být postaven na jiném již existujícím, přičemž jej smí modifikovat nebo rozšiřovat. Docker obraz lze vytvořit pomocí souboru zvaného *Dockerfile*. Každá instrukce v tomto souboru představuje jednu vrstvu výsledného Docker obrazu. Při jeho změně (za účelem vytvoření upraveného obrazu) se při sestavování výsledného obrazu zpracují pouze změněné vrstvy, což činí proces sestavení obrazu efektivnějším. Ze sestaveného Docker obrazu lze vytvářet spustitelné instance – kontejnery.

Docker klient předává démonu instrukce týkající se kontejnerů (jejich vytváření, spouštění, zastavování, mazání a podobně). Ve výchozím nastavení je kontejner značně izolován od ostatních kontejnerů a hostujícího stroje, na němž kontejner běží. Za pomoci klienta lze upravovat míru izolovanosti kotejneru (například jeho síťovou propustnost nebo přístup k souborovému systému). Docker kontejner je primárně konfigurován skrze Dockerfile, ale jeho nastavení lze upravovat i při startu. Všechny v kontejneru provedené změny jsou po jeho odstranění smazány (pokud nebyly persistovány mimo logický svazek kontejneru samotného).

Docker ve svém nitru využívá linuxovou technologii jmenných prostorů (anglicky *namespaces*). Na linuxových distrubích tak nástroj Docker funguje nativně, zatímco na jiných operačních systémech je nutné emulovat linuxové prostředí. Pro systém Microsoft Windows Docker využívá vrstvu WSL 2 nebo *Windows containers*, pro operační systém macOS poté jeho nativní virtualizace pro vytvoření linuxového prostředí. [7, 8, 9]

Jednoho z možných Docker klientů představuje i *Docker Compose*. Jedná se o orchestrační nástroj, jenž umožňuje definici a běh několika různých Docker

kontejnerů. Konfigurace pro Docker Compose se uvádí v souboru pojmenovaném *docker-compose.yml* ve formátu YAML. Následně lze všechny takto definované kontejnery spustit jediným příkazem, stejně tak je lze následně všechny ukončit. Docker Compose se typicky využívá k tvorbě jednotných vývojových nebo testovacích prostředí. [10]

4.2.2 Podman

Podman je open source nativní linuxový nástroj určený ke správě a běhu *Open Containers Initiative* (dále jen OCI) kontejnerů a kontejnerových obrazů. Narozdíl od platformy Docker nepracuje v režimu klient-server a zaměřuje se na možnost provozu kontejnerů pod uživatelským účtem bez administrátorských oprávnění (tzv. *rootless* kontejnery). Podman využívá model *fork-exec*, v rámci něhož lze správně monitorovat aktivity spojené se správou kontejnerů. Aby mohl v tomto režimu správně fungovat, spoléhá se nástroj Podman na kontejnerová běhová prostředí splňující OCI specifikaci, jež využívá ke komunikaci s linuxovým operačním systémem a spouštění kontejnerů. [11, 12]

Podman disponuje funkcionalitou, jež umožňuje sdružovat několik různých kontejnerů, zvanou pody. Účelem podu není pouhé logické seskupování spolu souvisejících kontejnerů, ale také například sdílení stejných jmenných prostorů mezi všemi kontejnery v podu, což následně umožňuje jejich snadné síťové propojení bez nutnosti vystavování síťových portů vně kontejner. Kontejnery seskupené do podů lze uvést do provozu jedním příkazem pro spuštění daného podu, což je činí vhodným orchestračním nástrojem pro správu různých kontejnerů. [13]

Utilitu Podman lze nativně provozovat pouze na linuxových distribucích. Ke zprovonění nástroje Podman na operačních systémech Microsoft Windows a macOS je třeba zřídit připojení ke stroji s operačním systémem Linux (i virtuálnímu), na němž je nutno nakonfigurovat a spustit SSH server. Podman klient se následně přes SSH připojuje k tomuto serveru, kde se k Podman službě připojí pomocí `systemd` soketů. V tomto nastavení tedy Podman funguje podle modelu klient-server. [14]

4.3 Vagrant

Vagrant je nástroj pro tvorbu kompletního vývojového prostředí, jenž je zasazeno do virtuálního stroje běžícím na hostujícím počítači. Cílem utility je celkové sjednocení vývojového prostředí napříč vývojáři a snížení času nutného na zprovoznění všech závislostí nutných pro vývoj.

Nástroj Vagrant za uživatele řeší většinu činností spojených s vytvořením vývojového prostředí – vytvoření virtuálního stroje včetně jeho konfigurace, síťové nastavení nebo například zřízení sdílených složek. Správa vývojového prostředí skrze Vagrant probíhá pouze pomocí několika příkazů.

Starší verze nástroje Vagrant umožňovaly spolupráci pouze s virtuálním strojem VirtualBox. Aktuální verze lze ale provozovat i nad dalšími (například VMWare nebo Parallels). Virtuální stroje jsou navíc nyní pouze jednou z možností, nad nimiž může Vagrant pracovat – zvolit lze prostou virtualizaci hostujícího počítače (pokud jí disponuje), kontejnerové řešení nebo vzdálené napojení do cloudu.

Vagrant lze konfigurovat pro různé projekty zvlášť. Na přítomnost nástroje Vagrant v projektu upozorňuje soubor *Vagrantfile* (vždy jeden na projekt). Vagrantfile představuje obyčejný textový soubor, jenž nese veškeré informace nutné k vytvoření vývojového prostředí (od operačního systému, který by měl na virtuálním stroji běžet, přes fyzické nastavení virtuálního stroje, například velikost paměti, až po software, jenž by měl na výsledném operačním systému nainstalován, nakonfigurován a spuštěn). Vagrantfile se typicky verzuje spolu s projektem, aby byl dostupný všem vývojářům. Jediným příkazem pak Vagrant přečte konfiguraci ve Vagrantfile a připraví celé vývojové prostředí.

Nástroj Vagrant lze rozšiřovat pomocí plug-inů. Protože je navíc Vagrant open source, mohou být vytvořené plug-iny oblíbené i mezi dalšími uživateli Vagrantu do nástroje začleněny. [15]

Kapitola 5

Testování softwaru

Následující kapitola seznamuje čtenáře s procesem testování softwaru, klasifikací testů dle různých kategorií a možnostmi automatického testování aplikací.

5.1 Proces testování

Na testování programů lze nahlížet z různých pohledů. Kromě technického hlediska je důležité zvážit i ekonomickou stránku testování a vnímání testování lidskou psychikou.

Samotná definice testování může být subjektivní a při její špatné volbě existuje možnost, že se k celému procesu testování přistupuje nesprávně. Testování neslouží k prezentaci bezchybného běhu programu nebo ujišťování, že aplikace vykonává to, co se od ní očekává. Proces testování by měl do testovaného softwaru přinášet určitou přidanou hodnotu. Vhodněji tak lze testování popsat jako proces, jehož cílem je spouštění programu za účelem objevení chyb. Předpokládá se tedy, že testovaná aplikace chyby obsahuje. Rozhodující faktor v přístupu k testování představuje také vyhodnocování úspěšnosti a neúspěšnosti testů. Za úspěšný lze považovat takový test, jenž objeví chybu a navede vývojáře k jeho opravě, nikoliv test, který kontinuálně prochází (ačkoliv se v takový test postupně mění každý, čímž je postupně budována důvěra v testovanou aplikaci).

Bude-li se na testování nahlížet jako na proces objevování chyb s předpokladem, že testovaný program chyby obsahuje, lze se následně ptát, zda je možné objevit všechny chyby aplikace. Typicky nelze odhalit veškeré chyby a nedostatky programu – příčinu představuje fakt, že obvykle není možné, nebo by bylo ekonomicky nevýhodné, otestovat všechny možné kombinace vstupů testovaného programu. Z ekonomického hlediska je vhodné před samotným započetím procesu testování připravit jeho strategii, založenou na výběru vhodných druhů testů pro daný program.

5.2 Základní principy testování

Jedním ze základních principů testování je správná definice vstupních dat a jim odpovídající přesný popis dat výstupních. Specifikace vstupu a výstupu by měla být součástí každého testovacího případu.

Další praktiku představuje princip, který tvrdí, že programátor by se měl vyvarovat testování vlastních programů. Problém spočívá v lidském přístupu – autor kódu ví, co by měla jeho aplikace vykonávat, čímž vzniká riziko, že testovací případy budou kontrolovat pouze úspěšné scénáře. Navíc je vývojář nucen zcela otočit svůj přístup k práci – prvně se snaží program vytvořit a následně by se jej měl pokusit narušit hledáním chyb.

S předcházejícím principem souvisí také fakt, že by ani organizace jako celek neměla testovat svůj vlastní software, ale spíše delegovat tuto činnost na nezávislou stranu.

Každé testování by mělo být zakončeno důkladným průzkumem testovacích výsledků. V případě nedostatečné (nebo zcela chybějící) analýzy výsledků může docházet k ignorování prvotních příznaků chyb, jež vyvstanou až v dalších fázích vývoje nebo testování.

Dalším platným principem testování je vhodná příprava vstupů pro jednotlivé testovací případy. Množina vstupních dat by měla být složena ze dvou druhů hodnot – validních a pro program přípustných, následně pak nevalidními a aplikací neočekávanými.

Důsledkem dodržování předchozí praktiky by měla být kontrola, zda program během svého běhu nevykonává neočekávané vedlejší činnosti.

Testovací případy by neměly být zahazovány. S tímto problémem se lze typicky setkat u interaktivních programů, u nichž je možné testovací případy vymýšlet a spouštět během jejich používání, ale po dokončení takového testovacího případu nejsou žádným způsobem persistovány. Ukládání testovacích případů a jejich následné spouštění po jakékoliv úpravě programu se nazývá regresní testování.

S dříve definovaným procesem testování souvisí další princip – plánování testovacího procesu by nemělo probíhat s předpokladem, že se nenaleznou žádné chyby.

Pravděpodobnost přítomnosti chyby v konkrétní části programu je přímo úměrná počtu objevených a opravených chyb v této části. Obvykle je tento trend pozorován v sekcích kódu, jež jsou více náchylné k chybám.

Posledním ze základních principů testování softwaru je povaha samotného procesu testování. Jedná se o výjimečně kreativní a zároveň náročný úkol. I přes existující rozsáhlé zdroje poskytující informace o postupech a možnostech tvorby testů se vždy přístup k testování konkrétní aplikace liší. [16]

5.3 Klasifikace testů

Proces testování lze dělit podle možnosti přístupu k testovanému programu nebo úrovně testování.

5.3.1 Znalost vnitřní struktury

Jednou z možností klasifikace testů je jejich rozdělení podle znalosti vnitřní struktury testovaného softwaru. V takovém případě se testy dělí na *black-box* a *white-box* testování.

Black-box testování je důležitou strategií v oblasti testování softwaru, během kterého se k testovanému softwaru přistupuje jako k černé skříňce, jejíž vnitřní chování není známo. Testování probíhá pouze na základě specifikace softwaru, tedy definici vstupních a výstupních dat. Má-li být metoda použita k nalezení všech chyb softwaru, je nutnou podmínkou vyčerpávající množina vstupů. To znamená využití každé možné kombinace vstupů jako testovací případ. Při nedodržení podmínky vyčerpávající množiny vstupů nelze očekávat nalezení všech chyb, protože lze vytvořit předpoklad, že pro všechny připravené testovací vstupy obsahuje program kontrolu, na základě níž vrátí korektní výstup. Vytvoření vyčerpávající množiny vstupů je technicky možné, ale ekonomicky a časově nevýhodné, což potvrzuje dva dříve popsané poznatky – software nelze testovat tak, aby bylo prokázáno, že neobsahuje žádné chyby. Dalším poznatkem je ekonomický dopad procesu testování – účelem testování by měla být maximalizace počtu nalezených chyb při zachování konečného počtu testovacích případů.

White-box testování je strategie založená na průzkumu programu, na základě něhož se následně vytvářejí testovací data. Analogií k vyčerpávající množině vstupů v případě *black-box* testování je vyčerpávající množina cest – pokud je pomocí testovacích scénářů zajištěn každý možný průchod testovaným softwarem, pak o něm lze prohlásit, že byl kompletně otestován. Analogicky je taktéž tento přístup ekonomicky a časově náročný. Vytvoření vyčerpávající množiny cest navíc předpokládá, že každá cesta v testovaném softwaru je testovatelná.

Oba výše popsané přístupy lze kombinovat za účelem vytvoření vhodné množiny testovacích případů a testovacích dat.

5.3.2 Úroveň testu

Podle úrovně abstrakce, jež se typicky řídí vodopádovým modelem, lze klasifikovat testy na jednotkové, integrační a systémové.

Na úrovni návrhu a specifikace požadavků na řešení se nacházejí systémové testy, jenž ověřují naplnění definovaných požadavků. Velmi často se kategorie systémových testů překrývá s *black-box* testováním.

Fázi předběžného návrhu odpovídají integrační testy a představují průnik *black-box* a *white-box* testů.

Poslední úrovní je konkrétní návrh programu, jemuž se přisuzují testy jednotkové. S nimi se asociují *white-box* testy, neboť se typicky jedná o testování napsaného kódu. [17]

5.4 Pokrytí testy

Mezi základní metriky při vyhodnocování testovací sady se řadí pokrytí, jenž určuje množství kódu, které je celou testovací sadou spouštěno. Pokrytí nabývá hodnot od 0 % do 100 % a funguje jako dobrá negativní indikace – pokud testovací sada dosahuje nízkého pokrytí, pravděpodobně není program dostatečně testován. Vysoké pokrytí ale nezaručuje celkovou kvalitu samotného procesu testování.

Metrika pokrytí se dělí na pokrytí kódu a pokrytí větví. Pokrytí kódu představuje poměr mezi počtem spuštěných řádků testovací sadou a celkovým počtem řádků kódu aplikace. Poměr mezi počtem prošlých větví testovací sadou a celkovým počtem všech větví programu se označuje jako pokrytí větví.

5.5 Jednotkové testy

Jednotkový test je automatický test, jenž ověřuje malou část kódu (jednotku). Vzhledem k rozsahu by jednotkové testy měly být rychlé. Jednotkové testy vždy běží zcela izolovaně – závisí-li testovaná jednotka na dalších jednotkách, musí být tyto závislosti nahrazeny jejich testovými alternativami.

5.5.1 Testové alternativy

Testové alternativy (v angličtině také zvané *test doubles*) slouží v jednotkových testech k nahrazení reálných závislostí, jež nelze v rámci procesu testování snadno nakonfigurovat a provozovat (nebo by tento proces trval delší dobu, čímž by se navýšil i čas nutný ke spuštění jednotkového testu). Typicky se dělí na mocky (anglicky *mocks*) a stuby (anglicky *stubs*).

Mock umožňuje emulovat a následně zkoumat chování odchozích interakcí – typ kolaborace, kdy testovaná jednotka mění stav nebo vyvolává akci v dané závislosti. Kupříkladu lze vytvořit mock pro e-mailový server, u něhož je po odpovídající interakci nutno zkontrolovat, zda se pokusil odeslat e-mailovou zprávu.

Stub emuluje chování příchozích interakcí – v kontextu testování se jedná o spolupráci mezi testovanou jednotkou a její závislostí, jejíž podstatou je pouze získání dat z dané závislosti. Příkladem může být vytvoření stubu jako náhradu relační databáze nebo souborového systému, ze kterých čte testovaný kód data.

5.5.2 Parametrizované testy

U malých částí kódu (metod, funkcí nebo tříd), jež jsou podrobeny jednotkovému testování, často dochází k jejich duplikaci, přičemž jediný rozdíl mezi jednotlivý testy spočívá v různých vstupních datech. Parametrizované testy adresují zmíněný problém – v prvním kroku se definuje tabulka se vstupy a jim odpovídajícími výstupy a v druhém je implementován samotný jednotkový test, jenž se postupně spouští se vstupy z tabulky definované v prvním kroku. Parametrizované testy tak nepředstavují odlišný druh testování ale spíše redukci testovacího kódu a zvýšení čitelnosti jednotkových testů.

5.6 Integrační testy

Smyslem integračních testů je ověření fungování programu jako celku. Narozdíl od jednotkých testů, jejichž cíl představuje izolované testování businessové logiky jednotlivých částí aplikace s nahrazenými závislostmi, integrační testy ověřují kolaboraci jednotek včetně jejich závislostí.

Integrační test pracuje s reálnými instancemi závislostí, jež jsou ve správě vývojáře nebo společnosti, která program vyvíjí. Pracuje-li aplikace se závislostmi, jež nelze společností spravovat, nahrazují se v rámci integračních testů testovými alternativami (podobně jako u testů jednotkových). [18]

5.7 Funkční a systémové testy

Funkční testování slouží k objevení nesouladů mezi programem, k němuž se přistupuje jako k *black-box* řešení, a specifikací rozhraní daného programu, na základě níž vznikají veškeré testovací případy.

Účel systémových testů spočívá v porovnání výsledného programu (nebo celého systému) proti specifikovaným požadavkům a cílům z analytické části vývoje. K vytváření systémových testů však nelze využít pouze na počátku zadané požadavky, neboť v jejich popisu typicky schází definice rozhraní aplikace, a proto se při tvorbě systémových testů nejprve prochází úvodní zadání s požadavky a následně se formulují testovací případy s využitím uživatelské dokumentace. Systémové testy by měly pokrývat funkční i nefunkční požadavky.

5.8 Regresní testování

Proces regresního testování se spouští po libovolných úpravách v programu (ať už se jedná o přidání nové funkcionality nebo opravu objevené chyby). Jeho smyslem je odhalení regrese v aplikaci, tedy ověření, zda veškeré původní funkcionality nebo vlastnosti programu zůstaly i po provedených změnách

funkční. Typicky se regresní testování provádí spuštěním určité podmnožiny již existujících testovacích případů. [16]

5.9 Automatizace procesu testování

Proces automatizace testování lze charakterizovat jako přechod z člověkem ručně spouštěných a vyhodnocovaných testů na jejich automatickou exekuci za užití podpůrných softwarových nástrojů.

Manuální a automatizované testování se od sebe liší na úrovni základních principů. Ručně prováděné testování lze typicky kategorizovat do dvou skupin – průzkumné a plánované.

Během průzkumného testování systému zůstává vývojáři (nebo osobě zaměřené přímo na testování) volný prostor na průzkum libovolné části programu a objevování chyb v něm. Metoda se obvykle využívá k nalezení co největšího počtu problémů, ale její úspěšnost se odvíjí od preciznosti a trpělivosti jedince, jenž testovaný systém zkoumá.

Plánované testy jsou lidmi předem připravené testovací scénáře, obvykle s definovanými vstupy a výstupy, a určeny jsou opět pro zpracování člověkem. Jejich provedení lze delegovat na jinou osobu nebo tým, v závislosti na kvalitě zpracování jednotlivých testovacích případů.

Automatizace procesu testování typicky nepřináší alternativu k první skupině manuálních testů – softwarové nástroje vyžadují přesnou definici svého chování při exekuci testovacích případů a postrádají „intuici“, jež do průzkumného testování vnáší člověk. Pokusy o vytvoření nástroje, jenž by dokázal zcela pokrýt rozsah průzkumných testů, se obvykle jeví jako těžko udržitelné.

Druhá skupina manuálního testování se svojí definicí více přibližuje konceptu automatického testování. V obou případech se u každého testovacího případu očekává specifikace, ačkoliv dokumentace testu určeného pro člověka dovoluje určitou benevolenci v jeho formulaci. Plánované manuální i automatizované testy vyžadují údržbu – každý zásah do programu může vynucovat úpravu testovacích případů. Ošetření selhání během testování představuje další společnou vlastnost obou přístupů, v tomto ohledu je pro automatizaci nutná přesná specifikace chování, protože software není narozdíl od člověka schopen posoudit závažnost a povahu selhání testovacího scénáře do takové míry, aby sám rozhodl o možnostech pokračování nebo například zopakování dalších testů. Podobné platí také pro závislosti mezi testovacími případy.

Proces automatizace testování se proto obvykle zakládá na dokumentaci existujících manuálních plánovaných testů a jejich následného převedení do odpovídajícího softwarového nástroje, jenž posléze spouští definované testovací případy sám. Automatizace procesu přináší ušetření času nutného k ručnímu provádění testů, sjednocuje testovací scénáře, což dovoluje jejich přenos například na server s kódem nebo na lokální počítač vývojáře. Vzhledem k softwa-

rovým nástrojem vyžadované preciznosti se také typicky jednotlivé testovací případy stávají jednoduššími a pochopitelnějšími.

5.9.1 Nástroje pro automatické testování

Automaticky spouštěné testy se často zapisují přímo ke kódu programu a k jejich spouštění je nutná přítomnost odpovídajícího softwarového nástroje, známého jako testovací framework. Jejich výhody spočívají v jednoduchém používání, formátovaném výstupu průběhů a výsledků jednotlivých testů nebo například možnosti spouštět pouze určité testy. Testovací frameworky také typicky umožňují spuštění vlastního kódu před nebo po samotném testu (případně skupinou testů). Určeny jsou obvykle pro běh jednotkových testů, ale lze je využít i pro spouštění testů integračních nebo systémových. Pro programovací jazyk Java existují například frameworky *JUnit* nebo *TestNG*. [19]

Kapitola 6

Datová úložiště desktopových aplikací

Kapitola čtenáři představuje možnosti persistence dat desktopových aplikací. Jmenovitě pojednává o textových souborech ve formátu XML a JSON a relační databázi SQLite.

6.1 XML

XML (*eXtensible Markup Language*) je otevřený formát navržený skupinou tvořící společnosti, instituce a jednotlivci zvanou *World Wide Web Consortium* (W3C). Lze jej kategorizovat jako značkovací jazyk. Účelem značek v jazyce XML je kontejnerizace informací do tzv. elementů, které se do sebe mohou dále zanořovat. XML dále zavádí pojem dokument, jenž se skládá z jediného vnějšího elementu, který obsahuje další elementy. Součástí XML dokumentu mohou být také dodatečné administrativní informace, například verze XML, použitá znaková sada nebo další metadata.

Z technického hlediska nelze XML považovat za značkovací jazyk, neboť jazyk formálně obsahuje množinu přípustných slov a gramatiku, pomocí nichž lze z jazyka tvořit výsledné věty. XML nedefinuje žádnou množinu přípustných elementů, namísto toho určuje přísná syntaktická pravidla, na základě kterých lze vybudovat jazyk vlastní. Takto vytvořené jazyky lze následně dodatečně validovat (více v podsekci Validace struktury a obsahu XML).

Motivem pro vytvoření XML byl chybějící způsob, jímž by bylo možné popisovat strukturovaná data. Původní elektronické formáty sloužily především ke grafické reprezentaci dat namísto popisu jejich struktury a významu. Jedním z prvních větších pokusů o vytvoření jazyka řešící zmiňovaný problém byl *Generalized Markup Language* (GML) navržený společností IBM. GML sloužil především ke kódování dat tak, aby jim rozumělo několik různých systémů. Vzhledem k popularitě jazyka GML se instituce ANSI rozhodla k vytvo-

ření standardu postaveném na GML. Výsledný formát *Standard Generalized Markup Language* se stal postupem času velmi oblíbeným, což vedlo k ratifikaci standardu mezinárodní organizací pro normalizaci (ISO). Dalším velkým objevem na poli generického popisu dat bylo HTML (*Hypertext Markup Language*). K dosažení jednoduchosti formátu HTML byly obětovány některé vlastnosti SGML a obecně generického popisu dat. Některé elementy jazyka HTML slouží pouze ke grafické reprezentaci dat nebo je HTML například benevolentní k uzavírání jednotlivých elementů. Standard SGML tak stále exceloval v popisu struktury a významu dat, nebyl ale vhodný pro použití na webu (především kvůli jeho vysoké komplexnosti) – tento fakt nechal následně vzniknout jazyku XML.

Mezi hlavní cíle standardu XML patřily především schopnost generického popisu dat (konkrétně nevytvářet obecný jazyk, jenž by dokázal popsat libovolná data, ale spíše poskytnout možnost vytvořit jazyk vlastní), jednoznačný výklad každého dokumentu, oddělení grafické reprezentace dat od popisu jejich významu a jednoduchost (oproti jazyku SGML). Zároveň bylo snahou navrhnout jazyk tak, aby jej bylo možno co nejstriktněji validovat.

Zmíněné vlastnosti činí formát XML vhodným k ukládání dat na fyzickém systému nebo výměně informací mezi různými systémy. Vzhledem k jeho popularitě již také existuje mnoho nástrojů, které zvládnou XML dokumenty různými způsoby zpracovávat a validovat (ať už se jedná o samostatné aplikace nebo knihovny pro programovací jazyky).

Zpracování XML dokumentů probíhá vždy sekvenčně, proto není zcela vhodný pro ukládání dat, k nimž je nutno přistupovat často a náhodně.

6.1.1 Validace struktury a obsahu XML

K validaci XML dokumentů se užívají schémata, která představují test, jímž musí XML dokument úspěšně projít, aby byl považován za validní vůči danému schématu. Kontrola dokumentu probíhá typicky na několika úrovních – strukturální, datové, integritní a businessové – přičemž za nejdůležitější je považována strukturální validace.

Prvním jazykem pro validaci XML dokumentů byl *Document Type Definition* (DTD), jenž vychází z SGML. DTD udává konečnou množinu elementů, jenž lze v XML dokumentu používat. Následně definuje možnosti zanoření definovaných elementů a jaká data mohou být do těchto elementů vložena, určuje pořadí jednotlivých elementů a jejich povinnost. Pro každý element ukládá také množinu povolených atributů. DTD představuje samostatný jazyk, jenž není založen na XML, a zameřuje se primárně na strukturální validaci dokumentů. Skrze DTD nelze přísněji validovat datový obsah elementu (z pohledu různých datových typů). Jazyku DTD také schází podpora pro jmenné prostory, jež jsou základní částí standardu XML.

Problémy s nedostatky jazyka DTD adresovalo konsorcium W3C, jež představilo koncept schémat a svůj standard *XML Schema*. Schémata nejsou na-

rozdíl od DTD součástí specifikace XML a vystupují tak jako volitelná technologická rozšíření. XML Schema je sám o sobě XML dokument, což přináší výhodu v možnostech jeho validace. XML Schema rozděluje definice na jednoduché a komplexní typy, přináší datové typy pro obsahy elementů a podporuje jmenné prostory.

6.1.2 Zpracování XML

Ačkoliv se standard XML řadí mezi otevřené, vzniklo na poli zpracování XML dokumentů několik standardních řešení. Pro datové toky událostí (anglicky *event streams*) se jím stal SAX, pro čtení XML dokumentu jako stromu objektů poté DOM.

Simple API for XML (SAX) představuje jeden z prvních způsobů pro práci s XML. SAX do programu postupně (od začátku dokumentu) zasílá jednotlivé události bez dalšího kontextu a zpracování nechává na aplikaci. Mezi události se řadí například začátek a konec dokumentu nebo začátek a konec elementu. Zpracování XML na základě toku událostí přináší výhody v podobě rychlosti, paměťové efektivitě nebo zachování integrity dokumentu (tok událostí lze pouze číst).

Document Object Model (DOM) představuje rozhraní pro paměťovou reprezentaci XML dokumentů. Při zpracování metodou DOM je každá část XML dokumentu považována za uzel a každý takový uzel je reprezentován svojí vlastní třídou v API. S jednotlivými uzly lze následně manipulovat, přidávat je nebo odstraňovat. XML dokument zpracovaný metodou DOM je vždy celý načten do paměti. [20]

6.2 JSON

JavaScript Object Notation (JSON) je otevřený datový formát odvozený z literálů programovacího jazyka JavaScript, což z něj činí jeho podmnožinu. Formát JSON se primárně využívá k výměně informací mezi systémy.

JSON formalizoval Douglas Crockford v roce 2006 v rámci RFC 4627. Organizace *Internet Engineering Task Force* (IETF) následně vydala svoji specifikaci (RFC 7159). Hlavní rozdíl mezi specifikacemi představuje přístup k validaci textu v JSON formátu. JSON vznikl jako náhrada standardu XML pro výměnu informací na webu (především kvůli velikosti a komplexitě XML dokumentů).

JSON specifikace připouští dvě možnosti kompozice dat – kolekci dvojic v podobě řetězcového klíče a hodnoty (objekt) nebo uspořádaný seznam hodnot (pole). Hodnotou se rozumí další kompozice (tedy objekt nebo pole), základní hodnota nebo *null* hodnota. Základní hodnoty mohou nabývat různých datových typů – číslo (celé nebo desetinné), řetězec nebo *boolean*.

Ačkoliv byl JSON formát založen na jazyce JavaScript, existují vzhledem k jeho jednoduché struktuře knihovny určené ke zpracování a produkci textu ve formátu JSON i pro další programovací jazyky. [21]

6.2.1 Validace textu ve formátu JSON

Samotná specifikace datového formátu JSON definuje pouze syntaxi a neobsahuje další možnosti strukturální nebo datové validace. Tento problém adresuje projekt *JSON Schema*.

JSON Schema je navrhovaná specifikace umožňující přísnější validaci textů ve formátu JSON. Princip kontroly je založen na konceptu schémat, která se také zapisují ve formátu JSON. JSON Schema dovoluje validovat samotnou strukturu JSON textu, povinnost klíčů nebo datové typy hodnot. Neomezuje se pouze na datové typy definované v JSON specifikaci, ale přidává možnost kontrolovat data v různých formátech (například formát data a času, IP adresy nebo e-mailové adresy). Dovoluje také specifikovat vlastní formát řetězcových dat pomocí regulárních výrazů. [22]

6.3 SQLite

SQLite je svobodný multi-platformní software poskytující relační databázi. Narozdíl od většiny ostatních relačních databází není nástroj SQLite postaven na architektuře klient-server. Ke svému provozu nevyžaduje běh několika různých procesů a persistenci dat do velkého množství souborů, což umožňuje přímou integraci SQLite do cílové aplikace. Jediným sdíleným zdrojem je jeden soubor, jenž obsahuje celou databázi – definici tabulek i samotná data. SQLite databázi tak lze snadno vytvořit, migrovat nebo zálohovat.

Vzhledem k malé velikosti celého nástroje je SQLite úsporný z pohledu provozu a nároků na zdroje cílového stroje. Lze jej tak využít i pro vestavěná nebo přenosná zařízení.

Své uplatnění nachází SQLite jako klasické relační úložiště dat pro aplikace, ale také jako cache, neboť pomocí SQLite lze vytvářet databáze přímo v paměti. Zavedení SQLite jako úložiště pro aplikace umožňuje sjednocení nejen businessových dat ale také konfigurací nebo jiných metadat, jež lze často modelovat jako relační data.

SQLite disponuje systémem založeném na dynamických datových typech, čímž se významně liší od jiných relačních databází, které typicky vyžadují nastavení statického datového typu na sloupci a nepřipouští vkládání dat jiného datového typu. Další rozdíl oproti ostatním relačním databázím představuje možnost přístupu k několika různým databázím v rámci jednoho připojení, což umožňuje načítání dat z různých databázových souborů v rámci jediného SQL dotazu.

Zjednodušení celého relačního systému na jednu binárku a jeden soubor obsahující celou databázi s sebou přináší omezení, kvůli kterým není SQLite

vhodný pro některá využití. Ačkoliv například podporuje transakce (jež plně odpovídají konceptu ACID), není SQLite navržen k velkému konkurentnímu zatížení, pro které jsou naopak databázové systémy založené na architektuře klient-server stavěny. Protože SQLite umisťuje veškerá data do jediného souboru na souborovém systému, není nástroj určen pro ukládání velkého množství dat. Extrémně velké soubory mohou způsobovat potíže s výkonem při náhodném přístupu. SQLite databáze také neposkytuje žádnou možnost autentizace nebo autorizace a spoléhá se pouze na oprávnění na úrovni souborového systému. API nástroje SQLite nabízí základní nastavení autentizace na aplikační úrovni, ale protože je celá databáze stále uložena v jediném souboru na souborovém systému, lze typicky tuto aplikační autentizaci obejít přímým přístupem k souboru. [23]

Kapitola 7

Návrh a realizace

Kapitola seznamuje čtenáře s návrhem řešení založeném na zvolení vhodných postupů a technologií analyzovaných v předchozích kapitolách s ohledem na dříve definované požadavky.

7.1 Návrh a příprava nového vývojového prostředí

V analytické části práce byly představeny různé metody tvorby vývojových prostředí.

První skupinu představovaly virtuální stroje, jmenovitě nástroje VirtualBox a VMWare Workstation Player. Virtualizované počítače představují velmi flexibilní řešení k provozu libovolného softwaru na téměř jakémkoliv stroji. Jejich komplexita v podobě emulování veškerého hardwaru umožňuje souběh různých nástrojů, jenž jsou následně v kompletní správě uživatele.

V další části byla představena kontejnerová řešení Docker a Podman, jenž narozdíl od virtuálních strojů pracují na úrovni procesů operačního systému. Jsou založená na nativních částech linuxových distribucí, jejich provoz je ale možný i na ostatních operačních systémech s využitím virtualizace. Kontejnerové nástroje také přináší standardizaci v podobě *Open Container Initiative*.

Posledním analyzovaným nástrojem byl Vagrant, jenž je vyvíjen přímo za účely sjednocování vývojových prostředí. Ve výchozím nastavení využívá ke své činnosti virtuální stroje a poskytuje možnost komplexního nastavování a konfigurace rozsáhlých vývojových prostředí.

Vzhledem k aktuálním potřebám nástroje Winch na vývojové prostředí není nutné volit komplexní nástroj Vagrant nebo spravovat konfiguraci virtuálních strojů. Kontejnerová řešení představují jednoduchou možnost k běhu jednotlivých databázových strojů, každý zcela izolovaný od ostatních. Tabulka 7.1 zachycuje splnění jednotlivých požadavků kontejnerovými nástroji Docker a Podman.

	DEV-1	DEV-2	DEV-3
Docker	✓	✓	✓
Podman	✓	✓	✓

Tabulka 7.1: Vyhodnocení požadavků na vývojové prostředí

Nástroje Docker i Podman splňují všechny definované požadavky na nové vývojové prostředí. Podman nabízí bezpečnější prostředí k provozování služeb a disponuje možností běhu kontejnerů v podech. Docker představuje více známý nástroj pro kontejnerizovaná řešení a poskytuje možnost sjednocovat různé služby pod *Docker Compose*.

Protože bude vybraný nástroj sloužit k provozu podporovaných databázových strojů primárně na lokálních prostředích vývojářů, nepředstavuje *rootless* vlastnost nástroje Podman velkou výhodu. Podobně také možnost provozu kontejnerů v rámci podů nepřináší zásadní přínos – jednotlivé databáze mohou (a v případě vývojového prostředí pro aplikaci Winch by i měly) běžet naprosto izolovaně.

Ze zmíněných důvodů bude tedy pro návrh a implementaci nového vývojového prostředí aplikace Winch uvažován Docker spolu s jeho nástrojem *Docker Compose*. Rozhodnutí ale pro vývojáře nepředstavuje omezení pouze na nástroj Docker, neboť Podmad disponuje velmi podobnou (a vě většině případů zcela totožnou) množinou příkazů, a pro spuštění jednotlivých databází zvlášť (Podman nedisponuje žádnou alternativou k *Docker Compose*) jej tak lze taktéž využít.

7.1.1 Docker Compose pro databázové stroje

Strukturu a definici jednotlivých databázových služeb definovaných v souboru `docker-compose.yml` umístěném v kořenové složce projektu Winch zachycuje listing 7.1.

```
version: "3.9"
services:
  db2:
    build: disl-winch-db2/src/test/resources/db2
    ports:
      - "50000:50000"
    volumes:
      - /db2:/database
    privileged:  true
  mssql:
    build: disl-winch-mssql/src/test/resources/mssql
    ports:
```

```
    - "1433:1433"
oracle:
  build: disl-winch-oracle/src/test/resources/oracle
  ports:
    - "1521:1521"
postgresql:
  build: disl-winch-postgresql/src/test/resources
  ports:
    - "5432:5432"
```

Listing 7.1: Soubor `docker-compose.yml`

Jednotlivé databázové stroje využívají definici obrazů (*Dockerfile*) v jednotlivých databázových modulech nástroje Winch. Databázový stroj IBM Db2 představuje oproti ostatním databázím výjimku ve svém fungování – k řádnému chodu vyžaduje propojení s hostitelským souborovým systémem, na který si ukládá data.

Všechny čtyři databáze lze spustit jediným příkazem `docker compose up`. První spuštění vyžaduje více času, neboť se stahují originální databázové obrazy z centrálních registrů. Další exekuce pak již využívají lokálně uložené obrazy.

Ačkoliv pro všechny typy podporovaných databází existují kontejnerové obrazy, nebylo možné dodržet přesné verzování uvedené v požadavku Pokrytí databázových závislostí (DEV-3), neboť pro některé verze databázových strojů chybí obrazy ve veřejných registrech. V prvním případě se jedná o databázi Oracle, jež je v produkčním prostředí podporována ve verzi 12c, ale v testovacím kódu se očekává edice *Express Edition* (XE), pro níž jsou obrazy dostupné pouze ve verzích 11g, 18c a 21c (vybrána byla verze 11g). V druhém případě problém představuje databáze Microsoft SQL Server, jež disponuje obrazem s verzí 2019, ale nikoliv s verzí 2016.

Základní kontejnerové obrazy všech databází se v jednotlivých databázových modulech anonymizačního nástroje Winch rozšiřují o inicializaci, jež typicky zahrnuje vytvoření uživatelů a jejich práv, schémat a tabulek tvořící jednoduché relační modely, které lze následně naplnit testovacími daty. Takto vytvořené databáze se pak využívají v rámci integračních testů aplikace Winch, případně do nich lze nahlížet i mimo běh testů v rámci debugování.

7.2 Návrh testovací metodiky

V rámci návrhu nové testovací metodiky pro anonymizační nástroj Winch budou navržena pravidla a doporučení pro tvorbu jednotkových, integračních a systémových testů.

7.2.1 Nezávislost jednotkových a integračních testů

Žádné jednotkové a integrační testy by na sobě neměly být závislé. Výsledek jednoho testu nesmí ovlivňovat průběh dalších testů, neboť provázanost mezi testy vede k nepředvídatelnému chování při spouštění celé testovací sady.

Jednotlivé testy nesou odpovědnost za přípravu a následné vrácení testovacího prostředí do původního stavu (ať se jedná o lokální vývojové prostředí nebo testovací prostředí pro automatické sestavování a testování nástroje Winch).

7.2.2 Refaktoring testů

Všechny testy v nástroji Winch by měly projít refaktoringem, jenž by měl zahrnovat následující kroky:

- sjednocení stylu kódu,
- využití parametrizovaných testů,
- používání stejných asercí.

Testy v nástroji Winch se od sebe často liší ve stylizaci kódu, což způsobuje nepřehlednost při úpravě existujících testovacích případů. Parametrizované testy pomáhají ke snižování duplicitního kódu při testování konkrétní funkcionality s více vstupy. Adepta na využití parametrizovaných testů představuje například testovací třída `NotNullDecoratorTest`, jejíž testovací případ zachycuje listing 7.2.

```
@Test
 void testDecorateValue() {
  Decorator decorator =  new NotNullDecorator('N/A')
  assert "N/A" == decorator.decorateValue( null)
  decorator =  new NotNullDecorator()
  assert "vstup" == decorator.decorateValue('vstup')
}
```

Listing 7.2: Testovací případ v třídě `NotNullDecoratorTest`

Všechny testovací metody v aplikaci Winch by měly aplikovat jednotný způsob asercí výstupních a očekávaných hodnot. Použití asercí různých druhů ztěžuje vyhodnocování výsledků testovacích metod, neboť každá kategorie užitých asercí formátuje případný chybový výstup jinak. Z tohoto důvodu budou všechny aserce využívat již zavedenou knihovnu JUnit. Externí knihovna zajistí konzistentní chování například v případě migrace nástroje nebo některých tříd na jiný programovací jazyk.

Pravidelné refaktorování testovací sady by se mělo stát rutinním procesem při vývoji a testování aplikace Winch.

7.2.3 Zaznamenávání průběhu testování

V rámci testování anonymizačního nástroje Winch bude využita knihovna určená pro logování, jež lze pro lokální testovací prostředí nastavit tak, aby výstup zůstal pouze na standardním výstupu, ale například pro testovací prostředí na serveru může logy zapisovat i do dedikovaných souborů. Zavedení logovací knihovny přináší i úrovně logování, jež jsou využity v produkčním kódu, ale své uplatnění mohou nalézt i v kódu testovacím.

7.2.4 Zavedení systémových testů

V anonymizačním nástroji Winch kompletně chybí automatické systémové testy, jež by ověřovaly businessové požadavky a E2E fungování aplikace Winch.

Do aplikace tedy budou zavedeny automatické testy, jež budou ověřovat funkčnost výsledného nástroje Winch v jeho distribuované podobě. Ukázkový scénářový test ověřující impleteční řešení práce se nachází v sekci Zpětná kompatibilita s nástrojem EA. Automatické spouštění podobně definovaných testů lze zařadit do projektu Winch pod vlastní Gradle úkol.

7.3 Návrh a implementace nové datové vrstvy

Po vyhodnocení výsledků bakalářské práce Implementace datové vrstvy pro anonymizační nástroj [1] bylo rozhodnuto o vytvoření zcela nového návrhu datové vrstvy nástroje Winch. Návrh přepracovaného datového modelu představený ve zmíněné práci vnáší do aplikace nutnost zpracovávat komplexní stromovou strukturu složek umístěnou na souborovém systému, jež lze nahradit vhodným formátem dat, a následně persistovat celý anonymizační model do jediného souboru.

Návrh prezentovaný v bakalářské práci také příliš složitě odstraňuje provázanost existujícího datového modelu s nástrojem Enterprise Architect, neboť zachovává některé původní koncepty struktury anonymizačního modelu. Jmenovitě se jedná například o ponechávání existence „božských“ tříd, jež disponují přístupem k celému modelu (v původní datové vrstvě je představují implementace rozhraní `IWMetaSingleton` a v nově navrhované potom implementace rozhraní `MetaFactory`), nebo zachovávání „proxy“ tříd, které ale neimplementují návrhový vzor proxy.

7.3.1 Volba formátu datového úložiště

V rámci teoretické části práci byly prozkoumány tři formáty dat, jenž lze využít k ukládání informací desktopových aplikací – XML, JSON a SQLite.

XML představuje textový formát založený na značkách. Jedná se o otevřený a dobře zdokumentovaný typ souboru, jenž se využívá k ukládání a přenosu dat. Jako nástroj určený k vytváření vlastního jazyka je schopen pokrýt

různé případy použití na všech dostupných platformách disponujících souborovým systém. XML také disponuje možností validovat dokumenty proti předem definované specifikaci.

Formát JSON je podobně jako XML otevřený standardizovaný textový typ souboru. Svoji strukturu staví na principu klíč-hodnota, umožňující flexibilní formátování libovolných dat. Vzhledem k jeho nezávisloti na konkrétní platformě nebo technologii jej lze využít k ukládání lokálních dat nebo komunikaci mezi systémy.

Nástroj SQLite přináší výhody klasických relačních databází založených na modelu klient-server v podobě jednoduchého multiplatformního a kompletně otevřeného řešení. Nástrojům různých charakterů a libovolným běhovým prostředím nabízí výhody dotazovacího jazyka SQL a možnost relačního modelování dat aplikace.

	DATA-1	DATA-2	DATA-3
XML	✓	✓	✓
JSON	✓	✓	✓
SQLite	✓	✓	✗

Tabulka 7.2: Vyhodnocení požadavků na formát nového úložiště

Dle vyhodnocení požadavků v tabulce 7.2 se mezi kandidáty na formát nového úložiště anonymizačního modelu řadí formáty XML a JSON. I přes flexibilitu relační databáze SQLite jako lokální persistence pro desktopové aplikace nebude v dalších návrzích zvažován kvůli její absenci snadné verzovatelnosti v systémech správy verzí.

Oba typy XML a JSON splňují požadavky definové v kapitole Požadavky na řešení. Subjektivně působí formát JSON čitelněji než XML, jenž vzhledem k nutnosti párování elementů navíc zabírá více místa na souborovém systému. Jako úložiště nové struktury anonymizačního modelu tak bude vybrán formát JSON.

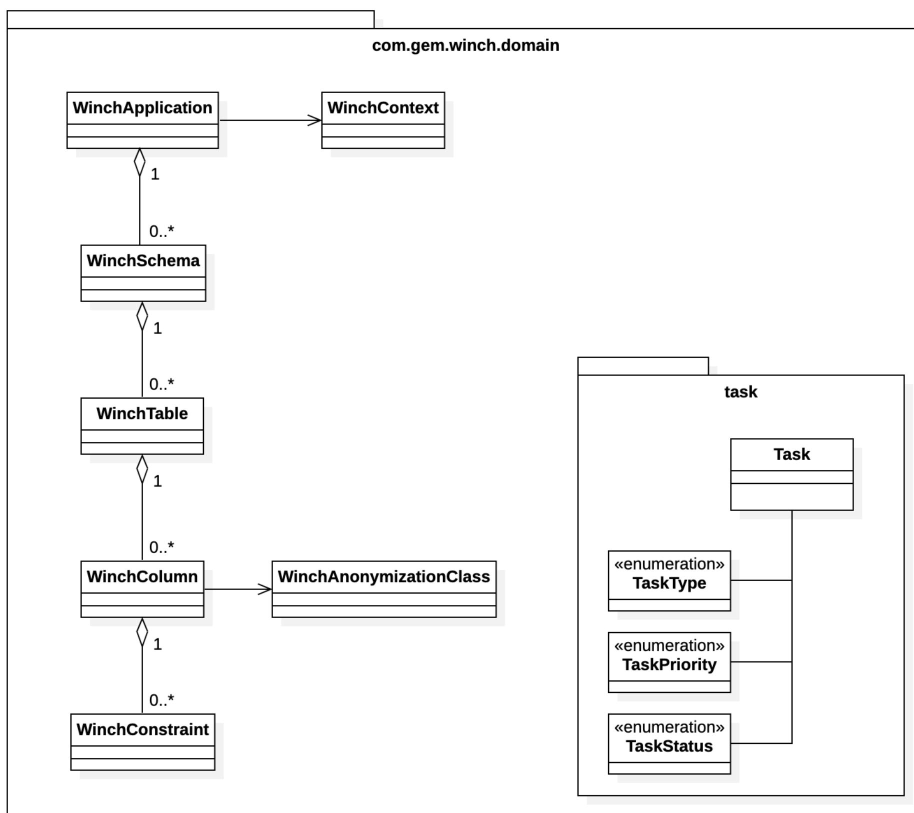
7.3.2 Nová architektura anonymizačního modelu

Vzhledem k požadavku Zpětná kompatibilita (DATA-4) a snaze vyhnout se kompletnímu přepsání anonymizačního nástroje Winch se podoba nové datové vrstvy odvíjí od stávající struktury anonymizačního modelu, jež byla představena v analytické části práce. Mapování tříd tvořících původní anonymizační model na nově navržené třídy zachycuje tabulka 7.3. Návrh logické struktury anonymizačního modelu je poté popsán zjednodušeným diagramem tříd na obrázku 7.1.

K přejmenování tříd (a jejich přesunu mezi balíčky) dochází především kvůli odstranění jmenné podobnosti mezi názvem třídy tvořící nový anonymi-

Původní třída	Odpovídající třída v nově navržené struktuře
`WApplication`	`WinchApplication`
`WSchema`	`WinchSchema`
`WTable`	`WinchTable`
`WColumn`	`WinchColumn`
`WConstraint`	`WinchConstraint`
`WAnonymizationClass`	`WinchAnonymizationClass`

Tabulka 7.3: Mapování původních tříd anonymizačního modelu na nové

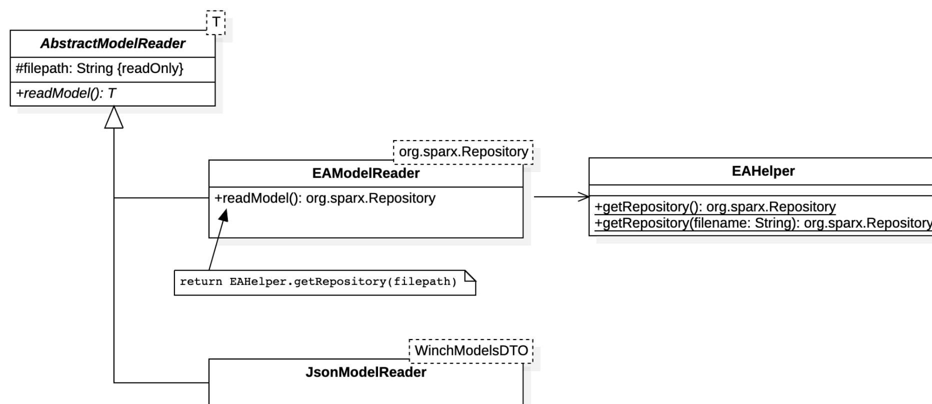


Obrázek 7.1: Návrh struktury anonymizačního modelu

zační model a jménem stereotypu v nástroji Enterprise Architect, neboť nově zavedené třídy již nejsou závislé na knihovně přístupující k modelům v EA.

Samotný anonymizační model reprezentuje třída `WinchApplication`, jež podobně jako třída `WApplication` ve stávající architektuře představuje vnější kontejner pro další třídy modelu.

Narozdíl od původních tříd anonymizačního modelu jsou nově navržené



Obrázek 7.2: Třídy pro čtení modelů z jejich datových úložišť

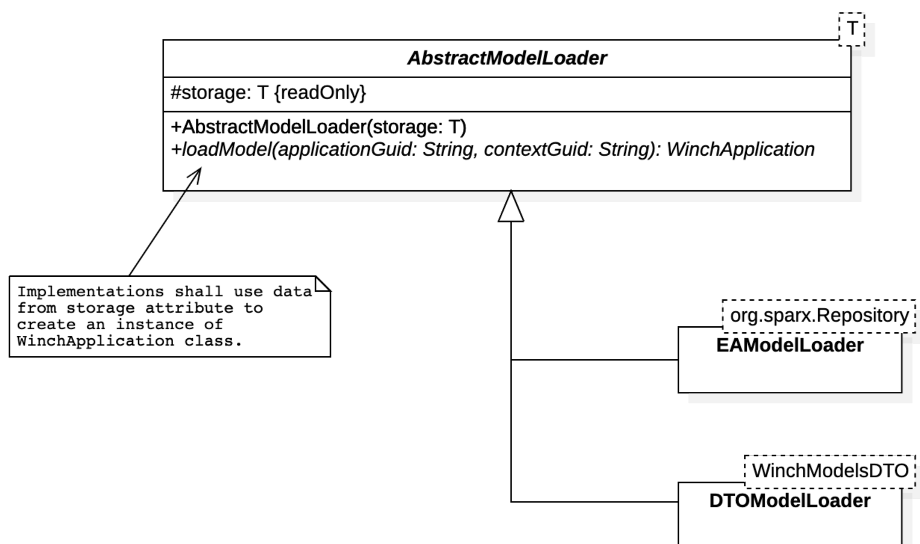
třídy zbaveny veškerých závislostí na externích knihovnách (s výjimkou nástrojů určených k logování). Závislost původních tříd anonymizačního modelu na knihovně zajišťující přístup k modelům v nástroji Enterprise Architect umožňovala nastavování všech potřebných vlastností daných tříd přímo z odpovídajích elementů v EA. Problém inicializace nově navržených tříd řeší nové třídy určené ke čtení modelů v daném formátu popsané diagramem na obrázku 7.2.

Abstraktní předek `AbstractModelReader` definuje jedinou parametrizovanou metodu `readModel`. Každý jeho konkrétní potomek pak představuje třídu, jež dovede přečíst informace z konkrétího úložiště, jenž v implementované metodě vrací. Vracené úložiště (které musí mít svoji vlastní reprezentaci v podobě Groovy třídy) by vždy mělo nést informace nutné k inicializaci tříd tvořících novou architekturu anonymizačního modelu popsanou na obrázku 7.1.

Potomek `EAModelReader` definuje jako úložiště anonymizačního modelu třídu `Repository` z knihovny zajiťující přístup k nástroji Enterprise Architect. Načítání samotného EA repozitáře deleguje na (v aplikaci již existující) pomocnou třídu `EAHelper`.

Třída `JsonModelReader` představuje potomka, jenž čte data anonymizačního modelu ze souborů ve formátu JSON, která jsou následně deserializována do obyčejné třídy `WinchModelsDTO`, jež není zatížena žádnými závislostmi. Z toho důvodu ji lze teoreticky využít i pro načítání anonymizačního modelu z jiných textových formátů. `JsonModelReader` využívá ke čtení JSON souborů Java knihovnu *jackson* namísto nativního Groovy parseru – knihovna *jackson* nabízí více možností konfigurace samotného čtení a zápisů textů ve formátu JSON.

Samotný abstraktní předek `AbstractModelReader` nedefinuje závislost na



Obrázek 7.3: Třídy pro čtení a přenos informací z úložišť modelů

žádné externí knihovně a v nástroji Winch Actor slouží především jako sjednocující prvek všech tříd načítajících specifická úložiště anonymizačních modelů.

Načítání informací z konkrétního úložiště a jejich následný přenos do tříd nově navržené struktury anonymizačního modelu obstarává další skupina tříd zaznamenaných na diagramu 7.3.

Každá třída dědící z abstraktního předka `AbstractModelLoader` by měla zajistit, že vracená instance třídy `WinchApplication` je okamžitě použitelná (kompletně nakonfigurovaná) ke spouštění veškerých procesů nástroje Winch, jenž anonymizační model vyžadují. Předek `AbstractModelLoader` slouží ke standardizaci tříd starajících se o inicializaci anonymizačního modelu (v podobném smyslu jako třída `AbstractModelReader`).

Popsaný způsob načítání anonymizačního modelu se od původní implementace liší v absenci vlastnosti zvané *lazy loading* – protože originální třídy tvořící anonymizační model přímo závisely na API nástroje Enterprise Architect, bylo v nich možné načítat určitá data až v okamžiku, kdy je daný proces potřeboval, neboť se třídy anonymizačního modelu mohly kdykoliv napojit na model v EA. Nynější návrh načítá veškeré informace před startem libovolného procesu, což poskytuje možnost načtení dat do struktur zcela nezávislých na originálním úložišti modelu. Pro zachování původního chování by bylo potřeba navrhnout další vrstvu kódu, s níž by mohly třídy anonymizačního modelu za běhu komunikovat, a jež by věděla, jak přistupovat ke konkrétnímu datovému úložišti. Protože ale procesy v nástroji Winch vyžadují načtení vždy celého anonymizačního modelu a samotný *lazy loading* nepřináší větší optimalizaci

ModelLoadingUtil
-modelFilepath: String {readOnly}
+loadModel(filepath: String, applicationGuid, contextGuid): WinchApplication
+getFileType(filepath: String): FileType
+getTaskFilePathByFileType(modelFilepath: String): String
+getTaskReaderWriterByFileName(modelFilepath: String): TaskReaderWriter

Obrázek 7.4: Třída `ModelLoadingUtil`

výkonu aplikace Winch, je možné jej odstranit.

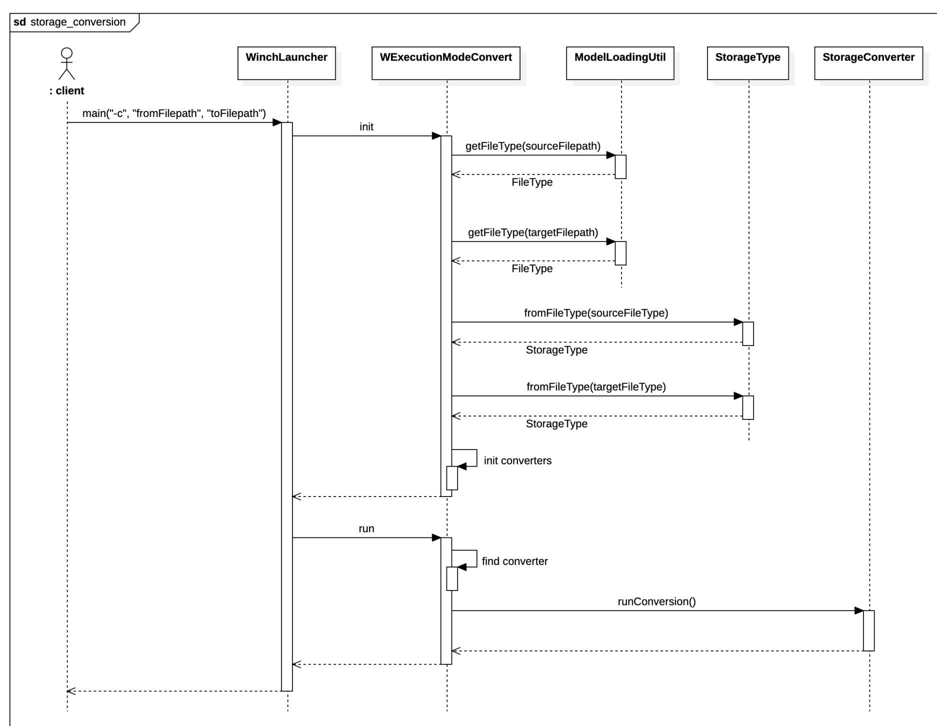
Potomek `EAModelLoader` abstraktní třídy `AbstractModelLoader` implementuje metodu `loadModel`, v rámci níž využívá atribut `storage` (datového typu `org.sparx.Repository` představující EA repozitář) k vytvoření a inicializaci instance třídy reprezentující anonymizační model `WinchApplication`. Proces mapování třídy `Repository` na třídu `WinchApplication` vycházel primárně z původního kódu určeného ke čtení anonymizačního modelu.

Taktéž v případě třídy `DTOModelLoader` se využívá vlastnost `storage` (v tomto případě datového typu `WinchModelsDTO`) ke čtení informací z daného úložiště a jejich kompletního namapování na třídu `WinchApplication`.

Navzájem korespondující potomci abstraktních tříd `AbstractModelReader` a `AbstractModelLoader` (ať se jedná o implementace načítajícího model z EA repozitáře nebo z JSON souboru) spolu přirozeně spolupracují v orchestrační třídě `ModelLoadingUtil` zdokumentované na diagramu 7.4.

Jejím cílem je poskytnutí programátorsky přívětivého rozhraní k načítání anonymizačního modelu z libovolného podporovaného úložiště. Tento cíl naplňuje metoda `loadModel`, jež na vstupu očekává cestu k souboru obsahující anonymizační modely, GUID aplikace (žádaný anonymizační model) a GUID kontextu (ke konfiguraci vybraného anonymizačního modelu). Metoda sama na základě přípony předané cesty k souboru rozhodne, o jaký druh úložiště se jedná, a na základě tohoto druhu pak inicializuje odpovídající potomky abstraktních tříd `AbstractModelReader` a `AbstractModelLoader`. K zachování zpětné kompatibility je umožňěno nepředávat cestu k souboru s anonymizačními modely – v takovém případě metoda předpokládá, že má model načíst z běžící instance nástroje Enterprise Architect.

Třída `ModelLoadingUtil` obsahuje další veřejné metody, jež vývojáři usnadňují práci nejen s anonymizačními modely. Jejich využití popisují následující podsekce.



Obrázek 7.5: Klientský pohled na konverzi datového úložiště

7.3.3 Přenos jednoho druhu úložiště na jiný

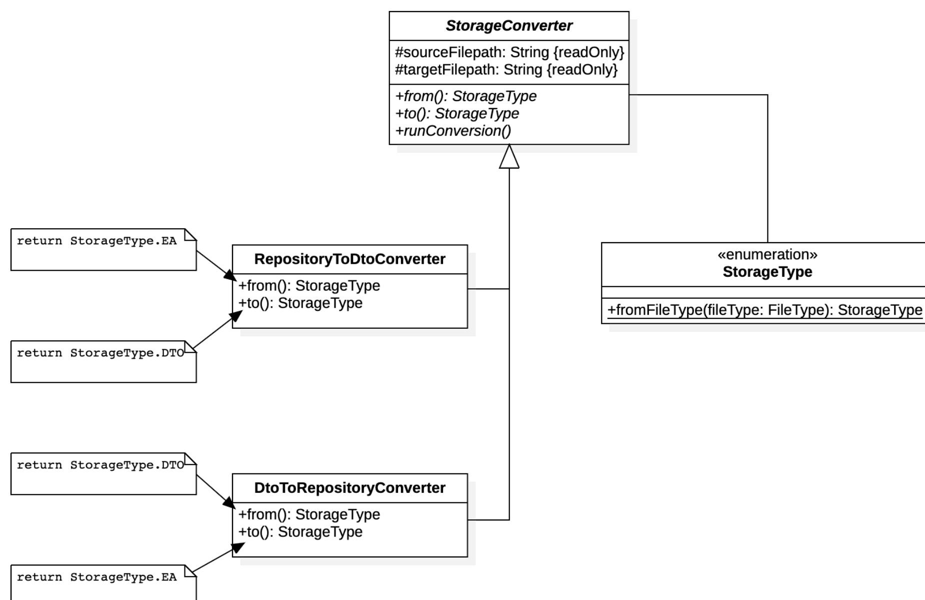
Požadavek Konverze formátů (DATA-5) na implementaci nové datové vrstvy nástroje Winch představuje i schopnost aplikace konvertovat jeden druh datového úložiště na jiný. Klientský pohled konverze jednoho zdroje dat na druhý popisuje sekvenční diagram na obrázku 7.5.

Třída `WExecutionModeConvert` představuje standardní Winch Actor mód, jenž se stará o orchestraci konverze úložišť. V iniciální fázi využívá veřejné metody třídy `ModelLoadingUtil` k získání typů souborů na vstupu. Souborové typy následně převede na typy úložišť využitím pomocných veřejných metod výčtové třídy `StorageType`. Nakonec je manuálně inicializována privátní kolekce konvertorů.

V exekuční fázi prochází třída `WExecutionModeConvert` svoji kolekci dostupných konvertorů (implementace abstraktní třídy `StorageConverter`) a při nalezení jej využívá ke spuštění samotné koverze.

Návrh a implementace samotných konvertorů je zachycena diagramem tříd na obrázku 7.6.

Metody `from` a `to` slouží k identifikaci směru konverze daného konvertoru. Samotná konverze probíhá v metodě `runConversion`, jež využívá zděděných



Obrázek 7.6: Konvertory datových úložišť

atributů nesoucích cesty ke zdrojovému a cílovému souboru s úložišti. V rámci ní dochází ke čtení anonymizačního modelu ze zdrojového souboru, mapování na třídy odpovídající úložišti cílového typu, a nakonec k zápisu zkonvertovaného anonymizačního modelu do cílového souboru.

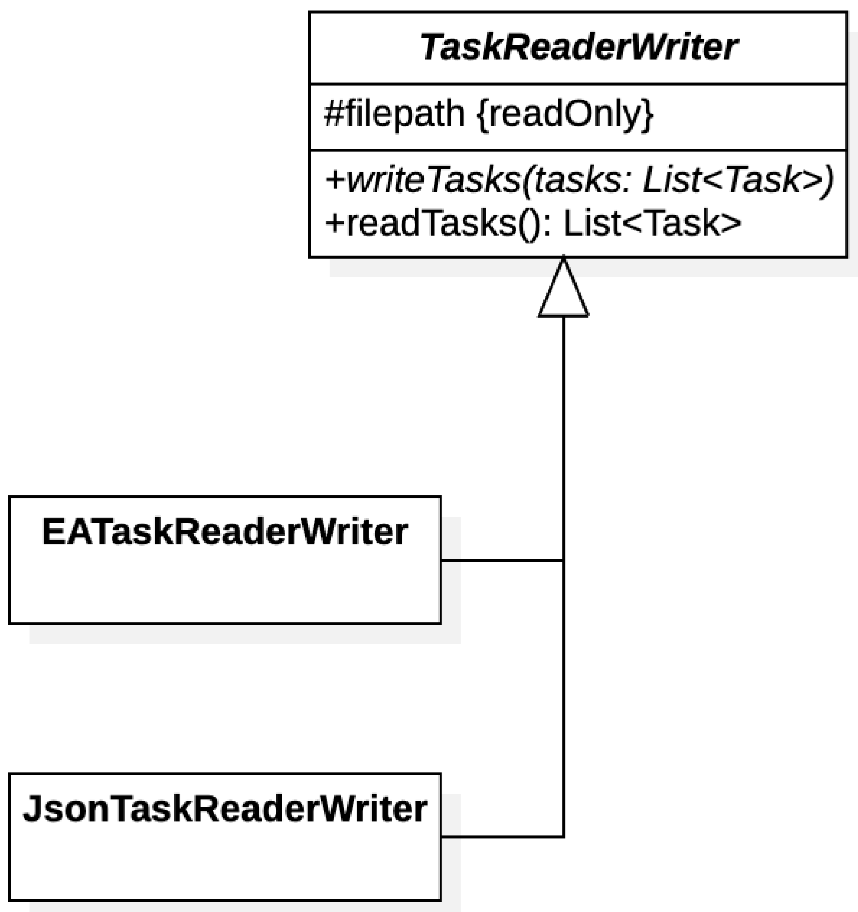
Implementace třídy `DtoToRepositoryConverter` představovala náročný úkol obnášející reverzní inženýrství původního kódu pro načítání anonymizačního modelu a průzkum dokumentace nástroje Enterprise Architect (konkrétně API pro automatizaci).

7.3.4 Proces objevování a persistence úkolů

Proces objevování citlivých údajů ztratil kvůli kompletnímu odstranění rozhraní `IWMetaSingleton` a jeho implementací možnost ukládat úkoly pro administrátora do nástroje Enterprise Architect (případně do souboru typu CSV pomocí třídy `CsvWMetaSingleton`, tato funkcionalita ale nebyla v kódu využita), jenž nyní navíc není jediným možným úložištěm anonymizačního modelu.

Za účelem generalizace čtení a zápisů *discovery* úkolů vznikla abstraktní třída `TaskReaderWriter`. Její struktura včetně potomků je zachycena na diagramu 7.7.

Atribut `filepath` nese cestu k souboru s úkoly – v případě původního úložiště anonymizačního modelu v repozitáři nástroje Enterprise Architect se



Obrázek 7.7: Abstraktní třída `TaskReaderWriter` a její implementace

TaskSingleton

-taskSingleton: TaskSingleton
-tasks: List<Task>

-TaskSingleton()
+getInstance(taskSingleton: TaskSingleton)
+addTask(name: String, test: String, isCompleted: boolean, priority: TaskPriority)
+addTask(task: Task)
+getTasks(): List<Task>

Obrázek 7.8: Třída `TaskSingleton`

může jednat o cestu k souboru s EA modelem nebo nemusí být cesta vyplněna vůbec, načež se úkoly ve třídě `EATaskReaderWriter` zapíší do běžící instance EA. Třída `JsonTaskReaderWriter` nezapisuje úkoly přímo do JSON souboru s anonymizačním modelem, namísto toho vytváří pro úkoly vlastní JSON soubor. Správný název souboru se vždy derivuje od názvu souboru nesoucí anonymizační model a o jeho poskytování se opět stará třída `ModelLoadingUtil`, konkrétně její metoda `getTaskFilePathByFileType`.

K přidávání a uchovávání úkolů vzniknuvších během procesu objevování citlivých údajů v databázi slouží třída `TaskSingleton`, jež implementuje návrhový vzor jedináček (anglicky *singleton*). Její struktura je zachycena na diagramu 7.8.

Metody pro přidávání úkolů vychází z původních implementací rozhraní `IWMetaSingleton` (konkrétně `DefaultWMetaSingleton`). Po dokončení procesu objevování citlivých údajů se úkoly vloží do kolekce `tasks` odkud k nim lze odkudkoliv přistoupit skrze metodu `getTasks`.

Třída `TaskSingleton` se využívá v abstraktním předkovi určeném ke správě procesu objevování `AbstractDiscoverTableManager`. V něm dochází ke vkládání úkolů do jedináčka. Následně je využit v existujícím exekučním módu `WExecutionModeDiscovery`, jenž byl rozšířen o inicializační fázi a načítání úkolů uložených do jedináčka po ukončení procesu objevování.

V rámci inicializační fáze se opět využívá třída `ModelLoadingUtil` k získání správného potomka abstraktního předka `TaskReaderWriter` a jeho uložení. Po ukončení objevování citlivých údajů v anonymizačním modelu se implementace třídy `TaskReaderWriter` (získané z inicializační fáze) využije k zápisu úkolů získaných ze třídy `TaskSingleton`.

Kapitola 8

Ověření řešení

Kapitola čtenáři představuje postup, pomocí něhož se vyhodnocovalo naplnění požadavků na nové vývojové prostředí, zavedení rozšířené metodiky testování a implementaci nové datové vrstvy nástroje Winch.

8.1 Vývojové prostředí

Nové vývojové prostředí bylo představeno v sekci Návrh a příprava nového vývojového prostředí. Za účelem jeho realizace byla vybrána multiplatformní kontejnerová platforma Docker spolu s jeho nástrojem *Docker Compose*, jenž umožňuje zprovoznit a inicializovat všechny relační databáze na lokálním nebo serverovém prostředí pomocí jediného příkazu.

Jednotlivé databázové stroje v podobě Docker obrazů jsou verzovány přímo v odpovídajících databázových modulech. Pro některé relační databáze musela být (vzhledem k absenci požadovaných verzí ve veřejných registrech Docker obrazů) zvolena jiná hlavní verze než je uvedeno v seznamu aplikací podporovaných databázových strojů.

Implementované řešení nového vývojového prostředí tak naplnilo první dva požadavky uvedené v podkapitole Přepracované vývojové prostředí. Třetí požadavek na pokrytí všech podporovaných databází se shodnými hlavními verzemi byl pro některé databáze splněn pouze částečně. Pokud by rozdíly mezi verzemi způsobovaly potíže, jež by nebylo možné replikovat na lokálním prostředí, lze pro danou databázi zvolit jinou metodu tvorby vývojového prostředí, jež byly popsány v teoretické části práce.

8.2 Metodika testování

V podkapitole Návrh testovací metodiky byl vytvořen návrh obsahující nová pravidla a doporučení pro testování anonymizačního nástroje Winch.

Samotný návrh pokrývá všechny požadavky definované v sekci Rozšířená testovací metodika a některé části nové metodiky byly aplikovány na implementaci (například užití parametrizovaných jednotkových testů zaznamenaných na listingu 8.1).

```
@RunWith(Parameterized. class)
 class ModelLoaderTaskFileTest {

    @Parameters(name = '{index}:
       ModelLoader.getTaskFilePathByFileType("{0}") = "{1}"')
     static Collection<Object[]> data() {
       Arrays.asList(
              [
                     ["C:\\Users\\JohnDoe\\model.eap",
                        "C:\\Users\\JohnDoe\\model.eap"],
                     ["/home/JohnDoe/model.eap",
                        "/home/JohnDoe/model.eap"],
                     ["..\\JohnDoe\\model.eap",
                        "..\\JohnDoe\\model.eap"],
                     ["model.eap", "model.eap"],

                     ["C:\\Users\\JohnDoe\\model.eapx",
                        "C:\\Users\\JohnDoe\\model.eapx"],
                     ["/home/JohnDoe/model.eapx",
                        "/home/JohnDoe/model.eapx"],
                     ["../JohnDoe/model.eapx",
                        "../JohnDoe/model.eapx"],
                     ["model.eapx", "model.eapx"],

                     ["C:\\Users\\JohnDoe\\model.json",
                        "C:\\Users\\JohnDoe\\tasks_model.json"],
                     ["/home/JohnDoe/model.json",
                        "/home/JohnDoe/tasks_model.json"],

                     ["..\\JohnDoe\\model.json",
                        "..\\JohnDoe\\tasks_model.json"],
                     ["../JohnDoe/model.json",
                        "../JohnDoe/tasks_model.json"],

                     [".\\model.json", ".\\tasks_model.json"],
                     ["./model.json", "./tasks_model.json"],

                     ["model.json", "tasks_model.json"],
              ] as Object[][]
```

```
        ) as Collection<Object[]>
    }

    @Parameter
     public String inputFilePath

    @Parameter(1)
     public String expectedTaskFilePath

    @Test
     void testGetTaskFilePathByFileType() {
        assert expectedTaskFilePath ==
            ModelLoadingUtil.getTaskFilePathByFileType(inputFilePath)
    }
}
```

Listing 8.1: Parametrizovaný test s využitím JUnit

Nasazení nově navržené metodiky vyžaduje kompletní přepis testovacího kód nástroje Winch, a protože implementace nové datové vrstvy představovala přepis většiny produkčního kódu, nebyla testovací metodika implementována v plném rozsahu.

8.3 Datová vrstva nástroje Winch

Následující podsekce představují ověření funkčnosti nově implementované datové vrstvy.

8.3.1 Zpětná kompatibilita s nástrojem EA

Ověření zpětné kompatibility původní a nové datové vrstvy probíhalo na základě porovnávání generovaných SQL příkazů pro cílové databáze. SQL soubory pro anonymizaci vytvořené spuštěním procesu generování nad původní datovou vrstvou (EA repozitářem) se musí přesně shodovat s SQL soubory vygenerovanými nástrojem Winch nad totožným anonymizačním modelem založeném na nové datové vrstvě (JSON soubor). K získání identického anonymizačního modelu v novém datovém formátu byl využit nový proces konverze mezi datovými úložišti popsaném v podkapitole Přenos jednoho druhu úložiště na jiný.

Testování shody vygenerovaných SQL souborů proběhlo v poloautomatickém režimu s využitím skriptu vyobrazeném na listingu 8.2. Skript ve své podstatě představuje automatizovaný scénářový test s následujícími kroky:

1. vygenerování souborů pro anonymizaci z modelu uloženého v běžící instanci EA,

2. zkopírování výsledných SQL souborů do testovací složky (`ea`),
3. spuštění konverze EA úložiště na formát JSON, spuštění procesu generování,
4. vygenerování souborů pro anonymizaci z modelu uloženého v JSON souboru,
5. zkopírování výsledných SQL souborů do testovací složky (`json`),
6. porovnání obsahů souborů v obou složkách.

```
@ echo  off

 set DB=mssql
 set APP_GUID={3E31E3FD-8152-4f39-AF14-4EDCAEAD271C}
 set CONTEXT_GUID={1C19FCFF-AF64-4961-A3B9-0BC2F37FA503}

 echo Running gendiff test
   for %DB%, APP_GUID=%APP_GUID%, CONTEXT_GUID=%CONTEXT_GUID%

@REM generate SQL files from running EA instance
 call .\disl-winch-%DB%.bat -g %APP_GUID% %CONTEXT_GUID%

@REM copy EA SQL files to current folder
 md ea
 xcopy /s /e /y %homedrive%%homepath%\GEMWinch\output\ .\ea\

@REM convert EA repository to JSON
 call .\disl-winch-%DB%.bat -c %CD%\WinchTest.eap
   %CD%\WinchTest.json

@REM generate SQL files from created JSON
 call .\disl-winch-%DB%.bat -g %APP_GUID% %CONTEXT_GUID%
   %CD%\WinchTest.json

@REM copy JSON SQL files to current folder
 md json
 xcopy /s /e /y %homedrive%%homepath%\GEMWinch\output\ .\json\

 echo Comparing SQL files generated from original EA with SQL
   files generated from converted JSON model
 fc ea\* json\*
 if  errorlevel 1  goto failed
```

```
 echo Passed
 goto clean

:failed
 echo Test failed, check the output above and generated SQL
   files
 exit 1

:clean
@REM clean after success
 del /F /Q ea
 del /F /Q json
```

Listing 8.2: Skript ověřující zpětnou kompatibilitu

Vyhodnocení testu zpětné kompatibility proběhlo úspěšně – anonymizační modely persistované v JSON souborech generují stejný anonymizační kód a konverze do nového formátu funguje bez ztráty informací důležitých pro proces anonymizace, čímž byly naplněny i zbývající požadavky na novou datovou vrstvu nástroje Winch – Zpětná kompatibilita (DATA-4) a Konverze formátů (DATA-5).

8.3.2 Regresní ověření

Po zdárném provedení testu zpětné kompatibility bylo cílem nahrazení původního datového úložiště anonymizačních modelů novým v celém testovacím kódu nástroje Winch.

Průchod všemi existujícími testy, ve kterých byla původní datová vrstva v podobě repozitáře v nástroji Enterprise Architect nahrazena novým úložištěm textovým úložištěm ve formátu JSON, zajistil, že v aplikaci Winch nedošlo k regresi, a tedy že lze nový formát a strukturu pro anonymizační model využít jako persistenci.

Záměna formátů v testovacím kódu také odebrala nutnost spouštět testovací sady na operačním systému Microsoft Windows s nainstalovaným nástrojem Enterprise Architect.

8.4 Shrnutí ověřovací fáze

V rámci ověření došlo k vyhodnocení nově navrženého a nasazeného vývojového prostředí pro nástroj Winch, jenž kromě přesného dodržení hlavních verzí podporovaných databázových strojů, dopadlo úspěšně. Následně byla navržena a vyhodnocena nová testovací metodika. Samotný návrh vyhovuje všem definovaným požadavkům, ale nedošlo k jeho implementaci v plném roz-

sahu aplikace Winch. Nakonec byla ověřena nová implementace datové vrstvy nástroje Winch, jež byla úspěšně otestována a vyhověla veškerým nastaveným požadavkům.

Závěr

Diplomová práce čtenáře seznámila s anonymizačním nástrojem Winch a jeho stávajícím stavem z pohledu vývojového prostředí, postupů testování a datovým modelem včetně typu úložiště.

Ze zadání, průzkumu nástroje a analýzy předcházející bakalářské práce věnující se implementaci datové vrstvy pro aplikaci Winch vzešly konkrétní požadavky na výsledné řešení.

K jejich naplnění bylo provedeno několik rešerší. Jmenovitě se práce zabývala možnostmi tvorby jednotných vývojových prostředí, jež by usnadnily budoucí rozvoje nástroje Winch, následně byla zkoumána problematika testování softwaru od základních principů přes kategorizaci jednotlivých druhů testů až po způsoby automatizace exekuce testů a rozdíly proti jejich manuálnímu provádění. Poslední teoretická část práce zkoumala vybrané formáty datových úložišť vhodných pro desktopové aplikace.

První fáze implementačního návrhu se zabývala zkoumáním existující bakalářské práce, jež měla za cíl implementovat novou datovou vrstvu pro nástroj Winch. Tento cíl byl převzat a začleněn do diplomové práce, původní myšlenky tvorby nového datového modelu ale nebyly v diplomové práci uvažovány z důvodu jejich vysoké komplexity a obavy, zda by při jejich dodržení opravdu došlo k platformové nezávislosti. Následovala tedy příprava a implementace vlastního řešení. Závislost aplikace Winch na nástroji Enterprise Architect byla zakořeněna v celém datovém modelu, implementace nezávislosti a přepracování datové vrstvy tak představovalo rozsáhlý refaktoring většiny produkčního kódu. Celé řešení si následně žádalo podrobné ověření zpětné kompatibility a kontroly, že v programu nedošlo k regresi. Výsledkem je zcela přepracovaný datový model, jenž nyní podporuje různé druhy úložišť, a aplikace Winch tak již není závislá na operačním systému Microsoft Windows. Nová architektura a její popis v této práci by navíc měla umožňovat jednoduché rozšiřování o další typy úložišť pro anonymizační modely.

Vzhledem k rozsáhlým úpravám a testování finální implementace nedošlo ke kompletnímu nasazení nově navržené testovací metodiky. Nová pravidla a

doporučení pro tvorbu testů vyžadují kompletní refaktoring testovacího kódu aplikace Winch a tento refaktoring by taktéž následně vyžadoval své vlastní ověření. Z těchto důvodů tak byly implementovány pouze příklady vycházející z nové metodiky testování a celé její nasazení podle vytvořené specifikace tak zůstává k implementaci v rámci jiných prací.

Anonymizační nástroj Winch byl v rámci diplomové práce navíc rozšířen o nové jednotné vývojové prostředí, jenž se verzuje přímo s kódem a je tak dostupné každému vývojáři, kteří si nyní nemusí předávat vývojové prostředí osobně.

Bibliografie

1. SCHUH, Matěj. *Implementace datové vrstvy pro anonymizační nástroj.* Praha, 2018. Bachelor's Thesis. České vysoké učení technické v Praze. Fakulta informačních technologií.
2. *What is a Virtual Machine? | VMware Glossary* [online]. 2022 [cit. 2022-02-03]. Dostupné z: https://www.vmware.com/topics/glossary/content/virtual-machine.html.
3. *What is a Hypervisor | VMware Glossary* [online]. 2022 [cit. 2022-02-03]. Dostupné z: https://www.vmware.com/topics/glossary/content/hypervisor.html.
4. *Oracle® VM VirtualBox®* [online]. 2022 [cit. 2022-02-03]. Dostupné z: https://www.virtualbox.org/manual/UserManual.html.
5. *VMware Workstation Player | VMware* [online]. 2022 [cit. 2022-02-03]. Dostupné z: https://www.vmware.com/products/workstation-player.html.
6. *What is a Container?* [Online]. 2022-01-16 [cit. 2022-01-16]. Dostupné z: https://www.docker.com/resources/what-container.
7. *Docker overview* [online]. 2021 [cit. 2022-01-17]. Dostupné z: https://docs.docker.com/get-started/overview.
8. *Install Docker Desktop on Windows* [online]. 2021 [cit. 2022-01-17]. Dostupné z: https://docs.docker.com/desktop/windows/install.
9. *Install Docker Desktop on Mac* [online]. 2021 [cit. 2022-01-17]. Dostupné z: https://docs.docker.com/desktop/mac/install.
10. *Overview of Docker Compose* [online]. 2021 [cit. 2022-01-17]. Dostupné z: https://docs.docker.com/compose.
11. *What is Podman? – Podman documentation* [online]. 2019 [cit. 2022-01-19]. Dostupné z: https://docs.podman.io/en/latest/index.html.

12. *Rootless containers with Podman: The basics* [online]. 2020-09-23 [cit. 2022-01-19]. Dostupné z: https://developers.redhat.com/blog/2020/09/25/rootless-containers-with-podman-the-basics.
13. *Podman: Managing pods and containers in a local container runtime* [online]. 2019-01-15 [cit. 2022-01-19]. Dostupné z: https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods.
14. *podman/mac_win_client.md at main ·containers/podman·GitHub* [online]. 2021-12-14 [cit. 2022-01-19]. Dostupné z: https://github.com/containers/podman/blob/main/docs/tutorials/mac_win_client.md.
15. HASHIMOTO, Mitchell. *Vagrant: Up and Running*. 1st. Sebastopol, California, USA: O'Reilly Media, Inc., 2013. ISBN 978-1-449-33583-0.
16. MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. *The Art of Software Testing*. 3rd. Hoboken, New Jersey, USA: Wiley, 2011. ISBN 978-1-118-13315-6.
17. JORGENSEN, Paul C. *Software Testing: a Craftsman's Approach*. 4th. Boca Raton, Florida, USA: CRC Press, 2013. ISBN 978-1-4665-6069-7.
18. KHORIKOV, Vladimir. *Unit Testing: Principles, Practices, and Patterns*. Shelter Island, New York, USA: Manning Publications, 2020. ISBN 9781617296277.
19. AXELROD, Arnon. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. New York, USA: Apress Media, 2018. ISBN 978-1-4842-3832-5.
20. RAY, Erik T. *Learning XML*. 2nd. Sebastopol, California, USA: O'Reilly Media, Inc., 2003. ISBN 978-0-596-00420-0.
21. SMITH, Ben. *Beginning JSON*. Berkely, California, USA: Apress Media, LLC, 2015. ISBN 978-1-4842-0202-9.
22. *Understanding JSON Schema* [online]. 2022-02-02 [cit. 2022-02-06]. Dostupné z: https://json-schema.org/understanding-json-schema/UnderstandingJSONSchema.pdf.
23. KREIBICH, Jay A. *Using SQLite*. Sebastopol, California, USA: O'Reilly Media, Inc., 2010. ISBN 978-0-596-52118-9.

Příloha **A**

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
├── readme_oponnent.txt .............. informace pro oponenta práce
├── src
│   ├── impl ......................... zdrojové kódy implementace
│   └── thesis ....................... zdrojová forma práce ve formátu LaTeX
└── text
    └── thesis_perner_2022.pdf ....... text práce ve formátu PDF
```