MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Automatické testování webových aplikací společnosti Unicorn

DIPLOMOVÁ PRÁCE

Jakub Szotkowski

Brno, podzim 2016

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: RNDr. Jaroslav Bayer

Poděkování

Na tomto místě bych chtěl poděkovat RNDr. Jaroslavu Bayerovi za jeho vedení a rady při tvorbě této diplomové práce. Dále bych chtěl poděkovat testerskému týmu společnosti Unicorn za pomoc při testování vyvíjené aplikace. Jmenovitě bych chtěl poděkovat Martinu Hefkovi, který mi s testováním pomohl nejvíce a přinášel mi přínosné rady k aplikaci i automatizovaným testům. S pomocí při testování aplikace v jiné společnosti než Unicorn mi pomáhal Milan Skipala, kterému bych chtěl tímto také poděkovat. Na závěr bych chtěl poděkovat celé mé rodině, která mne podporovala v průběhu celého studia.

Shrnutí

Cílem práce bylo vytvořit aplikaci, která umožní uživateli zapsat, vygenerovat a spustit automatizovaný test bez znalosti programovacího jazyka nebo testovacího frameworku. Aplikace je rozdělena na samostatné moduly, které spolu vzájemně kooperují. Testy jsou zapisovány do souboru XML a poté automaticky konvertovány do programovacího jazyka Java s podporou testovacího nástroje Selenium. Účelem aplikace je především umožnit testování projektů ve společnosti Unicorn. Při její realizaci byl kladen důraz na obecnost řešení i vlastního použití.

Klíčová slova

Selenium, testování, XML, automatizace, sql, Java, Windows, Linux, driver, Mozilla Firefox, Internet Explorer, Google Chrome, MySQL

Obsah

1 **Úvod** . . . 1
2 **Úvod do problematiky společnosti** . . . 3
2.1 *O společnosti Unicorn* . . . 3
2.1.1 Unicorn Systems . . . 3
2.1.2 Unicorn Universe . . . 3
2.1.3 Unicorn College . . . 4
2.1.4 Brněnská pobočka . . . 4
2.2 *Motivace k vytvoření programu* . . . 4
2.3 *Požadavky na aplikaci* . . . 4
3 **Testování softwaru** . . . 6
3.1 *Axiomy testování* . . . 7
3.1.1 Žádný program nelze otestovat kompletně . . . 7
3.1.2 Akceptace rizika . . . 8
3.1.3 Testování neprokáže neexistenci chyb . . . 8
3.1.4 Chyby se vyskytují ve skupinách . . . 8
3.1.5 Paradox pesticidů . . . 8
3.1.6 Všechny nalezené chyby se neopraví . . . 9
3.1.7 Těžko říci, zda je chyba chybou . . . 9
3.1.8 Specifikace softwaru nebývá konečná . . . 10
3.1.9 Testeři nejsou nejoblíbenějšími členy týmu . . . 10
3.1.10 Testování je technická disciplína . . . 10
3.2 *Úrovně testování* . . . 10
3.2.1 Testování programátorem . . . 11
3.2.2 Jednotkové testování . . . 11
3.2.3 Integrační testování . . . 11
3.2.4 Systémové testování . . . 12
3.2.5 Akceptační testování . . . 12
3.3 *Způsoby testování* . . . 12
3.3.1 Testování černé a bílé skříňky . . . 13
3.3.2 Statické a dynamické testy . . . 13
3.3.3 Manuální a automatické testy . . . 13
3.4 *Typy testů* . . . 13
3.4.1 Funkční a nefunkční testy . . . 14
3.4.2 Smoke testy . . . 14
3.4.3 Progresní a regresní testy . . . 14
3.4.4 Testy splněním a selháním . . . 15
4 **Automatizované testování** . . . 16
4.1 *Význam automatizovaného testování* . . . 16
4.2 *Realizace automatizace testů* . . . 16
4.3 *Výhody a nevýhody automatizování* . . . 17

4.4 *Nástroje pro automatizované testování* . . . 17
4.4.1 Selenium . . . 18
4.4.2 Tellurium . . . 21
4.4.3 WebAii Framework . . . 22
4.4.4 WatiN . . . 23
4.4.5 TestComplete . . . 23
4.4.6 Ranorex . . . 24
4.4.7 Protractor . . . 24
5 **Analýza a návrh** . . . 25
5.1 *Analýza požadavků* . . . 25
5.1.1 Funkční požadavky . . . 25
5.1.2 Nefunkční požadavky . . . 27
5.2 *Návrh komponent aplikace* . . . 27
5.2.1 GUI . . . 27
5.2.2 Import testů . . . 28
5.2.3 Generátor testů . . . 28
5.2.4 Konfigurace . . . 28
5.2.5 Logování . . . 29
5.2.6 Spouštěč testů . . . 29
5.2.7 Prohlížeč . . . 29
5.2.8 Databáze . . . 30
5.3 *Adresářová struktura aplikace* . . . 30
6 **Použité technologie** . . . 34
6.1 *Podporované prohlížeče* . . . 34
6.1.1 Mozilla Firefox . . . 35
6.1.2 Google Chrome . . . 35
6.1.3 Internet Explorer . . . 35
6.2 *Podporované databáze* . . . 35
6.2.1 MySQL . . . 36
6.2.2 Apache Derby . . . 36
6.3 *Programovací jazyk a vývojové prostředí* . . . 36
6.4 *Apache Maven* . . . 36
6.5 *Git* . . . 37
6.6 *Selenium* . . . 37
6.7 *JAXB* . . . 37
6.8 *Swing* . . . 37
7 **Implementace** . . . 38
7.1 *Moduly aplikace* . . . 38
7.1.1 Core . . . 38
7.1.2 Converter . . . 38
7.1.3 Generator . . . 39
7.1.4 Tester . . . 39

7.1.5 Application . . . 40
7.2 *Detaily implementace a uživatelské rozhraní* . . . 40
7.2.1 Tvorba testů . . . 40
7.2.2 Logování aplikace a testů . . . 45
7.2.3 Obrazovky aplikace . . . 45
8 **Praktické použití aplikace** . . . 50
8.1 *Příklady testů* . . . 50
8.1.1 Test akce Action . . . 50
8.1.2 Test akce SqlAction . . . 50
8.1.3 Kombinovaný test akcí . . . 50
8.2 *Použití na projektech* . . . 51
9 **Závěr** . . . 52
Literatura . . . 54
Rejstřík . . . 55
A **Prerekvizity a návod na spuštění** . . . 56
A.1 *Spuštění aplikace* . . . 56
B **XSD schéma** . . . 57
C **Příklad testu s Action** . . . 61
D **Příklad testu s SqlAction** . . . 66
E **Příklad kombinovaného testu** . . . 69
F **Příklad vygenerovaného testu** . . . 73

Kapitola 1

Úvod

V současnosti testování není už pouze doplňkovou činností při vývoji softwaru, ale stalo se právoplatnou součástí vývojového procesu. Proto je nutné hledat stále nové způsoby, jak si testování všech částí aplikace usnadnit. Dobrým pomocníkem jsou automatizované testy, které zvládnou vykonat stejnou činnost opakovaně bez chyb způsobených lidským faktorem. Pokud je navíc automatizován celý proces spouštění automatizovaných testů, mohou být testy pouštěny i v době, kdy neomezují běžné manuální testování.

Testovat lze programový kód i rozhraní aplikace. Aplikaci lze otestovat pomocí mnoha druhů a typů testů, z nichž každý se využívá v jiné fázi vývojového cyklu a na testování jiných částí aplikace. Dále lze testy rozdělit podle toho, kdo má za úkol jejich provádění. Některé testy píše přímo programátor, přičemž se většinou jedná o testování programového kódu. Naopak testeři mají na starost testování grafického rozhraní aplikace a provádění složitějších druhů testů. Práci testerů lze nahradit automatizovanými testy, ovšem je nutné zvážit v jakém poměru, jelikož některé testy nelze zautomatizovat.

Tématem práce je vytvoření nástroje, který umožní psát automatizované testy bez znalosti testovacích frameworků či znalosti jakéhokoliv programovacího jazyka. Jediným předpokladem je schopnost zapsat test ve validním XML souboru, který je v dalších fázích převeden na automatizovaný test a může být spuštět pomocí samostatné aplikace.

Text práce je rozdělen do devíti kapitol, z nichž první kapitola je úvodní, která má za cíl uvést čtenáře do řešeného problému a stručně popsat ostatní kapitoly. V dalších kapitolách jsou uvedeny jednotlivé části potřebné pro vznik aplikace.

Ve druhé kapitole se čtenář dozví blíže o společnosti Unicorn a o jejím působení. Dále je uvedena motivace pro vytvoření nového řešení pro psaní automatizovaných testů. V neposlední řadě jsou uvedeny požadavky, které musí výsledná aplikace splňovat.

Další kapitola je zaměřena na obecné informace týkající se testování. V první části je uveden postup pro odhalování chyb a axiomy spojené s testováním. V druhé části kapitoly jsou pak rozebrány jednotlivé úrovně, způsoby a typy testování.

Celá čtvrtá kapitola je věnována automatizovanému testování. Nejprve se čtenář dočte o významu automatizovaných testů, jak je správně realizovat a porovnány jsou jejich výhody a nevýhody vůči manuálnímu testování. V druhé části jsou uvedeny nástroje pro automatizované testování. Hlavní význam je věnován nástrojům z rodiny Selenia, jehož použití je v prerekvizitách pro vzniklou aplikaci. Mimo to, jsou ale uvedeny i jiné nástroje, tak aby měl čtěnář přehled o dalších možných variantách a věděl o jejich přednostech a nedostatcích.

Pátá kapitola se věnuje analýze a návrhu aplikace. V analytické části jsou uvedeny funkční a nefunkční požadavky, jejichž grafické zobrazení je znázorněno pomocí diagramu případů užití. V návrhové části jsou pomocí diagramu komponent znázorněny komponenty systému, jejich struktura je dále pro každou komponentu rozebrána zvlášť včetně diagramu tříd, používaných v rámci komponenty. Na závěr je vykreslena adresářová struktura aplikace a význam všech souborů a složek.

V šesté kapitole se objeví popis použitých technologií. V úvodu se řeší podporované prohlížeče webových stránek (dále pouze prohlížeče) a databázové systémy, které jsou ve zbytku práce označovány zkráceně jako databáze. Dále v kapitole nalezneme například použitý programovací jazyk, vývojové prostředí či verzovací systém.

Následující kapitola je rozdělena do dvou částí. V první části může čtenář nalézt informace o modulech, do kterých je aplikace rozdělena. Druhá se pak věnuje jednotlivých implementačním celkům, kterými jsou například tvorba testů, možnosti konfigurace nebo logování.

Osmá kapitola se zabývá praktickým použitím aplikace. Čtenář zde má možnost nalézt popisy příkladů zapsaných testů v XML nebo informace týkající se možného použití na projektech společnosti Unicorn, ale i jiných firem.

Závěrečná kapitola shrnuje splněné cíle vzniklé aplikace a uvádí možnosti pro následující vývoj, který může aplikaci rozšířit o další užitečné funkce v nových verzích.

Přílohová část práce obsahuje prerekvizity a postup pro spuštění aplikace. Dále je k vidění XSD schéma, na základě kterého je tvořen XML dokument obsahující zapsaný test. Uvedeny jsou zde také konkrétní příklady zapsaných testů v XML a příklad vygenerovaného testu v Javě.

Kapitola 2

Úvod do problematiky společnosti

Tato kapitola se pokusí čtenáři detailněji přiblížit oblast působení společnosti Unicorn a nastínit základní principy důležitých procesů při vzniku jejich produktů. Blíže se poté zaměříme na brněnskou pobočku, jejíž zaměstnanci pomáhali s návrhem tématu této diplomové práce. Mimo to zde budou rozebrány důvody a motivace firmy k vytvoření vlastního komplexního řešení pro automatizované testy, které bude splňovat jejich požadavky. Tato část nás dovede k vymezení zadání projektu pro praktickou část této diplomové práce.

2.1 O společnosti Unicorn

Společnost Unicorn je na trhu od roku 1990, kdy ji založil její majitel Vladimír Kovář. Jedná se o dynamickou skupinu společností, která nabízí rozsáhlé informační systémy a řešení v oblasti ICT[1]. Do této skupiny patří Unicorn Systems, Unicorn Universe a Unicorn College. Společnost patří mezi největší firmy v České republice a významně se angažuje i na evropské scéně. Unicorn může nabídnout svým zákazníkům zkušené zaměstnance se znalostmi z různých podnikatelských odvětví, mezi která patří bankovnictví, energetika či pojišťovnictví. [1]

2.1.1 Unicorn Systems

Unicorn Systems je jádrem společností Unicorn. Vykonává nejdůležitější část, která se týká vývoje informačních systémů a ICT řešení. Od svého vzniku vytvořila řadu špičkových a rozsáhlých řešení, pro nejrůznější oblasti. Mezi jedny z nejvýznamnějších projektů patří například ECP či Damas z oblasti energetiky, Sinis z pojišťovnictví nebo Finno z bankovnictví.

2.1.2 Unicorn Universe

Unicorn Universe je digitální stavebnice informačních systémů, která umožní vytvářet projekty založené na podobném přístupu jako např. sociální sítě. Dovoluje tedy vytvořit systém umožňující komunikaci, spolupráci a sdílení dat v rámci systému s podporou zabezpečeného přístupu. Tato řešení jsou poskytována prostřednictvím portálu Plus4U.

1. informační a komunikační technologie

2.1.3 Unicorn College

Jedná se o soukromou vysokou školu, která nabízí kvalitní vzdělání v oblasti IT, ekonomie a managementu. Soustředí se na vzdělávání odporníků v jednotlivých oborech, na které se společnost soustředí, tak aby maximalizovali konkurenční výhodu v jejich budoucím uplatnění. Dále Unicorn College zajišťuje školení svých zaměstnanců napříč všemi pobočkami, tak aby upevnili a posílili své znalosti pro uplatnění na projektech.

2.1.4 Brněnská pobočka

Brněnská pobočka se nachází nedaleko Modřic a aktuálně v ní pracuje přibližně 70 zaměstnanců, přičemž část z nich je helpdesk a náborový management, zbytek se poté věnuje projektům. Na pobočce jsou aktuálně vytvářeny projekty spojené s energetikou, ať už se jedná o hlavní projekt AMICA, tak o vedlejší CLB, CapCal, CGMES či již odevzdaný AIB HUB. Většinu z nich spojuje to, že se nejedná o jednorázové projekty, ale o dlouhodobé. Všechny programy jsou napsané v programovacím jazyce Java a mají webové rozhraní.

2.2 Motivace k vytvoření programu

Jak už bylo zmíněno v části 2.1.4, projekty, na kterých se ve firmě pracuje, jsou především dlouhodobé a je tedy nutné testovat stále stejné části, na které existují téměř neměnné scénáře. Testování těchto částí ručně je zdlouhavé a obírá testery o čas, který by mohli věnovat testování nových částí systému. Pro tento problém se nachází jednoduché řešení a tím jsou automatizované testy.

Bohužel to vede k dalšímu problému a tím je, kdo bude tyto testy psát. Tento člověk musí mít znalosti jak z programování, tak z testovacího frameworku, což obyčejně testeři nemají. Nabízí se hned několik řešení jak problém vyřešit, např. poslání testera na školení nebo najmutí nového testera se znalostmi pro automatizované testování. Většina řešení ale zhavaruje na finanční náročnosti.

Nakonec jsme ale zdárně dospěli k možnému řešení a to, vytvořit program, který umožní vygenerovat test a automatizovaně jej pustit bez jakékoliv znalosti programovacího jazyka či testovacího frameworku. Přičemž podmět pro vygenerování testu dokáže vytvořit kdokoliv s elementární znalostí strukturovaného zápisu.

2.3 Požadavky na aplikaci

Z předchozího odstavce známe nejdůležitější požadavek, který se týká strukturovaného zápisu testu. Test bude zapisován ve značkovacím jazyce XML[2], dále je nutné vytvořit XSD schéma pro validaci XML dokumentu. Mezi dalšími požadavky klienta bylo použití programovacího jazyka Java a testovacího frameworku Selenium, a to z důvodů jejich masivního

2. eXtensible Markup Language

používání na projektech. Vygenerované testy bude možné spouštět přimo z aplikace. Uživateli bude umožněno v konfiguraci nastavit prohlížeč, ve kterém se má test spouštět, přičemž podporovány by měly být alespoň nejvíce používané (Mozilla Firefox, Internet Explorer a Google Chrome).

Dalším rozšířením aplikace je podpora databáze. Uživateli bude umožněno nastavit v konfiguraci připojení k databázi a využívat rozšíření XML o nové značky, které umožní základní operace s databází. Tímto bude uživatel schopen porovnávat či ukládat data z/do databáze a webového rozhraní.

Nezbytnou součástí je ukládání výsledků testů do logovacích souborů, přičemž se bude ukládát každý jednotlivý průchod a každý krok prováděného testu. V případě, že krok bude obsahovat nějakou operaci, která má za následek uložení výsledku do souboru, bude i tento soubor uživateli přístupný po průchodu testu.

Kapitola 3

Testování softwaru

Testování je nedílnou součástí vývoje každého softwaru. Během testování jsou v aplikaci odhaleny chyby, které je nutné opravit před vydáním produktu. Druhů a typů testů je mnoho a každý se hodí na testování odlišných funkčností a v jiných fázích projektu. V této kapitole budou postupně stručně rozebrány a u každého bude řečeno k čemu se nejvíce hodí.

Chyby během testování je vhodné nalézt co nejdříve, tak aby jejich oprava nezapříčinila vznik dalších chyb. Navíc čím později je chyba odhalena, tím dražší se pro vývoj softwaru stává její oprava. Chyby jsou v softwaru hledány na základě specifikace, tak jak to definoval již Ron Patton[1]. Jeho definice obsahuje 5 základních podmínek, na jejichž základě lze rozpoznat, že se jedná o softwarovou chybu: [2]

„O softwarovou chybu se jedná, pokud:

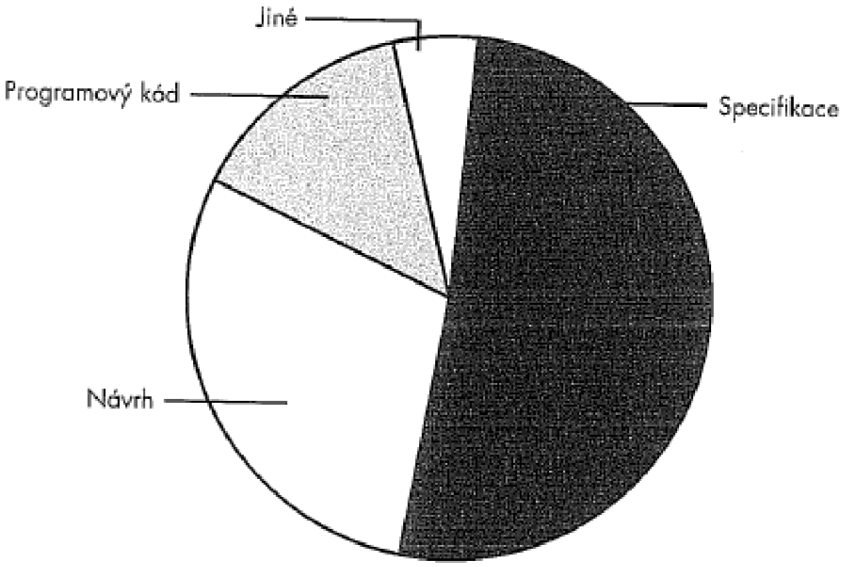
1. *Software nedělá něco, co by podle specifikace produktu dělat měl.*
2. *Software dělá něco, co by podle specifikace produktu dělat neměl.*
3. *Software dělá něco, o čem se produktová specifikace nezmiňuje.*
4. *Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.*
5. *Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo - podle názoru testera softwaru - jej koncový uživatel nebude považovat za správný.*

Splněna musí být vždy alespoň jedna, ale i více podmínek.“

Na základě těchto pravidel by se dalo odhadnout, že většina chyb vzniká při programování, kdy vývojář není schopen naprogramovat aplikaci dle specifikace, opak je ale pravdou. Bylo provedeno mnoho výzkumů nad různě velkými projekty (ať už velmi malé tak i obrovské) a výsledek byl vždy stejný, a to, že za většinu softwarových chyb může specifikace.

Důvodů, proč tomu tak je, může být více. V některých případech není specifikace produktu vůbec napsána, v dalších případech může být příliš abstraktní nebo se neustále mění. Často se také stává, že ji neproberou dostatečně všichni členové týmu, kterých se psaní specifikace týká. Na následujícím obrázku je grafově zobrazeno, kolik chyb v průměru vzniká na základě jednotlivých částí vývoje, viz 3.1.

1. profesor na University of Hull



Obrázek 3.1: Příčiny vzniku chyb (strana 15 [2])

3.1 Axiomy testování

Během testování libovolného softwaru se opakují stále stejné základní pravidla, které můžeme nazvat axiomy či obecné pravdy. Každé z těchto pravidel nám pomůže zlepšít celkový pohled na proces vývoje.

3.1.1 Žádný program nelze otestovat kompletně

Aplikaci nikdy nelze otestovat kompletně a to na základě nasledujících čtyř pravidel:

- počet vstupů je příliš velký,
- počet výstupů je příliš velký,
- počet průchodů programu je příliš velký,
- specifikace softwaru je subjektivní, to co jeden vidí jako správné chování, druhý může vidět jako chybu.

Právě díky poslednímu pravidlu je tato obecná pravda vždycky splněna, což můžeme ukázat na příkladu. Mějme aplikaci, která má za úkol uživatele pouze pozdravit. Ačkoliv nemá program vstupy a má pouze jeden průchod a jeden výstup, nelze ji otestovat kompletně. V případě, že výstup programu bude „Dobrý den“, tak anglicky komunikující uživatel může tohle chování reportovat jako chybu a tedy označit aplikaci za nedostatečně otestovanou.

3.1.2 Akceptace rizika

Aplikace nelze nikdy otestovat kompletně, proto je nutné příjmout rizika za některé části, které nemusí pracovat správně. V takovém případě je nutné, aby tester softwaru rozdělil množinu testovacích případů do rozumných podmnožin, které je schopen v rozumném čase otestovat. Podmnožiny by v sobě měly shromažďovat podobné případy užití, např. u kalkulačky by jednotlivé podmnožiny mohly být sčítání, odčítání, násobení a dělení. V každé části se otestují krajní případy užití a v případě kladných výsledků se označí část za splněnou a předpokládá se, že všechno pracuje správně.

3.1.3 Testování neprokáže neexistenci chyb

Na tento problém poukázal kromě Rona Pattona i Edsger W. Dijkstra: [3] *„Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.“* (v překladu: *„Testování programu může být velmi efektivní cestou, jak ukázat přítomnost chyb, ale je beznadějně nevhodné k prokázání jejich nepřítomnosti.“*)

V případě, že při testování nenalezneme žádné chyby, neznamená to, že v systému žádné chyby nejsou. Pouze to prokázalo skutečnost, že v našem konkrétním průchodu žádné chyby nenastaly. Takovýchto testů můžeme provést libovolné množství, ale nikdy nemůžeme zaručit, že se v systému už žádné další chyby nenalézají.

3.1.4 Chyby se vyskytují ve skupinách

Často se stává, že objeví-li tester nějakou chybu, záhy najde další, která se týká podobné funkcionality. Většinou je tohle chování způsobeno těmito příčinami:

- vývojáři mají špatné dny – V některých dnech jsou programátoři při psaní kódu téměř bezchybní. Ovšem někdy se do jejich kódu vkrádají chyby, které mohou být způsobeny nekoncentrovaností nebo nedostatečnou znalostí programované části. Proto jedna nalezená chyba může vést k dalším.
- vývojáři dělají podobné chyby – Každý vývojář má svou zaběhlou programovací techniku a v případě, že je náchylný na určitý druh chyb, bude se nacházet ve více částech aplikace.
- některé chyby jsou jen špičkou ledovce – Často se stává, že je v jádru softwaru závažný problém. Tester najde několik méně závažných chyb, které spolu zdánlivě nesouvisí, ale nakonec se zjistí, že mají společnou příčinu a navíc daleko závažnější.

3.1.5 Paradox pesticidů

S tímto termínem přišel již v roce 1990 Boris Beizer[2], který jej definoval pomocí účinku pesticidů na hmyz: [4] *„If you keep applying the same pesticide, the insects eventually build up*

2. americký softwarový inženýr

resistance and the pesticide no longer works.“ (v překladu: *„Pokud se budeme držet použití stejného pesticidu, hmyz si nakonec vytvoří odolnost a pesticid již nebude dále fungovat.“*)

Pokud tento princip aplikujeme na testování softwaru, zjistíme, že pokud budou prováděny stále stejné testy dokola, po určité době se testovaná část stane stabilní a imunní vůči prováděným testům. Je tedy nutné vymýšlet stále nové a jiné testy, tak aby se otestovaly i jiné připady užití a byla možnost nalézt nové chyby.

3.1.6 Všechny nalezené chyby se neopraví

Na rozsáhlejších projektech je téměř nemožné opravit všechny chyby, často je tedy nutné rozhodnout, které chyby jsou důležitější a je potřeba je primárně opravit. Důvodů, proč se nestihnou opravit všechny chyby, může být několik:

- nedostatek času – Na většině projektů se mnohdy dostaneme do situace, kdy se blíží termín odevzdání a projekt není ve fázi, ve které by měl být. Důvodem může být nedostatek programátorů nebo testerů, příliš mnoho funkčností či špatné odhady při návrhu a analýze aplikace.
- nejedná se o chybu – Oblíbenou hláškou programátorů je: *„It´s not a bug, it´s a feature.“* (v překladu: *„To není chyba, to je vlastnost.“*). V praxi to znamená, že dostanou k opravě chybu, která ve skutečnosti chybou není a tester pouze špatně pochopil, jak se má aplikace v určité chvíli zachovat. Někdy také může jít o změnu ve specifikaci, kdy se domnělá chyba změní ve funkcionalitu aplikace.
- příliš riskantní chyba – Kód softwaru je velice složitá věc a mnohdy se mohou při opravě rizikové chyby zavléci do aplikace další problémy. V případě, že je firma pod tlakem termínu vydání, je lepší možností tuto chybu neopravovat, prohlásit ji za známou a zákazníka o tomto problému informovat. Tím se zabrání vytvoření dalších potenciálních chyb, o kterých bychom nevěděli.
- oprava nestojí za to – Ačkoliv to vypadá jako nezodpovědnost, opravy některých chyb v praxi nejsou vždy nutností. Příkladem mohou být funkčnosti, které se používají minimálně. Dalším příkladem, kdy není nutně potřeba opravovat chyby, je použití alternativního způsobu průchodu aplikace, při kterém se reportovaná chyba neprojeví. Často rozhodnutí závisí na přijmutí rizika.

Do procesu rozhodování bývá zpravidla zapojován celý tým, tedy testeři, programátoři i projektoví manažeři. Každý z nich má odlišný názor na to, co se má a nemá opravit, a na základě konzultace se rozhodne, co nakonec opravit.

3.1.7 Těžko říci, zda je chyba chybou

Pokud vývojář, tester, dokonce ani zákazník chybu neobjeví, jedná se skutečně o chybu? Na tuto otázku může mít každý jinou odpověď, avšak pokud se budeme držet základních

podmínek, které definoval Patton, dospějeme k rozhodnutí, že se o chybu nejedná, jelikož ji nikdo nezpozoroval. Nikdo totiž nemůže popsat něco, co neviděl.

Na celý problém můžeme pohlížet z jiného úhlu. Jestliže systém používají dva uživatelé a jeden z nich nachází spoustu chyb, kdežto druhý téměř žádné, mohou mít na kvalitu výsledného softwaru zcela odlišný názor. Přitom se jen může lišit způsob jejich používání aplikace. První z nich poté označí aplikaci za plnou chyb a nepoužitelnou, naopak druhý ji označí za kvalitní a téměř bezchybnou.

3.1.8 Specifikace softwaru nebývá konečná

Problémem softwarového vývoje je zpravidla jeho délka, při které na trh přicházejí stále nové technologie a funkčnosti. Pokud bychom software vyvíjeli na základě stejné specifikace, která byla napsána na začátku vývoje, mohlo by se nám toto rozhodnutí vymstít. Produkt, který bychom odevzdali, by mohl být zastaralý a neschopný konkurence. Proto je důležité specifikaci revidovat a aktualizovat.

Z pohledu softwarového testera je nutné zakomponovat případné změny specifikace do testovacího procesu. Zpravidla to znamená, že průchod některých testovacích scénářů se omezí (dobře otestované části) a naopak se přidají nové testovací scénáře, týkající se nových funkcionalit.

3.1.9 Testeři nejsou nejoblíběnějšími členy týmu

Úlohou testera je kontrolovat a hledat chyby v programovém kódu svých kolegů. Je tedy pochopitelné, že není zrovna nejoblíbenějším členem vývojového týmu. Svůj obraz si mohou vylepšit včasným oznamováním chyb, které zajistí dřívější opravu systému a tím i dřívější stabilizaci aplikace. Ostatní členové pak nejsou vystavováni takovému tlaku z termínu odevzdání.

3.1.10 Testování je technická disciplína

Už dávno jsme za dobou, kdy se testování bralo pouze jako okrajová záležitost a aplikace byly testovány svými vývojáři. Dnešní systémy jsou příliš rozsáhlé, než aby je samotní vývojáři zvládali ještě testovat. Proto se testeři softwaru stali stabilními členy vývojových týmů a zastávají nepostradatelnou roli.

3.2 Úrovně testování

Základní úrovně testování lze definovat jako skupinu testů, pomocí kterých je aplikace testována v různých fázích vývojového cyklu. Dle vývojové fáze a časového horizontu, ve kterém se testování provádí, se dělí do následujích kategorií.

3.2.1 Testování programátorem

Po naprogramování části programu, je tento úsek ihned vyzkoušen programátorem, označováno jako „Assembly tests“ (v překladu: „montážní testy“). Ne vždy jsou tyto testy prováděny vývojářem, který kód napsal, ale člověkem s více zkušenostmi, např. seniorním vývojářem nebo softwarovým architektem. Tento proces je označován jako „code review“ (v překladu: „prozkoumání kódu“) a většinou se neprovádí po každém commitu[3], ale hromadně, např. na konci dne. V případě, že se při kontrole příjde na některé chyby a nedostatky, je vývojář upozorněn, aby si po sobě tyto chyby opravil.

Opravy těchto chyb jsou z pohledu vývoje nejlevnější a mohou zabránit zbytečnému prodražování projektu. V případě, že vývojář odevzdá po sobě neotestovaný kus kódu a ten vůbec nefunguje, tester nejprve musí nastudovat specifikaci, připravit testovací data a zahájit proces testování. Poté ovšem zjistí, že aplikace nefunguje a vrátí ji vývojáři. Ten chybu začne opravovat, ale tester zbytečně strávil čas přípravou na testování nefungující části místo aby se věnoval jiným částem aplikace.

3.2.2 Jednotkové testování

Po prověření kódu vývojářem přichází na řadu testy jednotek. U OOP[4] jsou těmito jednotkami samostatné třídy a metody. Testovanou jednotkou se rozumí samostatná nezávislá část vykonávaného programu, zpravidla to tedy znamená, že se snažíme každou metodu třídy otestovat zvlášť, tak aby nebyla ovlivněna okolním prostředím. K tomuto účelu se často využívá tzv. mockování [5], při kterém se získávání vnějších vstupů a výstupů nastaví na předem zvolenou hodnotu a ta je vždy pro test pevně vrácena. Tím je zajištěno, že se kontroluje jen funkčnost samotné metody.

Testy jednotek jsou zapisovány ve formě programového kódu, proto je píší zpravidla vývojáři. Dále jsou zodpovědni za jejich aktualizaci v případě, že se změní specifikace testované části. Pro tvorbu testů se využívají testovací frameworky, např. JUnit, NgUnit apod.

Je poměrně komplikované dopisovat jednotkové testy až po naprogramování aplikace. Mimo to ztrácí jejich dopisování význam, protože nám mají pomoci odhalit chybu co nejdříve. Kdežto po naprogramování aplikace už většinou proběhly jiné typy testů, které případné problémy měly odhalit. Pokud tedy chceme na projektu jednotkové testy používat, je vhodné se jimi zabývat již ve fázi návrhu. Často se taky používá termín „test-driven development“ (v překladu: „vývoj řízený testy“) [6], kdy jsou jednotkové testy napsány ještě před samotným zahájením vývoje aplikace. Po každém naprogramování celku je jeho funkčnost otestována těmito testy a pokud test neprojde, je chyba okamžitě odstraněna.

3.2.3 Integrační testování

V této fázi testování se poprvé dostává na scénu testovací tým. Musí být ověřena bezchybná komunikace mezi jednotlivými komponentami systému. Integraci lze testovat v rámci sys-

3. přidání nové verze do verzovacího systému
4. objektově orientované programování

tému, tedy komunikace mezi dvěma a více komponentami vyvíjené aplikace, nebo mezi komponentou a hardwarem či operačním systémem. Testuje se tedy integrace již dříve samostatně otestovaných komponent. Integrační testy mohou být manuální i automatické.

3.2.4 Systémové testování

Po ověření správné integrace komponent přichází na řadu systémové testování, které je stěžejní pro odevzdání aplikace zákazníkovi. Při tomto testování je aplikace prověřována jako celek a testy jsou proto prováděny v pozdějších fázích vývoje. Snaží se o testování aplikace z pohledu zákazníka, tedy jsou připraveny speciální testovací scénáře, které by měly obsahovat reálné průchody aplikace.

Zpravidla probíhá testování v několika kolech. Pokud je nalezena chyba, je v rámci tohoto kola opravena a znovu otestována v dalších kolech. Součástí systémového testování jsou funkční i nefukční testy. Slouží jako výstupní kontrola před odevzdáním zákazníkovi a je tedy neoddělitelnou součástí testovacího procesu, zkráceně řečeno: „Bez systémového testování bychom nemohli zaručit správnou funkčnost výsledného produktu.“

3.2.5 Akceptační testování

UAT[5] jsou akceptační testy na straně zákazníka. Do této fáze testování se dostaneme, pokud se během předchozích fází nevyskytnou závažné chyby, které by znemožnily předání aplikace do rukou klienta. Zákazník si, většinou se svým testerským týmem, provede akceptační testy, které jsou zpravidla prováděny podle testovacích scénářů, které jsou vytvořeny v kooperaci zákazníka s dodavatelem.

Testování aplikace probíhá na prostředích klienta, v případě nalezení chyb jsou reportovány zpět vývojovému týmu, který zajistí jejich opravu. Je důležité aby se obě strany dohodly, jak se budou reportované chyby a jejich opravy předávat. Většinou je dohodnut podobný princip jako při systémovém testování, a to v testovacích kolech, kdy jsou po určitý čas reportovány chyby a jejich opravy jsou poté naráz nasazeny v pravidelných časových intervalech na prostředí klienta (např. každý týden či 14 dnů).

3.3 Způsoby testování

Pro efektivní otestování projektu je nutné si na začátku stanovit správnou strategii testování, což zahrnuje rozhodnutí týkající se použitých druhů testů, v jakém časovém horizontu a míře budou použity či které nástroje a frameworky využijeme. Protože možných testů je velké množství, je důležité zvolit ty správné, tak aby otestování naší aplikace bylo co nejefektivnější. V případě špatné volby, může mít tohle rozhodnutí kritický dopad na termín odevzdání či cenu produktu.

5. User Acceptance Tests

3.3.1 Testování černé a bílé skříňky

Označení černé a bílé skříňky zde určuje, jakým způsobem má tester přístup ke zdrojovým kódům aplikace. Při použití testů černé skříňky nemá tester k dispozici žádné zdrojové kódy a systém kontroluje pouze z pohledu výstupních dat, které aplikace vrací. Díky tomu může velice dobře simulovat chování zákazníka, který rovněž nemá přístup k programovému kódu.

Naopak při testování bílé skříňky má tester výhodu zobrazení zdrojového kódu a díky tomu je schopen otestovat libovolný možný průchod aplikace. Díky známé struktuře softwaru je schopen otestovat nevalidní vstupy a provést různé další testy spojené s kontrolou kódu.

Kombinací mezi těmito dvěmi způsoby je šedá skříňka, kdy je například testerovi umožněn přístup do databáze. On je díky tomu schopen porovnat výstupy aplikace s výsledky v databázi, a tím i lépe reportovat případné nalezené chyby.

3.3.2 Statické a dynamické testy

Na základě toho, zda je nutné spustit pro testování aplikaci či nikoliv, můžeme testy rozdělit na statické a dynamické.

Statické testy nevyžadují běh software, jsou proto vhodné pro použití v ranných fázích vývojového cyklu, kdy ještě není vytvořena dostatečná část aplikace pro spuštění. Převážně se používá pro kontrolu specifikace požadavků před začátkem programování či analýzu zdrojových kódů.

Oproti tomu dynamické testy vyžadují spuštění a běh programu, jsou použity v pozdějších fázích vývoje a testují funkcionalitu aplikace.

3.3.3 Manuální a automatické testy

Testy lze rozdělit podle toho, zda jsou prováděny manuálně člověkem či automaticky pomocí softwaru. Přičemž je nutné rozhodnout, co je z pohledu testování výhodnější. Pokud je test nutné provádět často a opakovaně, je vhodné jej zautomatizovat. Důraz musí být ovšem kladen na fakt, že jeho zautomatizování a následná správa nesmí stát více prostředků než jeho manuální provádění. V takovém případě, je lepší test ponechat ve skupině manuálních. Dalším důvodem proč nechat test mezi manuálními, je nutnost zkontrolovat výsledky pomocí lidského úsudku.

3.4 Typy testů

Testy lze dále rozdělit podle typů, přičemž každý je vhodný na jiný druh otestování aplikace. Rozdíly mezi typy mohou být například ve způsobu přístupu k aplikaci či v jaké fázi jsou použity.

3.4.1 Funkční a nefunkční testy

Mezi funkční testy můžeme zařadit všechny testy, které kontrolují, zda aplikace funguje přesně tak jak má. Postupně jsou otestovány všechny funkčnosti a ověřuje se jejich správnost vůči specifikaci a požadavkům zákazníka. [7] Dále se kontroluje, zda nechybí žádné funkce, které jsou ve specifikaci.

Naproti tomu, nefunkční testy se zabývají testováním všech vlastností softwaru, které nesouvisí přímo s případy užití, ale mají vliv na správné vykonávání aplikace. Do tohoto typu testů patří především performance testy (v překladu: výkonové testy), které mají za úkol ověřit, jak se bude aplikace chovat v případě zátěže, např. přenos velkého množství dat, souběžné připojení velkého počtu uživatelů apod. Očekává se, že aplikace bude stále reagovat i při větší zátěži. Dále se testuje výkon aplikace vůči serveru, na kterém je nasazena, např. požadavky na paměť, síťové spojení, zatížení procesoru apod.

Na základě informací z předchozích dvou odstavců můžeme konstatovat, že funkční testy mají na výslednou bezporuchovost aplikace zásadní vliv. Proto se klade důraz na kontinuální začleňování těchto typů testů během celého procesu testování, přičemž se nebere v potaz rozsah testovaného produktu. Nejčastěji je využíváme v integrační, systémové a akceptační fázi, ve kterých je odhaleno největší množství chyb. Co se týče nefunkčního testování, určitě má také vliv na výslednou poruchovost aplikace, bohužel se na něj často zapomíná nebo jsou tyto testy prováděny minimálně.

3.4.2 Smoke testy

Občas nepřesně označovány jako tzv. „zahořovací testy“. Jedná se o druh testu, který je využit v okamžiku dokončení vývoje, tedy v době, kdy je aplikace možná spustit, zpravidla na konci fáze integračního testování. Jde o krátký test, který má za úkol vyzkoušet všechny části programu, zda jsou funkční a správně nainstalované. Dále může například nastavit konfiguraci programu pro další potřeby testování. Zaměřují se především na otestování hlavních částí aplikace, které bývají zřídka kdy upravovány.

V porovnání s ostatními typy testů, se jedná o rozsahově malé testy, jejichž provádění je často automatizováno nebo alespoň částečně automatizováno. Test je poté jednodušší spustit po každé instalaci aplikace a rychleji nám odhalí případné problémy se základními funkcemi systému.

3.4.3 Progresní a regresní testy

Progresní testy jsou používané pro kontrolu nových funkcí systému, využívají se ve všech fázích testování. Pro správné otestování nových funkcí potřebujeme přesnou specifikaci chování nových částí aplikace. Progresní testy nejsou příliš využívané a často je tento typ rozložen do jiných typů testů.

Regresní testy jsou využívané při opětovném testování vlastností a funkcí systému. Jejich smyslem je ověření, že nově vzniklé funkčnosti neměly vliv na ostatní části aplikace. Především na ty, které přímo nesouvisely s novým kódem a nikterak se neměnily.

Regresní testy jsou v praxi hojně využívané, proto je vhodné jejich provádění automatizovat, což nám umožní ušetřit čas při jejich opakovaném provádění a tester se může věnovat spíše novým funkčnostem. Regresní testy se tedy hlavně používají při retestech po opravách chyb.

3.4.4 Testy splněním a selháním

Často tyto typy testů můžeme v literatůře najít i pod označením pozitivní a negativní testy. Při provádění testů splněním, jsou aplikaci předhazována data, která jsou vždy pravdivá a měla by být splnitelná. Výstupy, které z aplikace dostáváme, jsou poté porovnávány s očekávanými výsledky od zákazníka.

Naopak při testech selháním jsou aplikaci podsouvána nestandardní data, která mají za účel aplikaci neočekávaně ukončit či se dostat do nějakého nepředvídatelného stavu. Během testování se ověřují mezivýsledky, přičemž se snažíme aby neobsahovaly nechtěné hodnoty. Často jsou obě kategorie v rámci jedné sady testů a navzájem se prolínají.

Oba typy testů se používají velice často, obvykle s použitím testování černé skříňky. Při provádění testů jsou většinou nejprve spuštěny testy se správnými daty a až po zdárném průchodu jsou spuštěny i testy s nesprávnými daty. Ovšem není to podmínkou a jednotlivé testy se mohou klidně prolínat v případě, že aplikace je na to připravená, tzn. je opravena většina chyb spojená s používáním správných vstupů.

Pro provádění testů selháním není jedinou povinností kontrolovat špatné výsledky, ale například i chybové hlášky, které jsou v těchto případech zobrazeny, např. chybová hláška „Error“ uživateli neřekne, co provedl špatně a jaký je skutečně problém. Nakonec bych zdůraznil, že tento typ testování nám může pomoci odhalit velké množství chyb a proto se jejich používání vřele doporučuje. A to především v systémové úrovni testování, kdy je testována aplikace jako celek.

Kapitola 4

Automatizované testování

Automatizované testování nám může přinést výraznou úsporu času i peněz pří vývoji softwaru, přesto je v praxi stále velice podceňovanou disciplínou. Přitom se nejedná jen o automatizování testovacích scénářů, ale teoreticky lze automatizovat celý průběh testování, ať už se jedná o vytvoření testovacích případů ze specifikace, instalaci testovacích prostředí či porovnání dosažených výsledků. Automatizaci testování je rozhodně vhodné použít v případě, že máme stabilní a téměř neměné testovací případy, které je nutné neustále opakovat a jejich manuální provádění by zabralo spoustu času.

Tato kapitola se věnuje nejprve základním faktům týkajících se automatizovaného testování, kde jsou závěrem shrnuty výhody a nevýhody jejich použití. Druhá část kapitoly se poté zaměřuje na frameworky pro automatizované testování a jejich srovnání.

4.1 Význam automatizovaného testování

Hlavním cílem zautomatizování testů je časová úspora při jejich pouštění. Dalo by se konstatovat, že automatizace by měla usnadnit a zefektivnit proces testování softwaru. Pokud jsou totiž testy prováděny bez zásahu lidského faktoru, jsme schopni se zbavit chyb, které z toho plynou, např. vložení nesprávných vstupních dat či špatné porovnání hodnot. Automatické testy lze rozdělit do stejných kategorií jako manuální testy a to především proto, že se většinou jedná o zautomatizování manuálních testů.

4.2 Realizace automatizace testů

Vhodné je automatizovat především testy, které se pouští často, je jich velké množství a příliš se nemění. Do této kategorie lze zařadit smoke testy, komparační testy, regresní testy nebo nefunkční testy. Pro realizaci je nutné splnit některé základní podmínky v postupu testování softwaru. Velice těžko se tvoří automatizované testy pro již existující aplikaci, z tohoto důvodu je potřeba přemýšlet nad automatizací již ve fázi analýzy a návrhu systému. Ve fázi vývoje se potom vytváří automatizované testy, které ověřují funkčnost již vzniklých, dále málo měněných, částí aplikace.

Z pohledu nákladů na vývoj softwaru je nejlepší automatizovat co největší část aplikace, přičemž je potřeba mít stále na paměti, že nejlepší je automatizovat celky, které se nemění nebo se do nich zasahuje minimálně. Zároveň je vhodné věnovat se částem, kde může

vznikat potenciálně největší množství chyb. Další pohled může být z pohledu délky životního cyklu aplikace. Pokud se plánuje vývoj pouze jedné verze nebo se jedná o kratší projekt v rámci několika měsíců, automatizované testy se nám příliš nevyplatí. Naproti tomu u dlouhodobých projektů je návratnost automatizace značná.

4.3 Výhody a nevýhody automatizování

Pokud chceme porovnávat výhody a nevýhody automatizovaného testování, je nutné přihlédnout vždy ke konkrétnímu projektu a k jeho specifikům. Tyto specifika mohou mít zásadní vliv při možnostech automatizace. Ačkoliv se tedy může zdát, že automatizace má vždy navrch oproti manuálnímu testování, nemusí tomu tak v případě některých projektů být.

Nejpodstatnější výhodou automatizovaných testů je bezesporu úspora času. Test spuštěný automaticky, který umožní kontrolu výsledků, ušetří čas pracovníkům, kteří jej předtím museli procházet manuálně. Pokud se jedná o kompletní automatizaci, tedy test je možné nastavit na průchod v libovolnou dobu, využije se toho a testy jsou spouštěny například každý den o půlnoci. Tím je zajištěna pravidelná kontrola aplikace a navíc neomezuje pracovníky v jejich manuálním testování. Další výhodou je odstranění chyb způsobených lidským faktorem, např. zadání špatných vstupních dat či kontrola výsledků.

Automatizace testování však nepřináší jen výhody, jak by se mohlo zdát z dosavadního povídání. Jako nevýhodu lze určitě považovat nutnost pravidelné aktualizace testů, a to především v případě, že se změní funkčnost aplikace zasahující do průchodu testu. Pokud by test nebyl aktualizován, může dojít k tomu, že budou reportovány chyby, které vůbec nejsou chybami. Ovšem může se také projevit opačný problém, kdy test neodhalí chybu v nové implementaci, což může zkomplikovat případné opravy. Další možnou nevýhodou je nutnost administrace dalšího prostředí, na kterém se budou testy provádět a dále vyčlenění testovacích dat, které budou pro automatizaci používány.

Pokud se tedy zamyslíme nad výhodami a nevýhodami, zpravidla dojdeme k závěru, že automatizované testování je jednoznačnou výhodou. Proč se tedy nepokusit zautomatizovat všechny části aplikace a pouze se starat o útržbu testů? Protože automatizace nikdy nemůže nahradit uvažování lidského uživatele a některé funkčnosti budou vždy potřebovat kontrolu pomocí lidského úsudku. Porovnání bych tedy zakončil tím, že je vhodné najít správnou míru automatizace a manuálních testů, a vzájemně jejich používání provázat.

4.4 Nástroje pro automatizované testování

Pro psaní automatizovaných testů existuje řada různých nástrojů, je tedy vhodné zvolit ten ideální pro potřeby každého projektu. Nástroje pro testování UI[1] můžeme rozdělit do základních dvou kategorií, kterými jsou webové aplikace a desktopové aplikace. V následující části se budeme věnovat především nástrojům pro testování webových aplikací.

1. uživatelské rozhraní

4.4.1 Selenium

Selenium je sada různých nástrojů s otevřeným kódem, z nichž každý má odlišný přístup pro podporu automatizace testování. Většina testerů, kteří píší automatizované testy, používá jeden nebo dva nástroje. Nicméně pokud se blíže podíváme na všechny nástroje, které Selenium nabízí, můžeme najít mnoho užitečných funkcí pro různé druhy testů. Každý ze sady těchto nástrojů obsahuje specifické funkce pro dané druhy testů, přičemž tyto operace jsou flexibilní, umožňují mnoho možností pro testování uživatelského rozhraní a porovnávání získaných výsledků vůči skutečnému chování aplikace.

Jednou z klíčových funkcí Selenia je spouštění vytvořených testů na různých prohlížečích. Selenium má integrovanou podporu pro použití v mnoha programovacích jazycích - Java, Ruby, Python, PHP, Perl, C#, Groovy. Dále není závislý na operačním systému, ale testy lze pustit na systémech Windows, Linux a Mac. První nástroj byl vyvíjen již v roce 2004 Jasonem Hugginsem. Od té doby se stal v komunitě velmi oblíbeným a dnes pravděpodobně nejrozšířenějším nástrojem pro automatizované testování. [8]

4.4.1.1 Selenium IDE

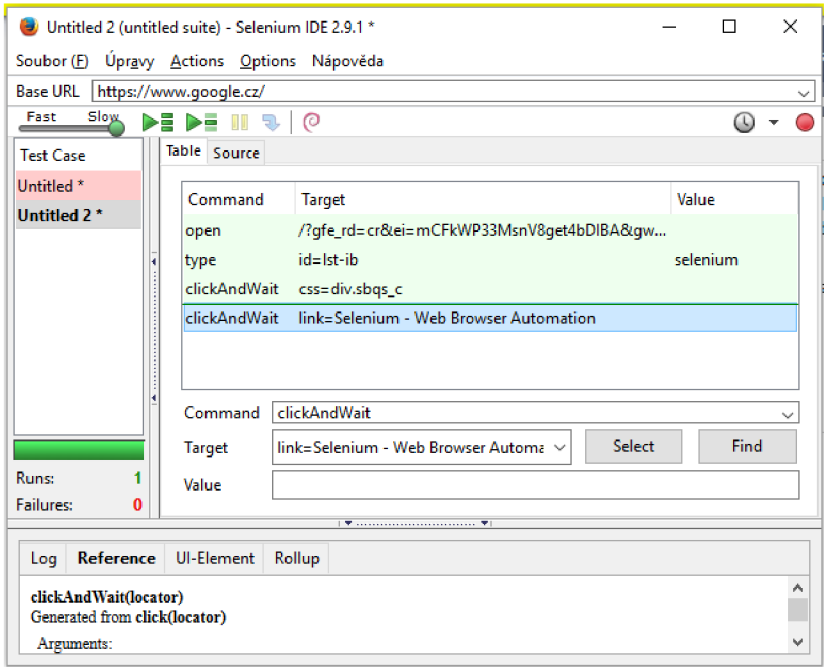
Jedná se o plugin určený do prohlížeče Firefox a umožňuje dva druhy zápisů testu. Prvním z nich je jednoduchý výběr funkcí, které Selenium pro HTML prvky nabízí, nastavení cesty k prvku na stránce a případně hodnota, která se má použít pro vykonání funkce. Pokud si to ukážeme na příkladu, tak jako funkci můžeme použít například „TYPE“, prvek bude textové políčko formuláře a hodnota bude text, který chceme vložit. Po provedení tohoto příkladu se do zvoleného políčka vypíše zvolený text.

Druhým způsobem, jak zapsat test v Selenium IDE, je zaznamenan posloupnost prováděných kroků přímo v prohlížeči, které jsou uloženy jako skript. V praxi to znamená, že tester projde testovací scénář a ten je automaticky zaznamenán do posloupnosti příkazů. Mezi oběma druhy zápisu lze kombinovat, což znamená, že pokud nahrajeme skript, tak jej můžeme manuálně upravit a naopak.

Nevýhodou Selenium IDE je možnost používat testy jen v prohlížečích Firefox, protože neexistuje plugin do ostatních prohlížečů. Sporným bodem mezi výhodou a nevýhodou může být nemožnost upravovat testy pomocí programovacího jazyka. V případě, že testy spravuje člověk bez znalosti programování, je to pro něj vhodná volba. Pokud ale máme člověka se zkušenostmi s programováním, je výhodnější použít některý z jiných nástrojů Selenia, které umožňují testy upravovat pomocí programovacího jazyka, který jim umožní použití dalších funkcí. Aby Selenium IDE částečně zmírnilo tuto nevýhodu, umožňuje vygenerování základu testu do podporovaných programovacích jazyků. Takto vygenerovaný test je ale nutné dále upravit, tak aby bylo umožněno jeho spuštění.

4.4.1.2 Selenium RC

Jedná se o nástroj, který umožňuje vytvářet testy pomocí široké škály programovacích jazyků. Jádrem je server, který představuje prostředník mezi klientem a cílovým počítačem



Obrázek 4.1: Selenium IDE

(serverem), umožňující spouštění a vypínání jednotlivých instancí prohlížečů. Tím se dostáváme k jeho obrovské výhodě v porovnání se Selenium IDE, kterou je možnost pouštění testů ve více prohlížečích. Server je v podstatě jedna třída napsaná v Javě, kterou není potřeba instalovat a lze ji jednoduše spustit pomocí JRE[2]. Funguje potom jako prostředník mezi prohlížečem a testovanou aplikací, kde jednotlivé požadavky jsou posílány skrze protokol HTTP. Pro vykonání konkrétní akce předává server pokyny prohlížeči pomocí javascriptové funkce.

Psaní testů pomocí Selenia v podporovaném programovacím jazyce, je umožněno pomocí sady klientských knihoven, speciálně vytvořených pro každý podporovaný programovací jazyk. Pokud chceme tyto knihovny využívat, stačí přidat referenci na požadované API[3]. To nám umožní využívat nové příkazy z jazyka Selenese, např. vyhledání prvků na stránce či přesměrování. Protože můžeme používat podporovaný programovací jazyk (např. Javu nebo C#), lze využít některý z frameworků pro jednotkové testování, které nám ulehčí např. porovnávání výsledků.

Na příkladu vidíme práci se Selenium RC. V prohlížeči Firefox je skrze vyhledávač Google vyhledán řetězec „selenium“ a klikne se na vyhledaný odkaz:

2. Java Runtime Environment - běhové prostředí pro Javu
3. Application Programming Interface - rozhraní pro programování aplikací

```
package com.example.tests;

import com.thoughtworks.selenium.*;
import org.openqa.selenium.Keys;

public class NewTest extends SeleneseTestCase {

  public void setUp() {
    setUp("http://www.google.com/", "*firefox");
  }

  public void loadSeleniumPageTest() {
    selenium.Open("http://www.google.com");
    selenium.Type("id=gs_htif0", "selenium" + Keys.ENTER);
    selenium.Click("link=Selenium - Web Browser Automation");
  }
}
```

4.4.1.3 Selenium WebDriver

Je založen na podobném principu jako Selenium RC, který se snaží ještě zdokonalit. Pro spouštění testů není potřeba mít spuštěn server, protože již ve svém API má integrovánu podporu pro jednotlivé prohlížeče. Pro spuštění testu ve zvoleném prohlížeči je tedy nutné mít driver, který jej podporuje. Výhodou je potom lepší přizpůsobení testu přímo pro zvolený prohlížeč. Příkazy mu nejsou podsouvány pomocí javascriptových funkcí, ale přímo pomocí zvoleného driveru. [9]

Protože se jednotlivé verze WebDriveru od sebe mírně liší, je potřeba si dát na tohle chování pozor a testy mírně upravovat pro potřeby jednotlivých prohlížečů. [10] Před začátkem provádění testu je potřeba nastavit v jakém prohlížeči se má test spouštět a tedy jaký driver má být inicializován. Mezi podporované prohlížeče patří Firefox, Chrome, Internet Explorer, Opera, Safari či Microsoft Edge. Mnoho driverů vzniká neoficiálně díky početné komunitě, proto můžeme nalézt podporu i pro méně známe prohlížeče.

Pro inicializaci WebDriveru si vystačíme zpravidla s bezparametrickým konstruktorem, který nám spustí prohlížeč ve výchozím nastavení. Je možné mu přidat další speciální nastavení pomocí „capabilities“ (v překladu: „schopností“), které umožňují například nastavit úvodní stránku prohlížeče. Pro vyhledávání elementů lze využít spoustu intuitivních metod a navíc je můžeme uložit jako objekty rozhraní IWebElement, což nám umožní daleko jednodušší volání metod a získávání atributů na jednotlivých prvcích.

Na uvedeném příkladě lze vidět práci se Selenium WebDriver, příklad provede stejnou činnost jako u Selenium RC:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.Keys;
```

```
public class NewTest {

  public static void main(String[] args) {
    WebDriver driver = new FirefoxDriver();
    driver.get("http://www.google.com/");
    WebElement element = driver.findElement(By.id("gs_htif0"));
    element.sendKeys("selenium" + Keys.ENTER);
    driver.findElement(By.linkText("Selenium - Web Browser Automation"))
      .click();
    driver.close();
  }
}
```

4.4.1.4 Selenium Grid

Tento nástroj umožňuje spouštět testy zároveň v různých prohlížečích a na různých strojích. Tím tester výrazně ušetří čas potřebný k provedení série testů, jelikož testování UI zabere výrazně více času než jednotkové testování kódu. Princip Selenium Grid je založen na Selenium RC, tedy na každé klientské stanici běží server a dálkově je řízen centrální stanicí, která posílá příkazy k provádění testů. [11]

4.4.2 Tellurium

Tellurium je nástroj s otevřeným kódem pro automatizované testování webových aplikací a je založeno na Selenium frameworku. Podobně jako Selenium IDE jej můžeme nainstalovat do Mozilly Firefox jako doplněk a nahrávat si jednotlivé skripty. Rozdílným přístupem oproti Seleniu je vyhledávání prvků. Tellurium je postaveno na konceptu UI Module, který se snaží zaměřit na znovu použitelné a snadno udržovatelné testy pro dynamicky se měnící aplikace. UI Module je seskupením UI (DOM[4]) prvků stránky. [13] Správa těchto prvků je řízena pomocí skriptovacího jazyka Groovy a odkazování na jednotlivé elementy je umožněno pomocí definovaného identifikátoru (uid). [12]

V příkladu se načte vyhledávání přes Google a vyhledá se textový řetězec: [14]

```
ui.Container(uid: "GoogleSearchModule", clocator: [tag: "td"]) {
  InputBox(uid: "Input", clocator: [title: "Google Search"])
  SubmitButton(uid: "Search", clocator: [name: "btnG", value: "Google
    Search"])
  SubmitButton(uid: "ImFeelingLucky", clocator: [value: "I'm Feeling
    Lucky"])
}
```

4. Document object model - objektově orientovaná reprezentace XML nebo HTML dokumentu

4.4.3 WebAii Framework

Obdobně jako v předchozím případě se jedná o framework pro automatizované testování webových aplikací. Rozdíl oproti předchozím je v případné úpravě frameworku, jelikož WebAii je sice zdarma, ale nemá otevřený kód, tedy uživatel nemá možnost případné nedostatky odstranit vlastní úpravou frameworku. Testy lze zapisovat v programovacích jazycích C# a Visual Basic .NET, je tedy spíše orientován pro použití na systémech Windows. Dokonce existuje plugin do Visual Studia, který ještě více usnadňuje psaní testů.

WebAii framework je postaven na dvou hlavních třídách. První z nich je třída *Browser*, která reprezentuje jednotlivé instance prohlížečů. Druhá je třída *Manager*, která se stará o životní cyklus prohlížeče a řídí komunikaci mezi prohlížeči a frameworkem. Testování je možné na prohlížečích Firefox, Internet Explorer, Chrome a Safari. V každém z nich je potřeba mírně upravit nastavení, což se dá přirovnat k nastavení driveru u Selenium WebDriver. Podrobné informace k nastavení prohlížečů jsou přehledně zpracované v dokumentaci frameworku. [15]

Obecně se dá říci, že se WebAii Framework a Selenium WebDriver stylem psaní testů příliš neliší. Poskytují podobný styl definování prohlížeče i orientace na stránce. Co by se dalo považovat za výhodu oproti Seleniu, je vyhledávání prvků na základě již vytvořených tříd. WebAii totiž ve svém API obsahuje velké množství tříd, které reprezentují jednotlivé prvky na stránce a tím nám umožní blíže specifikovat vyhledávání těchto prvků a používání specializovaných metod a vlastností.

Dále je vhodné se zmínit o třídě *Settings*, ve které se nastavuje specifické chování pro pouštěné testy. Mimo jiné v něm můžeme nastavit použitý výchozí prohlížeč (pokud není specifikován přímo pomocí třídy *Browser*), počáteční URL adresu, anotace prvků (označení prvků na stránce, které jsou právě používané) nebo čekací doba pro volání funkcí. Protože lze tyto konfigurace psát do souboru, usnadňuje nám to spouštění různých testů s odlišnými konfiguracemi, přičemž není potřeba kvůli každé změně upravit kód testu.

Příklad zachycuje práci s WebAii frameworkem:

```
[TestClass]
public class WebAiiFrameworkClass
{
  private Manager manager;

  [TestInitialize]
  public void TestInitialize()
  {
    Settings mySettings = new Settings();
    mySettings.Web.DefaultBrowser = BrowserType.InternetExplorer;
    manager = new Manager(mySettings);
    manager.Start();
    manager.LaunchNewBrowser();
  }

  [TestCleanup]
```

```
  public void TestCleanup()
  {
    manager.Dispose();
  }

  [TestMethod]
  public void NavigateToGoogle()
  {
    manager.ActiveBrowser.NavigateTo("http://www.google.com/");
  }
}
```

4.4.4 WatiN

WatiN je další z testovacích frameworků s otevřeným kódem, který umožňuje zapisovat automatizované testy v jazyce C#. Jeho vznik byl inspirován podobným frameworkem Watir[5], který je určen pro testy psané v jazyce Ruby. [16] Podporovány jsou pouze prohlížeče Internet Explorer a Firefox, což oproti ostatním nástrojům není příliš dobrá podpora. V dalších ohledech je dost podobný nástroji WebAii, za kterým bohužel mírně zaostává, např. v použití tříd pro reprezentaci prvků stránky nebo užší nabídce metod pro vyhledání prvků (nepodporuje např. Xpath). Dalším nedostatkem je minimální možnost konfigurace spouštěných testů, kterou je možné odstranit případnými opravami frameworku. Ovšem z praktického hlediska je vhodnější zvolit jiný nástroj, který již tuto podporu poskytuje.

V převzatém příkladě lze vidět vyhledání textu „WatiN“ skrze vyhledáváč Google a kontrola, zda se vyhledaný text na stránce nalézá: [17]

```
[Test]
public void SearchForWatiNOnGoogle()
{
  using (var browser = new IE("http://www.google.com"))
  {
    browser.TextField(Find.ByName("q")).TypeText("WatiN");
    browser.Button(Find.ByName("btnG")).Click();

    Assert.IsTrue(browser.ContainsText("WatiN"));
  }
}
```

4.4.5 TestComplete

TestComplete je komerční nástroj určený pro širokou škálu možností testování. Umožňuje pouštění zátěžových či jednotkových testů, testů pokrytí nebo třeba testy grafického rozhrání (desktopové i webové). Pro vytvoření testu jej stačí nahrát pomocí zaznamenání každého kroku, který uživatel provede. Úpravu testu lze provést v jednoduchém tabulkovém

5. Web application testing in Ruby

zobrazení. V případě složitějších změn lze testy upravovat pomocí skriptovacích jazyků, např. VBScript, JScript nebo C#Script. [18]

4.4.6 Ranorex

Podobně jako předchozí, je i Ranorex komerční nástroj, který poskytuje široké množství nejrůznějších funkcí pro testování grafického rozhraní. Nepoužívá žádný vlastní skriptovací jazyk, ale využívá programovací jazyk C# nebo VB.NET. Pro vyhledávání prvků grafického rozhraní se používá RanoreXPath, která by měla zajistit unikátní identifikaci prvku, a to ať už se jedná o desktopovou, webovou či mobilní aplikaci. Dále umožňuje nahrávat testy podobně jako Selenium IDE či TestComplete, pro tyto účely slouží aplikace Ranorex Studio. V neposlední řadě je potřeba zmínit, že se zaměřuje především na testování aplikací pod systémy Windows a má tedy dobrou podporu .NET knihoven. [19]

4.4.7 Protractor

Na závěr bych se chtěl zmínit o jednom zástupci nástrojů, které umožňují testování grafického rozhraní pouze pro specifickou webovou technologii. Na rozdíl od všech výšše zmíněných nástrojů, které nejsou závislé na použité webové technologii, Protractor je použitelný pouze na testování aplikací napsaných v Angularu. Pracuje na základě modelů, které se používají v jednotlivých prvcích a je tedy schopen detekovat všechny změny, které se na stránce objeví. Protože je navázán přímo na modely, nemusí se zabývat žádnými čekacími lhůtami, ale může automaticky provádět operace po ukončení předcházející či po objevení se požadované hodnoty.

Na příkladu lze vidět, jak se s Protractorem pracuje: [20]

```
describe('angularjs homepage todo list', function() {
  it('should add a todo', function() {
    browser.get('https://angularjs.org');

    element(by.model('todoList.todoText')).sendKeys('write first
      protractor test');
    element(by.css('[value="add"]')).click();

    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('write first protractor
      test');

    // You wrote your first test, cross it off the list
    todoList.get(2).element(by.css('input')).click();
    var completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).toEqual(2);
  });
});
```

Kapitola 5

Analýza a návrh

Kapitola je rozdělena do dvou částí, z nichž první je zaměřena na analýzu požadavků. Na základě požadavků zákazníka vznikla sada požadovaných funkčností, které budou přehledně znázorněny pomocí diagramu případů užití a provede se jejich rozdělení na funkční a nefunkční požadavky dle definice. [21] V druhé části kapitoly se zaměříme na návrhovou část a sice na rozdělení do jednotlivých komponent. Komunikace mezi komponentami bude zachycena pomocí diagramu komponent a činnost každého prvku bude zvlášť rozebrána. Každou komponentu ještě samostatně rozebereme pomocí diagramu tříd, který nám přehledně graficky ukáže vnitřní strukturu komponenty. Na závěr je uvedena adresářová struktura aplikace a je v ní vysvětleno, k čemu slouží jednotlivé složky.

5.1 Analýza požadavků

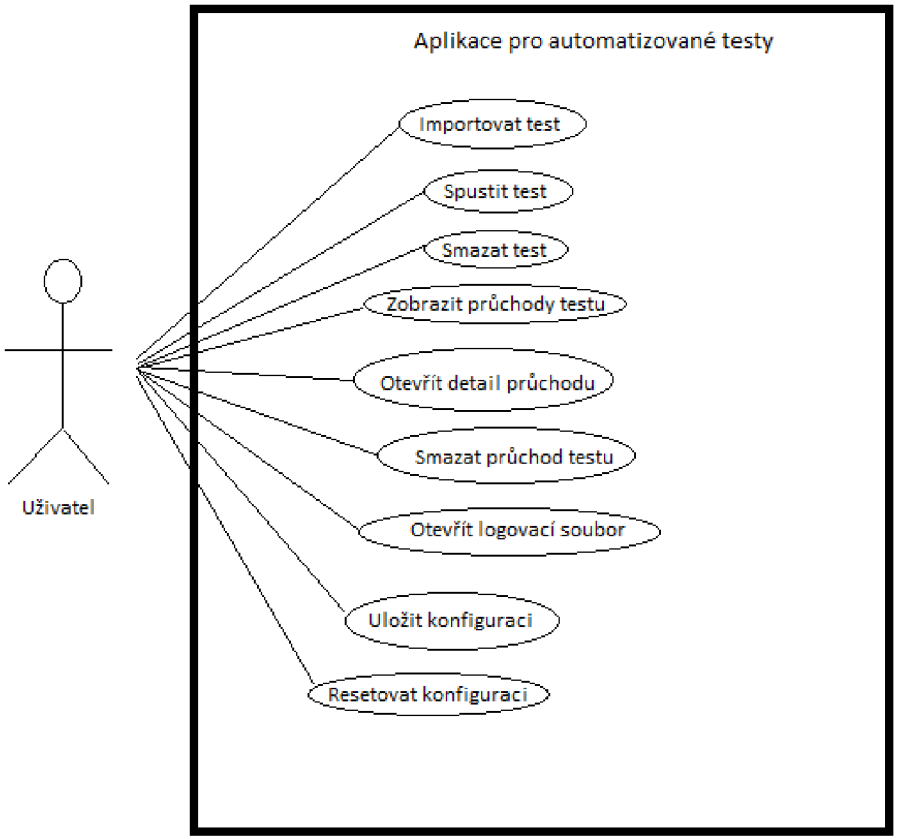
Požadavky na aplikaci vzešly od zákazníka, v rámci analýzy byly rozděleny na funkční a nefunkční. Požadavky byly dále graficky zpracovány do diagramu případů užití, který přehledně zobrazuje, co může uživatel se systémem provádět a jaké funkce mu jsou umožněny. [22] Množství požadavků na aplikaci není velké, ale spíše zahrnuje rozsáhlejší celky, výsledný diagram je krásně čitelný a přehledný (viz obrázek 5.1). Protože aplikaci a její funkce může používat kdokoliv, není nutné vytvářet více druhů uživatelů.

5.1.1 Funkční požadavky

Funkční požadavky specifikují základní chování aplikace a její funkce, které může uživatel v aplikaci využívat. Mezi funkční požadavky kladené na tuto aplikaci patří:

- Importování testu – aplikace umožní importovat test zapsaný v XML formátu pod zvoleným názvem do systému, tak aby byl převeden do Java kódu s podporou Selenia. Naimportované testy může uživatel v aplikaci dále využívat v ostatních případech užití. Během importu je test zvalidován pomocí XSD schématu.

- Správa testu – uživatel může vybírat jednotlivé testy a provádět na nich akce. Jednou z akcí bude jeho spuštění, kdy se provedou postupně všechny kroky testu, tak jak byly zapsány v XML souboru. Další akcí bude smazání existujícího testu. V neposlední řadě bude mít možnost uživatel prohlédnout jednotlivé průchody testů.



Obrázek 5.1: Diagram případů užití

- Logování – aplikace bude podporovat dva druhy logování. Všechny operace týkající se aplikace se budou logovat do jednoho globálního logovacího souboru. Operace týkající se průchodu spouštěných testů se budou logovat do samostatných souborů, jejichž prohlídku bude mít uživatel umožněnu z grafického rozhraní aplikace. Kromě prohlídky logovacích souborů bude uživateli umožněno otevření i ostatních souborů týkajících se průchodů testů.

- Konfigurace – uživatel může v GUI[1] aplikace nastavovat prohlížeč, ve kterém se budou testy spouštět. Dalším možností v nastavení bude volba použité databáze, v případě, že ji uživatel bude chtít využívat. V takovém případě proběhne kontrola připojení do databáze, aby průchody testů nekončily na chyby způsobené špatným připojením.

1. grafické uživatelské rozhraní

5.1.2 Nefunkční požadavky

Nefunkční požadavky nejsou přímo navázané na jednotlivé funkčnosti aplikace, ale spíše jako globální požadavky na celou strukturu aplikace. Do nefunkčních požadavků se dají zahrnout omezení a vlastnosti celého systému. Do nefunkčních požadavků této aplikace patří:

- Podpora prohlížečů – podporovány budou minimálně prohlížeče Internet Explorer, Mozilla Firefox a Google Chrome. Volitelně lze přidat podporu pro další internetové prohlížeče.
- Podpora databází – podporována bude databáze MySQL. Podpora dalších databází je volitelná.
- GUI aplikace – aplikace bude desktopová, každý uživatel si bude spouštět na lokální stanici vlastní instanci aplikace. GUI aplikace bude jednoduché a pro uživatele přehledné.

Hlavním důvodem, proč byla zvolena desktopová aplikace namísto webové, bylo odlišné rozhraní od spouštěných testů. Jelikož při každém provedení testu je spuštěna nová instance prohlížeče, mohlo by být toto chování pro uživatele matoucí. Dalším méně závažným důvodem je fakt, že aplikace obsahuje pouze 4 případy užití, pro které není nutné vytvářet složitější webové rozhraní.

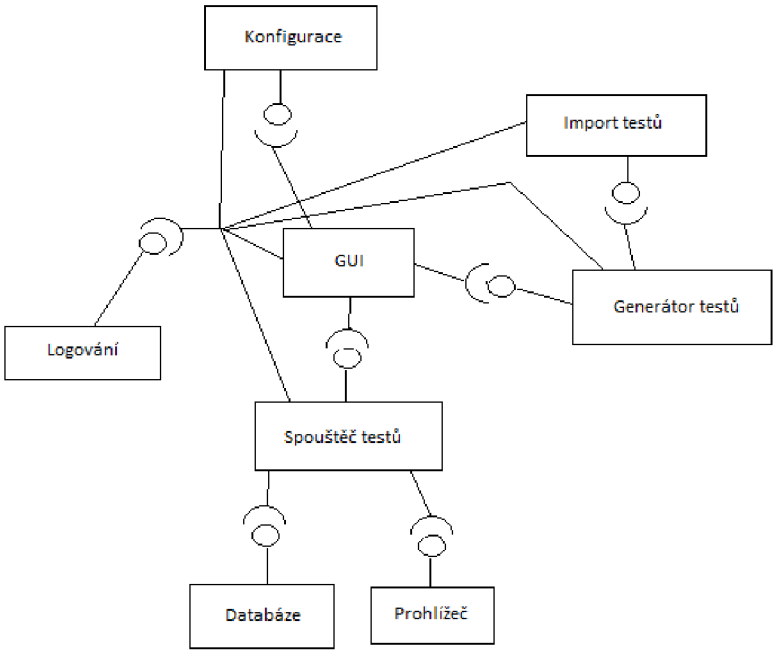
5.2 Návrh komponent aplikace

Na základě analýzy požadavků bylo navrženo rozdělení do jednotlivých komponent, které by spolu měly interagovat a navzájem spolupracovat. Pro zachycení spolupráce se nejvíce hodí použít diagram komponent, který graficky zachytí jednotlivé interakce. Každá komponenta poskytuje nebo naopak využívá rozhraní jiné komponenty, což je na diagramu znázorněno vazbami. [22]

V další části budou rozebrány jednotlivé komponenty podrobněji a jejich struktura bude přehledně zobrazena v diagramu tříd.

5.2.1 GUI

Základem komponenty je třída *Application*, která kromě atributů a metod, které jsou zobrazeny na obrázku 5.3, obsahuje i atributy pro jednotlivé grafické prvky. Dále jsou pro některé prvky vygenerovány metody na obsluhu obvyklých akcí, např. zachytávání kliknutí na tlačítko. Tyto atributy ani metody nejsou v diagramu zakresleny, protože by výrázně navýšily rozsáhlost a snížily přehlednost diagramu. Druhá třída slouží pro načítání hodnot elementů používaných pro seznamy.



Obrázek 5.2: Diagram komponent

5.2.2 Import testů

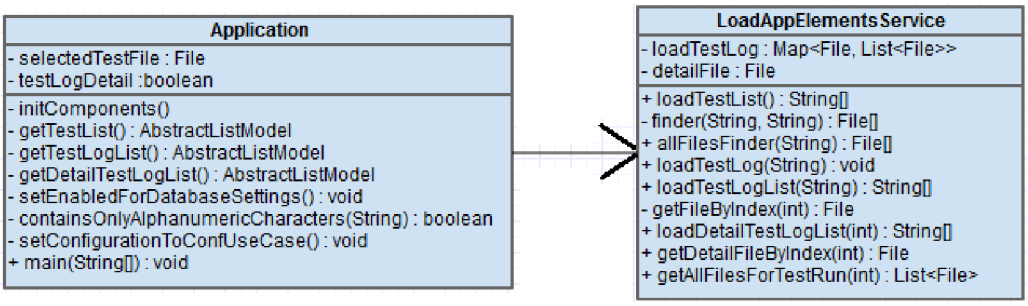
V této komponentě se mapuje test zapsaný v XML do Javy, k čemuž se používá třída *ConverterService* (obrázek 5.4). Pro mapování do Javy se používá celá hierarchie znázorněna na obrázku 5.5. Tato hierarchie je vygenerována podle XSD schématu.

5.2.3 Generátor testů

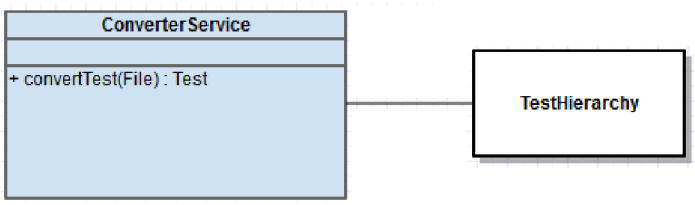
Posloupnost kroků testu, které byly převedeny do Javy při importu, je nutné zpracovat a vygenerovat z nich vlastní třídu spustitelného testu. K tomu slouží *GeneratorService* a k němu přidružené třídy (viz obrázek 5.6). Po vygenerování souboru s novým testem jej ještě dynamicky zkompilujeme, tak aby se mohl dále v aplikaci využívat.

5.2.4 Konfigurace

Konfigurace je načítána z konfiguračního souboru, jehož název je mimo jiné zaznamenán v konfiguračních konstantách. Ty obsahují klíče do tohoto souboru a další konstanty pro obecné použití v celé aplikaci. Pro úpravu souboru se používá třída *ConfigurationService*, která zároveň umožňuje i načítání zvolených parametrů (viz obrázek 5.7).



Obrázek 5.3: Diagram tříd pro GUI



Obrázek 5.4: Diagram tříd pro import testů

5.2.5 Logování

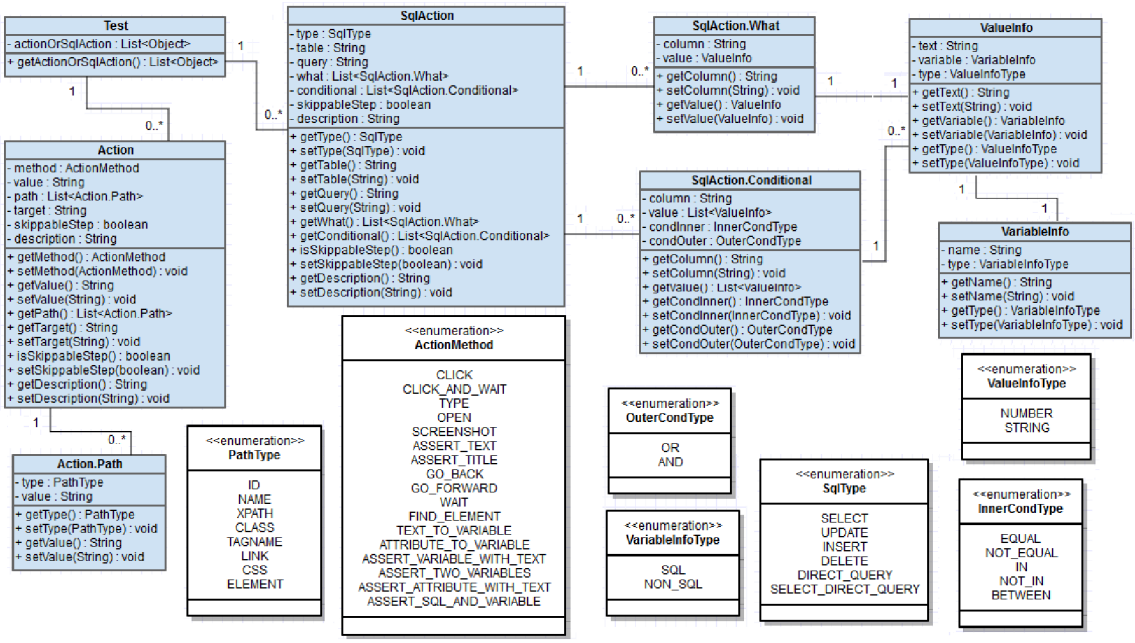
Na obrázku 5.8 můžeme vidět třídu, která nastavuje výchozí logování aplikace do souboru. Následně stačí v každé třídě, kde chceme akce logovat, vytvořit proměnnou pro tento logger a následně na něm už jen volat metody pro logování. Aplikace dále podporuje logování výsledků jednotlivých průchodů testů, které ovšem nepoužívá žádnou třídu, ale pouze vytváří textový soubor, do kterého zapisuje výsledky.

5.2.6 Spouštěč testů

Pro spuštění testů se používá třída *TestRunner*, která zavolá metodu z vygenerovaného testu. Pro volání metody se používají potomci abstraktního testu, který obsahuje základní metody pro nastavení před a po testu. Na diagramu 5.9 je znázorněno volání generovaného testu.

5.2.7 Prohlížeč

Tato komponenta poskytuje rozhraní pro nastavení prohlížeče, který bude použit při vykonávání testů (viz obrázek 5.10). Kromě volby druhu prohlížeče se nastaví i některé jeho základní vlastnosti.



Obrázek 5.5: Diagram tříd pro hierarchii testů

5.2.8 Databáze

Funguje podobně jako komponenta *Prohlížeč*, jen s tím rozdílem, že poskytuje rozhraní pro nastavení databáze (viz obrázek 5.11). Pro zvolený podporovaný typ databáze ustanoví nové spojení, tak aby mohly být volány sql dotazy.

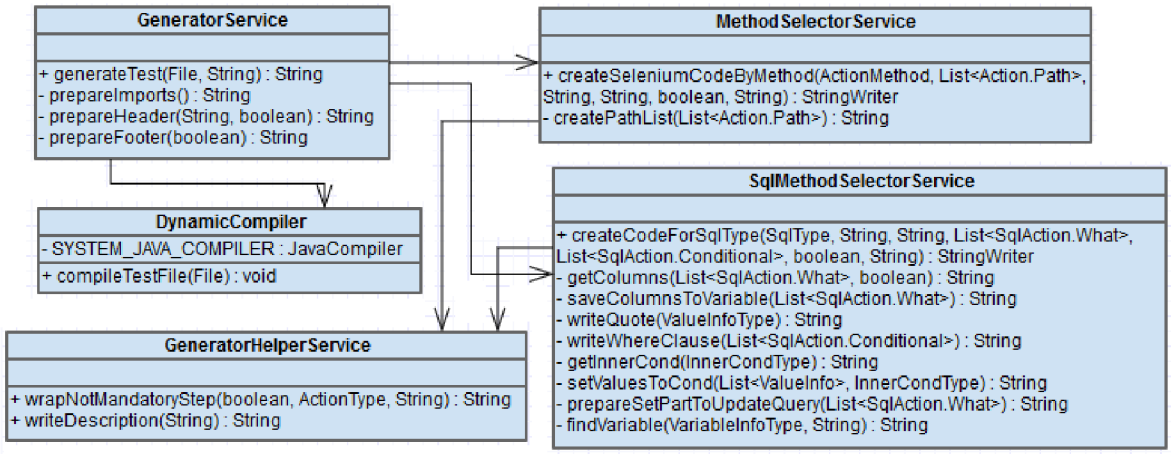
5.3 Adresářová struktura aplikace

Každá složka, obsažená v adresářové struktuře aplikace, má svůj význam. Jejich význam bude v následujících odrážkách vysvětlen:

- kořenová složka – Obsahuje jar archív, ve kterém je zabalena aplikace. Tu lze spustit přes příkazový řádek a nebo lze využít bat soubor pro automatické spuštění na systému Windows. Dále je součástí logovací soubor pro celou aplikaci. Součástí je i složka *data*, která obsahuje podložky pro jednotlivé části aplikace.

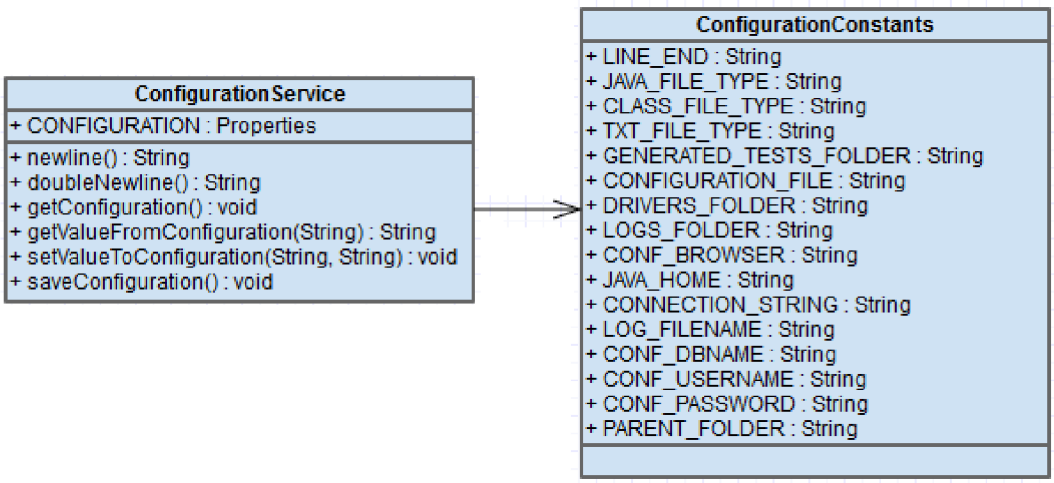
- conf – Složka obsahuje soubor s nastavením, ve kterém jsou uloženy jednotlivé konfigurační hodnoty.

- drivers – Ve složce jsou uloženy ovladače pro podporované prohlížeče. V případě, že chceme zprovoznit podporu pro nový prohlížeč, mimo jiné je nutné do této složky dohrát příslušný ovladač.

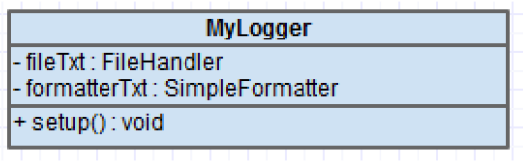


Obrázek 5.6: Diagram tříd pro generátor testů

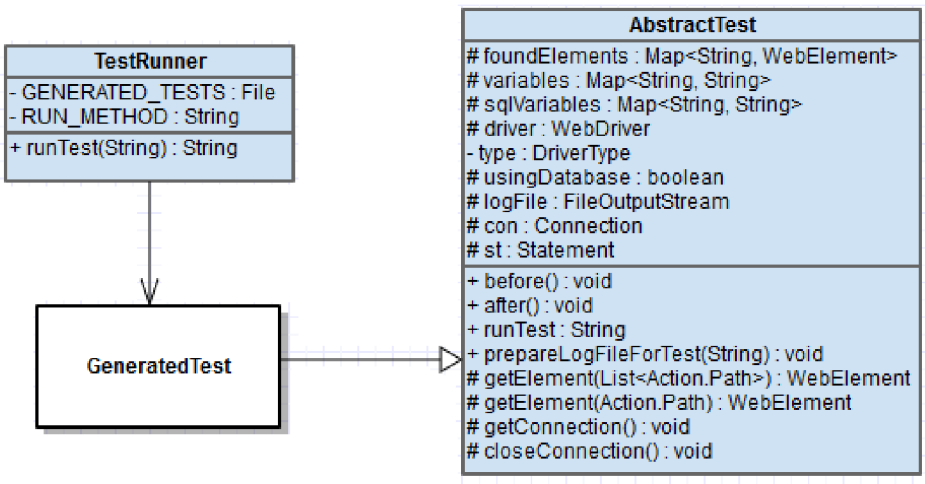
- generatedTests – Všechny testy, které chceme naimportovat do aplikace, jsou vygenerovány do této složky. Nachází se zde vygenerované Java soubory a zkompilované class soubory.
- testLog – Pro každý nově spuštěný test se vytvoří jedna podsložka se stejným názvem jako jméno testu. Do této složky se pak uloží logovací soubor pro každý průchod testu a případně další přidružené soubory týkající se průchodu.



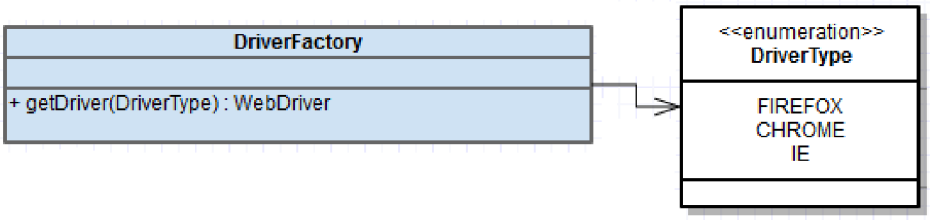
Obrázek 5.7: Diagram tříd pro konfiguraci



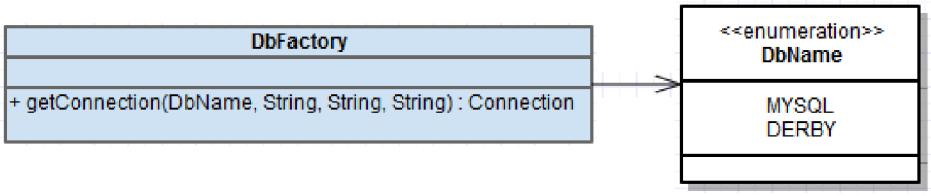
Obrázek 5.8: Diagram tříd pro logování



Obrázek 5.9: Diagram tříd pro spouštěč testů



Obrázek 5.10: Diagram tříd pro prohlížeč



Obrázek 5.11: Diagram tříd pro databázi

```
\Application
     \data
          \conf
               \configuration.properties
          \drivers
               \geckodriver.exe
               \geckodriver
               \chromedriver.exe
               \chromedriver
               \IEDriverServer.exe
          \generatedTests
                \Test.java
                \Test.class
          \testLog
                \Test
                     \Test_2016_12_24_12-12-12.txt
     \application.jar
     \application.log
     \RUN.bat
```

Příklad 5.3.1: adresářová struktura aplikace s testovacím testem Test

Kapitola 6

Použité technologie

Kapitola se věnuje použitým technologiím, s kterými se v aplikaci pracuje. Jednou z hlavních částí je samotné spuštění automatizovaných testů, proto bylo nutné vybrat prohlížeče a databáze, které bude aplikace podporovat. Tímto tématem se zabývá část 2.3 a 5.1.2. V této kapitole bude první část věnována základním informacím o prohlížečích a databázích. V dalších částech je zmíněn použitý programovací jazyk, vývojové prostředí a další technologie, které byly při vývoji využity.

6.1 Podporované prohlížeče

Jelikož systémy vyvíjené společností Unicorn podporují zpravidla tři hlavní prohlížeče, kterými jsou Mozilla Firefox, Internet Explorer a Google Chrome, byly tyto prohlížeče zvoleny za prioritně podporované i ve vzniklé aplikaci. S testerským týmem bylo dohodnuto, že případná podpora dalších prohlížečů může být realizována v dalších verzích aplikace.

Selenium využívá pro práci s aplikací verzi Selenium Webdriver, proto je každý prohlížeč ovládán pomocí ovladače, ke kterému je nutné nastavit cestu v adresářové struktuře. Jedná se o spustitelný soubor, který dokáže vyvolat a ovládat instance prohlížečů. Cesta je nastavena jako systémová proměnná. V kódu se pak využívají API pro jednotlivé prohlížeče, zpravidla pojmenované stejně jako ovladače. Ovladače jsou vydávány pravidelně při vytvoření nové verze prohlížeče, což má za následek, že každý ovladač umožňuje používat pouze verze, pro které byl vydán.

Aplikace při odevzdání podporuje tyto verze:

- Mozilla Firefox v. 50.1.0,
- Google Chrome v. 55.0,
- Internet Explorer v. 11.

V případě, že by při další aktualizaci některý z prohlížečů nefungoval a nedošlo přitom ke změně používaného API či ovladače, stačí pouze nahrát novější verzi ovladače do adresářové struktury aplikace pod stejným názvem (viz 5.3).

6.1.1 Mozilla Firefox

Mozilla Firefox je globálně jedním ze třech nejpoužívanějších prohlížečů[1]. Jedná se o multiplatformní webový prohlížeč, který je vyvíjen v rámci spolupráce dobrovolníků z Mozilla Corporation. Je postaven na rendrovacím jádře Gecko[2]. Zajímavostí je, že první verze Selenia byla uvedena jako plugin do prohlížečů Firefox, viz 4.4.1.1.

V aplikaci se používá pro práci s tímto prohlížečem verze Selenium WebDriver. Jelikož došlo při updatu prohlížeče z verze 46 na 47 k zásadním změnám v práci s profily, tak i pro Selenium se používají dva různé druhy ovladače. Do verze 46 stačilo pro práci s prohlížečem používat FirefoxDriver. Pokud uživatel aktualizoval na novější verzi, měl možnost ještě využít přenosnou verzi, je ovšem nutné nastavit cestu k této přenosné verzi.

Od verze 47 je nutné využít pro práci s prohlížečem MarionetteDriver. Ten se používá jako obálka pro ovladač GeckoDriver. Existují dva způsoby inicializace, první využívá původní FirefoxDriver a předá mu přes capabilities informaci o MarionetteDriveru. Druhý způsob používá přímo instanci MarionetteDriver. [24]

6.1.2 Google Chrome

Google Chrome je webový prohlížeč vyvíjený firmou Google, patří mezi nejpoužívanější prohlížeče. Založen je na rendrovacím jádře WebKit, respektive novější verze jsou postaveny na jeho odnoži Blink[3].

Pro práci se Seleniem se používá ChromeDriver, který je pravidelně aktualizovaný podobně jako verze prohlížeče. Často se stává, že jedna verze ovladače podporuje více verzí prohlížeče.

6.1.3 Internet Explorer

Jedná se o webový prohlížeč od společnosti Microsoft, který je součástí operačního systému Microsoft Windows. Z tohoto důvodu se stal jedním z nejpoužívanějších prohlížečů, ovšem počet jeho uživatelů každým rokem klesá. Poslední verze je 11, v nejnovějším operačním systému nahrazena novým prohlížečem Microsoft Edge, ovšem IE 11 se v něm stále vyskytuje z důvodu zpětné kompatibility.

Pro spolupráci se Seleniem se využívá IEDriver, jehož výhoda je, že se nemusí aktualizovat, protože s novější verzí IE se již nepočítá.

6.2 Podporované databáze

Primárně je ve společnosti Unicorn využívaná databáze MySQL. Proto když se rozhodovalo o podporovaných databázích, právě MySQL byla první volbou. Podpora dalších druhů da-

1. dle webu <http://gs.statcounter.com/>
2. bližší info na <https://developer.mozilla.org/en-US/docs/Gecko/FAQ>
3. bližší info na <http://www.chromium.org/blink>

tabází byla nepovinnou součástí, nakonec byla přidána ještě podpora pro databázi Apache Derby.

6.2.1 MySQL

Jde o celosvětově nejrozšířenější volně dostupnou relační databázi, která je vlastněna společností Sun Microsystems, dceřinou společností Oracle Corporation. Protože lze snadno implementovat, má dobrý výkon a je zdarma, tak je hojně využívaná pro použití ve webových aplikacích[4]. Z nejznámějších klientů, které využívají MySQL, lze uvést například Twitter, YouTube či Yahoo.

V rámci aplikace je komunikace mezi testy a databází umožněna pomocí třídy *DriverManager*, která naváže spojení s databází dle zadaných přihlašovacích údajů a adresy. Dále umožní volání jednotlivých dotazů a zpracování získaných výsledků.

6.2.2 Apache Derby

Apache Derby je volně dostupná relační databáze vyvíjena společností Apache Software Foundation[5]. Konfigurace databáze je snadná, proto je hojně využívaná na jednodušších projektech.

6.3 Programovací jazyk a vývojové prostředí

Jako programovací jazyk byla zvolena Java, protože se na většině projektů ve společnosti Unicorn používá. Dalším důvodem je dobře napsané API Selenia pro Javu včetně podrobné dokumentace. V neposlední řadě jsem vzal v úvahu své znalosti a preference k tomuto jazyku.

Pro tvorbu GUI bylo použito IDE Netbeans, které poskytuje grafický editor pro vytváření Swing aplikací. Co se týká zbytku kódu a obsluhy jednotlivých grafických prvků, ten byl napsán v IDE Eclipse Neon. Pro jeho volbu jsem se rozhodl na základě svých zkušeností a preferencí.

6.4 Apache Maven

Maven je nástroj pro správu, řízení a automatizaci sestavování aplikace. Primárně je používán pro aplikace psané v programovacím jazyce Java. Hlavní výhodou Mavenu je přídávání závislostí na cizí projekty, jejichž API chceme ve vlastní aplikaci využívat. Dále umožňuje přidávat závislosti mezi vlastními projekty a tím vytvářet stromovou strukturu, což nám umožní dekompozici na separátní části aplikace. [23]

4. bližší info na <https://www.mysql.com>
5. bližší info na <https://db.apache.org/derby/>

6.5 Git

Git je distribuovaný systém správy verzí, který byl vytvořen Linusem Torvaldsem[6]. Ten jej používal při vývoji jádra Linuxu. Git umožňuje spravovat verze aplikace v rámci celých týmů, každý člen tak může pracovat nad vlastní lokální větví a úpravy nahrávat do společné větve. Všechny úpravy jsou tak persistentně uloženy a každá verze je dostupná. Pod každou verzí jsou uloženy jen úpravy souborů, které proběhly a uživateli tak umožňuje zobrazit si změny, které se v daném commitu odehrály. [25]

6.6 Selenium

Selenium umožňuje spouštění generovaných testů ve vyvíjené aplikaci, z tohoto důvodu je vhodné, jej uvést mezi použité technologie. Zároveň je ale Seleniu věnována část 4.4.1, proto se jím dále v této kapitole nebudeme zabývat.

6.7 JAXB

JAXB[7] je technologie, která umožňuje mapování entit mezi Javou a XML soubory. Ke správnému namapování používá Java anotace, které označují jednotlivé XML elementy a atributy. Správně zapsaný XML soubor splňující XSD schéma je potom snadné převést do Java entit. [26]

6.8 Swing

Swing je knihovna uživatelských prvků pro platformu Java, které poskytují podporu pro desktopové grafické rozhraní. Pomocí Swingu lze vytvářet například okna, tlačítka, dialogy, seznamy apod. Původním nástrojem pro tvorbu grafického rozhraní bylo AWT[8], ovšem časem se přišlo na problémy (např. závislost na platformě), které způsobily přechod na Swing. [27]

6. finský programátor, zakladatel jádra OS Linux
7. Java Architecture for XML Binding
8. Abstract Window Toolkit, více na <http://www.javaworld.com/article/2077188/core-java/introduction-to-the-awt.html>

Kapitola 7

Implementace

V první části kapitoly se budeme věnovat jednotlivým modulům, do kterých je výsledná aplikace rozdělena. Úvodem bude vysvětlen princip funkčnosti jednotlivých modulů a jak spolu vzájemně kooperují. V druhé části se zaměříme na případy užití, jako je například vytváření testů či jejich spouštění.

7.1 Moduly aplikace

V této podkapitole jsou uvedeny moduly, do kterých byla aplikace rozdělena. Jako moduly jsou označeny jednotlivé projekty. Vytvořené moduly byly rozděleny podle logických částí vzniklé aplikace a navzájem mezi sebou spolupracují. Moduly jsou na sebe odkazovány pomocí Maven konfiguračních souborů (viz 6.4). Jejich sestavení má za úkol projekt *build_process*, který spustí sestavení současně na všech projektech, což zajistí jejich aktuálnost.

7.1.1 Core

Jádrem systému je modul *Core*, ve kterém najdeme společné funkce a služby pro celou aplikaci. Všechny moduly na něj mají referenci a mohou využívat jeho funkce.

V modulu nalezneme servisní třídu pro konfiguraci a soubor s konstantami, které jsou používané napříč celou aplikací. Další funkčností, kterou zde nalezneme, je vlastní logování do souboru. Využívá se při tom výchozího Java loggeru, kterému je předán definovaný soubor. Poslední třídou, kterou v tomto modulu nalezneme, poskytuje funkce pro načítání hodnot do elementů grafického rozhraní. Umožňuje načíst informace o vytvořených testech, jednotlivých průchodech a k nim přidruženým souborům. Přehledně je pak předá aplikačnímu modulu (jako seznam prvků), který je umožní zobrazit v seznamech na GUI.

7.1.2 Converter

Modul je určen pro převod testu z XML souboru do třídy napsané v Javě pomocí JAXB. Blíže se o této technologii dočtete v předchozí kapitole v části 6.7. Aby byl převod možný, je nutné mít vytvořeno XSD schéma, podle kterého se test validuje. Ten se podaří převést jen při úspěšné validaci. Toto schéma je součástí modulu *Converter*. Aby mohl být test převeden, je dále nutné mít vytvořenou hierarchii tříd, do kterých je XML namapováno. Protože by

bylo zdlouhavé a pracné tyto třídy psát ručně, využívá se pro vygenerování *jaxb2-maven-plugin*, který celou strukturu vytvoří za nás podle zvoleného XSD schématu. Díky tomu nemusíme po každé lehké úpravě XSD schématu upravovat strukturu tříd, ale pouze stačí spustit znovu proces sestavení.

7.1.3 Generator

Stěžejní funkcí tohoto modulu je vygenerování testu do spustitelné třídy. Vygenerovaná třída bude potomkem abstraktního testu a ve spustitelné metodě se bude nacházet posloupnost jednotlivých kroků, které byly zapsány v XML souboru.

Modul obsahuje hlavní servisní třídu, která má za účel vytvořit hlavičku, tělo i patičku generovaného testu. Pro vygenerování těla testu využívá dalších tříd z tohoto modulu, kde se pro vstup použije převedený test ze souboru XML. Můžeme zde tedy vidět kooperaci mezi moduly *Generator* a *Converter*. Pro každou akci je nutné vygenerovat odlišný kus kódu, proto byla logika rozdělena podle typu akcí (*SqlAction* a *Action*). Mimo to zde nalezneme i komponentu pro obecnou funkčnost, která napomáhá například obalit krok testu blokem, která umožní jeho případné přeskočení nebo přidání popisku do logovacího souboru.

Po vygenerování testu do vlastní třídy jej stále nemůžeme používat, protože se jedná o pouhý Java soubor, který ještě nebyl zkompilován. Proto je součástí tohoto modulu dynamický kompilátor, který umožní test zkompilovat a potom už jej můžeme používat ve zbytku aplikace. Pro úspěšnou kompilaci testu je nutné mít správně nastaveny systémové cesty pro JDK, blíže o prerekvizitách se dozvíte v příloze A.

7.1.4 Tester

Modul obsahuje funkčnost pro nastavení a spuštění vygenerovaných testů. Pojem nastavení je zde použit ve smyslu provedení určitých akcí před a po spuštění testu, čímž může být výběr a inicializace prohlížeče, otevření spojení do databáze nebo nastavení logovacího souboru. Pro nastavení prohlížeče je využívaná třída *DriverFactory*, ve které se zvolí prohlížeč z konfigurace a jsou mu upraveny výchozí vlastnosti pro lepší průběh testu (např. nastavení fullscreenu). Obdobně při použití databáze se přes třídu *DbFactory* nastaví použitá databáze a vytvoří se nové připojení.

V modulu se dále nachází abstraktní předek pro všechny generované testy, což nám umožní pro každý test pouštět obecné metody, které jsou pro všechny testy stejné. Vygenerovaný test poté jen přidá implementaci pro hlavní metodu, která spouští jednotlivé kroky. V předkovi se provádí metody před spuštěním a po ukončení testu.

Poslední komponentou je spouštěč testů, který načte zkompilovaný test ze složky a umožní jeho spuštění přes reflexi, tedy vyvolání metody za běhu aplikace. Na základě výsledku průchodu je vrácena výstupní hodnota, která je dále využita v aplikačním modulu.

7.1.5 Application

Aplikační modul obsahuje pouze jednu třídu, ovšem z pohledu uživatele tu nejzásadnější. Jedná se totiž o grafické rozhraní aplikace napsané s použitím technologie Swing (viz 6.8, tedy o rozhraní pro desktopovou aplikaci. Důvody pro volbu desktopové aplikace vysvětluji v sekci 5.1.2. Grafická stránka byla upravena v prostředí NetBeans, programový kód pro funkčnost prvků byl doplněn už klasicky v IDE Eclipse, stejně jako zbytek aplikace. Jednotlivé části aplikace budou postupně rozebrány v další podkapitole.

7.2 Detaily implementace a uživatelské rozhraní

V následují podkapitole se dočtete o jednotlivých případech užití, které se v aplikaci nacházejí. Naleznete zde popis obrazovek a jejich funkčností. Dále se dočtete o tvorbě testů či logování výsledků.

7.2.1 Tvorba testů

Testy jsou tvořeny zapsáním do souboru pomocí jazyka XML. Tento soubor je možné nahrát do aplikace, kde je z něj vygenerován test. Test musí být zapsán ve správné struktuře, což je validováno pomocí XSD[1] schématu. Schéma použité pro tvorbu testů si můžete prohlédnout v příloze B.

Ze schématu lze pochopit, že testy se skládají z jednotlivých akcí, přičemž akce můžeme rozdělit na *SqlAction* a *Action*. *SqlAction* akce se týkají pouze používání databáze, pokud tedy neplánujeme databázi používat, není nutné tyto akce využívat. Naproti tomu akce Action jsou využívány jak pro webové funkčnosti, tak pro pomocné funkce v Javě, např. čekací doby, ukládání do proměnných či různá porovnávání výsledků.

Zbytek podkapitoly bude věnován popisu jednotlivých metod, čtenář se zde dočte k čemu slouží a jak je správně zapsat do XML souboru, tak aby prošel validací. Nejdříve je vhodné rozebrat metody z akce *Action*, protože pomocí nich se testuje grafické rozhraní webové aplikace. Pro správné pochopení zápisu je nutné vysvětlit, z čeho se *Action* může skládat. Každá *Action* musí obsahovat název zvolené metody (method). Zbytek atributů je poté povinný či volitelný na základě zvolené metody. Mezi atributy, které dále může *Action* obsahovat, patří cesta k webovému elementu (path), dvě textové hodnoty použitelné pro různé účely (value a target), povinnost provést krok (skippableStep) a popis kroku pro záznam do logovacího souboru (description). Obecně lze poslední dvě hodnoty označit za volitelné, což znamená, že jejich vynecháním nezpůsobíme chybu při validaci. Povinnost vyplnění zbytku atributů (path, value a target) bude uvedena u každé metody.

Ještě je potřeba zmínit, že cestu k elementu lze vyhledat více způsoby, např. pomocí ID, jména, xpath apod. Proto než se pustíme do rozboru jednotlivých metod, bude vysvětleno vyhlédávání elementů. Na stránce lze elementy vyhledat pomocí více druhů identifikátorů a někdy lze dokonce jednotlivé prvky nacházet po aplikaci více pravidel. Proto lze cestu

1. XML Schema Definition

zapisovat v posloupnosti, ovšem pouze u určitých identifikátorů je posloupnost využita. A teď již k samotným identifikátorům:

- ID – Unikátní identifikátor, nalezneme ho podle HTML atributu id.
- NAME – Unikátní identifikátor, nalezneme ho podle HTML atributu name.
- XPATH – Unikátní cesta k prvku, která je generovaná na základě pozice umístění na stránce. Pokud se struktura stránky změní, může se změnit i xpath. Pro její sestavení můžeme například použít rozšíření FirePath doplňku FireBug. [28]
- CSS – Unikátní cesta k prvku, která je generovaná podle použitých stylů. Může se změnit, pokud se upraví struktura použitých stylů na stránce. Podobně jako xpath je možné ji sestravit pomocí rozšíření FirePath.
- LINK – Nalezení na základě textu odkazu, který by měl být unikátní.
- TAGNAME – Vyhledání podle značky elementu (tagu). Protože stejných značek může být na jedné stránce více, nejedná se o unikátní identifikátor. Proto lze vyhledání prvku zapisovat v posloupnosti kroků, přičemž se snažíme co nejvíce specifikovat hledaný element. V případě, že nedokážeme omezit vyhledání na přesně jeden prvek, je automaticky vybrán první nalezený po průchodu celé posloupnosti. Obecně lze zápis pomocí posloupnosti značek přirovnat k vyhledání pomocí XPATH, ovšem zápis může být lépe chápán a tím pádem i snáze opravitelný při změně struktury stránky. Pro lepší představu je uveden příklad zápisu posloupnosti značek:

```
<path>
  <type>TAGNAME</type>
  <value>div</value>
</path>
<path>
  <type>TAGNAME</type>
  <value>div</value>
</path>
<path>
  <type>TAGNAME</type>
  <value>p</value>
</path>
```

- CLASS – Vyhledání na základě použitých atributů class u jednotlivých elementů. Atribut class se používá pro styl, nemusí být tedy unikátní pro každý element, proto lze podobně jako vyhledávání podle TAGNAME psát do posloupnosti. Posloupnost nemusí obsahovat pouze vyhledání podle CLASS, ale může být kombinována s vyhledáním podle TAGNAME, což nám umožní snadnější vyhledání prvku.
- ELEMENT – Použití tohoto identifikátoru nám vybere element z již uložené proměnné v kódu. Jedná se tedy už o dříve nalezený element podle jiného pravidla,

který byl uložen do seznamu elementů používaných pod zadaným názvem. Tento způsob používání elementů je pro uživatele nejpřívětivější v případě, že chceme s prvkem pracovat opakovaně a nechceme jej stále vyhledávat pod jinými identifikátory. Uživateli si stačí pamatovat pouze jméno proměnné a může prvek opakovaně využívat.

Tímto jsme si prošli všechny způsoby, jak nalézt prvky stránky a v následující části si představíme samotné metody *Action*:

- OPEN – Otevře nebo přesměruje prohlížeč na zadanou url adresu, která je vyplněná ve značce value.
- TYPE – Umožní do nalezeného elementu zapsat požadovanou hodnotu. Text, který se do elementu zapisuje vložíme do hodnoty value.
- CLICK – Klikne na vyhledaný element.
- CLICK_AND_WAIT – Funguje stejně jako metoda CLICK, pouze umožní zadat dobu vyčkání než se provede další akce. Tuto dobu uživatel zadává do hodnoty value.
- ENTER – Simuluje stisknutí klávesy enter ve zvoleném elementu.
- CLEAR – Vymaže obsah vyhledaného elementu. Používá se pro textová pole obsahující text, který chceme odstranit.
- SELECT_VALUE – Vybere hodnotu z rozbalovacího seznamu na základě zadané hodnoty ve značce value.
- SCREENSHOT – Metoda nejprve vytvoří nový soubor, který bude obsahovat snímek obrazovky. Dále tento soubor překopíruje do složky s logovacím souborem pro průchod testu a nastaví mu nový název. Ten se skládá z identifikátoru průchodu (popsáno v části 7.2.2 v posledním odstavci) a uživatelem definované hodnoty ve značce value.
- GO_BACK – Vrátí prohlížeč na předchozí stránku, stejně jako kdybychom klikli na tlačítko back v prohlížeči.
- GO_FORWARD – Posune prohlížeč na následující stránku, stejně jako tlačítko forward v prohlížeči. Pokud se nelze posunout dopředu, neprovede se žádná akce.
- WAIT – Pokud chceme mezi kroky vytvořit úmyslnou časovou mezeru, lze ji zapsat pomocí této metody. Do značky value nastavíme čekací dobu v milisekundách, která uspí na danou dobu prováděcí vlákno.
- ASSERT_TITLE – Slouží k porovnání titulku stránky s textovou hodnotou zadanou ve value.

- ASSERT_TEXT – Porovná text získaný z nalezeného elementu s hodnotou value.
- FIND_ELEMENT – Nalezne element podle zadané cesty (zmíněno v předchozí části týkající se zadávání cest) a uloží jej do seznamu nalezených elementů pod proměnnou ze značky value.
- ATTRIBUTE_TO_VARIABLE – Získá hodnotu atributu z požadovaného elementu (název atributu vloží uživatel do značky value) a uloží ji do seznamu proměnných, které se dál mohou využívat pro zbytek testu. Uložení do seznamu probíhá pod názvem, který je zadán ve značce target.
- ASSERT_VARIABLE_WITH_TEXT – Porovná hodnotu uloženou v proměnné (její jméno použijeme ze značky target) se zadanou hodnotou ze značky value.
- ASSERT_ATTRIBUTE_WITH_TEXT – Funguje stejně jako předchozí metoda, pouze s tím rozdílem, že se nepoužije hodnota z proměnné, ale přímo z atributu elementu. Je tedy nutné vyplnit cestu k elementu a název požadovaného atributu ve značce value. Hodnota, se kterou se porovnává, je vyplněna ve značce target.
- ASSERT_TWO_VARIABLES – Porovná dvě hodnoty uložené v seznamu proměnných, jejich názvy jsou definovány ve značkách value a target.
- TEXT_TO_VARIABLE – Uloží text z nalezeného elementu do proměnné pod názvem ze značky value.
- ASSERT_SQL_AND_VARIABLE – Tato metoda je překlenovací mezi dvěmi typy akcí *Action* a *SqlAction*. Umožňuje porovnat hodnotu získanou z databáze a uloženou v seznamu proměnných pro sql data, s hodnotou ze seznamu proměnných pro informace o elementech. Příkladem může být například porovnání přihlášeného uživatele (login) z elementu stránky s hodnotou uloženou v databázi.

Ve všech metodách, které jsou určeny k porovnávání, jsou použity funkce z třídy *Assert*. Ta se používá v jednotkových testech a v případě, že porovnání není splněno, tak je i výsledek testu označen za neúspěšný. Stejného principu se využívá i v metodách definovaných v XML schématu, tedy pokud skončí porovnání neúspěchem, je vykonávání testu ukončeno na daném kroku. Výjimkou může být jen případ, kdy je krok označen za nepovinný a v případě jeho neúspěchu se pokračuje dále v provádění zbytku testu.

Druhý typ akcí je zaměřen čistě na používání s databází, bez nastavené databáze se tedy ani test neprovede. Obecně lze říci, že se jedná o CRUD operace[2], které lze zapsat dvojím způsobem. Prvním způsobem je použití přímo napsaného sql dotazu, druhý způsob se snaží vytvořit sql dotaz dynamicky na základě údajů zapsaných v XML dokumentu. V následující části budou blíže rozebrány jednotlivé metody, které lze využít pro akce *SqlAction*:

2. create, retrieve, update, delete (tedy operace na vytváření, získávání, aktualizaci a mazání záznamů)

- DIRECT_QUERY – Uživatel napíše přesný sql dotaz do značky query, ten je poté proveden bez jakékoli návratové hodnoty. Proto se tento typ metody používá pro operace nezávislé na programovém kódu.
- SELECT_DIRECT_QUERY – Protože předchozí typ metody neumožňuje návratovou hodnotu, byla pro dotaz SELECT vytvořena oddělená metoda. Uživatel do značky query napíše přímý sql dotaz, který se má provést. Po načtení hodnot je ale výsledek uložen do seznamu proměnných, a to na základě názvů nastavených ve značce what.
- INSERT – Vloží zadané hodnoty do tabulky (značka table). Informace o tom, do kterých sloupců a jaké hodnoty se mají vložit, se zapisuje do značky what.
- UPDATE – Aktualizuje hodnoty v tabulce (značka table), které se načtou ze značky what. Omezení aktualizovaných hodnot lze nastavit pomocí značky conditional, která vytvoří v sql dotazu WHERE klauzuli.
- DELETE – Smaže záznamy z tabulky (značka table). Pokud chceme smazat pouze určité záznamy, lze vytvořit WHERE klauzuli jejíž nastavení se zapíše do značky conditional.
- SELECT – Sestaví dotaz pro výběr hodnot z definované tabulky (značka table). Ve značce what se definují sloupce, které mají být dotazem vráceny a dále se v ní nastaví jména proměnných, do kterých jsou hodnoty ze sloupců uloženy. Pro větší specifikaci výsledků můžeme použít WHERE klauzuli zadanou ve značce conditional.

Metody INSERT, UPDATE, DELETE a SELECT se snaží o poskládání sql dotazu na základě vyplněných značek, což bylo u jednotlivých metod popsáno. Stále ale není jasné jak přesně má vypadat struktura značek what a conditional. Značka what se používá pro označení sloupců, které chceme v databází nalézt nebo které chceme upravit. Další použití je pro nastavení jmen proměnných, do kterých se uloží hodnoty ze získaných sloupců. Naproti tomu značka conditional se používá pro podmínky, které se používají ve WHERE klauzuli.

Co se týče struktury značky what, ta se skládá ze dvou dalších značek column a value. Do značky column se zapisuje název sloupce tabulky, který chceme použít a do value se ukládá typ a hodnota, pod kterou se bude sloupec ukládat či používat v dotazech. Styl zápisu bude více zřejmý z příkladů, které se nachází v následující kapitole (viz 8.1).

Zápis struktury značky conditional je podobně složitý jako značky what, protože využívá stejné elementy column a value. Navíc ještě obsahuje dvě další značky cond_inner a cond_outer, které spojují jednotlivé podmínky logickými spojkami, přičemž cond_inner značí vnitřní podmínku mezi sloupcem a porovnávanou hodnotou. Značka cond_outer obsahuje vnější podmínku, která spojuje více podmínek mezi sebou.

Posledními značkami, jejichž princip a způsob zápisu byl zatím vynechán jsou skippableStep a description. Tyto značky nejsou povinné a mohou být použity u všech typů akcí. Značka skippableStep označuje krok testu, který může být přeskočen a pokud jeho provedení způsobí nějakou chybu, tak je zalogována, ale pokračuje se dál ve vykonávání testu.

Druhou značkou je description, která přidává k danému kroku bonusový popis, jenž je vypsán do testovacího souboru. Tím umožní uživateli lépe identifikovat, které kroky skončili úspěšně a které naopak skončili chybou.

7.2.2 Logování aplikace a testů

V průběhu celé aplikace mohou nastávat různé výjimky, ať už při převodu testu z XML souboru do Javy, dynamické kompilaci či spuštění testu. Každá nečekaná údalost, ale i úspěšné provedení, musí být zalogováno, aby uživatel věděl, co se s aplikací nebo testem děje. Logy tedy můžeme rozdělit na dva základní typy, z nichž první se týká celé aplikace a druhý pouze jednotlivých průchodů testu.

Aplikační log je ukládán do jednoho souboru a pokud je aplikace spuštěna opakovaně, tak se navazuje na konec logovacího souboru. Logovány jsou téměř všechny akce na GUI aplikace a možné výjimky při standardních operacích, které jsou ošetřeny try .. catch bloky. Výhodou ukládání logu do souboru je, že údaje nejsou po vypnutí aplikace smazány a uživatel je může studovat i po ukončení aplikace.

Druhý typ logů, týkající se jednotlivých průchodů testu, loguje pouze informace o úspěšném či neúspěšném proběhnutí každého kroku. Pro každý průchod se vytvoří nový soubor, ten obsahuje pro každý krok jeho pořadí a informaci zda skončil úspěšně či neúspěšně. Navíc může ještě obsahovat poznámku ze značky description (viz 7.2.1). V případě, že krok skončí neúspěšně, je v logovacím souboru přidána i výjimka, která pád způsobila. Pokud je možné krok přeskočit v případě chyby, zaloguje se výjimka také, ale pokračuje se logováním dalších kroků.

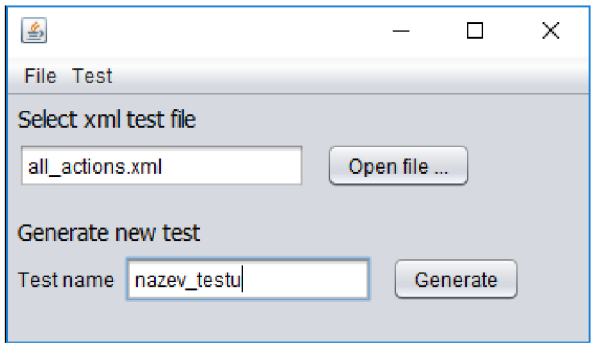
Co se týče názvu logovacích souborů, název souboru pro logování celé aplikace je *application.log*. Pro spuštěné testy jsou vytvářeny logovací soubory, které v názvu obsahují název testu a časovou známku (datum a čas spuštění), která identifikuje unikátně daný průchod.

7.2.3 Obrazovky aplikace

Jelikož se v aplikaci vyskytují jen 4 obrazovky, které navíc nejsou příliš složité, bude v následující podčásti stručně vysvětleno k čemu slouží a jak je používat.

7.2.3.1 Import testu

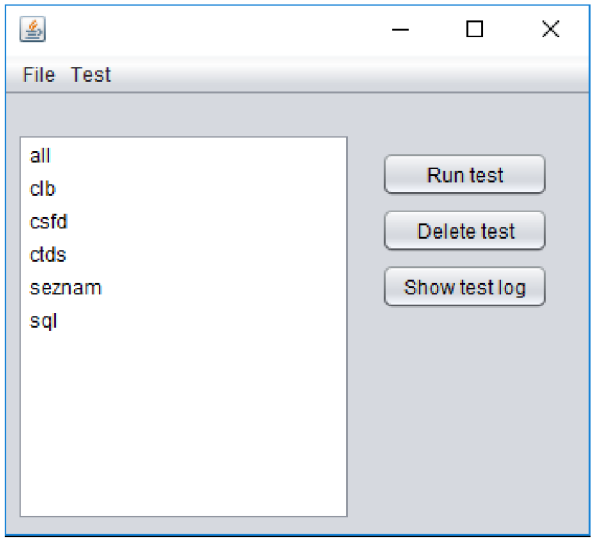
Na této obrazovce může uživatel vybrat soubor, který obsahuje zápis testu v XML. Po výběru souboru je nutné nastavit jméno generovaného testu a pomocí tlačítka spustit generování. Pokud již v aplikaci existuje test se stejným jménem, je uživatel o této skutečnosti informován a může si zvolit zda chce původní test přepsat. V případě, že během generování testu nenastane žádná chyba, je uživatel informován, že byl test vygenerován v pořádku. V opačném případě je informován o vzniklé chybě.



Obrázek 7.1: Obrazovka import testu

7.2.3.2 Spuštění testu

V levé části obrazovky se nalézá seznam s vygenerovanými testy, které je možné spouštět. Po výběru testu má uživatel na výběr ze tří možností: spuštění testu, smazání testu a prohlídka logovacího souboru. Možnost spuštění testu způsobí otevření prohlížeče a provedení zvoleného testu na základě definovaných kroků a nastavení. Pokud uživatel vybere smazání testu, je mu zobrazena potvrzující hláška a v případě kladné odpovědi je test odstraněn. Logovací soubory zůstanou zachovány pro případ, že by si je chtěl uživatel dále procházet, ovšem jsou mu dostupné už jen přes adresářovou strukturu, nikoliv přes obrazovku. Poslední možností je zobrazení logovacích souborů pro vybraný test, který otevře novou obrazovku (viz 7.2.3.3).



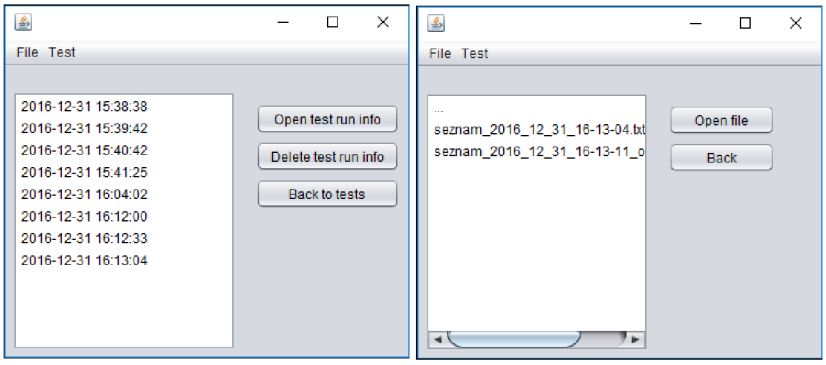
Obrázek 7.2: Obrazovka spouštění testu

7.2.3.3 Průchod logovacích souborů

Na první pohled je tato obrazovka velice podobná spouštění testů. V levé části opět nalezneme prvek pro zobrazení seznamu, který nyní obsahuje seznam všech průchodů testu, které byly provedeny. Je zobrazen vždy ve formátu data a času, tak aby uživatel mohl jednoznačně identifikovat, kdy se daný průchod testu odehrál. Po výběru průchodu má uživatel na výběr z více možností. Kliknutím na tlačítko smazat, vymaže všechny údaje o daném průchodu včetně souborů, které jsou na průchod navázány (snímky obrazovky). Kliknutím na tlačítko návratu se vrátíme na obrazovku se spouštěním testů.

Nejzajímavější je možnost zobrazit si detaily vybraného průchodu. Po zvolení této možnosti se lehce upraví grafické zobrazení obrazovky. V seznamu nyní vidíme název logovacího souboru a zbytek souborů navázaných na daný průchod. Pokud si některý ze souborů vybereme, tak jej můžeme nechat otevřít ve výchozím programu pro tento typ souboru. Tlačítkem zpět se vrátíme na předchozí obrazovku s výběrem průchodů.

Tento případ užití je koncipován podobně jako souborový systém, tedy umožňuje otevřít možnost po dvojkliku na element, případně se vrátit na předchozí stránku dvojklikem na možnost „...“. Uživateli to umožňuje další možnost pohybu na obrazovce, které je z jeho pohledu intuitivní. Obecně lze tento případ užití považovat za ulehčení práce uživateli při zobrazení výsledků testů, jelikož pouze zobrazuje data, které lze najít v adresářové struktuře.



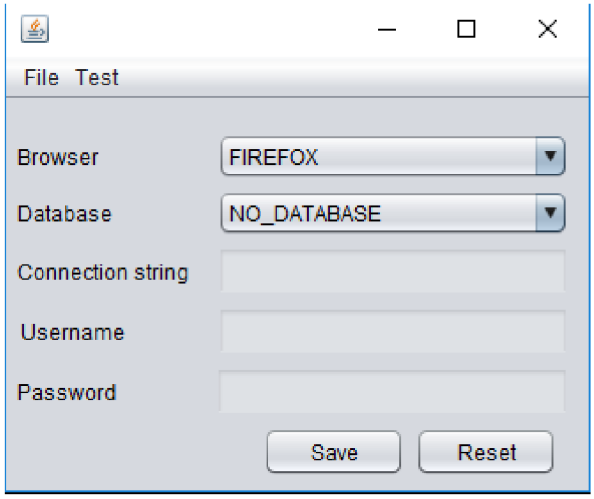
Obrázek 7.3: Obrazovky průchodu a detailu logovacích souborů

7.2.3.4 Konfigurace

Poslední obrazovkou je konfigurace, ve které má uživatel možnost nastavit prohlížeč pro spouštění testů a databázi. Hodnoty jsou načítány z konfiguračního souboru a změny, které uživatel provede a potvrdí, se opět do tohoto souboru uloží. Pro snadný výběr prohlížeče slouží rozbalovací seznam, který obsahuje pouze podporované prohlížeče a neumožní tedy uživateli zadat nějaký z nepodporovaných. Výběr použité databáze probíhá obdobně, tedy ze seznamu se vybere správný typ databáze. Navíc se ale nastavují hodnoty pro připo-

jení, kterými jsou uživatelské jméno, heslo a připojovací řetězec (connection string). Tyto hodnoty se zapisují do samostaných políček. U volby databáze můžeme nastavit možnost NO_DATABASE, která označuje, že se nebude používat žádná databáze, a tedy ani nepůjdou spustit testy, které obsahují akce *SqlAction*.

Na konci obrazovky se nacházejí dvě tlačítka. Tlačítkem „Reset“ se vrátí všechny změny do původního stavu bez jakékoliv změny v konfiguračním souboru. Naopak po kliknutí na tlačitko „Save“ se hodnoty z formuláře pokusí aplikace zvalidovat a pokud validace proběhne úspěšně, jsou hodnoty uloženy do konfiguračního souboru. Validace především vyzkouší zda pro zadané údaje lze návazat spojení do databáze.



Obrázek 7.4: Obrazovka konfigurace

```
Test step 1 finished with result: SUCCESS (otevře stránku csfd)
Test step 2 finished with result: SUCCESS
Test step 3 finished with result: SUCCESS
Test step 4 finished with result: FAIL
no such element: Unable to locate element: {"method":"xpath",
  "selector":".//*[@id='news-4591']/div[1]/a"}
  (Session info: chrome=54.0.2840.99)
  (Driver info: chromedriver=2.25.426923
  (0390b88869384d6eb0d5d09729679f934aab9eed),
  platform=Windows NT 10.0.10586 x86)
  (WARNING: The server did not provide any stacktrace information)
Command duration or timeout: 172 milliseconds
For documentation on this error, please visit:
http://seleniumhq.org/exceptions/no_such_element.html
Build info: version: '3.0.1', revision: '1969d75',
time: '2016-10-18 09:49:13 -0700'
System info: host: 'Jakub-PC', ip: '192.168.0.101',
os.name: 'Windows 10', os.arch: 'x86', os.version: '10.0',
java.version: '1.8.0_101'
Driver info: org.openqa.selenium.chrome.ChromeDriver
Capabilities [{applicationCacheEnabled=false, rotatable=false,
mobileEmulationEnabled=false, networkConnectionEnabled=false,
chrome={chromedriverVersion=2.25.426923
(0390b88869384d6eb0d5d09729679f934aab9eed),
userDataDir=C:\Users\Jakub\AppData\Local\Temp\scoped_dir18584_25794},
takesHeapSnapshot=true, pageLoadStrategy=normal, databaseEnabled=false,
handlesAlerts=true, hasTouchScreen=false, version=54.0.2840.99,
platform=XP, browserConnectionEnabled=false, nativeEvents=true,
acceptSslCerts=true, locationContextEnabled=true,
webStorageEnabled=true, browserName=chrome, takesScreenshot=true,
javascriptEnabled=true, cssSelectorsEnabled=true}]
Session ID: d4553d0b0a8c66656b209a53b8b1c7f8
*** Element info: {Using=xpath, value=.//*[@id='news-4591']/div[1]/a}
Test step 5 finished with result: SUCCESS (vyhledá film Kingsman)
Test step 6 finished with result: SUCCESS
Test step 7 finished with result: SUCCESS
Test step 8 finished with result: SUCCESS
Test step 9 finished with result: SUCCESS
```

Příklad 7.2.1: příklad výpisu testu s neúspěným nepovinným krokem

Kapitola 8

Praktické použití aplikace

Kapitola se věnuje praktickému použití vzniklé aplikace. Uvedeny jsou příklady vzniklých testů, které byly vytvořeny pro testování některých aplikací společnosti Unicorn. Dále jsou uvedeny důvody, proč lze aplikaci používat i na jiných projektech i v jiných firmách.

8.1 Příklady testů

Při testování aplikace ve společnosti Unicorn vzniklo pár testů, které zkoušeli jednotlivé funkce webového rozhraní a připojení do databáze. V této podkapitole budou uvedeny popisy demonstrativních příkladů, které naleznete v přílohové části práce (viz C, D, E). Popis se týká jednoho testu pro webové rozhraní, jednoho pro testování databáze a jednoho testu, který obsahuje oba dva typy akcí.

8.1.1 Test akce Action

V testu se nejprve přihlásí uživatel jako administrátor a následně se nastaví základní konfigurace, s kterou testeři spouští zbytek testů. Po úspěšném uložení konfigurace se zkontroluje, podle jiného případu užití, správně nastavený pracovní den.

8.1.2 Test akce SqlAction

V krocích testu je nejprve zjištěn pracovní den modelu se zadaným názvem, následně je pod tímto dnem vložen nový model a v dalším kroku jsou upraveny jeho vlastnosti. Tento model může být použit pro výpočty ve zbytku aplikaci. Pro demonstraci byl ještě doplněn poslední krok, který model z databáze smaže a tím po sobě uklidí.

8.1.3 Kombinovaný test akcí

Na začátku je uživatel znovu přihlášen do systému, tentokrát ale pod rolí operátor. Otevře se vytváření filtru, ve kterém se nastaví jeho preference a uloží se. Poté je dle názvu vytvořeného filtru zkontrolován nový záznam v databázi a zda byl filtr exportován. Nakonec je filtr odstraněn, aby nezpůsoboval problémy při opakovaném průchodu testu.

8.2 Použití na projektech

Vytváření testů v XML a jejich spouštění z aplikace bylo testováno na systému AMICA. Pár dalších testů bylo vytvořeno také pro systém CLB. Oba tyto systémy jsou dlouhodobě tvořeny ve společnosti Unicorn. Na obou projektech se aplikace osvědčila při testování jednodušších částí aplikací. Testování probíhalo na operačních systémech Windows.

Kromě zmíněných projektů, ale proběhlo testování i na běžných serverech, jako je seznam, csfd, google či serialzone. Na všech serverech šlo bez problému otestovat základní funkčnosti, proto se lze domnívat, že může být aplikace použita i na jiných projektech. Tohle tvrzení dokazuje i to, že byla aplikace poskytnuta dalšímu testerovi z jiné firmy, kde aplikaci rovněž vyzkoušel a setkal se s kladnými výsledky.

Na další způsob, jak využít vygenerované testy, mne přivedl tester, který má ve společnosti Unicorn na starost psaní automatizovaných testů. Když mu bylo vysvětleno, jak aplikace funguje, přišel na využití vygenerovaných testů do Java souboru, tedy ještě před zkompilováním. Metoda, která je v něm zapsána, lze použít jako kostra složitějšího testu. Může být obohacena o další složitější metody, které přidají možnosti pro složitější funkce. Vyvíjená aplikace nebyla pro tohle využítí primárně určena, ale schopný tester se znalostmi Selenia a automatizovaných testů, ji takhle může použít. Příklad vygenerovaného testu můžete vidět v příloze F.

Kapitola 9

Závěr

Cílem práce bylo vytvořit nástroj, který umožní uživateli zapsat, vygenerovat a spustit automatizovaný test bez znalosti programovacího jazyka či testovacího frameworku. Nástroj, který v rámci práce vznikl, tyto požadavky splňuje. Kromě testování GUI umožňuje i spolupráci s databází, což přináší další možnosti jak otestovat aplikaci. Program lze spustit na operačních systémech Windows a Linux, a to díky Javě, která není platformově závislá.

Co se týče struktury práce, tak v první části jsou zmíněny způsoby a druhy testování, nástroje pro automatizované testování a provedená analýza s návrhem vzniklé aplikace včetně použitých technologií. V druhé části práce se pozornost zaměřuje na konkrétní implementaci a rozdělení do modulů. Také jsou zde uvedeny praktické výsledky získané při testování aplikace.

Aplikace umožňuje především testování jednodušších případů užití, protože je důraz kladen na obecnost a možnost nasazení na nejrůznějších projektech. Proto je důležité aplikaci dále rozvíjet a rozšiřovat o nové funkce, aby poskytovala uživateli ještě více možností pro testování. Jednou z takových funkcí je stahování souborů, které je ovšem komplikované z důvodu odlišného chování pro jednotlivé prohlížeče. Proto je potřeba vymyslet způsob, jakým ukládání souborů poskytnout. Další funkčností, která se nabízí, je rozšíření podporovaných prohlížečů a databází. Podporu prohlížečů lze rozšířit pomocí nových ovladačů, ovšem každé přidání nového ovladače vyžaduje zásah do kódu aplikace.

Literatura

[1] *O společnosti*, Unicorn a.s., [online], [cit. 2016-12-01], <https://unicorn.com/cz/company-profile>. 2.1

[2] Patton, R.: *Testování softwaru*, Computer Press, Praha, 2002, 80-7226-636-5. 3, 3.1

[3] Dijkstra, E.: *The humble programmer*, [online], [cit. 2016-12-03], <https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF>. 3.1.3

[4] Beizer, B.: *Software Testing Techniques*, Dreamtech, 2003, 9788177222609. 3.1.5

[5] Link, J.: *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufmann, 2003, 9780080520179. 3.2.2

[6] Astels, D.: *Test-driven Development: A Practical Guide*, Prentice Hall PTR, 2003, 9780131016491. 3.2.2

[7] Jorgensen, P.: *Software Testing: A Craftsman's Approach, Fourth Edition*, CRC Press, 2013, 9781466560697. 3.4.1

[8] *Introduction*, Selenium Project, [online], [cit. 2016-12-09], <http://www.seleniumhq.org/docs/01_introducing_selenium.jsp>. 4.4.1

[9] *Selenium WebDriver*, Selenium Project, [online], [cit. 2016-12-09], <http://seleniumhq.org/docs/03_webdriver.html>. 4.4.1.3

[10] De Busscher, R.: *Browser compatibility with Webdriver*, [online], [cit. 2016-12-09], <http://jsfcorner.blogspot.cz/2011/05/browser-compatibility-with-webdriver.html>. 4.4.1.3

[11] *How it works*, Selenium Project, [online], [cit. 2016-12-10], <http://selenium-grid.seleniumhq.org/how_it_works.html>. 4.4.1.4

[12] Fang, J.: *Tellurium User Guide: Tellurium UI Objects*, Selenium Project, [online], [cit. 2016-12-10], <http://code.google.com/p/aost/wiki/UserGuide070UIObjects>. 4.4.2

[13] Mongolu, V.: *Tellurium Automated Testing Framework*, Selenium Project, [online], [cit. 2016-12-10], <http://www.methodsandtools.com/tools/tools.php?tellurium>. 4.4.2

[14] Fang, J.: *Introducing the Tellurium Automated Testing Framework*, Selenium Project, [online], [cit. 2016-12-10], <https://www.infoq.com/articles/tellurium_intro>. 4.4.2

[15] *Getting Started with Telerik Testing Framework*, Telerik, [online], [cit. 2016-12-11], <http://docs.telerik.com/teststudio/testing-framework/getting-started>. 4.4.3

[16] Osherove, R.: *The Art of Unit Testing*, Manning Publications Co., 2009, 978-1-933988-27-6. 4.4.4

[17] *WatiN Overview*, WatiN, [online], [cit. 2016-12-20], <http://watin.org/>. 4.4.4

[18] *TestComplete Automated Testing*, SmartBear Software, [online], [cit. 2016-12-20], <http://smartbear.com/products/qa-tools/automated-testing-tools>. 4.4.5

[19] *User Guide*, Ranorex GmbH, [online], [cit. 2016-12-20], <http://www.ranorex.com/support/user-guide-20.html>. 4.4.6

[20] *Protractor - end to end testing for AngularJS*, Protractor, [online], [cit. 2016-12-20], <http://www.protractortest.org/>. 4.4.7

[21] Windle, D. a Abreo, L.: *Software Requirements Using the Unified Process: A Practical Approach*, Prentice Hall Professional, 2003, 9780130969729. 5

[22] Weilkiens, T.: *Systems Engineering with SysML/UML Modeling, Analysis, Design*, Burlington: Elsevier, 2008, 978-008-0558-318. 5.1, 5.2

[23] Sonatype Company: *Maven: The Definitive Guide: The Definitive Guide*, "O'Reilly Media, Inc.", 2008, 9780596551780. 6.4

[24] ALAN: *How to use the Firefox Marionette Driver*, Selenium Simplified, [online], [cit. 2016-12-25], <http://seleniumsimplified.com/2016/04/how-to-use-the-firefox-marionette-driver/>. 6.1.1

[25] Loeliger, J. a McCullough, M.: *Version Control with Git: Powerful tools and techniques for collaborative software development*, "O'Reilly Media, Inc.", 2012, 9781449345044. 6.5

[26] McLaughlin, B.: *Java and XML Data Binding*, "O'Reilly Media, Inc.", 2002, 9780596002787. 6.7

[27] Loy, M. a Eckstein, R.: *Java Swing*, "O'Reilly Media, Inc.", 2002, 9780596004088. 6.8

[28] Bradens, L.: *Testing XPath and CSS locators FirePath style across browsers*, Selenium Simplified, [online], [cit. 2016-12-27], <https://autumnator.wordpress.com/2013/05/02/testing-xpath-and-css-locators-firepath-style-across-browsers/>. 7.2.1

Rejstřík

F
framework, 1, 4, 11, 12, 16, 19, 21–23, 52

G
Google, 19, 21, 23, 35, 51
Google Chrome, 5, 20, 22, 27, 34, 35

H
HTML, 18, 41

I
Internet Explorer, 5, 20, 22, 23, 27, 34

J
Java, 4, 18, 19, 25, 28, 31, 36–40, 45, 51, 52
JAXB, 37, 38
JDK, 39

M
Maven, 36, 38
mock, 11
Mozilla Firefox, 5, 18–23, 27, 34, 35
MySQL, 27, 35, 36

S
Selenium, 1, 4, 18, 19, 21, 22, 25, 34–37, 51
    Grid, 21
    IDE, 18, 19, 21, 24
    RC, 19–21
    WebDriver, 20, 22, 34, 35
software, 1, 6–10, 13, 14, 16
Swing, 36, 37, 40

U
UI, 17, 21
Unicorn, 1–4, 34–36, 50, 51

W
Windows, 18, 22, 24, 30, 35, 51, 52

X
XML, 1, 2, 4, 5, 25, 28, 37–40, 43, 45, 51
XSD, 2, 4, 25, 28, 37–40

Příloha A

Prerekvizity a návod na spuštění

Pro správný běh aplikace je vyžadováno mít nainstalovaný JDK 1.8, který je vyžadován ke spuštění, ale také k dynamické kompilaci vzniklých testů. Dále je nutné nastavit systémové proměnné:

- JAVA_HOME – cesta do adresáře s nainstalovaným JDK (např. C:\Program Files\Java\jdk1.8.0_111),
- Path – mimo jiné musí obsahovat cestu k JDK včetně adresáře bin (např. C:\Program Files\Java\jdk1.8.0_111\bin), ideálně uvedenou na prvním místě, tak aby nebyla ovlivněna jinou cestou.

Dále je nutné mít nainstalované prohlížeče, které chceme používat, a to ve verzích, které jsou podporované. Případně lze dohrát novější ovladače, jak je popsáno v 6.1. Pokud chceme používat i databázi, musí být spuštěná a obsahovat všechny schémata a tabulky, které v testech požadujeme.

A.1 Spuštění aplikace

Jelikož se jedná o Java aplikaci, lze ji spustit nezávisle na operačním systému (podpora je zajištěna pro Windows a Linux). Přes příkazový řádek lze aplikaci spustit příkazem:

```
java -cp application.jar cz.fi.muni.automation.application.Application
```

Příloha B

XSD schéma

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fi.muni.cz/automation"
  xmlns:tns="http://fi.muni.cz/automation" elementFormDefault="qualified">
  <xs:simpleType name="PathType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="ID" />
      <xs:enumeration value="NAME" />
      <xs:enumeration value="XPATH" />
      <xs:enumeration value="CLASS" />
      <xs:enumeration value="TAGNAME" />
      <xs:enumeration value="LINK" />
      <xs:enumeration value="CSS" />
      <xs:enumeration value="ELEMENT" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ActionMethod">
    <xs:restriction base="xs:string">
      <xs:enumeration value="CLICK" />
      <xs:enumeration value="CLICK_AND_WAIT" />
      <xs:enumeration value="TYPE" />
      <xs:enumeration value="OPEN" />
      <xs:enumeration value="ENTER" />
      <xs:enumeration value="CLEAR" />
      <xs:enumeration value="SELECT_VALUE" />
      <xs:enumeration value="SCREENSHOT" />
      <xs:enumeration value="ASSERT_TEXT" />
      <xs:enumeration value="ASSERT_TITLE" />
      <xs:enumeration value="GO_BACK" />
      <xs:enumeration value="GO_FORWARD" />
      <xs:enumeration value="WAIT" />
      <xs:enumeration value="FIND_ELEMENT" />
      <xs:enumeration value="TEXT_TO_VARIABLE" />
      <xs:enumeration value="ATTRIBUTE_TO_VARIABLE" />
      <xs:enumeration value="ASSERT_VARIABLE_WITH_TEXT" />
      <xs:enumeration value="ASSERT_TWO_VARIABLES" />
      <xs:enumeration value="ASSERT_ATTRIBUTE_WITH_TEXT" />
      <xs:enumeration value="ASSERT_SQL_AND_VARIABLE" />
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="SqlType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="SELECT" />
      <xs:enumeration value="UPDATE" />
      <xs:enumeration value="INSERT" />
      <xs:enumeration value="DELETE" />
      <xs:enumeration value="DIRECT_QUERY" />
      <xs:enumeration value="SELECT_DIRECT_QUERY" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="InnerCondType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="EQUAL" />
      <xs:enumeration value="NOT_EQUAL" />
      <xs:enumeration value="IN" />
      <xs:enumeration value="NOT_IN" />
      <xs:enumeration value="BETWEEN" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="OuterCondType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="OR" />
      <xs:enumeration value="AND" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ValueInfoType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="NUMBER" />
      <xs:enumeration value="STRING" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="VariableInfoType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="SQL" />
      <xs:enumeration value="NON_SQL" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="variableInfo">
    <xs:sequence>
      <xs:element name="name" type="xs:string"></xs:element>
      <xs:element name="type" type="tns:VariableInfoType"></xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="valueInfo">
    <xs:sequence>
      <xs:choice minOccurs="1" maxOccurs="1">
```

```
        <xs:element name="text" type="xs:string" />
        <xs:element name="variable" type="tns:variableInfo" />
      </xs:choice>
    <xs:element name="type" type="tns:ValueInfoType" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="action">
  <xs:sequence>
    <xs:element name="method" type="tns:ActionMethod" />
    <xs:element name="value" type="xs:string" />
    <xs:element name="path" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="type" type="tns:PathType" />
          <xs:element name="value" type="xs:string" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="target" type="xs:string" />
    <xs:element name="skippableStep" type="xs:boolean"
      default="false"></xs:element>
    <xs:element name="description" type="xs:string"></xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="sqlAction">
  <xs:sequence>
    <xs:element name="type" type="tns:SqlType"></xs:element>
    <xs:element name="table" type="xs:string"></xs:element>
    <xs:element name="query" type="xs:string"></xs:element>
    <xs:element name="what" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:sequence>
          <xs:element name="column" type="xs:string" />
          <xs:element name="value" type="tns:valueInfo" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="conditional" minOccurs="0"
      maxOccurs="unbounded">
       <xs:complexType>
        <xs:sequence>
          <xs:element name="column" type="xs:string" />
          <xs:element name="value" type="tns:valueInfo"
            maxOccurs="unbounded" />
          <xs:element name="cond_inner" type="tns:InnerCondType" />
          <xs:element name="cond_outer" type="tns:OuterCondType" />
        </xs:sequence>
      </xs:complexType>
```

```
      </xs:element>
      <xs:element name="skippableStep" type="xs:boolean"
        default="false"></xs:element>
      <xs:element name="description" type="xs:string"></xs:element>
    </xs:sequence>
  </xs:complexType>

   <xs:element name="Test">
     <xs:complexType>
       <xs:sequence>
          <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="Action" type="tns:action"/>
            <xs:element name="SqlAction" type="tns:sqlAction"/>
          </xs:choice>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
</xs:schema>
```

Příloha C

Příklad testu s Action

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Test xmlns:ns2="http://fi.muni.cz/automation"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="automation.xsd">
  <ns2:Action>
    <ns2:method>OPEN</ns2:method>
    <ns2:value>http://localhost:8080/ctds</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>TYPE</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>name</ns2:value>
    </ns2:path>
    <ns2:value>aswissgridhe1</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>TYPE</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>password</ns2:value>
    </ns2:path>
    <ns2:value>test</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>CLICK</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>submitButton</ns2:value>
    </ns2:path>
    <ns2:description>login</ns2:description>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>WAIT</ns2:method>
    <ns2:value>5000</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>OPEN</ns2:method>
```

```
  <ns2:value>http://localhost:8080/ctds/configuration/
    configuration.seam</ns2:value>
  <ns2:description>open configuration</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>WAIT</ns2:method>
  <ns2:value>3000</ns2:value>
  <ns2:description>wait after opening configuration</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>FIND_ELEMENT</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>groupSelect</ns2:value>
  </ns2:path>
  <ns2:value>selectConfGroup</ns2:value>
</ns2:Action>
<ns2:Action>
  <ns2:method>SELECT_VALUE</ns2:method>
  <ns2:path>
    <ns2:type>ELEMENT</ns2:type>
    <ns2:value>selectConfGroup</ns2:value>
  </ns2:path>
  <ns2:value>INTG</ns2:value>
  <ns2:description>select INTG group</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>change_group_button</ns2:value>
  </ns2:path>
</ns2:Action>
<ns2:Action>
  <ns2:method>WAIT</ns2:method>
  <ns2:value>3000</ns2:value>
  <ns2:description>wait after change group</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLEAR</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>BussinessDay.global</ns2:value>
  </ns2:path>
  <ns2:description>clear business day field</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>TYPE</ns2:method>
```

```
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>BussinessDay.global</ns2:value>
  </ns2:path>
  <ns2:value>20161224</ns2:value>
  <ns2:description>set parameter value</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK_AND_WAIT</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppConfiguration']/div/div/div[3]/form/
      table[2]/tbody/tr/td/input[1]</ns2:value>
  </ns2:path>
  <ns2:value>10000</ns2:value>
  <ns2:description>save configuration</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>SELECT_VALUE</ns2:method>
  <ns2:path>
    <ns2:type>ELEMENT</ns2:type>
    <ns2:value>selectConfGroup</ns2:value>
  </ns2:path>
  <ns2:value>DACF_PROCESS</ns2:value>
  <ns2:description>select DACF_PROCESS group</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>change_group_button</ns2:value>
  </ns2:path>
</ns2:Action>
<ns2:Action>
  <ns2:method>WAIT</ns2:method>
  <ns2:value>3000</ns2:value>
  <ns2:description>wait after change group</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLEAR</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>D-1.input.mandatoryTsoList.global</ns2:value>
  </ns2:path>
  <ns2:description>clear mandatory tso list field</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>TYPE</ns2:method>
```

```
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>D-1.input.mandatoryTsoList.global</ns2:value>
  </ns2:path>
  <ns2:value>CH,CZ,DE</ns2:value>
  <ns2:description>set mandatory tso list</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLEAR</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>dacf.2ndcycle.scaling.enabled.global</ns2:value>
  </ns2:path>
  <ns2:description>clear scaling</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>TYPE</ns2:method>
  <ns2:path>
    <ns2:type>ID</ns2:type>
    <ns2:value>dacf.2ndcycle.scaling.enabled.global</ns2:value>
  </ns2:path>
  <ns2:value>1</ns2:value>
  <ns2:description>set scaling</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK_AND_WAIT</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppConfiguration']/div/div/div[3]/form/
      table[2]/tbody/tr/td/input[1]</ns2:value>
  </ns2:path>
  <ns2:value>10000</ns2:value>
  <ns2:description>save configuration</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>OPEN</ns2:method>
  <ns2:value>http://localhost:8080/ctds/home.seam</ns2:value>
  <ns2:description>open home page</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>WAIT</ns2:method>
  <ns2:value>3000</ns2:value>
  <ns2:description>wait after opening home page</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>ASSERT_TEXT</ns2:method>
  <ns2:path>
    <ns2:type>CLASS</ns2:type>
```

```
      <ns2:value>date</ns2:value>
    </ns2:path>
    <ns2:value>(24.12.2016)</ns2:value>
    <ns2:description>assert right setted business day</ns2:description>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>SCREENSHOT</ns2:method>
    <ns2:value>home_page</ns2:value>
    <ns2:description>home page screenshot</ns2:description>
  </ns2:Action>
</ns2:Test>
```

Příloha D

Příklad testu s SqlAction

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Test xmlns:ns2="http://fi.muni.cz/automation"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="automation.xsd">
  <ns2:SqlAction>
    <ns2:type>SELECT</ns2:type>
    <ns2:table>gm_operational.model</ns2:table>
    <ns2:what>
      <ns2:column>business_day</ns2:column>
      <ns2:value>
        <ns2:text>bd</ns2:text>
      </ns2:value>
    </ns2:what>
    <ns2:what>
      <ns2:column>ts</ns2:column>
      <ns2:value>
        <ns2:text>timestamp</ns2:text>
      </ns2:value>
    </ns2:what>
    <ns2:conditional>
      <ns2:column>model_name</ns2:column>
      <ns2:value>
        <ns2:text>RS_Default_UCT.uct</ns2:text>
        <ns2:type>STRING</ns2:type>
      </ns2:value>
      <ns2:cond_inner>EQUAL</ns2:cond_inner>
    </ns2:conditional>
    <ns2:description>select bd from model</ns2:description>
  </ns2:SqlAction>
  <ns2:SqlAction>
    <ns2:type>INSERT</ns2:type>
    <ns2:table>gm_operational.model</ns2:table>
    <ns2:what>
      <ns2:column>business_day</ns2:column>
      <ns2:value>
        <ns2:variable>
          <ns2:name>bd</ns2:name>
          <ns2:type>SQL</ns2:type>
```

```
      </ns2:variable>
      <ns2:type>STRING</ns2:type>
    </ns2:value>
  </ns2:what>
  <ns2:what>
    <ns2:column>ts</ns2:column>
    <ns2:value>
      <ns2:variable>
        <ns2:name>timestamp</ns2:name>
        <ns2:type>SQL</ns2:type>
      </ns2:variable>
      <ns2:type>STRING</ns2:type>
    </ns2:value>
  </ns2:what>
  <ns2:what>
    <ns2:column>model_name</ns2:column>
    <ns2:value>
      <ns2:text>zkusebni_model</ns2:text>
      <ns2:type>STRING</ns2:type>
    </ns2:value>
  </ns2:what>
  <ns2:what>
    <ns2:column>model_id</ns2:column>
    <ns2:value>
      <ns2:text>123456789123456789</ns2:text>
      <ns2:type>NUMBER</ns2:type>
    </ns2:value>
  </ns2:what>
  <ns2:description>insert new model</ns2:description>
</ns2:SqlAction>
<ns2:SqlAction>
  <ns2:type>SELECT_DIRECT_QUERY</ns2:type>
  <ns2:query>SELECT model_id FROM gm_operational.model
    WHERE model_name LIKE '%zkusebni_model%'</ns2:query>
  <ns2:what>
    <ns2:column>model_id</ns2:column>
    <ns2:value>
      <ns2:text>modelId</ns2:text>
      <ns2:type>NUMBER</ns2:type>
    </ns2:value>
  </ns2:what>
  <ns2:description>get id from new model</ns2:description>
</ns2:SqlAction>
<ns2:SqlAction>
  <ns2:type>UPDATE</ns2:type>
  <ns2:table>gm_operational.model</ns2:table>
  <ns2:what>
    <ns2:column>model_type</ns2:column>
```

```
      <ns2:value>
        <ns2:text>1</ns2:text>
        <ns2:type>NUMBER</ns2:type>
      </ns2:value>
    </ns2:what>
    <ns2:what>
      <ns2:column>type</ns2:column>
      <ns2:value>
        <ns2:text>1</ns2:text>
        <ns2:type>NUMBER</ns2:type>
      </ns2:value>
    </ns2:what>
    <ns2:conditional>
      <ns2:column>model_id</ns2:column>
      <ns2:value>
        <ns2:variable>
          <ns2:name>modelId</ns2:name>
          <ns2:type>SQL</ns2:type>
        </ns2:variable>
        <ns2:type>NUMBER</ns2:type>
      </ns2:value>
      <ns2:cond_inner>EQUAL</ns2:cond_inner>
    </ns2:conditional>
    <ns2:description>update inserted model</ns2:description>
  </ns2:SqlAction>
  <ns2:SqlAction>
    <ns2:type>DELETE</ns2:type>
    <ns2:table>gm_operational.model</ns2:table>
    <ns2:conditional>
      <ns2:column>model_id</ns2:column>
      <ns2:value>
        <ns2:variable>
          <ns2:name>modelId</ns2:name>
          <ns2:type>SQL</ns2:type>
        </ns2:variable>
        <ns2:type>NUMBER</ns2:type>
      </ns2:value>
      <ns2:cond_inner>EQUAL</ns2:cond_inner>
    </ns2:conditional>
    <ns2:description>delete inserted model</ns2:description>
  </ns2:SqlAction>
</ns2:Test>
```

Příloha E

Příklad kombinovaného testu


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Test xmlns:ns2="http://fi.muni.cz/automation"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="automation.xsd">
  <ns2:Action>
    <ns2:method>OPEN</ns2:method>
    <ns2:value>http://localhost:8080/ctds</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>TYPE</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>name</ns2:value>
    </ns2:path>
    <ns2:value>uswissgridhe1</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>TYPE</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>password</ns2:value>
    </ns2:path>
    <ns2:value>test</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>CLICK</ns2:method>
    <ns2:path>
      <ns2:type>ID</ns2:type>
      <ns2:value>submitButton</ns2:value>
    </ns2:path>
    <ns2:description>login</ns2:description>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>WAIT</ns2:method>
    <ns2:value>5000</ns2:value>
  </ns2:Action>
  <ns2:Action>
    <ns2:method>OPEN</ns2:method>
```

```
  <ns2:value>http://localhost:8080/ctds/SECURITY_MONITOR/
    startSecurityMonitor.seam</ns2:value>
  <ns2:description>open security monitor</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>WAIT</ns2:method>
  <ns2:value>3000</ns2:value>
  <ns2:description>wait after opening security monitor</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK_AND_WAIT</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[2]/div[1]/
      div[4]/div[1]/input</ns2:value>
  </ns2:path>
  <ns2:value>2000</ns2:value>
  <ns2:description>open filter management</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK_AND_WAIT</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/input[2]
      </ns2:value>
  </ns2:path>
  <ns2:value>2000</ns2:value>
  <ns2:description>open creating new filter</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>TYPE</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[2]/div[1]/
      input[1]</ns2:value>
  </ns2:path>
  <ns2:value>filter auto test</ns2:value>
  <ns2:description>set filter name</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>SELECT_VALUE</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[3]/div/
      select[2]</ns2:value>
  </ns2:path>
  <ns2:value>N-1 limit [%]</ns2:value>
  <ns2:description>set filter name</ns2:description>
```

```
</ns2:Action>
<ns2:Action>
  <ns2:method>TYPE</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[3]/div/
      span/span/input[1]</ns2:value>
  </ns2:path>
  <ns2:value>80</ns2:value>
  <ns2:description>set filter condition</ns2:description>
</ns2:Action>
<ns2:Action>
  <ns2:method>CLICK_AND_WAIT</ns2:method>
  <ns2:path>
    <ns2:type>XPATH</ns2:type>
    <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[5]/
      input[3]</ns2:value>
  </ns2:path>
  <ns2:value>10000</ns2:value>
  <ns2:description>save new filter</ns2:description>
</ns2:Action>
<ns2:SqlAction>
  <ns2:type>SELECT</ns2:type>
  <ns2:table>gm_operational.sm_filter</ns2:table>
  <ns2:what>
    <ns2:column>id</ns2:column>
    <ns2:value>
      <ns2:text>filterId</ns2:text>
    </ns2:value>
  </ns2:what>
  <ns2:conditional>
    <ns2:column>name</ns2:column>
    <ns2:value>
      <ns2:text>filter auto test</ns2:text>
      <ns2:type>STRING</ns2:type>
    </ns2:value>
    <ns2:cond_inner>EQUAL</ns2:cond_inner>
  </ns2:conditional>
  <ns2:description>select id from filter by name</ns2:description>
</ns2:SqlAction>
<ns2:SqlAction>
  <ns2:type>SELECT</ns2:type>
  <ns2:table>ctds.db_export_observer</ns2:table>
  <ns2:what>
    <ns2:column>id</ns2:column>
    <ns2:value>
      <ns2:text>exportId</ns2:text>
    </ns2:value>
```

```
    </ns2:what>
    <ns2:conditional>
      <ns2:column>export_id</ns2:column>
      <ns2:value>
        <ns2:variable>
          <ns2:name>filterId</ns2:name>
          <ns2:type>SQL</ns2:type>
        </ns2:variable>
        <ns2:type>NUMBER</ns2:type>
      </ns2:value>
      <ns2:cond_inner>EQUAL</ns2:cond_inner>
    </ns2:conditional>
    <ns2:description>check if was created record in export table
      </ns2:description>
  </ns2:SqlAction>
  <ns2:Action>
    <ns2:method>CLICK_AND_WAIT</ns2:method>
    <ns2:path>
      <ns2:type>XPATH</ns2:type>
      <ns2:value>.//*[@id='angAppSecurityMonitor']/div/div/div[2]/
        table/tbody/tr/td[7]/a[2]/img</ns2:value>
    </ns2:path>
    <ns2:value>2000</ns2:value>
    <ns2:description>cleaning - remove filter</ns2:description>
  </ns2:Action>
</ns2:Test>
```

Příloha F

Příklad vygenerovaného testu

```
import cz.fi.muni.automation.tester.test.AbstractTest;
import cz.fi.muni.automation.core.configuration.*;
import cz.fi.muni.automation.converter.entity.*;
import cz.fi.muni.automation.tester.db.*;
import java.io.*;
import java.util.Calendar;
import java.util.Date;
import java.text.SimpleDateFormat;
import org.junit.Assert;
import java.util.List;
import java.util.ArrayList;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.apache.commons.io.*;
import org.openqa.selenium.support.ui.*;
import org.openqa.selenium.*;

public class all extends AbstractTest {

  @Override
  public String runTest() throws InterruptedException {
    usingDatabase = true;
    if (usingDatabase && DbName.NO_DATABASE.toString().equals(
      ConfigurationService.getValueFromConfiguration(
      ConfigurationConstants.CONF_DBNAME).toUpperCase())) {
      return "Test needs database and no database is configured";
    }
    try {
      int i = 1;
      List<Action.Path> list = new ArrayList<>();
      Action.Path p = new Action.Path();
      ResultSet set;
      try {
        super.before();
        super.prepareLogFileForTest("all");
        driver.get("http://localhost:8080/ctds");
        logFile.write(("Test step " + i++ + " finished with result:
          SUCCESS " + ConfigurationService.newline()).getBytes());
```

```
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.ID); p.setValue("name"); list.add(p);
super.getElement(list).sendKeys("uswissgridhe1");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS " + ConfigurationService.newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.ID); p.setValue("password"); list.add(p);
super.getElement(list).sendKeys("test");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS " + ConfigurationService.newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.ID); p.setValue("submitButton");list.add(p);
super.getElement(list).click();
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (login)" + ConfigurationService.newline()).getBytes());
Thread.sleep(5000);
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS " + ConfigurationService.newline()).getBytes());
driver.get("http://localhost:8080/ctds/SECURITY_MONITOR/
  startSecurityMonitor.seam");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (open security monitor)" + ConfigurationService
  .newline()).getBytes());
Thread.sleep(3000);
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (wait after opening security monitor)" +
  ConfigurationService.newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[2]/
  div[1]/div[4]/div[1]/input"); list.add(p);
super.getElement(list).click();
Thread.sleep(2000);
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (open filter management)" + ConfigurationService
  .newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/
  input[2]"); list.add(p);
super.getElement(list).click();
Thread.sleep(2000);
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (open creating new filter)" + ConfigurationService
  .newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[2]/
```

```
  div[1]/input[1]"); list.add(p);
super.getElement(list).sendKeys("filter auto test");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (set filter name)" + ConfigurationService.newline())
  .getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[3]/
  div/select[2]"); list.add(p);
new Select(super.getElement(list))
  .selectByVisibleText("N-1 limit [%]");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (set filter name)" + ConfigurationService.newline())
  .getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[3]/
  div/span/span/input[1]"); list.add(p);
super.getElement(list).sendKeys("80");
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (set filter condition)"
  + ConfigurationService.newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[5]/
  input[3]"); list.add(p);
super.getElement(list).click();
Thread.sleep(10000);
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (save new filter)" + ConfigurationService.newline())
  .getBytes());
set = st.executeQuery("SELECT id FROM gm_operational.sm_filter
  WHERE name = 'filter auto test'  ");
set.next();
sqlVariables.put("filterId",set.getString("id"));
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (select id from filter by name)" +
  ConfigurationService.newline()).getBytes());
set = st.executeQuery("SELECT id FROM ctds.db_export_observer
  WHERE export_id = " + sqlVariables.get("filterId") + "  ");
set.next();
sqlVariables.put("exportId",set.getString("id"));
logFile.write(("Test step " + i++ + " finished with result:
  SUCCESS (check if was created record in export table)" +
  ConfigurationService.newline()).getBytes());
list = new ArrayList<>();p = new Action.Path();
  p.setType(PathType.XPATH);
  p.setValue(".//*[@id='angAppSecurityMonitor']/div/div/div[2]/
```

```
          table/tbody/tr/td[7]/a[2]/img"); list.add(p);
        super.getElement(list).click();
        Thread.sleep(2000);
        logFile.write(("Test step " + i++ + " finished with result:
          SUCCESS (cleaning - remove filter)" + ConfigurationService
          .newline()).getBytes());
      } catch (final org.openqa.selenium.WebDriverException ex) {
        logFile.write(("Test step " + i++ + " finished with result:
          FAIL" + ConfigurationService.newline()).getBytes());
        logFile.write((ex.getMessage() + ConfigurationService.newline())
          .getBytes());
      } catch (final SQLException ex) {
        logFile.write(("Test step " + i++ + " finished with result:
          FAIL" + ConfigurationService.newline()).getBytes());
        logFile.write((ex.getMessage() + ConfigurationService.newline())
          .getBytes());
      } catch (final Exception ex) {
        logFile.write(("Test step " + i++ + " finished with result:
          FAIL" + ConfigurationService.newline()).getBytes());
        logFile.write((ex.getMessage() + ConfigurationService.newline())
          .getBytes());
      } finally {
        super.after();
        logFile.flush();
        logFile.close();
      }
    } catch (final IOException ex) {
      System.err.println(ex.toString());
    }
    return "Test finished";
  }

}
```