Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Návrh, implementace a integrace nástroje pro vytváření automatických integračních testů

Diplomová práce

Bc. Michal Kuchta

Vedoucí práce: Mgr. Vít Barabáš

Garant práce: Ing. Marta Vohnoutová

České Budějovice 2020

Bibliografické údaje

Kuchta, M., 2020: **Návrh, implementace a integrace nástroje pro vytváření automatických integračních testů**. [Design, implementation and integration of the tool for automatic integration test creation. Mgr. Thesis, in Czech.]– 61 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Diplomová práce se zabývá vývojem nástroje pro vytváření automatických integračních testů. V teoretické části jsou popsané již existující dostupné nástroje pro automatické testování. Hlavní část diplomové práce je zaměřena na návrh, implementaci a integraci nástroje pro vytváření integračních testů. Součástí práce je také návrh a konfigurace virtuálního prostředí pro spouštění vytvořených integračních testů. Výsledná aplikace je následně otestována na vzorovém příkladu.

Klíčová slova

Automatizace, Testování, Virtualizace, Java

Annotation

This diploma thesis deals with the development of a tool for automatic integration tests creation. In the theoretical part, the already available tools for automatic testing are described. The main part of the thesis is focused on design, implementation and integration of the tool for creation of integration tests. Part of the work is also design and configuration of virtual environment for running created integration tests. The application is then tested on an example.

Key words

Automatization, Testing, Virtualization, Java

Prohlášení

Prohlašuji, že svoji diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. V platném znění souhlasím se zveřejněním své diplomové práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích, Dne 15.5.2020 Podpis autora…….

Poděkování

Chtěl bych poděkovat vedoucímu mé práce, Mgr. Vítu Barabášovi, a Ing. Martě Vohnoutové, za ochotu, vstřícný přístup, odborné rady a konstruktivní připomínky při vypracování této diplomové práce.

Také bych chtěl poděkovat své rodině, známým a zejména mé přítelkyni, za nekonečnou trpělivost a podporu při psaní této práce.

V poslední řadě bych rád poděkoval svému kolegovi Adrianu Czarnomskimu, se kterým probíhala úzká spolupráce při analýze a návrhu řešení nástrojů, vytvořených v této diplomové práci.

Obsah

1 Úvod

Testování je proces, při kterém uživatel musí otestovat určitou funkcionalitu aplikace. V případě, kdy je aplikace testována manuálně, musí být jednotlivé části otestovány postupně. Čas pro otestování celé aplikace se tak výrazně zvýší.

Tento problém se v dnešní době snaží vyřešit nástroje pro automatické testování, které si kladou za cíl snížit čas potřebný k testování. Těchto nástrojů je dostupných hned několik. Většina z nich je však buď placená, nebo nesplňuje kritéria společnosti ENGEL z hlediska testování softwaru.

Samotný návrh zadání této diplomové práce vznikl ve společnosti ENGEL, kdy bylo potřeba vytvořit nástroj, schopný jednoduše vygenerovat integrační test pro již existující testovací framework. Ten slouží k internímu testování aplikací ve společnosti ENGEL.

Diplomová práce se věnuje dostupným nástrojům pro vytváření automatických integračních testů. Dále obsahuje samotný návrh a popis implementace nástroje pro vytváření integračních testů. Součástí práce je také návrh a konfigurace virtuálního prostředí pro spouštění již existujících testů. Tento nástroj je následně otestován na vzorovém příkladu.

2 Cíle práce

Hlavním cílem této práce je navrhnout a vytvořit nástroj pro vytváření integračních testů. Takto vytvořený test bude dále využíván pro testování software ve společnosti ENGEL.

Práce má za cíl několik bodů:

1) Popsat dostupné nástroje pro vytváření automatických integračních testů
2) Popsat jednotlivé části vývoje tohoto nástroje od analýzy a návrhu až po jeho implementaci
3) Navrhnout a vytvořit nástroj pro vytváření automatických integračních testů
4) Navrhnout a konfigurovat virtuální prostředí pro unifikované spouštění již existujících integračních testů
5) Otestovat vytvořený nástroj na vzorovém příkladu

3 Slovník pojmů

Slovník pojmů obsahuje seznam důležitých pojmů, které jsou během práce zmíněny.

3.1 Testování (test)

Úkon, pomocí kterého se zjistí, zda určitá věc, například počítač nebo auto, funguje správně nebo efektivně[1].

V softwarovém prostředí se jedná o proces, při kterém se zjišťuje, zda určitá část softwaru nebo celý software funguje tak, jak bylo předem definováno.

3.2 Integrační testování

Integrační testování je proces, ve kterém jsou jednotlivé softwarové nebo hardwarové komponenty mezi sebou testovány, aby byla ověřena správná funkcionalita a interakce mezi nimi. [1]

3.3 Tester

Také znám pod zkratkou SQE (*software quality engineer*). Člověk zodpovědný za testování softwaru a zajištění jeho výsledné kvality. Jeho hlavním úkolem je hledání chyb během testování vyvíjeného softwaru a následně jejich dokumentace, spolu s vytvářením nových testovacích případů, tzv. *test casů*.

3.4 Test case (testovací případ)

Dokumentace, ve které jsou určeny vstupy a očekávané výstupy pro určitý test. Pomocí této dokumentace je testována určitá funkcionalita aplikace.

*Test cas*e je detailněji popsán v teoretické části práce.

3.5 Test run

Množina testovacích případů, související s určitou částí aplikace. Tyto testovací případy jsou poté provedeny např. před vydáním aplikace nebo při testování specifické části aplikace.

[1] Definice testu. Dostupná z: https://dictionary.cambridge.org/dictionary/english/test

3.6 Framework

Sada předem definovaných částí softwaru, které může vývojář použít, rozšířit nebo upravit pro potřeby své aplikace. Při použití frameworku nemusí softwarový vývojář začínat od začátku v případě, že chce vytvořit novou aplikaci. Frameworky jsou složené z kolekce objektů a může být využit jak jejich design, tak i logika. [2]

4 Teoretická část

Teoretická část práce obsahuje stručný přehled dvou hlavních typů testování, manuálního a automatického, a jejich využití.

Dále jsou v této kapitole popsány některé dostupné nástroje pro automatické testování.

4.1 Test case

Dokumentace, ve které jsou určeny vstupy a očekávané výstupy pro specifický test. Také je zde popsána množina podmínek nebo jednotlivých kroků, které se mají provést během tohoto testu. [3]

Pomocí *test casu* je krok po kroku ověřeno, zda určitá funkcionalita softwarové aplikace, například odeslání e-mailu, funguje, jak bylo definováno. Při vytváření test casů mohou být také zjištěny nedostatky logiky aplikace. [4]

Kroky *test casu* by měly být napsány tak, aby z nich bylo snadno pochopitelné, co je potřeba v každém kroku udělat (například klikni na tlačítko). Vždy by měla být popisována pouze jedna specifická funkcionalita a v co nejméně krocích. [5]

Test Scenario ID	Login-1			**Test Case ID**	Login-1A		
Test Case Description	Login – Positive test case			**Test Priority**	High		
Pre-Requisite	A valid user account			**Post-Requisite**	NA		
Test Execution Steps:							
S.No	**Action**	**Inputs**	**Expected Output**	**Actual Output**	**Test Browser**	**Test Result**	**Test Comments**
1	Launch application	https://www.facebook.com/	Facebook home	Facebook home	IE-11	Pass	[Priya 10/17/2017 11:44 AM]: Launch successful
2	Enter correct Email & Password and hit login button	Email id : test@xyz.com Password: ******	Login success	Login success	IE-11	Pass	[Priya 10/17/2017 11:45 AM]: Login successful

Obrázek 4.1 Ukázka testovacího případu [6]

Příkladem může být *test case* popisující přihlášení na webovou stránku pomocí přihlašovacích údajů. Tyto údaje označují jednotlivé kroky potřebné pro splnění daného testu. Výstupem zmíněného testu by byla kontrola, zda došlo k úspěšnému přihlášení na stránku.

Test case by neměl být popsán pro jedno konkrétní řešení. Může být vytvořen jak pro pozitivní, tak i pro negativní průběh testu.

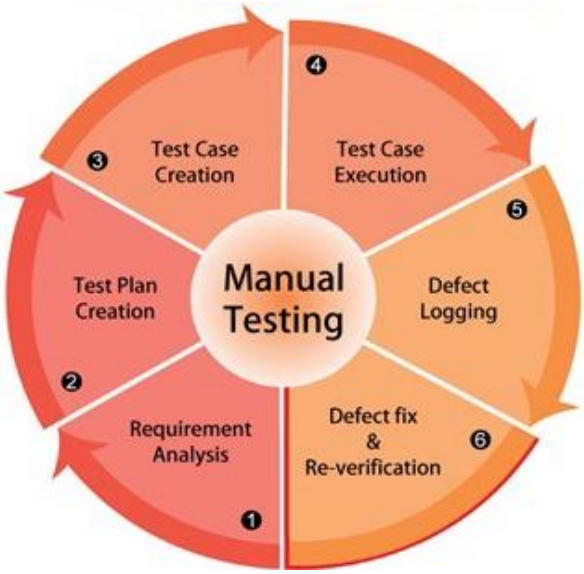
4.2 Manuální testování

Testování je proces sloužící k nalezení chyb ve vyvíjeném softwaru a ujištění, zda všechny části aplikace splňují předem daná kritéria. Testování je prováděno software kvality inženýrem, který má na starost ověření funkčních požadavků aplikace. [7]

Typ testování, v němž je množina test casů ověřující funkcionalitu aplikace testována bez použití jakýchkoliv automatických nástrojů.

Výhodou tohoto druhu testování kvality softwaru je možnost objevit nejasnosti v rané fázi vývoje před tím, než je produkt odeslán zákazníkovi.

Také pomáhá ověřit požadavky, při kterých není testována funkcionalita aplikace. Mezi tyto požadavky lze zmínit např. ověření, zda je aplikace uživatelsky přívětivá. [8]



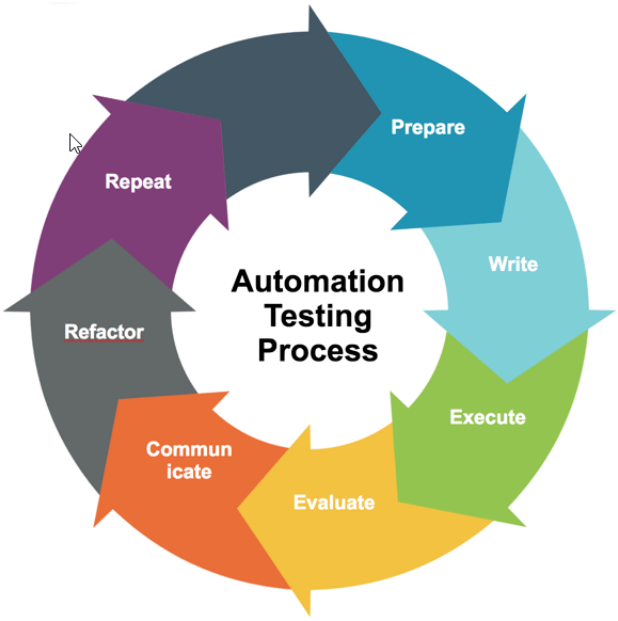
Obrázek 4.2 Postup při manuálním testování [9]

Nevýhodou je samotná náročnost testování po časové stránce. Oproti automatickému testování musí být nejprve vytvořen test case pro danou funkcionalitu. Pomocí test casu je následně uživatelem, krok po kroku, otestována určitá část aplikace. Samotné ověření funkcionality závisí zejména na schopnosti uživatele zpracovat a vyhodnotit daný test case.

Testování více-vláknových (multi-threading) operací, výkonosti nebo zabezpečení je velmi těžko proveditelné bez použití nástrojů pro automatické testování. [10]

4.3 Automatické testování

Automatizace je automatické provádění úkolů bez pravidelného přerušování. Cílem je optimalizovat proces, minimalizovat a postupně odstranit potřebu člověka zasahovat do prováděných úkolů. Automatizace vede k vyšší efektivitě, protože není potřeba využití lidských zdrojů. Lze automatizovat i úkoly, které přesahují lidské schopnosti, jako je například výroba elektronických zařízení. [11]



Obrázek 4.3 Proces automatického testování [12]

Typ testování, zahrnující automatické provádění *test casu* pomocí nástroje sloužícího k automatizaci. Tester napíše script, který může být spuštěn v opakujících se intervalech. [13]

Čas provádění testů při automatickém testování je výrazně nižší než u testování manuálního. Je však potřeba vzít v úvahu, že script, který bude schopen automaticky testovat aplikaci, musí být manuálně vytvořen.

Script může být vytvořen buď programově, za použití frameworku určeného k testování, nebo za použití rozšíření prohlížeče, kde jsou následně nahrány jednotlivé kroky testu. Tento script musí být upraven při jakýchkoliv změnách ve struktuře aplikace. [14]

Výhodou tohoto typu testování jsou dlouhotrvající testy, které jsou často vynechávány při manuálním testování. Díky automatickému testování mohou být tyto testy prováděny

snadno a bez dozoru a mohou být spouštěny na více počítačích s různými konfiguracemi. [15]

Nevýhodou je testování uživatelského rozhraní. Sebemenší změny v uživatelském rozhraní mohou vést k selhání testu. Je velmi obtížné vytvořit spolehlivý *test case* na uživatelské rozhraní pro více zařízení nebo pro různá rozlišení aplikace. [16]

4.4 Record and play (codeless automatization)

Testovací nástroj, ve většině případů použitý jako rozšíření prohlížeče, pomocí kterého jsou zaznamenávány uživatelské akce na webové stránce. Uživatelské akce jsou automaticky generovány do testovacího scriptu, který může být následně spuštěn (přehrán). Výhoda těchto nástrojů je možnost vytvořit test během pár sekund a následně ho opakovaně využívat. [17]

Record and play nástroje jsou však těžko udržovatelné. V případě častých změn na webových stránkách dochází k nespolehlivosti těchto nástrojů a v důsledku toho také ke snížení efektivity testování. Nahraný test například nelze spustit, protože některý webový element již neexistuje. V takovém případě musí být test vytvořen znovu. [18]

Problémem těchto nástrojů je testování mezi prohlížeči, tzv. *cross-browser testing*. Test je obvykle vytvářen na jednom prohlížeči, kde je nainstalovaný nástroj pro nahrávání. Jakmile je test nahrán, je možné jej spustit. V případě, že má být test spuštěn na dalších prohlížečích, může dojít k problémům, které jsou způsobeny změnami na webové stránce v rámci prohlížeče. [19]

Nejefektivnější použití těchto nástrojů je v případě, že webová stránka zůstává neměnná, tedy že se elementy nacházejí na stejném místě a pod stejným názvem. Většinu ze zmíněných problémů řeší novější nástroje, které ale nejsou dostupné zdarma.

Jedním z nejznámějších a nejpoužívanějších nástrojů, dostupných zdarma, je *Selenium IDE*.

4.5 Nástroje pro automatické testování

Tato část obsahuje popis některých dostupných nástrojů pro automatické vytváření testů. Jsou zmíněny jak volně dostupné nástroje, tak i placené.

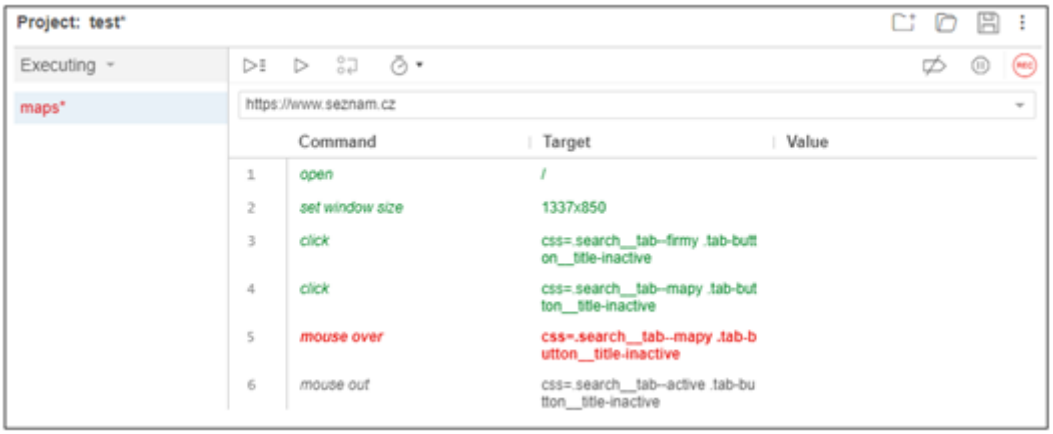
4.5.1 Selenium IDE[2]

Jedno z nejznámějších rozšíření prohlížeče, dostupné v prohlížečích Chrome a Firefox, sloužící k jednoduchému vytvoření automatických testů bez znalosti programovacího jazyka.

Patří do skupiny tzv. *record-and-play* (nahraj a spusť) nástrojů. Pomocí nástroje jsou nahrávány jednotlivé uživatelské akce v prohlížeči, například kliknutí na element, přes existující Selenium příkazy. Všechny příkazy obsahují parametry, které definují obsah daného elementu. [20] Nahrané testy mohou být uloženy a následně spuštěny v prohlížeči.

Vytvořený test je možné editovat, a to i během nahrávání testu.

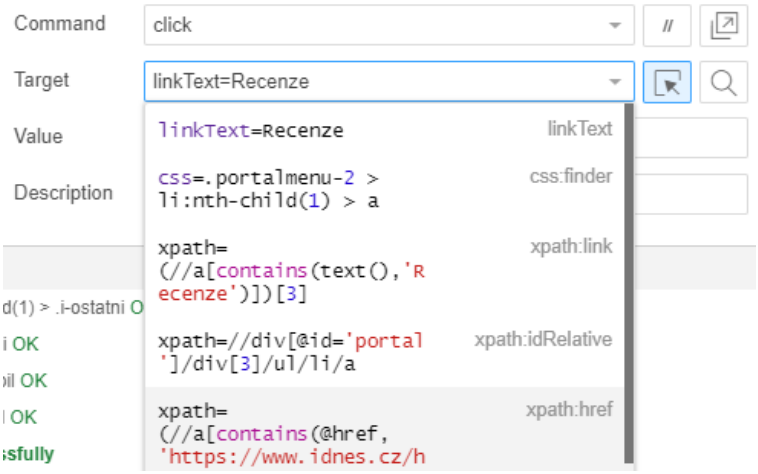
Selenium IDE umožňuje upravit rychlost provádění testu nebo přeskočit některé kroky během provádění.



Obrázek 4.4 Příklad nahraného Selenium testu

Problematika tohoto nástroje je v detekci elementů během provádění testu. Pro identifikaci elementů je využito několik identifikátorů, například *xpath:link* nebo *xpath:idRelative*. Element je možné detekovat také podle kaskádových stylů.

[2] Integrated Development Environment



Obrázek 4.5 Identifikátory elementu v Selenium IDE

V případě, že HTML[3] stránka zůstává neměnná, nemá Selenium IDE sebemenší problém identifikovat prvky na webové stránce. Pokud je stránka dynamická a hledaný element je pozměněn, není Selenium schopné tento element detekovat ani za použití několika lokalizátorů (viz obrázek 4.5). To následně vede k zacyklení celého testu, který se stále snaží idenfitikovat již neexistující element.

Selenium IDE je možné využít pouze pro testování webových aplikací a jen v prohlížečích Chrome a Firefox. Testování mobilních nebo desktopových aplikací tento nástroj v současné době neumožňuje. V dnešní době, kdy je automatické testování velmi rozšířeno a podpora velkého množství prohlížečů nebo zařízení je základní částí většiny dostupných nástrojů, je toto velký problém.

I přes mnoho nevýhod je stále využíván komunitou díky snadnému vytvoření jednoduchých automatických testů. Ve větších firmách, využívajících nástroje pro automatizaci, své místo nenajde.

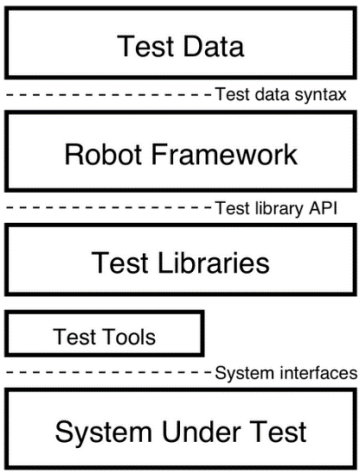
4.5.2 Robot framework

Open-source automatizační framework, nezávislý jak na operačním systému, tak i na webovém prohlížeči. Slouží pro akceptační testování, akceptační programovaní řízení testů (ATDD[4]) nebo robotickou optimalizaci procesu.

Silnou stránkou frameworku je princip testování, fungující na základě klíčových slov. Klíčová slova jsou uložena v již existující knihovně. Pokud klíčové slovo neexistuje, je

[3] Hypertext Markup Language
[4] Acceptance Test Driven Development

možné framework rozšířit o vlastní uživatelskou knihovnu. Ta obsahuje další vytvořená klíčová slova. Knihovny jsou implementované buď v jazyku Java nebo Python.



Obrázek 4.6 Architektura frameworku Robot [21]

V momentě, kdy má být test proveden, jsou zanalyzována a rozdělena testovací data pomocí frameworku. Následně jsou využita klíčová slova, poskytována knihovnami, pro interakci s tímto systémem. Komunikace systému s knihovnami probíhá buď přímo, nebo pomocí jiných testovacích nástrojů.

Test je spuštěn z příkazové řádky a průběh testu tak není vizuálně zobrazen. Zobrazen je jen výsledek, zpráva a log, v HTML a XML[5] formátu. [22]

```
*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Obrázek 4.7 Příklad vytvořeného testu v Robot frameworku [23]

Nevýhodou frameworku je absence *record-and-play* nástroje, který je v dnešní době podporován mnoha nástroji pro automatické testování.

[5] Extensible Markup Language

Nahrávání testu je možné zrealizovat pomocí rozšíření (nástrojů), která jsou k dispozici. Tyto nástroje nejsou dlouhodobě udržované, a jejich kompatibilita s frameworkem proto není zaručena. Mezi tyto nástroje patří *ChromeRobot* nebo *FireRobot*.

4.5.3 Sahi Pro

Nástroj pro automatické testování webových aplikací za pomoci *sahi* scriptu. Script není nic jiného než rozšíření jazyka Javascript, používající stejnou syntaxi, ale přidávající schopnost efektivně komunikovat s prohlížečem. Umožnuje také provádět systémové akce, jako je například čtení ze souborového systému nebo přístup do databáze.

Script je defaultně vytvořen (automaticky generován) v programovacím jazyku Javascript. Pro vytvoření scriptu může být použit i jazyk Java.

Spustitelný script může být vytvořen pomocí režimu nahrávání, kdy jsou veškeré uživatelské akce zachyceny a následně automaticky převedeny do tohoto scriptu. Díky tomu je test vytvořen během několika sekund. Script je možné vytvořit i manuálně, a to v již zmíněných programovacích jazycích. [24]

```
_setValue(_textbox("user"), "test");
_setValue(_password("password"), "secret");
_click(_submit("Login"));
_setValue(_textbox("q"), "2");
_setValue(_textbox("q[1]"), "1");
_setValue(_textbox("q[2]"), "1");
_click(_button("Add"));
_click(_cell("Rs.200[1]"));
_assertExists(_textbox("total"));
_assert(_isVisible(_textbox("total")));
_assertEqual("1150", _textbox("total").value);
_click(_button("Logout"));
```

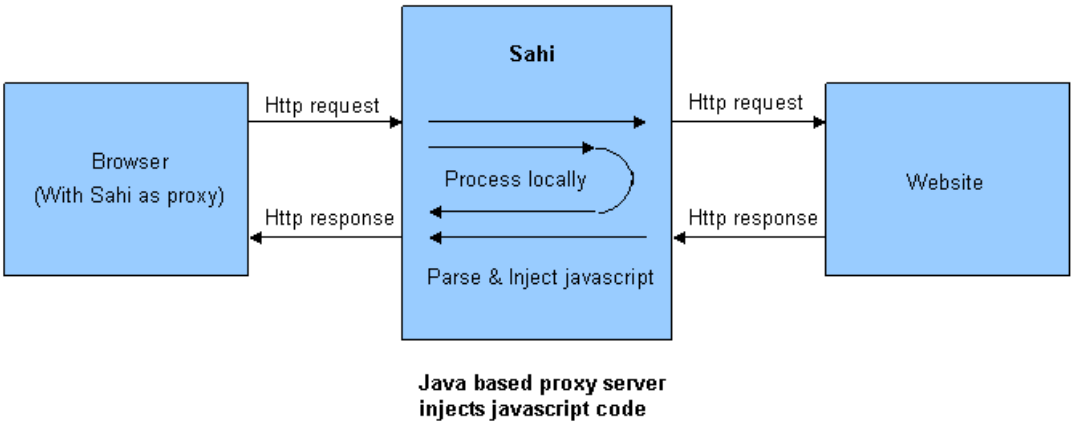
Obrázek 4.8 Příklad vytvořeného Sahi scriptu [25]

Syntaxe scriptu, zobrazena na obrázku 4.8, je velmi jednoduchá a může být snadno pochopena i nezkušeným uživatelem.

Sahi Pro obsahuje robustní identifikační API[6], ve kterém nejsou pro identifikaci elementů ve webové aplikaci použity *XPath* ani CSS[7] selektory. Místo toho je vytvořen

[6] Application Programming Interface
[7] Cascading Style Sheets

wrapper okolo DOM[8] Javascriptu. Díky vložení vlastního javascriptu do webové stránky je zajištěna vetší kontrola nad stránkou. Prvky na webové stránce mohou být následně lokalizovány i v momentě, kdy není nastavené jejich id nebo jsou použity vlastní HTML tagy.



Obrázek 4.9 Sahi architektura [26]

Tento nástroj není volně dostupný a funguje formou ročního předplatného, které se pohybuje okolo 700 dolarů za jednu licenci. V případě použití *concurrent* licence je cena 999 dolarů. K dispozici je i verze *Sahi Open Source*, která je dostupná zdarma . Ta ale neobsahuje všechny funkce *Sahi Pro*. [27]

4.5.4 TestComplete

Jedno z nejlepších možných řešení, pro automatické testování, dostupných na trhu.

TestComplete umožnuje automatické UI[9] testování pro desktopové, webové i mobilní aplikace. Díky nástroji pro identifikaci objektů, využívajícího umělou inteligenci, a podpoře mnoha testovacích frameworků, jsou vytvářeny stabilní a škálovatelné testy na uživatelské rozhraní i v případě, že jsou provedeny větší změny ve struktuře stránky. [28]

UI test je možné vytvořit několika způsoby. Může být vytvořen bez napsání jediné řádky kódu díky funkci *record-and-play*. Test lze vytvořit také z řady operací, pomocí tzv. *keyword driven testing*. Operace jsou specifikovány klíčovými slovy, simulujícími uživatelské akce, jako je kliknutí na myš nebo stisknutí klávesy. [29]

[8] Document Object Model
[9] User Interface (uživatelské rozhraní)

Nechybí ani možnost napsat test manuálně, a to za použití jednoho ze sedmi programovacích jazyků, které nástroj podporuje. Mezi těmito jazyky lze najít Javascript, Python, VBScript nebo také C#. [30]

Možností, které tento framework nabízí, je nespočet. Zmíněny mohou být také *data-driven testing*, kontrola *cross-browser* validace kaskádových stylů nebo systém pro reporty a analýzu testu, které dokáží získat informace i v průběhu provádění testu.

Nevýhodou tohoto nástroje je jeho vysoká cena. Ta se pohybuje v mezích od 5 do 10 tisíc eur za přenosnou licenci [31].

S přihlédnutím k tomu, co všechno tento nástroj nabízí, není cena nijak závratná. Pro velké firmy se jedná o jedno z nejlepších řešení na poli automatického testování softwaru.

5 Použité nástroje a technologie

Tato kapitola obsahuje stručný popis nástrojů a technologií, které byly použity při psaní této diplomové práce.

5.1 Java 8

Java je silně typovaný, objektově orientovaný, programovací jazyk. Jeho výhodou je kompilování do instrukční sady *bytecode* a binárního formátu definovaného v Java Virtual Machine (JVM) specifikaci. [32]

Dostupný z https://www.java.com/en/download/.

5.2 Vaadin

Vaadin je framework pro vývoj webových aplikací v programovacím jazyce Java navržený k jednoduchému vývoji a údržbě webového uživatelského prostředí. [33]

Jeho silnou stránkou je možnost vytvořit grafické rozhraní přes Java implementaci.

Dostupný z https://vaadin.com/.

5.3 Eclipse IDE 2019-06

Vývojové prostředí, dostupné zdarma, pro vývoj webových či desktopových aplikací v programovacích jazycích Java nebo Javascript.

Dostupné z *https://www.eclipse.org/ide/.*

5.4 Oracle VM VirtualBox

Open-source software, volně dostupný za podmínek GNU General Public License (GPL) verze 2.

Multiplatformní virtualizační nástroj pro počítače, nezávislý na nainstalovaném operačním systému. Je ho tak možné použít například na systému Windows, Linux nebo Mac OS.

Umožnuje vytvořit a spustit na nainstalovaném systému virtuální stroje, na kterých mohou být spuštěny další operační systémy. Je tak možné na operačním systému Windows spustit operační systém Linux, který běží ve virtuálním prostředí. [34]

Velkou výhodou je možnost virtuální stroj pozastavit, kopírovat nebo přenášet mezi hosty. Virtualbox obsahuje možnost vytvořit tzv. *snapshot*, který uloží aktuální stav

virtuálního stroje. Těchto stavů může být vytvořeno více. Takto vytvořený *snapshot* může být použit v případě, že dojde k problému s virtuálním strojem, a je potřeba ho obnovit do momentu, kdy byl funkční. [35]

Nástroj je dostupný z *https://www.virtualbox.org/*.

6 Praktická část

Praktická část práce je rozdělena do několika dílčích částí. První část se stručně věnuje problematice testování ve firmě ENGEL. Dále praktická část obsahuje popis dvou nástrojů, které byly v rámci této diplomové práce vytvořeny. Tyto nástroje pomáhají řešit zmíněnou problematiku testování.

6.1 Problematika současného procesu testování ve firmě ENGEL

Tato kapitola obsahuje stručný úvod do problematiky současného procesu testování ve firmě ENGEL spolu s návrhem řešení, které odstraňuje zjištěné nedostatky.

6.1.1 Analýza současného procesu testování ve firmě ENGEL

Analýza současného procesu testování softwaru ve firmě ENGEL již byla provedena a není tedy součástí této práce. Zmíněná analýza byla provedena kolegou Czarnomskim a je součástí jeho bakalářské práce[10].

Ze zjištěné analýzy lze zdůraznit několik bodů:

Nedostatek času na testování aplikace

Řízení projektů ve firmě funguje podle agilní metodiky vývoje SCRUM. Projekt je implementován ve dvoutýdenních sprintech a otestovat novou funkcionalitu v daném sprintu, aby byly případné chyby v implementaci nalezeny a opraveny, je problematické.

Otestování jedné části aplikace může zabrat i několik hodin

Firma ENGEL nabízí dlouholetou podporu jejich produktů. Časová náročnost testování aplikace se tak zvyšuje, protože je potřeba otestovat jak starší, tak i novější verzi aplikace. *Test case* ale může zůstat pro více verzí stejný.

Stejný test je proveden vícekrát kvůli jiné verzi aplikace nebo různým parametrům

Tester musí manuálně otestovat stejnou funkcionalitu více než jednou s různým nastavením (různými parametry) aplikace.

[10]Czarnomski Adrian: *Návrh a optimalizace procesu automatizovaného integračního testování*

6.1.2 Řešení současného procesu testování ve firmě ENGEL

Na základě analýzy, kdy byly zjištěny možnosti vylepšení současného procesu testování, došlo k návrhu řešení, které je celkem rozděleno do tří částí:

- Nástroj pro tvorbu a správu *test casů*
- Testovací framework zodpovědný za provedení vytvořeného integračního testu
    - **Není součástí této práce –** testovací framework je součástí bakalářské práce kolegy A. Czarnomskiho
- Příprava virtuálního prostředí, na kterém je test spuštěn

Na základě analýzy současného řešení byla vytvořena nová webová aplikace, která umožní testerovi vytvořit nebo upravovat integrační testy.

Aplikace obsahuje parametry pro nastavení konfigurace ve virtuálním prostředí spolu s příkazy, potřebnými pro automatické provedení uživatelských akcí na testované aplikaci. Výstupem aplikace je integrační test uložený ve formátu Java, který je využit již existujícím testovacím frameworkem.

Vytvořený test je spuštěn ve virtuálním prostředí, které je nastaveno na základě konfigurace daného testu.

K přípravě virtuálního prostředí slouží další webová aplikace, která byla pro tento účel vytvořena. Zde je nastaveno, na kterém systému se mají vybrané testy spustit, a na jaké verzi aplikace mají být tyto testy provedeny.

6.2 Nástroj pro tvorbu automatických integračních testů

Tato část diplomové práce se věnuje analýze, návrhu a implementaci nástroje pro vytváření integračních testů.

Součástí této kapitoly jsou i ukázky některých důležitých částí implementace, bez kterých by aplikace nemohla fungovat.

6.2.1 Analýza řešení nástroje pro tvorbu integračních testů

Na základě zjištěných požadavků z kapitoly *6.1.2* došlo k návrhu řešení.

Integrační test by měl být vytvořen na jakémkoliv systému bez nutnosti dalšího zásahu do počítače (instalace doplňků třetí strany, změny v uživatelském nastavení). Bylo tedy nutné projít všechny dostupné možnosti a vybrat nejoptimálnější řešení pro implementaci jak frontendové, tak backendové části aplikace.

Samotné vytvoření integračního testu by nemělo zabrat více než pár minut. Aplikace by měla být uživatelsky přívětivá, aby jí dokázal používat i nezkušený uživatel. Zde byl kladen velký důraz zejména na vzhled a rozložení aplikace. Bylo také nutné zajistit velkou stabilitu aplikace a funkčnost integračního testu, který bude pomocí nástroje vytvořen. Velká část analýzy byla věnována ošetření vstupních i výstupních dat.

Nedílnou součástí analýzy bylo zjištění, jak má vypadat struktura výstupního testu. Toto je klíčový bod, bez kterého by nástroj pro tvorbu integračních testů nemohl fungovat. Zde často probíhala komunikace s kolegou A. Czarnomskim, který měl na starost vývoj testovacího frameworku. Ten musel definovat všechny informace o struktuře testu, např. v jakém formátu má být test uložen nebo jaká data musí test obsahovat.

6.2.2 Návrh řešení nástroje pro tvorbu integračních testů

Po analýze byla zvolena webová aplikace, která je zcela nezávislá jak na operačním systému, tak i na webovém prohlížeči, a je jí možné používat kdekoliv bez větších omezení. V dnešní době, kdy jsou webové aplikace oblíbené, existuje pro ně spoustu dostupných zdrojů a jejich vývoj jde stále vpřed, se jedná o ideální řešení.

Důraz byl kladen na to, aby aplikace byla jednoduchá a přehledná, a bylo možné vytvořit test i bez znalosti programovacího jazyka.

Systém také zajišťuje validaci vstupních dat, které uživatel zadá. Není tak možné vložit nebo podvrhnout hodnotu, kterou systém nepodporuje, do jakékoliv části aplikace. Pokud by k tomu došlo (manuálně), je aplikace schopna vymazat, případně opravit, nevalidní data v momentě, kdy je test editován.

Pro vývoj aplikace byl zvolen programovací jazyk Java, který je ve firmě využíván, a jsou s ním bohaté zkušenosti. Ten byl doplněn o framework *Vaadin*. Tento framework se ve firmě ENGEL pro vývoj aplikací dlouhodobě používá, a proto je z hlediska dalšího rozvoje aplikace ideální. Navíc je vydán pod licencí *Apache License*[11]*, version 2.0,* a je tedy zcela zdarma k využívání či případným modifikacím.

Vaadin byl použit pro implementaci grafického rozhraní webové aplikace. Jeho silnou stránkou je schopnost vytvořit celé uživatelské rozhraní aplikace pomocí Java kódu. Díky tomu může být výsledná HTML stránka vytvořena rychle i bez větších znalostí jazyka HTML nebo Javascript. Není tak nutné používat další nástroje pro tvorbu čistě frontendové části.

6.2.2.1 Grafické rozhraní

Jak již bylo zmíněno, hlavní důraz při vývoji této aplikace byl kladen na jednoduchost a přehlednost aplikace během vytváření integračního testu.

Samotný grafický návrh jednotlivých částí aplikace byl proto několikrát konzultován s vedoucím práce a také s testerem z firmy ENGEL, aby bylo dosaženo co největší optimalizace při pracování s tímto nástrojem.

Grafické rozhraní dílčích částí aplikace bylo naimplementováno tak, aby měl tester co největší „*know-how*“ během jednotlivých kroků při vytváření testu. To je klíčový bod, bez kterého by nemohl být správně vytvořen integrační test.

Problém při tvorbě grafického rozhraní, který bylo nutné vyřešit, bylo nastavení kaskádových stylů pro jednotlivé komponenty, které framework nabízí. Vaadin obsahuje již předdefinované kaskádové styly (*padding, margin* nebo *width* elementu) pro všechny komponenty. Tyto styly jsou pro většinu komponent velmi špatné a ve značné míře nepoužitelné. Zmínit lze např. přetečení textu u některých komponent (treegrid, table). Většina defaultních kaskádových stylů musela být přepsána za použití css pravidla *!important,* aby bylo dosaženo správného, a hlavně funkčního, grafického rozhraní.

[11] Jedná se o svobodnou softwarovou licenci

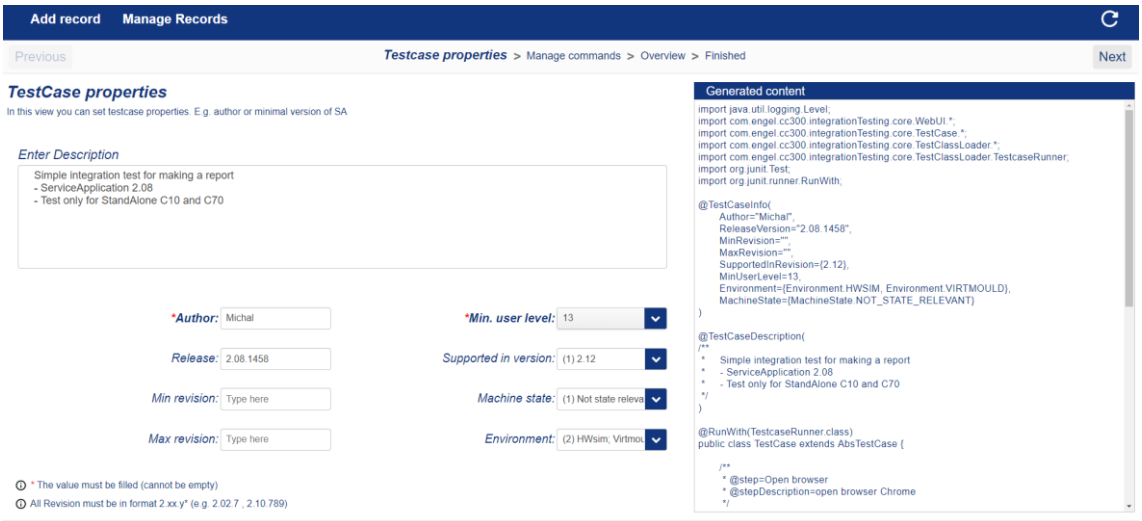
I přes zmíněné problémy byl stále zvolen tento framework, a to zejména pro snadné a rychlé vytvoření webové stránky a také proto, že je již ve firmě používán.

Samotná aplikace byla rozdělena do dvou hlavních částí:

- Vytvoření nového integračního testu
- Správa (mazání, editace) již existujících integračních testů

Pro každou část byla vytvořena vlastní webová stránka, vycházející ze společné grafické šablony. Tato šablona je univerzální, a v případě přidání další funkcionality, může být snadno použita bez větších obtíží.

Design aplikace je zobrazen na spodním obrázku.



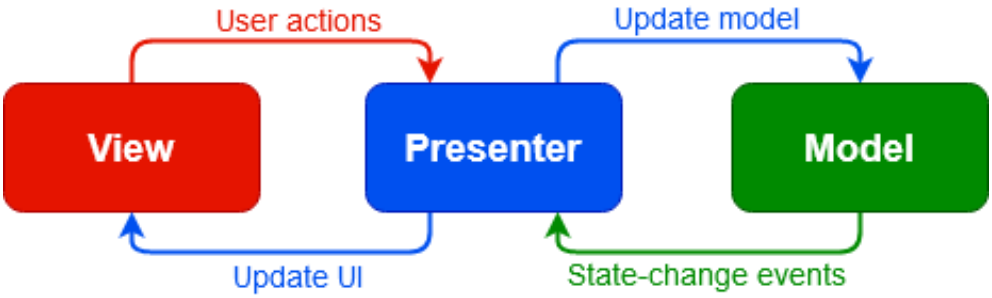
Obrázek 6.1 Ukázka uživatelského rozhraní aplikace

Cílem bylo udělat uživatelské rozhraní co nejjednodušší, aby se v něm uživatel snadno a rychle zorientoval. Na každé stránce jsou zobrazeny pouze potřebné komponenty k vytvoření daného testu spolu s doplňkovými informacemi, které souvisí s danou částí. Je tedy přesně určeno, jaká akce má být v daný okamžik provedena.

6.2.2.2 Architektura

Celá aplikace byla psána dle softwarové architektury MVP[12]. Díky tomu bylo možné rozdělit logiku aplikace a uživatelské rozhraní do částí, které jsou na sobě nezávislé. V budoucnu tak může být framework Vaadin nahrazen jiným frameworkem, a to bez větších obtíží.

[12] Model-view-presenter



Obrázek 6.2 MVP architektura

Obrázek 6.2 zobrazuje rozdělení jednotlivých částí MVP architektury. Cílem této architektury je oddělit logiku aplikace od uživatelského rozhraní.

Jak je vidět na obrázku, výhoda MVP je, že *model* nikdy nekomunikuje přímo s *view* a *view* nekomunikuje přímo s *modelem*. Veškerá komunikace probíhá přes *presenter*, který získává data z modelu a předává je do *view*. V samotném *view* tedy není žádná logika, ale veškerou logiku obstarává vždy *model*.

6.2.2.3 Struktura integračního testu

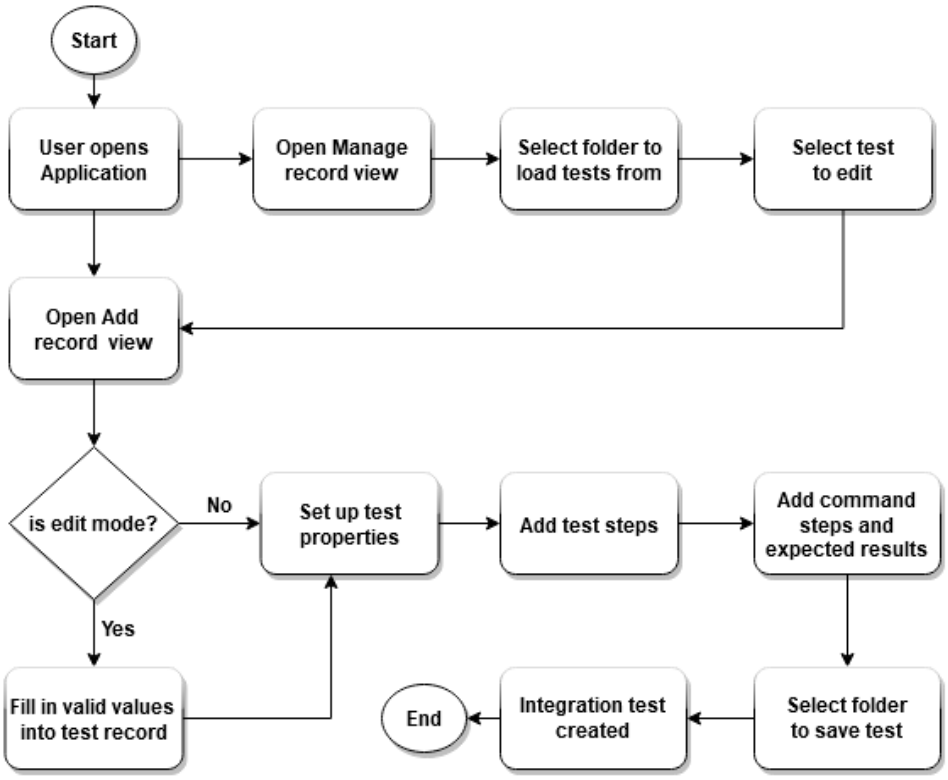
Hlavním cílem této práce bylo vytvoření aplikace, která umožní testerovi (SQE[13]) pohodlně vytvořit integrační test podle jeho požadavků.

Vytvoření integračního testu je rozděleno do několika kroků:

- Nastavení konfigurace pro integrační test
- Vytvoření jednotlivých kroků, tvz. *steps*“, integračního testu
- Přidání očekávaných vstupů a výstupu pro každý krok

K pochopení, jak je integrační test rozdělen do kroků, je na spodním obrázku zobrazen vývojový diagram nástroje pro vytvoření a správu integračních testů.

[13] Software quality engineer



Obrázek 6.3 Vývojový diagram nástroje pro tvorbu integračních testů

Nastavení konfigurace integračního testu

Tato část obsahuje nastavení konfigurace pro daný integrační test.

Konfigurace odpovídá reálnému nastavení ve firmě ENGEL a slouží k přípravě virtuálního prostředí, na kterém bude integrační test spuštěn. Konfigurace byla vytvořena na základě komunikace s interním zaměstnancem firmy, zodpovědným za testování vyvíjených aplikací. Ten specifikoval parametry, potřebné ke správnému otestování aplikace. Bez těchto parametrů není možné správně ověřit funkcionalitu aplikace.

Při vytváření testu je uživatelem přednastaveno, na kterém virtuálním stroji a s jakými parametry má být tento integrační test spuštěn.

Všechny parametry procházejí validací a nemůže tak být vložena hodnota, která není v interním zázemí firmy povolena. V případě, že by byla některá hodnota podvrhnuta (skrze editaci souboru v operačním systému), dojde k odstranění této nevalidní hodnoty v rámci editování daného testu. Díky tomu je zajištěna velká stabilita aplikace.

Obrázek 6.4 Ukázka validní konfigurace integračního testu

Vytvoření jednotlivých kroků

Jak již bylo zmíněno v kapitole *4.1 Test case*, každý *test case* obsahuje seznam akcí, které jsou během testu prováděny postupně za sebou, aby byla otestována určitá funkcionalita. Tyto akce zde reprezentuje krok (*step)*.

Krok není nic jiného než metoda, která obsahuje seznam vstupů a očekávaných výstupů během jednoho kroku spolu s popisem, k čemu daná metoda slouží. Takto vytvořené metody jsou následně spuštěny testovacím frameworkem během provádění integračního testu. Jednotlivé kroky je možné mezi sebou prohazovat a měnit tak pořadí prováděných metod při procesu testování.

```
/**
 * @step=StartBrowser
 * @stepDescription=Start up browser
 */
@Step(1)
public void startBrowser(){
    /** @commandSteps start */
    this.webUITesting.setURL("127.0.0.1");
    this.webUITesting.setPORT("8082");
    this.webUITesting.launchBrowser(Browser.CHROME);
    /** @commandSteps end */

    /** @expectedResults start */
    this.webUITesting.isVisible("//*[@id=\"ROOT-2521314\"]/div");
    /** @expectedResults end */
}
```

Obrázek 6.5 Vygenerovaný krok s očekávanými vstupy a výstupy

Přidání očekávaných vstupů a výstupu pro každý krok

Hlavní částí implementovaného řešení je realizace uživatelských vstupů na testované aplikaci. Ta je rozdělena do dvou částí:

- Uživatelské vstupy
- Očekávané výstupy

Pod uživatelskými vstupy si lze představit sled akcí, které musí tester během jednoho kroku manuálně provést. Jako uživatelský vstup si lze představit např. kliknutí na určité tlačítko na webové stránce nebo změnu jazyka v aplikaci pomocí *comboboxu*.

Očekávané výstupy označují chování aplikace v momentě, kdy jsou všechny uživatelské vstupy (v daném kroku) provedeny. Může sem patřit např. kontrola, zda se po zadání správných přihlašovacích údajů otevřela nová stránka s tímto přihlášeným uživatelem. Dalším příkladem může být kontrola, zda po přepnutí jazyka byl změněn text v aplikaci.

Vytvořený integrační test může vypadat následovně:

```
import org.junit.Test;
import org.junit.runner.RunWith;

@TestCaseInfo(
    Author="Tester",
    ReleaseVersion="2.08.00",
    MinRevision="2.08.00",
    MaxRevision="2.14.00",
    SupportedInRevision={2.14,2.12, 2.08},
    MinUserLevel=13,
    Environment={Environment.ALL_CC300_FAMILY},
    MachineState={MachineState.ALL_OPERATIONAL_STATE}
)

@TestCaseDescription(
/**
 * Simple test for export user
 */
)

@RunWith(TestcaseRunner.class)
public class ExportUserTest extends AbsTestCase {

    /**
     * @step=StartBrowser
     * @stepDescription=Start up browser
     */
    @Step(1)
    public void startBrowser(){
        /** @commandSteps start */
        this.webUITesting.setURL("127.0.0.1");
        this.webUITesting.setPORT("8082");
        this.webUITesting.launchBrowser(Browser.CHROME);
        /** @commandSteps end */
    }

    /**
     * @step=navigate
     * @stepDescription=Navigate to manage users view.
     */
    @Step(2)
    public void navigate(){
        /** @commandSteps start */
        this.webUITesting.clickOn("//*[@id=\"ROOT-2521314\"]/div/div[2]/div[3]/div/span");
        /** @commandSteps end */
    }
```

Obrázek 6.6 Příklad vytvořeného integračního testu ve formátu Java

Na obrázku 6.6 je zobrazen výsledný formát testu, který byl vytvořen pomocí nástroje pro tvorbu integračních testů. Výstupem nástroje je vytvořená Java třída, obsahující všechny potřebné části, které byly popsány. Tato třída bude dále využita testovacím frameworkem při novém procesu testování.

6.2.3 Implementace nástroje pro tvorbu integračních testů

Tato část obsahuje ukázku a popis důležitých části logiky aplikace. Jednotlivé ukázky nejsou samostatně funkční.

Generování struktury integračního testu

Hlavní částí této aplikace je vygenerování integračního testu.

Pokud uživatel při vytváření testu provede určitou uživatelskou akci (nastavení stavu mašiny, změna podporované verze), je automaticky vygenerována aktuální struktura tohoto testu. Tato struktura je po celou dobu vytváření testu viditelná v okně *generated content.* Vytvoření struktury zajišťuje metoda *generateContentForClass*, popsaná níže.

```
private String generateContentForClass(List<RecordedCommandData> data,
        PropertiesData propertiesdata) {
    this.propertiesdata = propertiesdata;
    this.data = data;
    StringBuilder sb = new StringBuilder();
    sb.append(getRecordImports()).append(NEW_LINE);
    sb.append(createAnnotationWithProperties(propertiesdata));
    sb.append(getTestCaseDescription(propertiesdata
            .getDescriptionTextAreaValue()));
    sb.append(RUN_WITH_TESTCASE_RUNNER_CLASS).append(NEW_LINE);
    sb.append(getExtension(this.classFileName)).append(NEW_LINE)
            .append(NEW_LINE);
            data.forEach(
            record -> sb.append(
                    createRecordedData(record, data
                    .indexOf(record) + 1))
            .append(NEW_LINE));
    sb.append(StringUtils.RIGHT_CURLY_BRACER);
    return sb.toString();
}
```

Samotná metoda, zodpovědná za vygenerování obsahu integračního testu, je složena z několika dílčích částí. Aby struktura testu mohla být vygenerována, je nutné získat informace, potřebné k jejímu vytvoření. O zajištění správných informací se starají datové objekty *RecordedCommandData* a *PropertiesData*.

První zmíněný, *RecordedCommandData*, obsahuje veškeré informace o jednotlivých krocích, které se mají v tomto integračním testu provést. Tato data obsahují nejdůležitější

informaci v celém testu, seznam jednotlivých vstupů a očekávaných výstupů během testu. Přesněji během jednoho kroku testu.

PropertiesData obsahují konfiguraci integračního testu. Ta je následně využita při přípravě virtuálního prostředí před samotným provedením testu.

Nastavení vstupní konfigurace pro integrační test

Aplikace umožnuje uživateli nastavit u některých konfiguračních parametrů více hodnot. Klasická *ComboBox* komponenta ale tuto funkcionalitu nepodporuje. Bylo nutné najít náhradu za tuto komponentu. Po analýze byla objevena volně dostupná knihovna *MultiComboBox*, vytvořená přímo pro framework Vaadin. Ta umožnuje zvolit více hodnot.

Pro správné nastavení konfigurace testu musel být upraven výstupní formát pro parametry, které umožnují zvolit více hodnot. Toho bylo docíleno pomocí metody *convertComboBoxDataToEnums*.

```
public String convertComboBoxDataToEnums(Set<MultiComboBoxData> state) {
    StringBuilder sb = new StringBuilder();
    List<MultiComboBoxData> list = new ArrayList<>();
    list.addAll(state);
    sb.append("{");
    list.forEach(data -> {
        sb.append(data.getComboBoxState());
        if (list.indexOf(data) != list.size() - 1) {
            sb.append(", ");
        }
    });
    sb.append("}");
    return sb.toString();
}
```

Tato metoda vezme vybraná vstupní data a ta následně převede do stanoveného formátu. S těmito převedenými daty lze poté dále pracovat.

Získání dostupných příkazů

Důležitou částí pro vytvoření integračního testu jsou příkazy, sloužící pro ovládání grafického rozhraní během provádění testu. Pro zajištění příkazů byla vytvořena metoda *loadAvailableCommmands*.

```
public List<AvailableCommandsData> loadAvailableCommands() {
    List<AvailableCommandsData> recordCommandData = new ArrayList<>();

    String fileContent = getFileContentFromFile(getWebUiFileLocation());
    List<String> fileContentList = Arrays
        .asList(fileContent.split(StringUtils.NEW_LINE));
    fileContentList.stream()
            .filter(line -> fileContentList.indexOf(line) > 0
                    && fileContentList.get(
                        fileContentList.indexOf(line) - 1)
                    .contains(METHOD_FIND_PATTERN))
            .forEach(line -> {
                AvailableCommandsData availableCommandsData =
                    getAvailableCommandsData(line);
                recordCommandData.add(availableCommandsData);
            });

    return recordCommandData;
}
```

Metoda získává data přímo ze třídy *WebUiTesting.java*, která je součástí již vytvořeného testovacího frameworku. Z ní jsou vždy získány aktuální dostupné příkazy, které framework podporuje. Nemůže se tedy stát, že by byl vytvořen test s příkazem, který není podporován.

Obsah třídy je filtrován na základě klíčových slov, která určují použitelné metody pro integrační test. Z nich je následně vytvořen list objektů *AvailableCommandData*.

K získání těchto objektů byla vytvořena metoda *getAvailableCommandsData*, pomocí které je rozdělen vstupní řetězec na několik dílčích částí. Následně, pomocí návrhového vzoru *builder*, je vytvořen nový datový objekt se všemi potřebnými vlastnostmi, jako je například jméno metody, počet parametrů nebo datový typ těchto parametrů.

```
private AvailableCommandsData getAvailableCommandsData(String line) {
    String[] separatedMethod = line.replaceAll("[){]",
        StringUtils.EMPTY_STRING).split("\\(");
    String[] methodTypeAndName = separatedMethod[0]
        .replace("public", StringUtils.EMPTY_STRING).trim()
        .split(StringUtils.SPACE);
    List<String> parameterList = getMethodParameters(
        separatedMethod[separatedMethod.length - 1]);
    List<String> paramTypes = new ArrayList<>();
```

```
        parameterList.forEach(param -> {
            paramTypes.add(param.split(StringUtils.SPACE)[0]);
        });

        return AvailableCommandsData.builder()
            .setMethodName(methodTypeAndName[1])
            .setMethodReturnType(methodTypeAndName[0])
            .setMethodParameters(parameterList)
            .setParametersTypes(paramTypes)
            .setMethodCommandHints(line).build();
    }
```

Validace příkazů

Jedním z klíčových prvků při provádění testu je ověření, že každý příkaz je validní, tedy že splňuje určitá vstupní kritéria. Jako kritérium se dá označit například správný datový objekt, správný počet parametrů nebo validní hodnota těchto parametrů.

Validaci příkazů zajišťuje metoda *isValueMatchingCommandParametrs*.

```
    private boolean isValueMatchingCommandParameters(String value) {
        List<String> parameterTypes = getSelectedCommand()
            .getParameterTypes();
        if (parameterTypes.size() == ONE_PARAMETER) {
            return checkIfValueIsValidForOneParameter(value,
                 parameterTypes);
        } else if (parameterTypes.size() == TWO_PARAMETERS) {
            return checkIfValueIsValidForTwoParameters(value,
                 parameterTypes);
        }
        return parameterTypes.isEmpty();
    }
```

Nejprve je zjištěn počet validních parametrů spolu s jejich datovými typy. Na základě zjištěných parametrů je následně provedena validace těchto parametrů.

V případě, že příkaz obsahuje dva parametry, je provedena kontrola obou parametrů pomocí metody *checkIfValueIsValidForTwoParameters*.

```
    private boolean checkIfValueIsValidForTwoParameters(String value,
         List<String> types) {
        this.parameterTypes = types;
        String[] values = value.split(",");
        if (values.length == TWO_PARAMETERS) {
            boolean isFirstParameterValid = values[0].trim()
                    .matches(getRegexForType(types.get(0)));
```

```
            boolean isSecondParameterValid = values[1].trim()
                    .matches(getRegexForType(types.get(1)));

            return isFirstParameterValid && isSecondParameterValid;
        }
        return false;
    }
```

Parametry jsou vždy vkládány do jednotného textové pole a jsou odděleny čárkou. Nejprve je rozdělen textový řetězec s parametry pomocí metody *split*. Následně je provedena validace obou parametrů pomocí regulárního výrazu.

K ověření, zda příkaz má správný formát, byla naimplementována metoda *getRegexForType*. Ta vrátí, dle zadaného typu, regulárního výraz, podle kterého je každý parametr ověřen.

```
    private String getRegexForType(String type) {
        switch (type) {
        case "String":
            return STRING_MATCHER;
        case "int":
            return DIGIT_MATCHER;
        case "Browser":
            return SUPPORTED_BROWSER_MATCHER;
        default:
            throw new UnsupportedOperationException("not supported");
        }
    }
```

Použití existujících příkazů

K zajištění co největší efektivity při vytváření integračních testů byla do aplikace doimplementována funkce pro použití již existujících příkazů. Díky tomu může uživatel vytvořit test z již existujících příkazů, uložených v listu, a to za pomocí pár kliknutí. Doba vytvoření testu je tak mnohonásobně nižší.

Vytvoření listu obsluhuje metoda zobrazená níže. Ta, v momentě přidání příkazu, automaticky kontroluje, zda příkaz v tomto listu existuje či nikoliv. Pokud příkaz není v listu obsažen, dojde k jeho přidání.

```
    private void addCommandToCachedCommandsIfNotExistsYet() {
        CommandData commandData = new CommandData(getSelectedCommand(),
            getCommandText());
```

```
        Optional<CommandData> isCommandAlreadyIncludedInCacheTree =
            cachedCommands.stream()
                .filter(e -> e.getCommandTextFormat()
                .contentEquals(commandData.getCommandTextFormat()))
                .findAny();

        if (!isCommandAlreadyIncludedInCacheTree.isPresent()) {
            cachedCommands.add(commandData);
            listener.onCachedCommandChanged(getCachedCommands());
        }
    }
```

FileChooser

Nedílnou součástí aplikace je možnost vytvořený test uložit do souborového systému nebo zvolit adresář, ze kterého budou načteny již vytvořené testy.

Během implementace těchto funkcionalit bylo zjištěno, že Vaadin neobsahuje komponentu *FileChooser* a uživatel nemá možnost, jak uložit soubor na specifické místo na disku. Po následné analýze tohoto problému byl, jako nejlepší řešení, zvolen *JFileChooser*, který je součástí balíčku *javax.swing*. i ten měl však svoje limity.

Velkým problémem bylo zobrazení dialogu pro uložení. *JFileChooser* potřebuje jako parametr komponentu, ve které má být zobrazen. Ta musí být z balíčku *javax.swing*. Jako parametr je možné vložit i hodnotu *null*. S tímto nastavením funguje *JFileChooser* sporadicky. V případě použití více monitorů je dialog vždy zobrazen. Pokud je aktivní jen jeden monitor, dochází v některých případech ke skrytí dialogu za další otevřená okna. To může způsobit situaci, kdy aplikace vypadá, že neodpovídá, i když tomu tak není.

Tento problém byl vyřešen přepsáním defaultní metody *createDialog*, která nyní vždy otevře dialog v popředí.

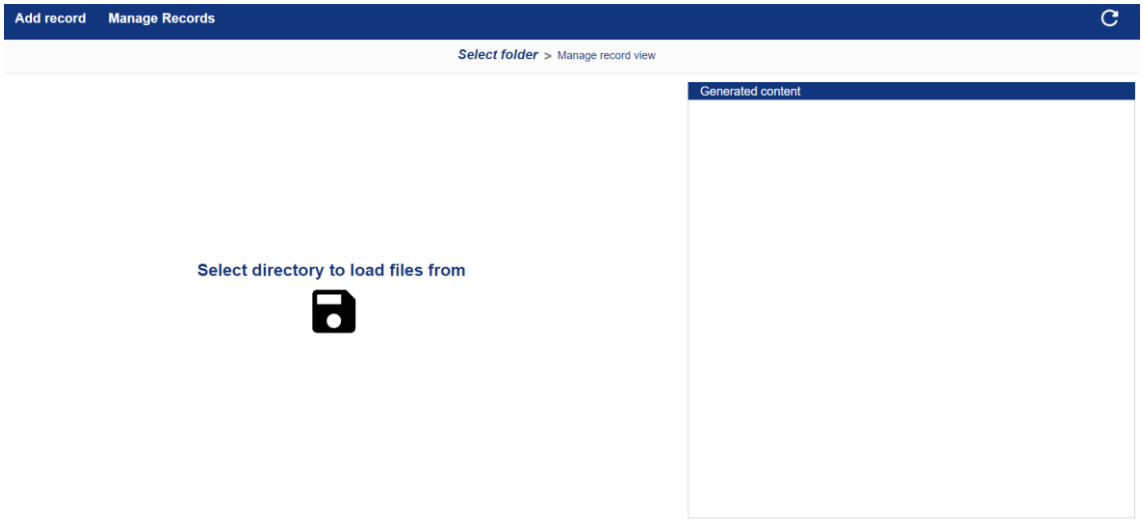
```
  private JFileChooser getJFileChooser() {
          JFileChooser jfc = new JFileChooser() {

              @Override
              protected JDialog createDialog(Component parent) {
                  JDialog dialog = super.createDialog(parent);
                  dialog.setModal(true);
                  dialog.setAlwaysOnTop(true);
                  return dialog;
              }
```

```
                jfc.setDialogType(JFileChooser.SAVE_DIALOG);
                jfc.setVisible(true);
                return jfc;
            }
    }
```

6.2.4 Ukázka nástroje pro vytváření integračních testů

Tato část práce ukazuje příklad využití nástroje pro vytváření integračních testů v praxi.



Obrázek 6.7 Základní stránka aplikace

Na obrázku je zobrazena hlavní stránka aplikace, která se uživateli otevře vždy při zapnutí aplikace. Uživatel je vyzván, aby zvolil adresář, ze kterého budou načteny již vytvořené integrační testy. Jakmile je kliknuta ikona diskety, otevře se dle systému standartní dialogové okno.

V levé horní části obrazovky jsou vidět navigační karty, které obsahují dvě možnosti:

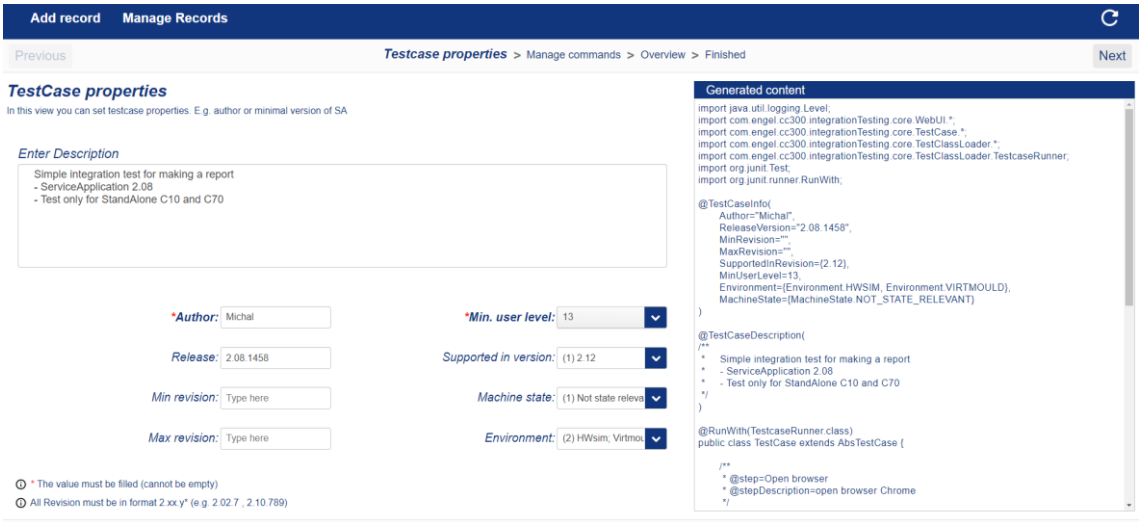
- *Add record* – vytvoření nového integračního testu
- *Manage record* – správa již existujících integračních testů

Obrázek 6.8 Načtení integračních testů ze složky

Navigační menu zůstávají stejná po celou dobu vytváření testu. Jakmile je vybrána složka, ze které mají být načteny integrační testy, otevře se okno s jejich seznamem. To je zobrazeno na obrázku 6.*8 Načtení integračních testů ze složky*. Spolu s tabulkou dostupných testů je vidět aktuální vybraná složka, která je zobrazena hned nad tabulkou.

Složku je možné změnit pomocí tlačítka *change directory*, které je dostupné nalevo od aktuálního obsahu testu.

Pokud je vybrán test, automaticky je zobrazen jeho obsah v pravé části aplikace. Je tak vždy vidět, co daný test obsahuje. Spolu s načtením obsahu dojde také k povolení tlačítek, která se nacházejí napravo od seznamu testů. Ty slouží k mazání nebo editaci vybraného testu.



Obrázek 6.9 Vytváření projektu

První okno, viditelné po zvolení karty *Add record* v levém horním menu, obsahuje prvotní nastavení integračního testu. Jedná se o parametry, které se nastavují před prováděním testu, a slouží ke správnému nastavení virtuálního prostředí pro testování. Lze nastavit například na jakém stroji má být test prováděn, v jakém stavu musí být tento testovaný stroj nebo na jaké verzi aplikace má být integrační test spuštěn.

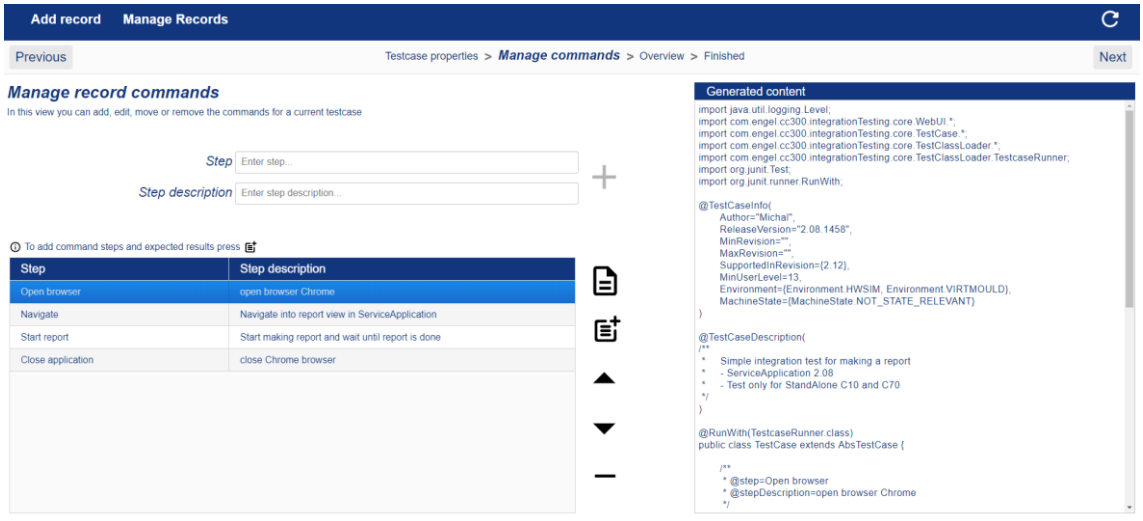
K testu je také možné přidat jeho popis, tzv. *description*. Ten nemá žádný vliv na nastavení vstupní konfigurace testu a slouží pouze jako přídavná informace k danému integračnímu testu. Může obsahovat například popis, k čemu daný test slouží.

V pravé části obrazovky je po celou dobu vytváření integračního testu zobrazen aktuální obsah, který je v případě jakékoliv uživatelské změny vždy nově vygenerován.

Součástí okna je také navigační část obsahující tlačítka *previous* a *next*, určená pro přepnutí fáze vytváření. Navigační tlačítka jsou povolována dynamicky na základě vyplněných parametrů.

Jednotlivé fáze vytváření, spolu s aktuální, jsou zobrazeny v horní části obrazovky.

Po celou dobu vytváření testu je možné vyčistit všechna existující data a začít znovu. Tato možnost je dostupná v pravém horním rohu.



Obrázek 6.10 Vytváření kroků integračního testu

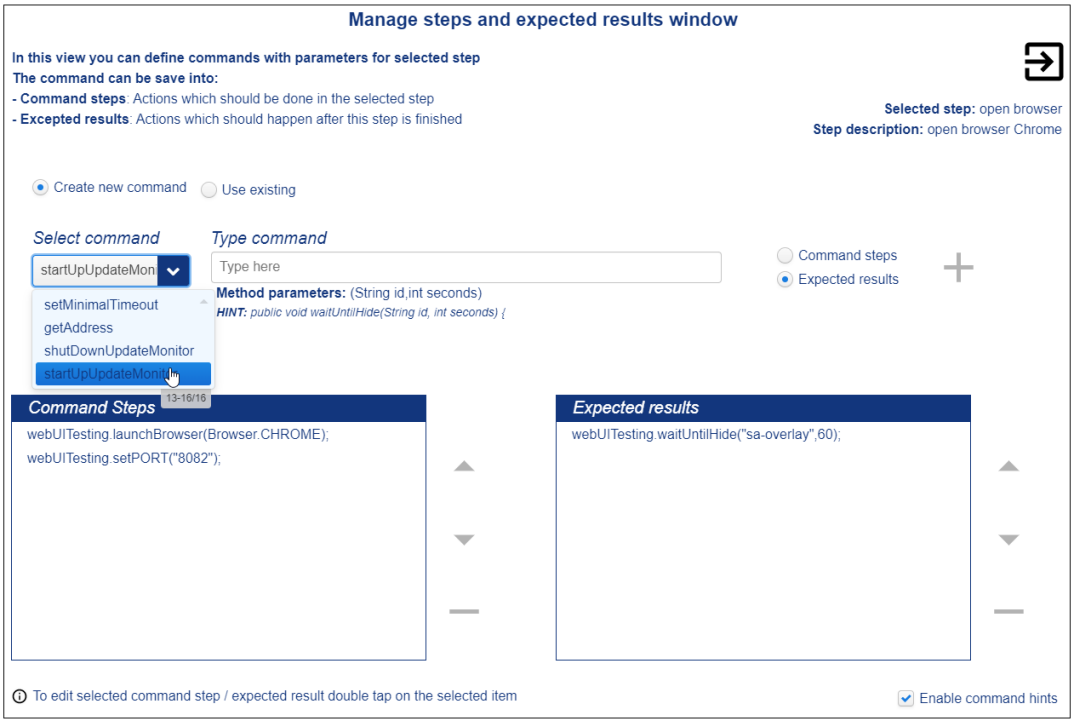
Další fází vytváření integračního testu je přidání jednotlivých kroků, které je zobrazeno na horním obrázku. Každý krok si lze představit jako množinu vstupů a očekávaných výstupů, které jsou prováděny za sebou.

Aby mohl být krok přidán, musí být vyplněno pole *step* a *step description*. *Step description* není nic jiného než informace pro uživatele, k čemu je daný krok určen. *Step* slouží jako identifikační metoda, která je následně provedena v integračním testu. Aby bylo možné přidat záznam do tabulky, musí být obě textová pole vyplněna a *step* již nesmí existovat v tabulce.

Každý krok je vložen do tabulky, podle které je určeno pořadí kroků testu. Pokud dojde ke změně v tabulce, například ke změně pořadí elementů nebo k odebrání elementu z tabulky, je automaticky změněn vygenerovaný obsah. Je tedy zaručeno, že je pořadí jednotlivých příkazů správné.

Pokud je vybrán element z tabulky, dojde automaticky k povolení ovládacích tlačítek, která se nacházejí vpravo od tabulky. Ovládací tlačítka umožnují měnit pořadí prováděných metod (kroků), změnit jejich název nebo přidat další informace pro tento test pomocí tlačítka *add command steps and expected results.*

Tato funkcionalita je zobrazena na dalším obrázku.



Obrázek 6.11 Vyskakovací okno pro vybraný *step*

Na obrázku je zobrazeno vyskakovací okno, které slouží k přidání jednotlivých příkazů k vybranému kroku. Všechny příkazy jsou přesně dané a je pouze definována jejich hodnota. Hodnot může být i více, dle zvoleného příkazu. Parametry příkazu jsou

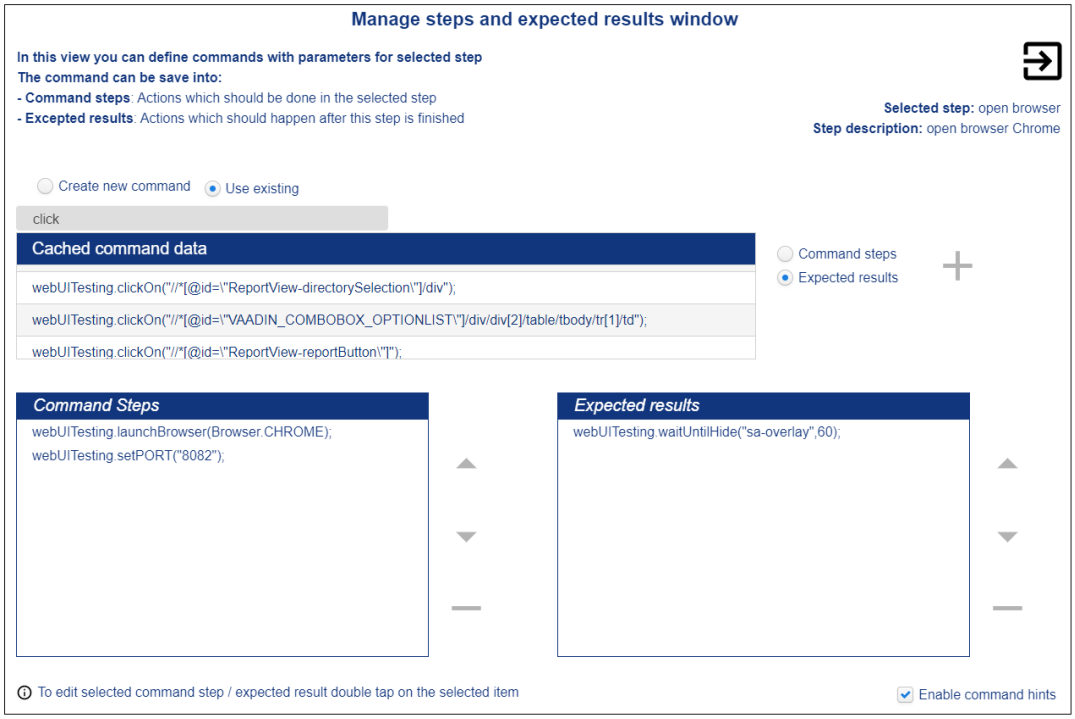
spolu s nápovědou zobrazeny pod textovým polem. Nápovědu je možné zrušit pomocí checkboxu v pravém dolním rohu.

Pro všechny metody je kontrolován datový typ proměnných. Pokud je zadán špatný datový typ parametru, nelze vložit příkaz do tabulky. Je tak zajištěno, že integrační test neselže během jeho provádění kvůli špatnému datovému typu u metody. Příkaz může být přidán do jedné ze dvou tabulek, pomocí kterých je určeno pořadí provádění příkazů ve zvoleném kroku.

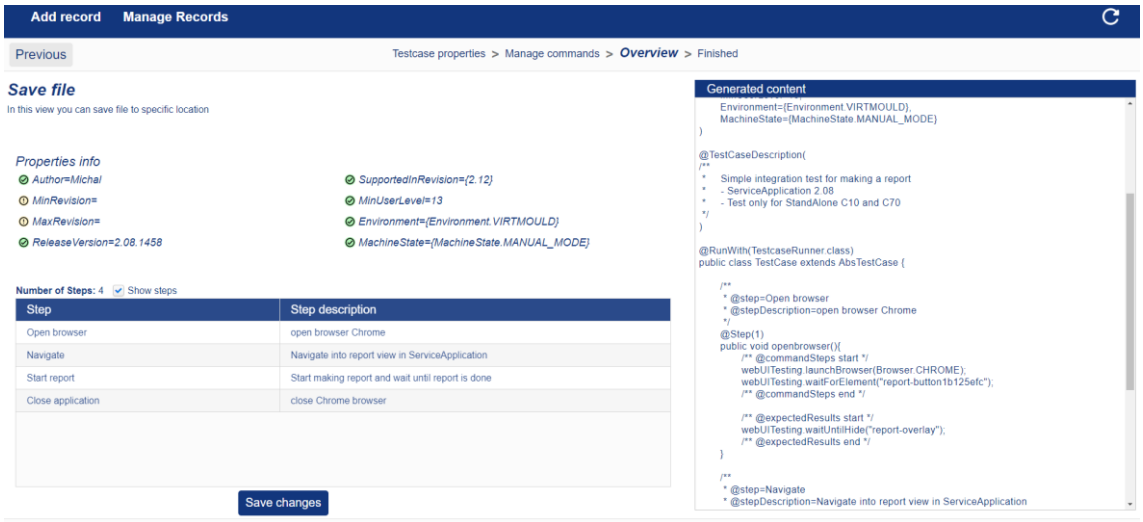
V pravém horním rohu se nachází tlačítko pro ukončení okna. Pokud došlo ke změnám v tabulce, je možno tato data uložit nebo pokračovat beze změn.

Pod tlačítkem pro ukončení okna je vidět aktuální krok, do kterého jsou příkazy přidávány. V pravé dolní části je možnost povolit nebo vypnout nápovědu k příkazům.

Do obou tabulek mohou být vloženy i již existující příkazy. Ty jsou dostupné při zvolení možnosti *use existing* v přepínacím menu, které se nachází nad volbou příkazu. Pokud je vytvořen nový příkaz, je automaticky vložen do tohoto seznamu. V seznamu je možné vyhledávat a příkazy tak mohou být snadno vloženy do jedné z tabulek. Tato funkcionalita je zobrazena na spodním obrázku.



Obrázek 6.12 Vložení z již existujících příkazů

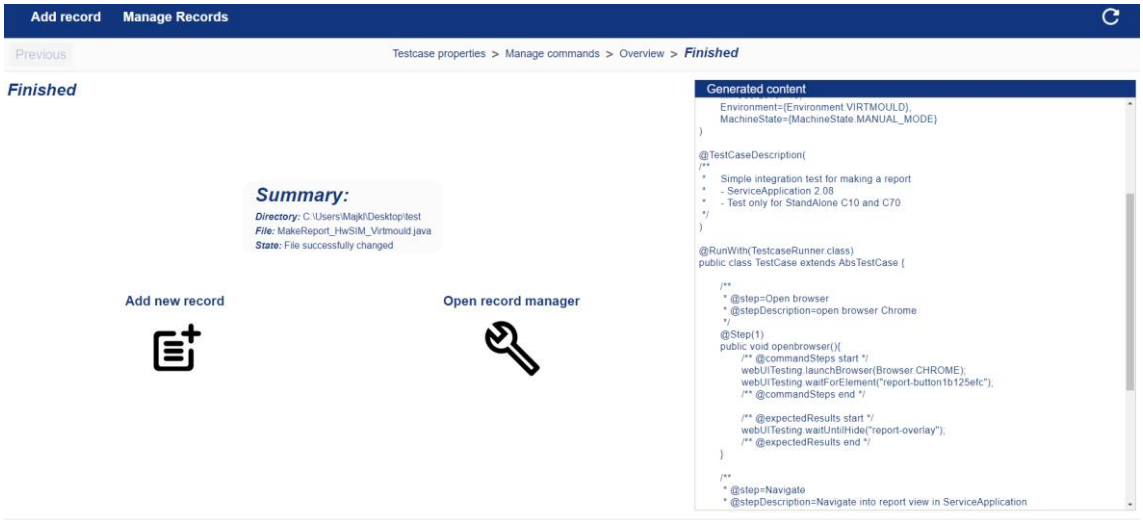


Obrázek 6.13 Přehled testu

Předposlední fází vytváření integračního testu je jeho celkový přehled, tzv. *Overview*, který je zobrazen na horním obrázku. Ten obsahuje všechny důležité informace o testu, který má být vytvořen. Jsou zde zobrazeny všechny informace o vstupní konfiguraci testu.

V levé spodní části jsou zobrazeny všechny vytvořené kroky testu spolu s možností ukončit vytváření testu a uložit (vygenerovat) test do souboru.

Spodní obrázek ukazuje poslední fázi vytváření testu, kterou je jeho sumarizace. Ta obsahuje informace o vytvořeném testu (kam byl uložen, zda byl úspěšně vytvořen) a možnost vytvořit nový test nebo zobrazit již vytvořené testy.



Obrázek 6.14 Úspěšné uložení testu

6.3 Nástroj pro unifikované spouštění již existujících integračních testů

Kapitola obsahuje analýzu, návrh a implementaci konfigurace virtuálního prostředí pro unifikované spouštění již existujících integračních testů. Jedná se o druhou aplikaci, která byla v této diplomové práci vytvořena.

6.3.1 Analýza nástroje pro spouštění integračních testů

Další částí, které se tato diplomová práce věnuje, je nastavení prostředí pro spouštění integračních testů.

Než došlo k návrhu a samotné implementaci tohoto řešení, byla provedena analýza požadavků pro spouštění integračních testů. Analýza je jednou z nejdůležitějších částí vývoje software. Pomáhá zjistit případné nejasnosti v projektu ještě před tím, než dojde k samotnému návrhu a následné implementaci nástroje. Byly stanoveny základní požadavky, které by měl nástroj splňovat:

- Zvolení vstupní konfigurace
- Příprava a spuštění specifického stroje (prostředí)
- Zajištění potřebných dat (vybrané testy, testovací framework) na zvoleném prostředí a provedení dostupných testů v tomto prostředí

Zvolení vstupní konfigurace

Základní požadavkem nástroje je umožnit uživateli, aby si zvolil integrační testy, které mají být provedeny při *test runu*, a virtuální prostředí, na kterém má být *test run* spuštěn.

Možností, jak tohoto požadavku docílit, je hned několik. Nejjednodušší cestou by bylo manuální vytvoření konfiguračního souboru, který by obsahoval výše zmiňované informace (seznam testů, testovací prostředí). Vzhledem k tomu, že bude nástroj využíván zejména software quality inženýrem, by toto řešení nebylo optimální. Tester by musel znát přesnou syntaxi konfigurace. V případě velkého množství integračních testů by mu samotné nastavení konfigurace zabralo spoustu času.

Očekává se, že uživatel, který bude nástroj používat, nemá žádné nebo minimální znalosti programovaní. Pro nastavení vstupní konfigurace by proto měla být vytvořena aplikace, ve které budou zvoleny všechny požadované vlastnosti pro *test run*.

Příprava a spuštění zvoleného prostředí

Hlavním požadavkem nástroje je možnost provedení více *test runů* s různými konfiguracemi v jeden okamžik. Bylo nutné zajistit dostupnost více běhových prostředí.

Vzhledem k požadavkům se jako vhodné řešení ukázalo být použití virtualizačního nástroje, pomocí kterého je možné nakonfigurovat a vzdáleně spustit testovací stroj. Těchto strojů může být spuštěno v jeden okamžik i více a je tak možné najednou testovat více integračních testů s různými konfiguracemi. Navíc, díky použití virtuálního systému, není nijak ovlivněn uživatelský systém.

Tento nástroj musí pro splňovat následující kritéria:

- Obsahuje knihovny pro ovládání virtuálních strojů skrze jazyk Java
- Možnost spravovat více virtuálních strojů v jeden okamžik
- Dostupný zdarma

Zajištění potřebných dat na zvoleném prostředí

Jednou z nejdůležitějších částí této analýzy bylo zajištění všech dat, která jsou nutná k provedení testu, na vybraném virtualizačním prostředí. Zde musely být zjištěny dostupné možnosti, které umožní přesun všech potřebných dat do virtuálního obrazu.

Provedení dostupných testů

Poslední částí analýzy bylo zjištění, jak spustit dostupné testy na zvoleném virtuálním prostředí. Bylo nutné získat více informací o testovacím frameworku, např. jak se spouští nebo které parametry potřebuje ke správnému provedení testu. Zde probíhala komunikace s kolegou A. Czarnosmskim. Ten se staral o vývoj tohoto frameworku a dokázal přesně specifikovat, jak správně nastavit a spustit testovací framework se všemi potřebnými parametry.

6.3.2 Návrh řešení nástroje pro spouštění integračních testů

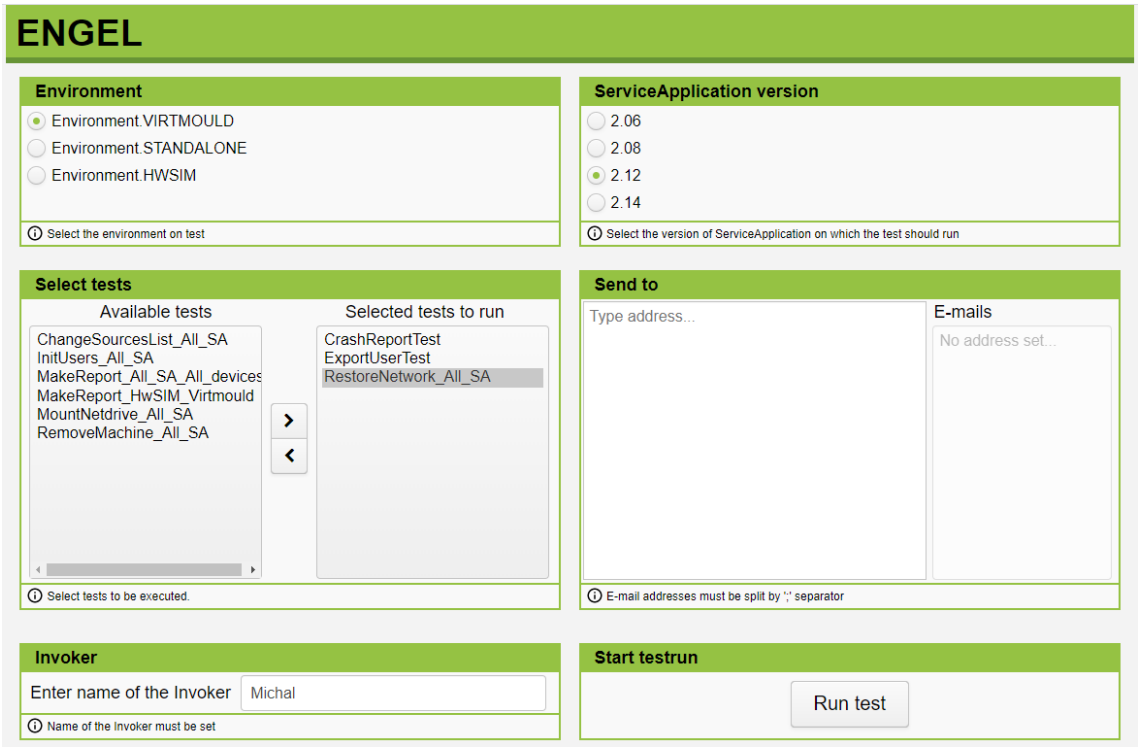
Na základě zjištěné analýzy byl pro přípravu virtuálního prostředí zvolen nástroj *Oracle VirtualBox,* verze 5.2.30. Ten je dostupný zdarma, umožnuje spravovat více virtuálních strojů v jeden okamžik a co je hlavní, má dostupné knihovny pro ovládání virtuálních strojů pomocí jazyku Java, který je pro funkcionalitu naší aplikace rozhodující.

Nástrojů pro virtualizaci je dostupných hned několik. Jako další lze zmínit například *VMware*, *Parallels* nebo *Hyper-V*. V případě potřeby může být *VirtualBox* nahrazen jiným nástrojem, který musí splňovat kritéria, specifikována v analyzační části.

Pro nastavení vstupní konfigurace pro *test run* byla vytvořena jednoduchá webová aplikace, která byla napsána v programovacím jazyce Java s použitím frameworku Vaadin, který byl využit pro grafické rozhraní aplikace. Obě technologie jsou ve společnosti ENGEL používány. Je tak zajištěn snadný rozvoj aplikace i mimo tuto diplomovou práci.

V aplikaci je zvolen seznam testů, které mají být otestovány. Spolu se seznamem testů je zvoleno testovací prostředí a verze aplikace, na které mají být testy provedeny. Poté dojde k provedení *test runu*.

Výsledný design aplikace je zobrazen na spodním obrázku.



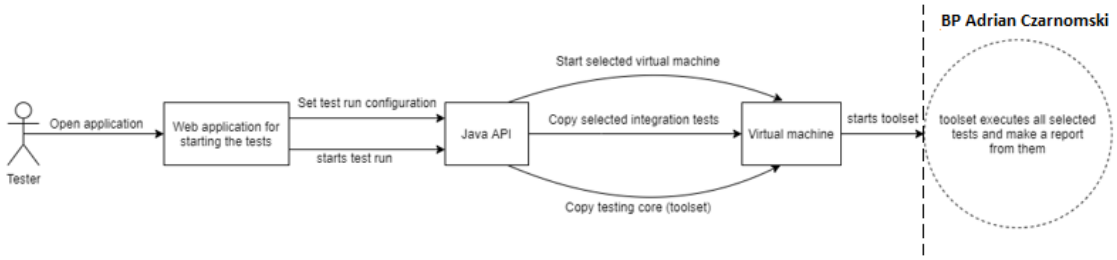
Obrázek 6.15 Design aplikace pro spouštění již existujících testů

Na základě stanovených parametrů (*Environment*, *ServiceApplication version*) jsou zobrazeny dostupné integrační testy a uživatel si pouze specifikuje, které testy chce provést. Jakmile jsou zvoleny testy a je vyplněno pole *Invoker*, může uživatel spustit test pomocí tlačítka *Run test*, které nastaví a spustí virtuální prostředí.

Jak je vidět na obrázku 6.16, uživatel nejdříve zvolí konfiguraci pro *test run* spolu se seznamem testů, které mají být provedeny. Poté dojde k zapnutí *test runu*, při kterém je spuštěn zvolený virtuální stroj pomocí Java VirtualBox API, a k překopírování vybraných testů do tohoto stroje spolu s testovacím frameworkem. Jakmile je zvolený virtuální stroj spuštěn, dojde k provedení všech dostupných testů na tomto virtuálním stroji a jejich vyhodnocení.

Vytvořená aplikace se stará pouze o přípravu a nastavení virtuálního prostředí a spuštění testovacího frameworku, nikoliv o samotné provedení a vyhodnocení testů.

Provedení a vyhodnocení jednotlivých testů během *test runu* není součástí této práce. Tato část byla navržena a naimplementována kolegou A. Czarnomskim a je součástí jeho bakalářské práce.



Obrázek 6.16 Flow diagram nástroje pro spouštění již existujících testů

Nastavení virtuálního prostředí a jeho spuštění

Pro nastavení virtuálního prostředí byl použit již existující virtuální stroj, který se ve firmě ENGEL používá. Z něj byly následně vytvořeny ostatní virtuální stroje, které obsahují všechna nastavení, potřebná pro správné provedení testů na zvoleném běhovém prostředí. K výběru tohoto stroje přispělo i to, že všechny manuální testy ve společnosti jsou prováděny na tomto virtuálním stroji, který je vždy upraven pro potřeby aktuálního *test runu*. Toto řešení se proto se jeví jako ideální.

Pro zajištění správné funkcionality byla do virtuálního stroje přidaná knihovna *Java jre1.8 a* byl doinstalován prohlížeč *Chromium* verze 79.0.3945 spolu se všemi závislostmi.

K zaručení automatického spouštění testů na zvoleném stroji byl naimplementován skript *starttestrun.sh*, který byl přidán do seznamu aplikací, automaticky zapnutých po načtení systému. Tento skript zapne testovací framework s potřebnými parametry.

```
#!/bin/bash
testfolder="/media/sf_netdrive/test_to_run"
finishedfolder="/media/sf_netdrive/test_finished/"

java -jar $testfolder/core.jar --test-case-folder $testfolder
    --results-folder $finishedfolder
```

Pokud bude chtít uživatel otestovat specifický test na verzi aplikace 2.14 a nastaví v konfiguraci aplikace parametr *Environment.Virtmoud*, dojde k zapnutí virtuálního stroje, který odpovídá těmto parametrům. V tomto případě bude spuštěn druhý virtuální stroj, zobrazen na spodním obrázku (*2.14_Virtmould*).

Obrázek 6.17 Ukázka virtuálních strojů pro testování

6.3.3 Implementace nástroje pro spouštění integračních testů

V této části práce jsou popsány některé důležité části implementace nástroje pro spouštění integračních testů. Části kódu nejsou samostatně funkční. Vzhledem k tomu, že budou oba vytvořené nástroje v této diplomové práci použity ve firmě ENGEL, není kompletní zdrojový kód dostupný.

Spuštění virtuálního stroje

O spuštění správného virtuálního stroje se stará metoda *startVirtualMachine* ze třídy *StartTestModel*.

```
private boolean startVirtualMachine(String machineName) {
   try {
       VirtualBoxManager virtBoxManager = getVBoxManager();
       IVirtualBox vbox = virtBoxManager.getVBox();
       IMachine machine = vbox.findMachine(machineName);
```

```
            ISession session = virtBoxManager.getSessionObject();
            IProgress progress = machine.launchVMProcess(session, "", "");

            while (!progress.getCompleted().booleanValue()) {
                progress.waitForCompletion(2000);
            }

            progress.releaseRemote();
            unlockMachine(session);
            return true;
        } catch (VBoxException e) {
            LOGGER.log(Level.WARNING, e.getMessage(), e);
            throw new VBoxException("Failed to start machine", e);
        }
    }
```

Nejprve je získána instance třídy VirtualBoxManager, pomocí které probíhá komunikace s virtualizačním nástrojem přes SOAP[14] protokol. Aby bylo možné komunikovat přes tento protokol, je nutné, aby běžel proces VirtualBox serveru (*VboxWebSvc*) na hostovaném serveru. Následně je získána instance objektu IVirtualBox, přes kterou je vyhledán zvolený virtuální stroj na konkrétním VirtualBox hostu. Jakmile je zvolený stroj nalezen, dojde k jeho spuštění.

Jak již bylo zmíněno výše, o komunikaci s virtualizačním nástrojem se stará metoda *getVboxManager*. Přes tuto metodu je na základě operačního systému navázáno spojení na specifickou adresu, kde je dostupná služba *VboxWebSvc*. Ta slouží ke komunikaci s nástrojem VirtualBox přes SOAP protokol. Nakonec dojde k ověření, zda bylo spojení se zmíněnou službou úspěšně vytvořeno.

```
    private VirtualBoxManager getVBoxManager() {
        VirtualBoxManager virtBoxManager = null;
        try {
            virtBoxManager = VirtualBoxManager.createInstance(null);
            if (SystemUtils.isSystemWindows()) {
                virtBoxManager.connect(VBOX_ADDRESS_LOCAL, null, null);
            } else {
                virtBoxManager.connect(VBOX_ADDRESS_LINUX, null, null);
            }

            IVirtualBox vbox = virtBoxManager.getVBox();
```

[14] Simple Object Access Protocol

```
            if (vbox == null) {
                throw new NullPointerException("No vboxManager found");
            }
            return virtBoxManager;
        } catch (VBoxException e) {
            LOGGER.log(Level.WARNING, e.getMessage(), e);
            throw new VBoxException("No virtualBoxManager found");
        }
    }
```

Výhodou použití SOAP protokolu je možnost komunikovat s více VirtualBox hosty v jeden okamžik. Tato komunikace probíhá přes webové služby pomocí HTTP protokolu a jazyk XML.

6.4 Testování vytvořených nástrojů

Oba nástroje byly po celou dobu vývoje testovány samotným vývojářem, aby došlo k odstranění chyb již během ranné fáze vývoje. K nalezení chyb přispěl také kolega, zodpovědný za vývoj nástroje pro provedení testů (testovací framework). Ten měl vždy přístup k aktuální verzi nástrojů skrz verzovací nástroj Git. Díky jeho zpětné vazbě bylo možné rychle a efektivně reagovat na nalezené problémy v implementaci.

K zajištění co největší kompatibility nástrojů probíhalo testování vždy na dvou operačních systémech: Windows a Linux. Testování na sytému Linux probíhalo vzhledem k dostupným možnostem na virtuálním stroji.

Operační systém Mac OS není v tuto chvíli ve firmě používán pro testovací účely. Nebylo tak možné otestovat implementované nástroje na tomto systému.

Pro nalezení případných chyb v návrhu uživatelského rozhraní byla webová aplikace testována v prohlížečích Chrome a Firefox, které patří v dnešní době mezi nejpoužívanější.

Hlavním cílem testování bylo ověřit, zda nástroj pro vytvoření a editaci testu splňuje kritéria, která byla specifikována v kapitole *Návrh řešení nástroje pro tvorbu integračních testů*, tedy zda jsou vstupní data ošetřena a uživatel nemůže vložit nevalidní hodnotu. V případě, že by data byla podvrhnuta v samotném souboru, mělo by v rámci editace dojít k automatickému opravení nevalidních hodnot, případně k jejich smazání.

Samotné testování bylo rozděleno celkem do dvou částí. První část se věnovala testování nástroje pro tvorbu integračních testů. V druhé části byl testován nástroj pro konfiguraci a spouštění integračních testů.

Výsledky testování jsou popsány v další kapitole.

6.4.1 Zhodnocení výsledků

První část se věnovala testování nástroje pro tvorbu integračních testů. Nejprve bylo testováno ošetření vstupních dat, které je, z hlediska použití vytvořeného testu v procesu testování, velmi důležité. Uživatel se snažil během vytváření testu několikrát vložit do všech vstupních polí v aplikaci jak validní, tak nevalidní hodnoty. Validace vstupů dopadla velmi dobře, ani jednou se nepodařilo vložit nevalidní hodnotu do systému.

Dále bylo testováno úspěšné vygenerování testu. Pro měření úspěšnosti bylo vytvořeno celkem 30 testů s různými parametry, aby došlo k co největšímu pokrytí při

testování. Tyto testy byly následně vyzkoušeny na testovacím frameworku. Ten byl schopen všechny vytvořené testy spustit a úspěšně provést. Lze tedy konstatovat, že tato část implementace dopadla nadmíru úspěšně.

Následně byla ověřena schopnost nástroje editovat již existující test. Nejprve bylo testováno, zda dokáže aplikace úspěšně načíst a znovu uložit upravený test, obsahující pouze validní hodnoty. Celkem bylo editováno 30 již vytvořených testů. Zde nebyl zjištěn žádný problém a aplikace byla schopná upravit a správně uložit test ve všech případech.

Nejdůležitější částí bylo zjistit, jak se aplikace zachová v případě, že je test poškozen a obsahuje nevalidní (podvrhnutá) data. Pro tyto účely bylo manuálně upraveno 15 testů, do kterých byly vloženy nevalidní hodnoty. Z 15 testovaných případů dokázala aplikace načíst test a opravit nevalidní data celkem 14krát. V jednom případě došlo během načtení souboru k chybě kvůli datovému formátu, který byl v aplikaci špatně rozpoznán. Tato chyba byla následně opravena a při druhém cyklu bylo opraveno všech 15 testů.

Druhou částí bylo testování nástroje pro konfiguraci a spouštění integračních testů. Zde se nejdříve zjišťovala schopnost systému nastavit a spustit zvolený virtuální stroj. Ze 30 případů byl vždy úspěšně spuštěn zvolený stroj a byla překopírována potřebná data na tento stroj.

Nakonec bylo testováno automatické spuštění testovacího frameworku a provedení zvolených testů na virtuálním stroji. Úspěšně byl spuštěn testovací framework celkem 28krát z celkových 30 pokusů. Jednou se správně nenačetl webový prohlížeč, nebylo tak možné provést zvolené testy. Při dalším neúspěšném pokusu se nepodařilo automaticky přihlásit uživatele do systému a bylo nutné se přihlásit do systému manuálně. Po manuálním přihlášením byl testovací framework spuštěn. Toto však nebyl problém vytvořeného nástroje, ale virtuálního obrazu, na kterém byly testy prováděny.

Na spodní tabulce jsou zobrazeny výsledky testování obou těchto aplikací.

Nástroj pro tvorbu integračních testů	
Ošetření vstupních dat	✓
Úspěšné vygenerování testu	✓
Automatické opravení nevalidních dat v rámci editace	✓
Nástroj pro konfiguraci a spouštění integračních testů	
Příprava a spuštění zvoleného virtuální stroje	✓
Automatické spuštění zvolených testů na virtuálním stroji	✓

Tabulka 6.1 Vyhodnocení testování

6.5 Další rozvoj

Vytvořené nástroje dávají základ pro budoucí automatické testování ve firmě ENGEL, kde je také plánován jejich další rozvoj. Aby byl plně využit potenciál obou vytvořených nástrojů je v plánu tyto nástroje rozšířit a integrovat do dalších interních systémů.

Cílem do budoucna je přidat další funkcionality do obou nástrojů např.

- Vytvoření testu automaticky na základě uživatelských akcí
- Obnovení stavu při pádu aplikace

7 Závěr

Prvním cílem práce bylo popsat již existující nástroje pro automatické testování. Díky tomu bylo získáno větší povědomí o problematice automatického testování, které je v dnešní době stále aktuální. Také byly zjištěny možnosti automatizace testování v současné době.

Hlavním cílem této diplomové práce bylo vytvoření nástroje pro tvorbu integračních testů. Aby bylo dosaženo tohoto cíle, bylo nutné projít celým procesem vývoje aplikace. Na základě již existující analýzy došlo k návrhu řešení, díky kterému mohl být nástroj pro tvorbu integračních testů vytvořen a otestován.

Součástí práce bylo také navrhnout a nakonfigurovat virtuální prostředí pro spouštění námi vytvořených integračních testů. Pro splnění tohoto byla nejprve provedena analýza. Na základě této analýzy následně došlo k vytvoření aplikace, která řeší nastavení konfigurace a správu virtuálních strojů pro spouštění vybraných testů.

Oba vytvořené nástroje byly otestovány na vzorových příkladech a byla ověřena jejich funkcionalita.

Na základě těchto informací lze konstatovat, že se podařilo splnit všechny vytyčené cíle této diplomové práce.

V současné době jsou nástroje ve fázi testování a nasazení ve společnosti ENGEL, kde poskytují solidní základ pro implementaci automatických testů v procesu vývoje software.

Seznam pramenů a literatury

[1] *IEEE Standard Glossary of Software Engineering Terminology: integration testing* 1990. [cit. 2020-03-10]. ISBN 1-55937467-X.

[2] Building Object-Oriented Frameworks [online]. [cit. 2020-02-15]. Dostupné z: https://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf

[3] IEEE Standard Glossary of Software Engineering Terminology: Test case [online].

[cit. 2019-08-21]. ISBN 1-55937467-X.

[4] Test Case [online]. [cit. 2019-08-25]. Dostupné z: http://softwaretestingfundamentals.com/test-case

[5] *How to write a good test case: How do you make your test cases great?* [online].

[cit. 2019-08-25]. Dostupné z: https://www.tesena.com/write-good-test-case

[6] Sample Test Case Template with Test Case Examples: Sample Test Cases [online]. In: . [cit. 2020-04-18]. Dostupné z: https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2017/10/Test-Case-Example1.jpg

[7] SHARMA, Lakshay. *Manual Testing: What is Manual Testing ?* [online].

[cit. 2019-08-28]. Dostupné z: https://www.toolsqa.com/software-testing/manual-testing

[8] *Manual Testing: Why do we need Manual Testing?* [online]. [cit. 2019-08-28].

Dostupné z: https://artoftesting.com/manualTesting/manual-testing.html

[9] Manual testing process: Procedure of Manual Testing [online]. In: . [cit. 2020-04-18].

Dostupné z: https://miro.medium.com/max/676/1*qBcFakOclmDcX0CG6UQpRg.jpeg

[10] *Manual Testing: Disadvantages of Manual Testing* [online]. [cit. 2019-08-28]. Dostupné z: https://artoftesting.com/manualTesting/manual-testing.html

[11] *TECHNOLOGY Q&A: WHAT IS AUTOMATION?* [online]. [cit. 2019-08-30]. Dostupné z: https://www.hcltech.com/technology-qa/what-is-automation

[12] Automation Testing Process [online]. In: . [cit. 2020-04-18]. Dostupné z: https://testguild.com/wp-content/uploads/2018/11/AutomationTestingProcess.png

[13] *Automation Testing: What is Automation Testing?* [online]. [cit. 2019-08-30]. Dostupné z: https://artoftesting.com/manualTesting/automation-testing.html

[14] *What is Automated Testing?: Common Misconceptions About Automated Testing* [online]. [cit. 2019-08-30]. Dostupné z: https://smartbear.com/learn/automated-testing/what-is-automated-testing

[15] *What is Automated Testing?: Automated Testing is Better Than Manual Testing* [online]. [cit. 2019-08-30]. Dostupné z: https://smartbear.com/learn/automated-testing/what-is-automated-testing

[16] *Automation Testing: What not to Automate?* [online]. [cit. 2019-08-30]. Dostupné z: https://artoftesting.com/manualTesting/automation-testing.html

[17] CloudQA, a leader in Record and Playback testing tools: What is a record and playback testing tool? [online]. [cit. 2019-08-01]. Dostupné z: https://cloudqa.io/cloudqa-advanced-record-playback-testing-tools

[18] CloudQA, a leader in Record and Playback testing tools: Reliability of record and playback testing tools [online]. [cit. 2019-08-01]. Dostupné z: https://cloudqa.io/cloudqa-advanced-record-playback-testing-tools

[19] Angie Jones. *10 features every codeless test automation tool should offer: 7. Cross-browser support* [online]. [cit. 2019-08-01]. Dostupné z: https://techbeacon.com/app-dev-testing/10-features-every-codeless-test-automation-tool-should-offer

[20] *Selenium IDE* [online]. [cit. 2019-07-18]. Dostupné z: https://www.seleniumhq.org/docs/02_selenium_ide.jsp

[21] Introduction [online]. In: . [cit. 2020-04-18]. Dostupné z: https://robotframework.org/#examples

[22] *INTRODUCTION [online]. [cit. 2020-02-24]. Dostupné z: https://robotframework.org/#introduction*

[23] Introduction: Examples [online]. In: . [cit. 2020-04-18]. Dostupné z: https://robotframework.org/#examples

[24] *Introduction* [online]. [cit. 2020-02-23]. Dostupné z: https://sahipro.com/docs/introduction/index.html

[25] Sahi Script: Sample Code [online]. In: . [cit. 2020-04-18]. Dostupné z: https://sahipro.com/docs/introduction/sahi-script.html

[26] Architecture [online]. In: . [cit. 2020-04-18]. Dostupné z: https://sahipro.com/docs/assets/images/sahi_architecture.gif

[27] Sahi Pro: Pricing [online]. In: . [cit. 2020-04-18]. Dostupné z: https://sahipro.com/pricing

[28] *Why UI Automation with TestComplete?* [online]. [cit. 2020-02-24]. Dostupné z: https://smartbear.com/product/testcomplete/features/automated-ui-testing

[29] *Keyword Driven Testing in TestComplete* [online]. [cit. 2020-02-24]. Dostupné z: https://smartbear.com/product/testcomplete/features/keyword-driven-testing

[30] *Choose from Multiple Programming Languages to Write Your Functional Tests* [online]. [cit. 2020-02-24]. Dostupné z: https://smartbear.com/product/testcomplete/features/automated-ui-testing

[31] TestComplete: Pricing [online]. In: . [cit. 2020-04-18]. Dostupné z: https://smartbear.com/product/testcomplete/pricing

[32] *Java™ Programming Language* [online]. [cit. 2020-02-20]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html

[33] Overview [online]. [cit. 2020-02-15]. Dostupné z: https://vaadin.com/docs/v8/framework/introduction/intro-overview.html

[34] *Chapter 1. First Steps* [online]. [cit. 2020-02-20]. Dostupné z: https://www.virtualbox.org/manual/ch01.html

[35] *Chapter 1. First Steps: 1.1. Why is Virtualization Useful?* [online]. [cit. 2020-02-20]. Dostupné z: https://www.virtualbox.org/manual/ch01.html#virt-why-useful

Seznam tabulek

Seznam obrázků

Seznam zkratek

IDE	Integrated Development Environment
HTML	Hypertext Markup Language
ATDD	Acceptance Test Driven Development
XML	Extensible Markup Language
API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
UI	User Interface
MVP	Model-View-Presenter
SQE	Software Quality Engineer
SOAP	Simple Object Access Protocol