Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Automatizované testování webových aplikací

Bc. Michal Bujalka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Michal Bujalka

Systémové inženýrství a informatika
Informatika

Název práce

Automatizované testování webových aplikací

Název anglicky

Automated testing of web applications

Cíle práce

Cílem práce je vytvořit a ověřit překladač (wrapper) nad testovacím nástrojem Selenium WebDriver tak, aby bylo umožněno vytvářet automatizované testy běžným uživatelům bez programátorských znalostí.
Dílčí cíle:
• Charakteristika jazyka HOCON a možnosti využití
• Zajištění implementace v překladači pro všechny prohlížeče (možnost volby prohlížeče pomocí konfiguračního souboru HOCON)
• Zajištění výstupu testování do textového dokumentu a příkazové řádky
• Porovnání s manuální testováním a vytvářením testovacích scénářů přímo v jazyku Java

Metodika

Teoretická část je založena na studiu odborné literatury a relevantních zdrojů. V této části se práce bude věnovat testování webových aplikací, jejich technikami a postupy. Dále bude charakterizován nástroj pro automatizované testování Selenium a programovací jazyk Java, který je jeho nedílnou součástí. Bude proveden popis konfiguračního jazyka HOCON, v kterém budou vytvářeny testovací scénáře.

Praktická část je zaměřena na automatizované testování webových aplikací pomocí nástroje Selenium WebDriver a programovacího jazyka HOCON. Nejprve bude vytvořený překladač programovacího jazyka HOCON pro nástroj Selenium. Ověření překladače bude provedeno na vybraných webových aplikacích. Po uskutečnění těchto testů budou výsledky porovnány s manuálním testováním a vytvářením testovacích scénářů přímo v jazyku Java.

Na základě syntézy teoretických poznatků a výsledků praktické časti budou formulovány závěry práce.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

Automatizované testování, Webové aplikace,Selenium, Java, HOCON, XPath

---

Doporučené zdroje informací

AVASARALA, Satya. Selenium WebDriver Practical Guide.1. Packt Publishing, 2014. ISBN 9781782168850;1782168850;.

GARG, Navneesh. Test Automation using Selenium WebDriver with Java: Step by Step Guide. Test Automation Using Selenium with Java, 2014. ISBN 0992293510

HEROUT, Pavel. XSLT 2.0 a SVG prakticky. 1. vyd. České Budějovice: Kopp, 2010. ISBN 9788072324064;8072324063;.

JONES, REX. Absolute Beginner Java 4 Selenium WebDriver: Come Learn How To Program For Automation Testing. 2016. ISBN 1530408369

PATTON, Ron. Testování softwaru. Vyd. 1. Praha: Computer Press, 2002. ISBN 9788072266364;8072266365;.

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 9. 8. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 31. 03. 2022

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Automatizované testování webových aplikací" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. března 2022 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Janu Masnerovi, Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnoval. Poděkování patří též Ing. Kristýně Kaslové za její pomoc a rady při tvorbě této diplomové práce a také firmě Atos IT Solutions and Services s.r.o. za poskytnutí diplomní praxe a odborných konzultací v rámci diplomové práce.

Automatizované testování webových aplikací

Abstrakt

Diplomová práce se zabývá automatizovaným testováním webových aplikací nástrojem pro automatizované testování Selenium WebDriver. Teoretická východiska vymezují oblast vývoje a testování softwaru, zejména automatizovaného testování. Dále je představen nástroj Selenium WebDriver a popis vytváření automatizovaných testů v programovacím jazyku Java. Nakonec je definován jazyk HOCON a jeho možné využití. Vlastní práce se skládá z návrhu, vytvoření a ověření překladače pro nástroj Selenium WebDriver, aby bylo možné vytvářet automatizované testy pomocí jazyku HOCON běžnými uživateli. Po vytvoření překladače je provedeno ověření vytvořeného překladače oproti manuálnímu testování a automatizovanému testování v programovacím jazyku Java. Ověření je prováděno na vybrané webové aplikaci, pro kterou jsou vytvořeny ukázkové testovací případy. Po provedeném ověření je zhodnocení automatizovaného testování pomocí překladače prováděno z hlediska rychlosti, délky kódu a ceny.

Klíčová slova: automatizované testování, webové aplikace, vývoj softwaru, testování softwaru, Selenium WebDriver, Java, HOCON, XPath, HTML, webové prohlížeče

Automated testing of web applications

Abstract

This Master's thesis is about automated testing of web applications with use of automated testing tool Selenium WebDriver. The theoretical background summarizes the area of software development and testing, especially automated testing. As a next step, there is Selenium WebDriver introduction along with the description of creating automated tests in the Java programming language. After that, the thesis focuses on the HOCON language and it's possibilities. The Practical Part consists of design, development and verification of a wrapper for the Selenium WebDriver tool, in order of creating automated tests in HOCON by regular users. After creating the HOCON wrapper, there is authentication running of the created wrapper against manual testing and automated testing in the Java programming language. Verification is performed on a selected web application for which test cases are created. After the verification process, there is a performance comparison test of the translator in terms of speed, code length and price.

Keywords: automated testing, web applications, software development, software testing, Selenium WebDriver, Java, HOCON, XPath, HTML, web browsers

Obsah

1 Úvod

Webové aplikace nebo také webové stránky jsou v dnešní době tak úzce spjaté s lidským životem, že bez nich a internetu si málokdo dokáže představit život. Ať už se jedná o firmy či instituce, které je využívají pro své samotné fungování či jiné účely, tak jsou nepostradatelná i pro jednotlivce v soukromém životě, kteří je využívají na nákupy, platby, komunikaci, vzdělávání, zábavu apod. Většina z nich jsou velice komplexní a složité aplikace, určené k plnění těch nejrůznějších požadavků uživatele, čímž roste riziko možné chyby. Veškeré aplikace mají tedy jedno společné, a to, aby byly kvalitní, bez chyb a splňovaly dané potřeby uživatelů. Splnění těchto požadavků je součástí vývoje aplikace, což je nesmírně složitý proces, začínající prvotní myšlenkou aplikace až po její běžné používání. Na kvalitu přitom má vliv celá řada faktorů, např. kvalifikovanost pracovníků, kteří se na vývoji podílí, dále standardizované náležitosti kvality softwaru. Nejdůležitějším faktorem však zůstává samotné testování.

Testování někdy bývá mylně vnímáno jako činnost prováděná až v samotném závěru vývoje. Tomu by tak nemělo být, neboť chyba může nastat kdykoliv v průběhu celého vývoje softwaru a náklady na opravu dané chyby se potom zvyšují s časem. Je proto zapotřebí, aby testování bylo přítomné od raných fází vývoje, čímž se zamezí odhalení chyb v pozdějších fázích. Testování je možné dvěma způsoby, a to manuálně nebo automatizovaně. Manuální testování provádí manuální testeři, jež na základě testovacích scénářů aplikaci manuálně otestují, přičemž mohou odhalit množství chyb různého charakteru nespecifikovaného v testovacím scénáři. Časová náročnost provedení je tedy individuální, stejně tak kvalita a efektivita testování, které se ve finále promítnou do nákladů. Automatizované testování je oproti tomu prováděno pomocí softwaru, který na základě vytvořeného kódu (skriptu) aplikaci otestuje v kratším čase a s libovolným počtem opakování. Oba způsoby se tedy zaměřují na jiný pohled testování a přináší své výhody. Je tedy vhodné oba uvedené způsoby kombinovat.

V případě automatizovaného testování se jedná o specializované pracovníky – automatizovaný testery, kteří jsou velmi cenění, a tudíž těžko sehnatelní na trhu IT práce. Jejich práce spočívá ve vytváření automatizovaných testů pro dané aplikace pomocí vybraných nástrojů automatizovaného testování, kde nejpoužívanějším nástrojem

automatizovaného testování webových aplikací je nástroj Selenium WebDriver. Automatizované testy v Seleniu jsou vytvářeny pomocí nejznámějších programovacích jazyků, takže je nezbytné, aby tvůrci disponovali značnou znalostí vybraného programovacího jazyka.

Právě na tuto problematiku cílí tato diplomová práce. Snaží se odpovědět na otázku, zda a jakým způsobem je možné zjednodušit vytváření automatizovaných testů, aby byly k dispozici i běžným uživatelům bez programátorských znalostí.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této diplomové práce je vytvořit a ověřit univerzální překladač (wrapper) nad nástrojem Selenium WebDriver, určený pro automatizované testování webových aplikací, tak aby bylo umožněno vytvářet automatizované testy bez programátorských znalostí.

Dílčí cíle diplomové práce jsou:

- Charakteristika jazyka HOCON a možnosti jeho využití.
- Zajištění implementace v překladači pro všechny prohlížeče (možnost volby prohlížeče pomocí konfiguračního souboru HOCON).
- Zajištění výstupu testování do textového dokumentu a příkazové řádky.
- Porovnání s manuální testováním a vytvářením testovacích scénářů přímo v jazyku Java.

2.2 Metodika

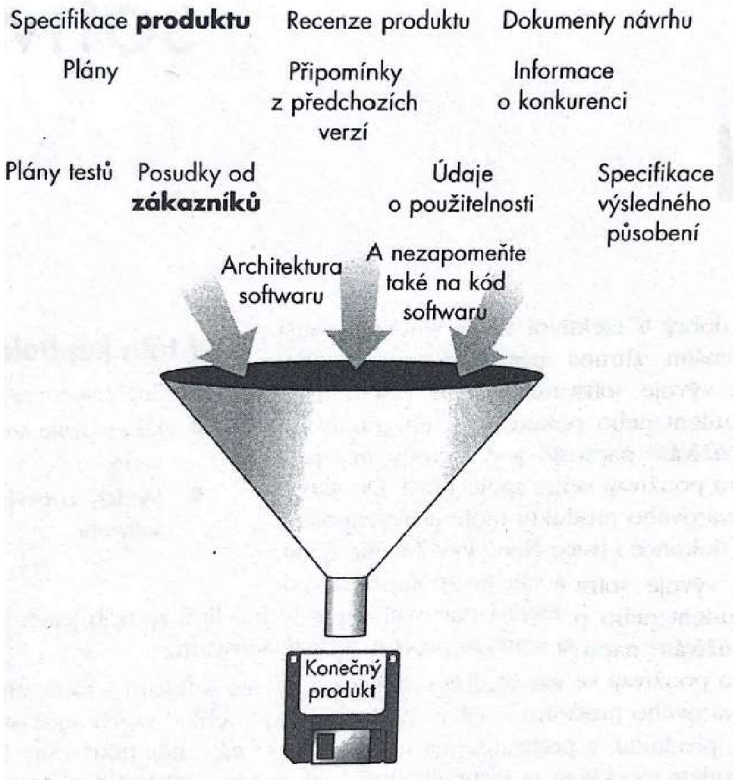
Práce je rozdělena dvě části: teoretickou a praktickou. Teoretická část je založena na studiu odborné literatury a relevantních zdrojů. Tematikou teoretické části je vývoj a testování softwaru. V části věnující se problematice testování bude kromě manuálního testování popsáno zejména automatizované testování. V souvislosti s tím bude věnovaná pozornost vybranému nástroji pro automatizované testování Selenium WebDriver a konfiguračnímu jazyku HOCON. Praktická část bude zaměřena na vývoj samotného překladače a jeho ověření. Nejdříve bude vytvořen návrh překladače a popsáno jeho fungování a následně na základě vybraných nástrojů bude vytvořen kód překladače. Ověření překladače bude prováděno pomocí porovnání testování manuálního a automatizovaného v programovacím jazyku Java ve vybraných webových prohlížečích. Na konci práce budou jednotlivé dosažené časy testování zhodnoceny a porovnány s vytvořeným překladačem. Součástí bude i porovnání délky kódu a ceny automatizovaného testu.

3 Teoretická východiska

3.1 Vývoj softwaru

Testování je nedílnou součástí procesu vývoje každého softwaru, a tudíž každý tester by měl přinejmenším zhruba porozumět celkovému procesu vývoje softwaru. Metody vývoje softwarového produktu se liší podle toho, zda je daný program vytvářen jedním programátorem nebo celou řadou programátorů velké společnosti. U velkých společností mnohdy bývají do vytváření nového softwarového produktu zapojeny jednotky až desítky členů projektového (vývojového) týmu, z nichž každý má svou specifickou roli a všichni vzájemně spolupracují podle předem stanoveného plánu. Přesně specifikovaná pracovní náplň, způsob vzájemné komunikace a rozhodování jednotlivých lidí je součástí procesu vývoje softwaru. [1]

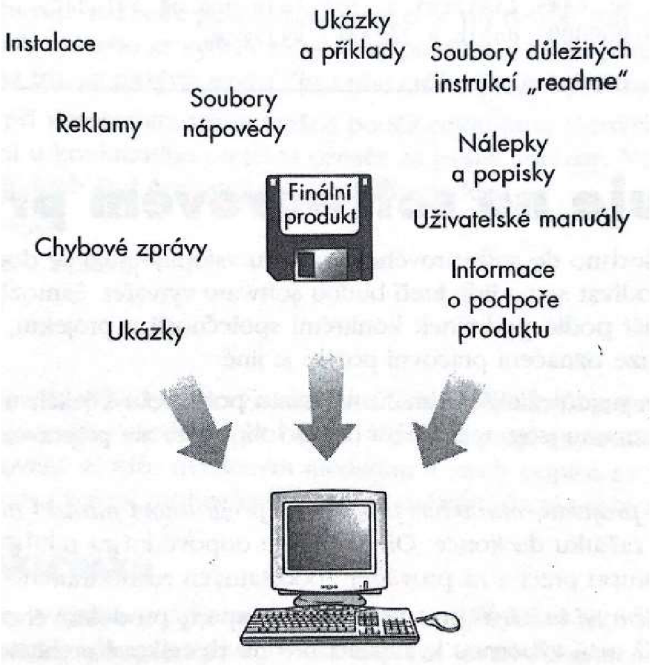
Na následujícím obrázku č. 1 je znázorněna veškerá práce, včetně abstraktních součástí, která může vstupovat do softwarového produktu.



Obrázek 1 – Veškerá práce vstupující do softwarového produktu [1]

Součástí samotného softwaru (produktu) není pouze finální program, který si uživatel spustí u sebe na počítači, ale řada skrytých komponent, které jsou mnohdy ignorovány. Pro softwarového testera je nezbytné vnímat podstatu těchto komponent, protože se dají testovat a ve všech se mohou objevovat chyby. [1]

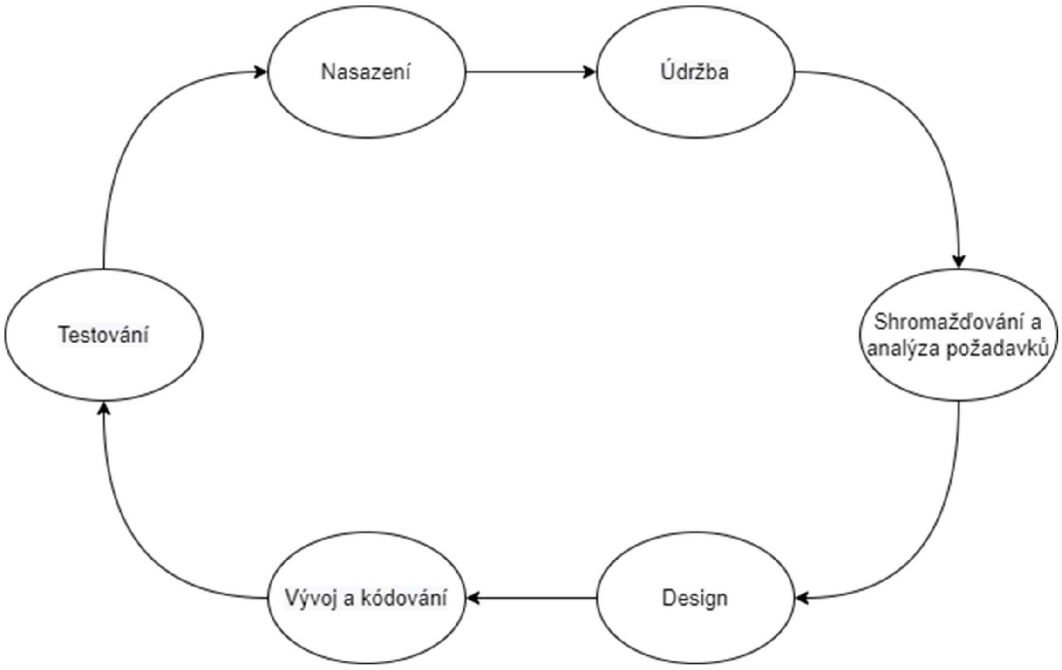
Na následujícím obrázku č. 2 je znázorněno, z jakých částí se finální softwarový produkt skládá.



Obrázek 2 – Součásti finálního softwarového produktu [1]

3.1.1 Životní cyklus vývoje softwaru

Proces, podle něhož je vytvářen softwarový produkt, a to od jeho úplného začátku (prvotního záměru) až do zveřejnění a uvedení na trh, se nazývá životní cyklus vývoje softwaru, běžně označován jako SDLC (Software Development Life Cycle). SDLC definuje jednotlivé kroky zahrnuté ve vývoji softwaru v každé jeho fázi. Jedná se tedy o kompletní cyklus vývoje, což jsou veškeré úkoly spojené s plánováním, vytvářením, testováním a nasazováním softwarového produktu, což je schematicky znázorněno na obrázku č. 3. Účelem SDLC je dodávat vysoce kvalitní produkt, který bude odpovídat požadavkům zákazníka. [2] [3]



Obrázek 3 – Životní cyklus vývoje softwaru (SDLC)

Jednotlivé etapy SDLC jsou:

1. Shromažďování a analýza požadavků
2. Design
3. Vývoj a kódování
4. Testování
5. Nasazení
6. Údržba. [2] [3]

1. Shromažďování a analýza požadavků

První fáze SDLC je shromažďování a analýza požadavků. Jedná se o nejdůležitější a základní etapu SDLC, která bývá prováděna vedoucími členy týmu. Během této etapy jsou od zákazníka shromažďovány veškeré relevantní informace pro vývoj produktu podle jejich očekávání. Jakékoliv nejasnosti je nutné řešit v této fázi. Když je shromažďování požadavků dokončeno, provede se prvotní analýza, kterou je ověřena proveditelnost požadavků, včetně plánování. Plánování určuje náklady a zdroje potřebné k proveditelnosti projektu. Veškeré nejasnosti jsou diskutovány se zákazníkem okamžitě. Jakmile jsou požadavky na software zcela pochopeny, je vytvořen dokument SRS (Software Requirement Specification). Tento dokument by měl být zcela pochopitelný pro vývojáře a také zkontrolován zákazníkem pro další použití. [2] [3]

2. **Design**

Pro tuto fázi SDLC jsou požadavky shromážděné v dokumentu SRS použity jako reference pro softwarové architekty, aby přišli s nejlepší architekturou pro daný vyvíjený software. Na konci této etapy je vývojáři dokument SRS přeměněn na dokument SDS (Software Design Specification). Tento dokument je zkontrolován veškerými zúčastněnými stranami a diskutována zpětná vazba a návrhy na vylepšení. [2] [3]

3. **Vývoj a kódování**

V této fázi začíná „skutečný" vývoj softwaru. Vývoj (implementace) začíná, jakmile vývojáři obdrží dokument SDS. Na základě toho je vyvinutý kód softwaru. Veškeré požadavky na software jsou implementovány v této fázi. [2] [3]

4. **Testování**

S testováním se začne, jakmile je kódování dokončeno a jednotlivé moduly jsou uvolněny k testování. V této etapě SDLC je softwarový produkt důkladně testován a veškeré zjištěné závady či chyby jsou vráceny vývojářům, aby je opravili. Opakované testování neboli regresní testování se provádí až do okamžiku, kdy software naplňuje zákazníkova očekávání. Testeři se odkazují na dokument SRS, aby se ujistili, že software odpovídá standardům zákazníka. [2] [3]

5. **Nasazení**

Jakmile je produkt otestován (v závislosti na jeho složitosti), je nasazen v produkčním prostředí nebo je provedeno první UAT (User Acceptance testing) v závislosti na očekávání zákazníka. UAT se provádí v tzv. testovacím prostředí, které je tvořeno replikou prostředí produkčního. Testování je prováděno společně se zákazníkem. Pokud je zákazník s vyvíjeným softwarem spokojen, je nasazen do produkčního prostředí. [2] [3]

6. **Údržba**

Po nasazení do produkčního prostředí se vývojáři starají o údržbu softwaru. Údržba zahrnuje jakékoliv problémy spojené s fungováním softwaru v produkčním prostředí a jejich následnou opravu nebo případné vylepšení a další rozvoj softwaru. [2] [3]

3.1.1.1 Modely SDLC

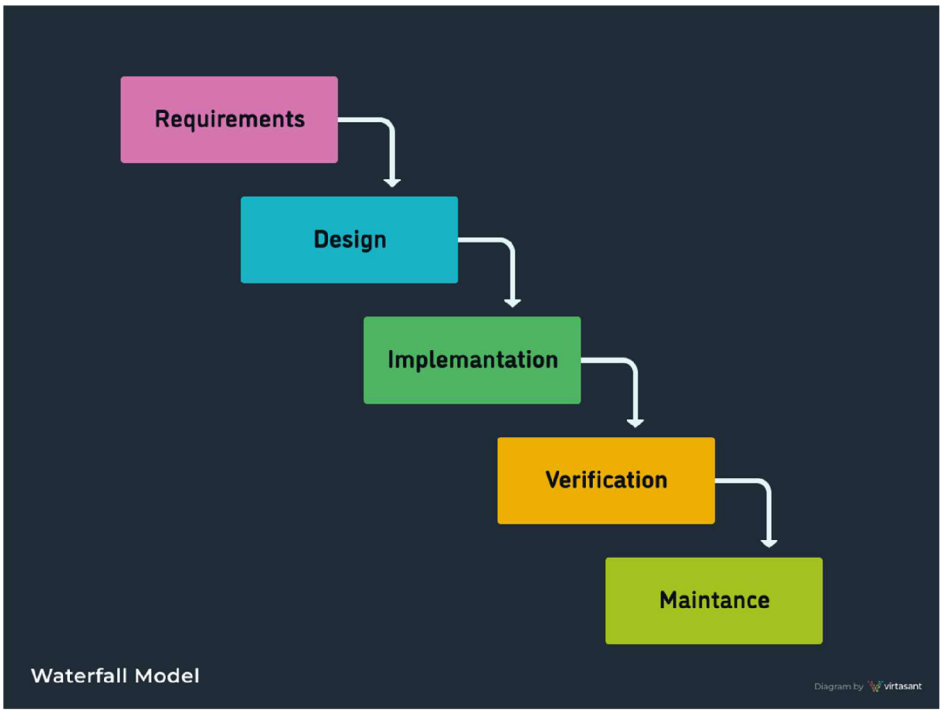
Jsou definovány a navrženy různé modely SDLC, které jsou dodržovány během procesu vývoje softwaru. Tyto modely jsou také označovány jako „metody procesů vývoje softwaru“. Každý model sleduje řadu kroků jedinečných pro svůj typ, aby byl zajištěn úspěch v procesu vývoje softwaru. Modely SDLC mohou mít odlišné přístupy, ale základní fáze a aktivita zůstává u všech modelů stejná. Mezi nejznámější modely SDLC patří:

- Vodopádový model
- Prototypování
- Iterativní model
- Spirálový model
- V-model
- Agilní vývoj. [4]

Dále je uveden základní popis zmíněných modelů a pro základní představu i jejich grafické znázornění.

Vodopádový model

Prvním modelem SDLC, jenž se ve vývoji softwaru používal v 70. až 90. letech, byl vodopádový model (Waterfall Model). Model je znázorněn na obrázku č. 4. Je postaven na postupném, lineárním toku kroků. Každá fáze musí být dokončena před přechodem na další, jelikož každá fáze závisí na výsledcích fáze předchozí. Jakmile je fáze dokončena, nelze se vrátit zpět, jelikož metoda umožňuje pouze směr vpřed prostřednictvím jednotlivých fází. Hlavní výhodou tohoto modelu je jeho jednoduchost, která umožňuje snadněji sledovat a implementovat daný model vývoje softwaru. Veškeré činnosti, které je nutné provést, jsou dobře definovány a analyzovány. [4]

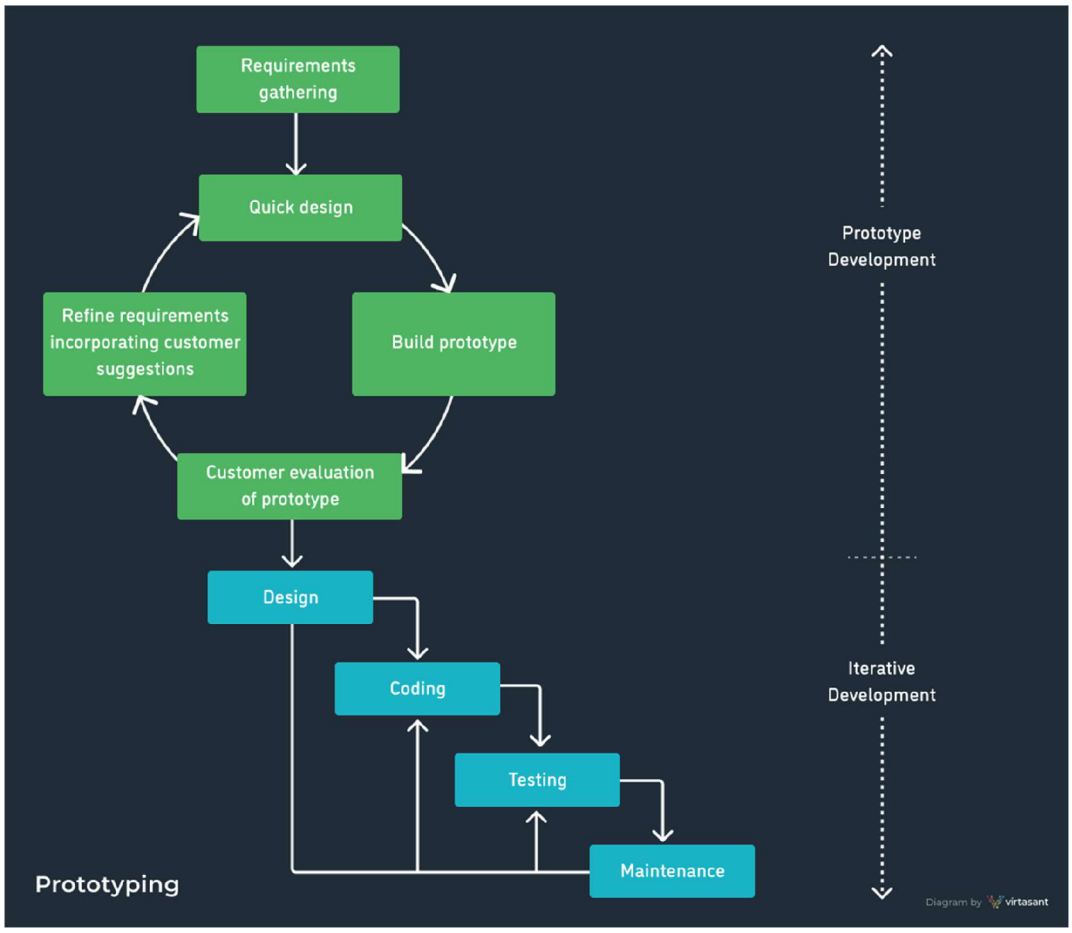


Obrázek 4 – Waterfall Model [4]

Tato metoda je nejstarší a dá se říci, že ne vždy fungovala efektivně. Výsledkem toho bylo, že průkopníci softwaru vyvinuli nové metodiky, jejichž cílem bylo vylepšit nebo nahradit vodopádový model. [4]

Prototypování

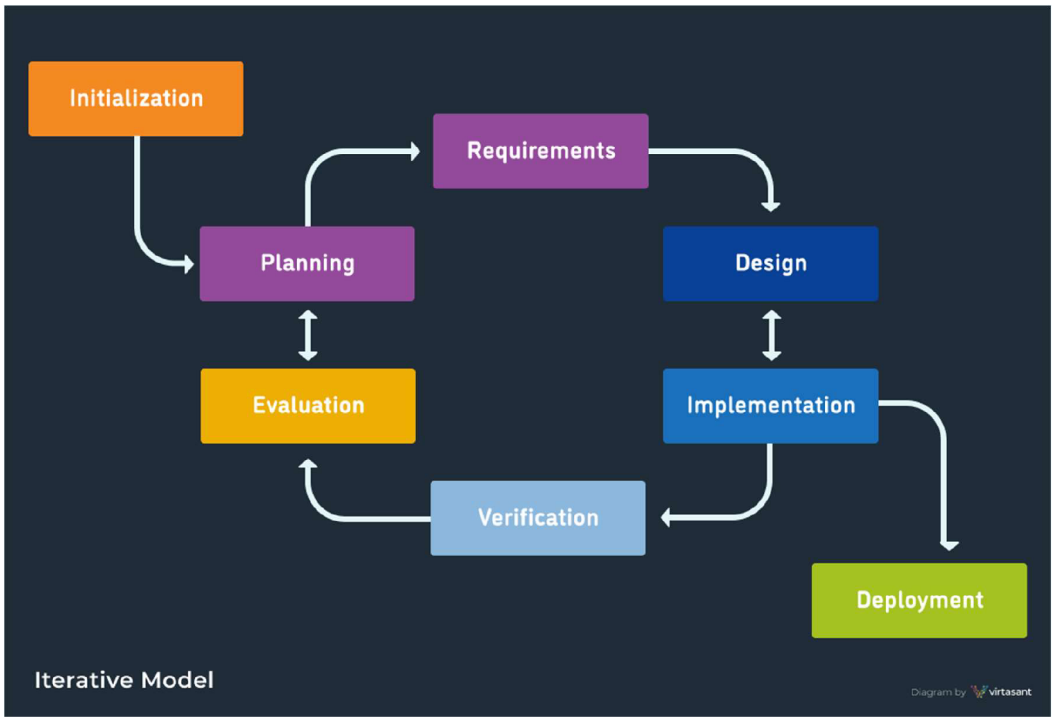
Jedná se o první metodiku, která byla vyvinuta (v polovině 70. let) jako alternativa k vodopádovému modelu. Tato metoda spočívá ve vytváření jednotlivých prototypů za účelem shromažďování zpětné vazby od potenciálních uživatelů, či zákazníka, je znázorněna na obrázku č. 5. Odtud jsou prototypy vyvinuty do konečných požadavků softwaru. [4]



Obrázek 5 – Prototypování [4]

Iterativní model

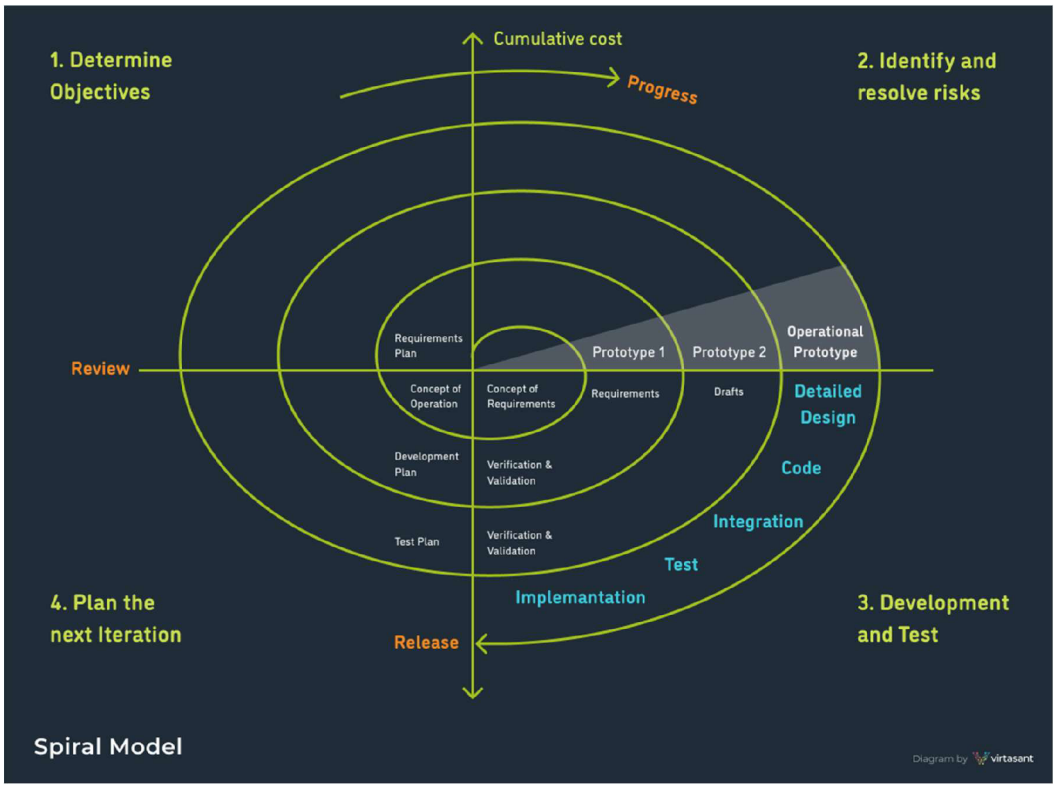
Tato metoda je jeden z raných předchůdců agilního vývoje. Byl zde kladen důraz na iterativní a přírůstkové akce během vývoje softwaru. Na začátku této metody jsou známy pouze hlavní požadavky. Na jejich základě je vytvořena vývojovým týmem softwaru první rychlá a levná verze softwaru, jinými slovy prototyp. Jednotlivé iterace procházejí vždy všemi fázemi SDLC a tyto cykly se opakují během vývoje až do dokončení, jak je uvedeno na obrázku č. 6. Bylo (a někdy dosud stále je) zvykem, že se na jednotlivých fázích SDLC pracovalo současně. [4]



Obrázek 6 – Iterativní model [4]

Spirálový model

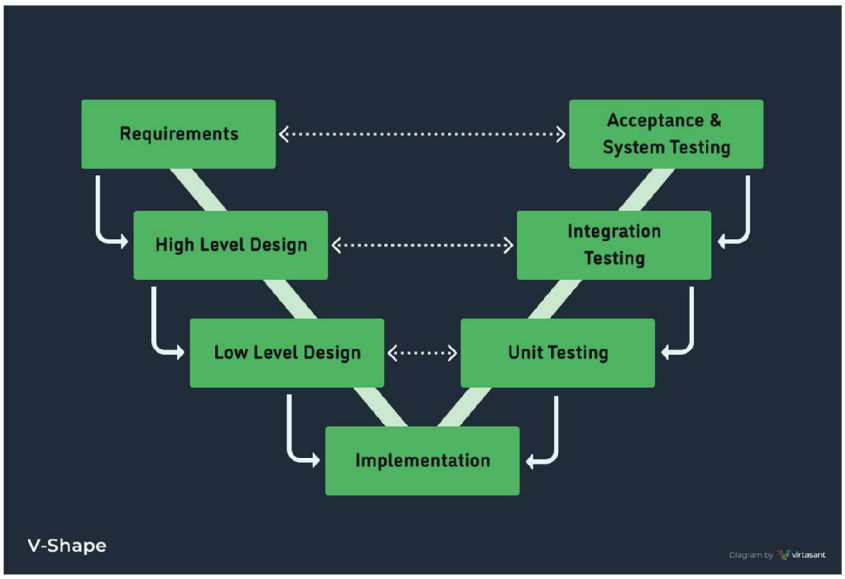
Základní myšlenku spirálového modelu lze popsat tak, že na začátku nelze definovat všechno úplně podrobně. Začne se tedy s malými definicemi těch nejdůležitějších funkcí, ty jsou následně otestovány a připomínkovány uživatelem, či zákazníkem, a až poté se přejde na další úroveň vývoje. Celý proces se opakuje, dokud se nedojde k požadovanému výslednému produktu. Model je znázorněn na obrázku č. 7. [4]



Obrázek 7 – Spirálový model [4]

V-model

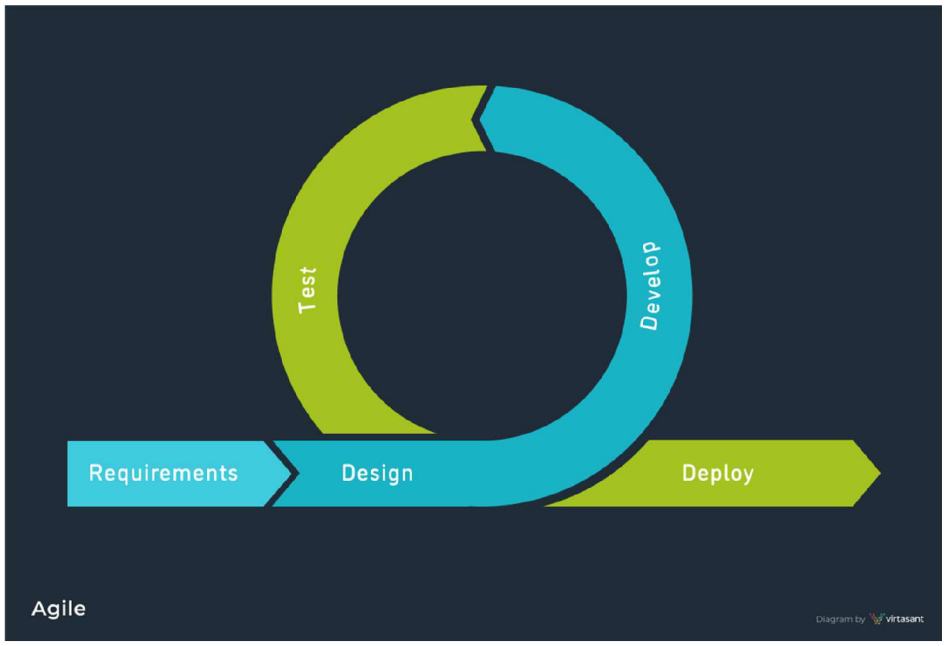
V-model je pojmenován podle dvou klíčových konceptů, a to: Validace a Verifikace a je znázorněn na obrázku č. 8. Model demonstruje vztahy mezi každou etapou životního cyklu vývoje softwaru a jeho přidruženou etapu testování (ověřování). [4]



Obrázek 8 – V-model [4]

Agilní model

Agilní model či agilní vývoj, někdy též označován jako CI/CD (Continuous Integration / Continuous Delivery), je znázorněn na obrázku č. 9. Model je zaměřen zejména na flexibilitu při vývoji produktu. Software je při tomto vývoji rozdělen do ucelených (a smysluplných) vývojových bloků a je dodán v pořadí dohodnutém se zákazníkem. Software je dodáván průběžně, takže zákazník může aplikaci používat už od jejího prvního nasazení a nemusí čekat na dodání kompletní aplikace. Každý požadavek, ať již od zákazníka nebo přímo od vývojářů, je začleněn do tzv. backlogu (seznam úkolů), naceněn ze strany vývojářů a je udělena priorita realizace ze strany zákazníka. Ke každému požadavku se většinou vyjadřuje celý vývojový tým, čímž je zajištěn objektivní pohled na problematiku i objektivní odhad pracnosti. Testování agilního vývoje bývá prováděno testery nebo automatizovaně, aby bylo zaručeno, že realizací nového požadavku nedošlo k poškození stávající funkcionality. Změnové požadavky tudíž nejsou nasazovány ve velkých blocích, ale ihned po jejich dokončení. [4]



Obrázek 9 – Agilní model [4]

3.1.2 Projektový tým softwaru

Na většině softwarových projektů, které jsou určitým způsobem vytvářeny, se podílí projektový tým. Skladba pracovníků týmu se podstatně liší podle podmínek společnosti a daného projektu, ale role bývají z velké části stejné (podobné), pouze označení pracovní pozice se liší. Projektový tým vyvíjeného softwaru zpravidla tvoří:

- Projektoví manažeři
- Architekti (systémoví inženýři)
- Analytici
- Programátoři (vývojáři)
- Testeři
- Techničtí specialisté. [1]

3.1.3 Kvalita softwaru

S neustále hlubší integrací a rostoucí složitostí i nákladností jak kritických, tak podpůrných systémů logicky rostou i požadavky na jejich kvalitu. Kvalitní software je v dnešní době samozřejmostí. Přitom mnozí si ale pod pojmem „kvalita“, „kvalitní“ nebo „nekvalitní“ představují něco poněkud odlišného. První snahy globálně definovat kvalitu softwaru a její atributy daly vzniknout různým modelům kvality, především řadě norem ISO/IEC 9126 (Softwarové inženýrství – jakost produktu), ISO/IEC 14598 (Softwarové inženýrství – hodnocení softwarového produktu) a ISO/IEC 12119 (Softwarové balíky – Požadavky na jakost a zkoušení). Tyto normy však trpí vážnými nedostatky, a tak jsou postupně nahrazovány jednotným systémem norem ISO/IEC 25000-25099 v rámci projektu SQuaRe (Software Quality Requirements and Evaluation – Požadavky na kvalitu softwaru a její hodnocení). [5]

Jedna z norem projektu SQuaRe, ISO/IEC 25010, definuje kvalitu softwaru, jako *míru, do jaké softwarový produkt splňuje stanovené a implicitní potřeby, je-li používán za stanovených podmínek*. Norma definuje model kvality produktu a model kvality užití. Kvalita softwaru sestává z osmi charakteristik: Funkčnost, Účinnost, Kompatibilita, Použitelnost, Bezporuchovost, Bezpečnost, Udržovatelnost a Přenositelnost. Každá z těchto charakteristik má navíc další podcharakteristiky, které jsou předmětem měření, přičemž jedna z měr je i testování. [5] [6]

3.1.3.1 Zajišťování a řízení kvality

Zajišťování kvality softwaru je někdy mezi veřejností mylně chápáno jako testování softwaru. Hlavní ideou SQA (Software Quality Assurance, v překladu „zajišťování kvality softwaru“) je, že kvalitní proces vývoje softwaru zajišťuje kvalitní produkt. Tento proces je komplexní aktivitou, která zajišťuje, aby veškeré standardy, procesy a procedury použité během vývoje a údržby softwaru byly vhodné a správně dodržované. Oproti tomu řízení kvality softwaru (Software Quality Control) je zaměřeno na výstupy z jednotlivých procesů (dílčí produkty), u nichž kontroluje, zda odpovídají daným specifikacím a požadavkům (včetně implicitních). Kromě samotného testování se využívají techniky statické analýzy, jako inspekce, revize či strukturované procházení. Řízení kvality je tedy zaměřeno na nalézání defektů (chyb), jejich odstranění a následné prověřování správnosti provedených úprav (změn). [1] [5] [7]

V souvislostí s těmito procesy se často vyskytují pojmy jako Verifikace a Validace (někdy také zkráceně označovány jako V&V aktivity). Jedná se o velmi blízké, avšak nezaměnitelné koncepty, které jsou na sobě závislé. Verifikační a validační aktivity ověřují, zda software odpovídá specifikacím a jako celek poskytuje přesně to, co uživatel potřebuje – tedy zda splňuje svůj účel. Význam těchto dvou pojmů lze chápat jako:

- Verifikace: Vytvářím produkt správně?
- Validaci: Vytvářím správný produkt? [1] [5] [7]

3.2 Testování softwaru

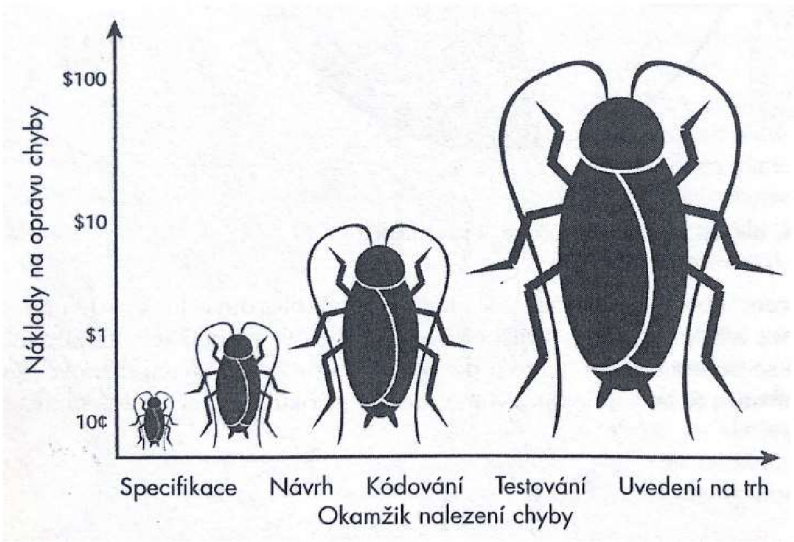
Testování softwaru je součástí zajišťování a řízení kvality softwaru, kde působí jako jeden z procesů (metod) řízení kvality softwaru. Přesněji se jedná o metodu nebo proces, jak zkontrolovat, zda vyvíjený softwarový produkt odpovídá požadavkům zákazníka a zajistit, aby výsledný produkt byl bez kritických vad, a to za účelem výroby kvalitního produktu. Testování je řízený proces, jelikož musí být prováděno s nějakým záměrem. Předchází mu plánování ve formě návrhu testů a po něm následuje jejich vyhodnocení. Testování softwaru je nezbytným a velice důležitým procesem vývoje softwarového produktu, bez kterého není možné v dnešní době vyvíjet softwarový produkt. [1] [8]

Neexistuje žádná přesná definice testování softwaru a lze říci, že co publikace a autor, to jiný názor na to, co je a není testování softwaru. Jednou z definic testování

softwaru v užším smyslu je: ***proces řízeného spouštění softwarového produktu s cílem zajistit, zda splňuje specifikované či implicitní potřeby uživatelů***. Z této definice vyplývá, že testování zkoumá softwarový produkt, čímž získává informace o jeho kvalitě, která je chápána jako stupeň naplnění požadavků a přání uživatelů. Obsahem testování je tedy především sběr a analýza testování, přičemž nalezené ***defekty*** jsou z tohoto pohledu vedlejším produktem celého procesu. [5]

3.2.1 Důvody testování

Hlavním důvodem testování je odhalování chyb před okamžikem, kdy mohou být škodlivé. Chyby v procesu vývoje softwarového produktu lze nalézt po celou dobu, od prvopočátku, tj. od plánování přes programování a následné testování, až po okamžik uveřejnění produktu. Existuje tzv. Pattonův graf nákladů na opravu chyby, znázorněný na obrázku č. 10, který odpovídá Boehmově zákonu – čím dříve se na chybu přijde, tím je její oprava levnější. U tohoto grafu je vhodné věnovat pozornost logaritmickému měřítku na ose Y a sledovat, jak postupem času rostou náklady na opravu chyb. [1]



Obrázek 10 – Pattonův graf nákladů na opravu chyby [1]

Výše nákladů zde postupem času roste vždy na desetinásobek. Chyba, která by byla odhalena a opravena ještě v raných stadiích sestavování specifikace, nemusí stát téměř nic, nebo minimum například 10 centů. Jestliže stejná chyba by byla objevena během kódování nebo testování softwaru, bude stát 1 až 10 dolarů. A jestliže by ta samá chyba byla nalezena až zákazníkem ve finálním produktu, náklady můžou vyskočit na 100 dolarů a více. [1]

3.2.2 Nejdůležitější termíny v testování

Terminologie v testování se míchá a různí. Běžně se používají termíny jako: chyba (error bug), vada (flaw), omyl (mistake), selhání (failure), porucha (fault), defekt (defect), závada (glitch), odchylka (variance), problém (problem), nepříjemná událost (accident) anomálie (anomaly), nekonzistence (inconsistency), incident (incident). Tyto uvedené termíny používané pro chyby softwaru mohou popisovat občas odlišné skutečnosti, které jsou na sobě navzájem časově závislé. [7]

Je nutné dříve či později tyto termíny v testování rozlišovat neboť terminologie (podpořená ISTQB klasifikací) je již vcelku sjednocená, a to následovně:

1. Lidé se mohou dopustit omylu nebo je chyba ve specifikaci (návrhu).
2. Vlivem lidské chyby se do systému naimplementuje defekt (defect, bug).
3. Porucha (fault) nastane, pokud se tato část kódu začne provádět. Je to projev jedné či více chyb, přičemž ale na místo, kde je defekt, vůbec nemusí program dojít, např. v mrtvém kódu, málo pravděpodobné větvi atp.
4. Selhání (failure) je projevem poruchy, která způsobila uvedení programu do nekorektního stavu nebo jeho běh nezamýšlenou cestou. [7]

3.2.3 Základní axiomy testování softwaru

Axiom je tvrzení, jehož pravdivost je obecně přijímána, a není tedy nutné ji dokazovat. U testování softwaru je možné formulovat řadu axiomů, ale mezi ty nejvýznamnější a nejznámější se řadí: [1] [5]

1. **Žádný netriviální program není možné otestovat úplně**

Pokud bychom chtěli zcela otestovat softwarový produkt, znamenalo by to, že by byly odhaleny všechny defekty. Testování by v takovém případě zahrnovalo veškeré správné i nesprávné vstupní hodnoty, stavy programu a přechody mezi nimi či možné vlivy prostředí, které je třeba nasimulovat. V praxi, s výjimkou triviálních systémů, je takovéto kompletní otestování téměř nemožné. [1] [5]

2. **Testování je věci odhadu rizika**

Jelikož není úplné otestování softwaru možné, tester či analytik testování vybírá k testování scénáře podle vlastního uvážení. Tím dochází k riziku, že nezachytí možný defekt ve scénářích, které nevybral. [1] [5]

3. **Testování může prokázat jen přítomnost defektů, nikoli jejich absenci**

Testováním je možné prokázat přítomnost defektů v softwaru, avšak není možné prokázat jejich bezchybnost, především kvůli nemožnosti kompletního otestování softwaru. Tudíž nehledě na to, kolik času zabere testování softwaru, není možné jej označit za bezchybný, vyjímaje triviálních aplikací. [1] [5]

4. **Čím více defektů je nalezeno, tím více jich v produktu je**

Toto tvrzení vychází z toho, že defekty se vyskytují ve skupinách, jelikož lidé často dělají stejné chyby, nebo mohou mít své špatné dny, či některé chyby jsou skutečně pouhou špičkou ledovce. Přitom platí také opačné tvrzení k tomuto axiomu „z chyb vznikají další chyby“. [1] [5]

5. **Programátor nemá testovat svůj kód**

Bývá zvykem, že programátor by neměl testovat svůj vlastní kód, s výjimkou testování jednotek, a to hned z několika důvodů. Jeden z hlavních důvodů je, že programátor neporozumí specifikacím zcela správně, a tak požadovanou funkcionalitu naimplementuje špatně. Při následném testování defekt není odhalen, neboť vychází z předpokladu získaných nesprávným pochopení samotného zadání. Dalším důvodem je, že programátoři se při testování zaměřují spíše na prokázání, že jejich kód funguje správně. Potom v případě, kdy zapomene ošetřit určité případy či situace, tak je nejspíš zapomene i otestovat. [1] [5]

3.2.4 Testovací tým

V dnešní době je velmi vzácné setkat se s tím, že by testování a související aktivity byly svěřeny jednomu pracovníkovi. Bývá zvykem, že podle potřeb projektu a daných omezení je sestaven různě velký tým, jehož členové mají specifické role. Obecně jsou ve struktuře testovacích týmů středních a větších projektů zastoupené role:

- Inženýr QA (QA Engineer)
- Manažer testování (Test manager)
- Vedoucí testování (Test lead)
- Tester – automatizované testování (Test automator)
- Analytik testování (Test analyst či Test engineer)
- Tester (Tester). [5] [9]

Inženýr QA

Tato role je často nepřesně používána pro pracovníky, jejichž skutečnou pracovní náplní je testování. Jak již ale bylo vysvětleno v kapitole 3.1.3.1, testování není QA (Quality Assurance). Tato role se z pohledu kvality zabývá sledováním, hodnocením a snahou zlepšovat procesy, nikoli jejich výstupní produkty. Sleduje každou fázi procesu vývoje softwaru a zajišťuje, aby vyvinutý software splňoval standardy kvality. [5] [9]

Manažer testování

Nejvyšší role na pomyslné pyramidě testovacího týmu. Manažer testování je odpovědný za veškeré aktivity související s testováním (či řízením kvality vůbec) a odpovídá vedení projektu či vyššímu managementu. [5] [9]

Vedoucí testování

U velkých projektů bývá zvykem, že jednotlivé části vyvíjeného softwaru (například backend a frontend) jsou testovány samostatným týmem, za který odpovídá vedoucí testování. V případě menších projektů odpovídá za testovací tým celkově. Prvním úkolem vedoucího testování bývá vytvoření plánu testování (test plan) v souladu se strategií testování, kontrola jeho dodržování a případné korektivní akce. Dále má na starosti systém pro hlášení defektů, zadávání úkolů jednotlivým členům týmu a dohlížení na jejich plnění. Z dílčích zpráv pak pravidelně generuje hlášení o celkovém stavu testování dané části softwaru a tak dále. Kromě toho se může účastnit samotného navrhování i provádění testů. [5] [9]

Tester – automatizované testování

V případě projektů využívajících automatizované testování obvykle týmy disponují specializovanými pracovníky zaměřujícími se výhradně na tuto činnost, především pokud

se jedná o robustnější komerční nástroje určené pro testování komplexních aplikací. Specializovaní pracovníci většinou používají nástroje automatizovaného testování (frameworky) jako Selenium, Cucumber nebo další k efektivnímu navrhování a psaní nových testovacích případů. [5] [9]

Analytik testování

Analytik má za úkol vytváření testovacích scénářů a jednotlivých testovacích případů. Analytik testování zajišťují, aby funkční připravenost aplikace byla přijatelná před jejím nasazení do testovacího nebo produkčního prostředí. [5] [9]

Tester

Tester je zodpovědný zejména za provádění testů podle připravených scénářů, za záznam získaných výsledků (tedy rozdíl mezi skutečným a očekávaným stavem), případně vytváří hlášení o nalezených defektech. Po opravě defektu je úkolem testera přetestování a správa stavu nahlášeného defektu včetně výsledků původního scénáře. [5] [9]

3.2.5 Dokumentace testování

Nezbytnou součástí správného a efektivního testování je existence dokumentace, na základě které je testování prováděno. V této kapitole budou popsané ty nejdůležitější a nejznámější dokumenty (dokumentace), v které v proces testování vznikají.

3.2.5.1 Testovací plán

Test plan neboli testovací plán či plán testu, je hlavní dokument pro celý proces testování. Zpravidla bývá vytvářen jako první dokument testování a zahrnuje např. testovací strategii, cíle testování, harmonogram, výstupy a zdroje potřebné k provedení testování. Definice testovacího plánu podle ISTQB je: "Testovací plán je dokument popisující rozsah, přístup, zdroje a plán zamýšlených testovacích aktivit." [10]

3.2.5.2 Testovací případ

Jeden z nejvíce používaných pojmů při testovaní je testovací případ (někdy také jako testovací scénář), běžně označován jako TC (test case). Definice testovacího případu podle IEEE 1012:2004 je definována jako: „sada vstupů, podmínek pro spuštění a očekávaných výsledků, vyvinutá pro konkrétní účel, jako je provedení specifické cesty v programu nebo

ověření souladu s konkrétním požadavkem.“ Jinými slovy se jedná o seznam kroků při testování jednoho konkrétního případu, který u testované položky může nastat. Součástí je také výčet množiny vstupů (dat), podmínek a očekávaných reakcí (výsledků). Nezbytnou součástí jsou také vstupní podmínky neboli výchozí stav testované aplikace. [5] [7]

Specifikace obecného testovacího případu je detailně popsána v normě IEEE829:2008 následovně:

- Unikátní identifikátor testovacího případu.
- Účel nebo zaměření testu.
- Specifikace všech vstupních dat potřebných k provedení testu.
- Specifikace výstupních hodnot, vlastností a očekávaného chování.
- Potřeby na prostředí, jako je hardware, software či jiné.
- Požadavky na provedení testu, jako jsou vstupní a výstupní podmínky nebo zvláštní procedury, které musí být vykonány.
- Případné závislosti na ostatních testovacích případech, které musí být provedené před spuštěním. [5]

3.2.5.3 Testovací sada

Testovací sada (test suite), někdy také překládáno do češtiny jako testovací scénář, je skupina logicky souvisejících testovacích případů. Oproti testovacímu případu slouží pro komplexnější testy větších celků. Testovací sady se dají libovolně seskupovat, díky čemuž je možné provádět náročnější i méně náročnější skupiny testů. [7]

3.2.6 Testovací metody

V testování softwaru se udává, že existuje až 200 různých typů testů, což často způsobuje nepochopení a nedorozumění, co a kdy se vlastně testuje a co má být správným výsledkem daného testování. Testy lze dělit do mnoha kategorií dle různých specifikací. Nejčastější třídění testů je definováno následujícími kategoriemi (testovací metody):

- Statické vs. dynamické testovací metody
- Znalosti kódu
- Úrovně testování (fáze testů)
- Dimenze kvality. [5] [7]

3.2.6.1 Statické vs. dynamické testovací metody

Statické testy jsou koncipovány tak, že není vyžadováno spuštění programu. Jejich hlavní výhodou je možnost začít s testováním již v raných fázích vývoje. Patří sem zejména procházení kódu a statická analýza. Dynamické testy jsou koncipovány tak, že probíhají vždy nad spuštěným programem, případně jeho částí. [1] [7]

3.2.6.2 Znalosti kódu

Na základě úrovně znalosti kódu testovaného softwaru lze dělit testy do tří skupin, a to: černou, bílou a šedou skřínku.

Černá skřínka

Reprezentuje testy, kdy není známé vnitřní prostředí softwaru a na software se pohlíží z venku. Testování je z pohledu uživatele, který nemá žádné znalosti o tom, jak systém uvnitř funguje či jak je vytvořen, ani by se o to neměl zajímat. Jedná se tedy o způsob simulující reálný způsob používání aplikace. Typickým představitelem jsou např. funkční a akceptační testy a toto testování je typické pro pozdější fáze testů. [5] [7]

Bílá skřínka

Jedná se o testování z pohledu programátora, kdy jsou testy psané na základě znalosti zdrojového kódu a vnitřní logiky a zaměřují se na potencionální slabá místa, oblasti s vyšší složitostí. Jde o velmi důkladné testování, které se používá zejména u počátečních fází testů a zcela typické je pro jednotkové testy. [5] [7]

Šedá skřínka

Při návrhů testů šedé skřínky jsou k dispozici nejen specifikace požadavků na systém, ale také dokumenty používané vývojáři a v omezené míře zdrojový kód. Testy jsou tedy tvořeny z pohledu uživatele techniky bílé skřínky, ale zároveň se používají také techniky černé skřínky. To umožňuje navrhovat testy, které testují funkčnost sytému a zároveň se zaměřují na potencionální slabá místa, oblasti s vyšší složitostí nebo situace, které by byly jinak s největší pravděpodobností opominuty. [5] [7]

3.2.6.3 Úrovně testování (fáze testů)

Tato metoda, někdy také známa jako fáze testů, definuje testování na různých úrovních vývoje softwaru. Na základě V-modelu (viz kapitola 3.1.1.1) jsou rozlišovány následující fáze/úrovně testování korespondující s jednotlivými úrovněmi vývoje softwaru:

1. Testování jednotek (Unit testing)
2. Integrační testování (Integration testing)
3. Systémové testování (Systém testing)
4. Akceptační testování (Acceptance testing). [5] [7]

1. Testování jednotek (Unit testing)

Jedná se o první fázi testování, je označováno také jako jednotkové testy, prováděno přímo programátorem, který software vyvíjí. Testovány bývají nejmenší části programu, které lze chápat jako nejmenší testovatelné součásti programu. Při testování je cílem otestovaní veškerých jednotek nezávisle na ostatních, samostatně, a prokázat správné chování. Testování probíhá tak, že kód testu funguje tak, že poskytované vstupy s co možná největší variabilitou jsou ověřovány oproti očekávaným hodnotám, přičemž při jiném výsledku test ohlásí chybu. Výhodou testování jednotek je opakovatelnost a automatizace testu, navíc nenutí programátory se nad kódem zamýšlet více než při určování případů k testování. Nevýhodou je zachycení pouze konkrétních chyb souvisejících s přímým chováním jednotek, nikoli chyby při jejich integraci do systému jako celku. [5] [7]

2. Integrační testování (Integration testing)

Pokud veškeré jednotky (respektive moduly) fungují správně, dalším krokem je jejich integrace do stabilního celku ve fázi integračního testování. Toto testování bývá již prováděno samotnými testery, jejichž cílem je otestování různých části softwaru v kombinaci s ostatními. Integrace je prováděna tak, že do systému jsou postupně přidávány jednotlivé jednotkově otestované moduly, které fungují správně a zároveň nenarušují stabilitu dosud fungujícího celku. Hlavním cílem této fáze je nalézání chyb vzniklých spojením a následnou interakcí jednotlivých modulů, což musí náležitě odrážet testovací případy. [5] [7]

3. **Systémové testování (System testing)**

Díky úspěšnému integračnímu testování je systém dostatečně stabilní a je možné začít s fází systémového testování. Fáze systémového testování je nejdůležitější fáze, jelikož se důkladně ověřuje, zda systém splňuje požadavky zákazníka. Jinak řečeno systémový test by měl končit přesvědčením, že zákazník bude produkt akceptovat. Testování v této fázi se neomezuje pouze na zásadní funkční testy, ale ve většině případů zahrnuje i řadu jiných typů testů k ověření mimofunkčních požadavků. [5] [7]
Nejpoužívanějšími testy systémového testování jsou:

- Funkční testy – systém splňuje specifikované či implicitní požadavky.
- Testy robustnosti – otestování chybných vstupů, neočekávané situace apod.
- Testy použitelnosti – hodnotí míru zaučení uživatele do systému a jeho použití.
- Testy interoperability – ověření komunikace s ostatními systémy (typicky od jiných dodavatelů).
- Testy spolehlivosti – ověřují, jak dlouho je systém schopný pracovat, aniž by došlo k selhání.
- Výkonnostní testy – měření výkonu systému pomocí sledování vybraných ukazatelů. [5] [7]

4. **Akceptační testování (Acceptance testing)**

Po úspěšné fázi systémového testování je produkt připraven na nejdůležitější validační aktivitu – akceptační testování zákazníkem. Toto testování je taktéž označováno zkratkou UAT. Hlavním smyslem akceptačních testů je zjistit, zda software splňuje tzv. akceptační kritéria. Tyto testy při převzetí produktu zákazníkem jsou předem domluveny a odsouhlaseny oběma stranami, jejich nesplnění může být důvodem odmítnutí produktu. [5] [7]

3.2.6.4 Dimenze kvality

Funkčnost, která je zpočátku považována jako jediná testovatelná kategorie, je ve skutečnosti jen jednou z dimenzí kvality (dimenze typy testů). Pro další dimenze se běžně používá pojem jako mimofunkční požadavky, nefunkční požadavky, či nefunkcionální požadavky. Každá z těchto dimenzí by měla být pokryta alespoň jedním testem pro zajištění kvalitního softwaru. [7]

Dimenze kvality jsou definované normou ISO/IEC 25010:2011 a slouží především k pochopení, že testovat jen funkčnost je nedostatečné a že pohledů na kvalitu je více. [6]

- Funkčnost (Functionality)
    - Tato charakteristika představuje míru, do jaké produkt nebo systém poskytuje funkce, které splňují stanovené a předpokládané potřeby při použití za specifikovaných podmínek.
- Výkon (Performance)
    - Tato charakteristika představuje výkon ve vztahu k množství zdrojů použitých za uvedených podmínek.
- Kompatibilita (Compatibility)
    - Míra, do jaké si produkt, systém nebo komponenta může vyměňovat informace s jinými produkty, systémy nebo komponentami a/nebo vykonávat požadované funkce při sdílení stejného hardwarového nebo softwarového prostředí.
- Použitelnost (Usability)
    - Míra, do jaké může být produkt nebo systém používán určenými uživateli k dosažení stanovených cílů s účinností, efektivitou a spokojeností v určeném kontextu použití.
- Spolehlivost (Reliability)
    - Stupeň, do kterého systém, produkt nebo komponenta plní specifikované funkce za specifikovaných podmínek po stanovenou dobu.
- Bezpečnost (Security)
    - Míra, do jaké produkt nebo systém chrání informace a data, aby osoby nebo jiné produkty nebo systémy měly stupeň přístupu k datům odpovídající jejich typům a úrovním oprávnění.
- Udržitelnost (Maintainability)
    - Tato charakteristika představuje stupeň účinnosti a efektivity, se kterou lze produkt nebo systém modifikovat za účelem jeho zlepšení, nápravy nebo přizpůsobení změnám prostředí a požadavků.
- Přenositelnost (Portability)
    - Stupeň účinnosti a efektivity, se kterou lze systém, produkt nebo komponentu přenést z jednoho hardwaru, softwaru nebo jiného provozního nebo uživatelského prostředí do jiného. [6]

3.2.7 Typy testů

Kromě již zmíněných testů v rámci kapitoly testovacích metody jsou v této kapitole vybrány další nejpoužívanější typy testů.

3.2.7.1 Testování podle scénáře

Testování podle scénáře je jeden z nejčastějších typů testování, které je velmi účinné. Jeho principem je, že na základě specifikace nebo způsobu, jakým uživatel s aplikací pracuje, se vytvoří testovací scénáře. Testovací scénář obsahuje podrobný popis kroků, které má tester během testu provést a současně i očekávané reakce systému. Jedná se o jeden ze základních typů manuálního testování, které mohou provádět i junior testeři. [7]

3.2.7.2 Průzkumné testování

Jedná se o komplementární techniku k testování podle scénářů. Průzkumné testování je expertní metoda, která zpravidla bývá prováděna zkušeným test inženýrem, jehož úkolem je na základě provedeného průzkumného testu revidování a rozšiřování seznamu testovacích scénářů. Poté mohou testování dle takto upravených testovacích scénářů provádět i méně zkušení testeři, kteří tímto způsobem získají dané know-how. Stejně jako testování podle scénáře, je i průzkumné testování zaměřeno na funkčnost a použitelnost aplikace, ale liší se tím, že nemá předem stanovený plán postupu testování. Jedná se tak o scénář, kdy tester si libovolně „hraje“ s aplikací a na základě svých zkušeností a znalostí nalézá případné chyby. [7]

3.2.7.3 Regresní testy

Během vývoje softwaru je často některá jeho součást modifikována, ať jde opravu chyby, rozšíření funkčnosti nebo změnu kódu s cílem zlepšení výkonnosti. Při každém takovém zásahu a s každou změnou vzniká riziko, že může být do již otestovaného systému nebo jednotlivých komponent zavlečen defekt. Z toho důvodu jsou prováděna tzv. regresní testování, což je ověření, že po změně určité části systému zůstávají ostatní součástí funkční a ve stejném stavu. Regresní testování ve vztahu k jednotlivým fázím testování je prováděno na jednotkové, integrační a systémové úrovni. Regresní testy už ze své podstaty musí využívat již existující testovací případy, aby zachytily případný rozpor mezi aktuálními a dříve získanými výsledky. V podstatě lze používat jakékoliv funkční testy, které již dříve úspěšně proběhly. Množství regresních testů tvoří tzv. balík regresních testů,

jež obsahuje testy pokrývající všechny součásti aplikací. Testování pomocí tohoto balíku by mělo být prováděno pravidelně u každého buildu (nové verze) a ideálně automatizovaně. U regresního testování se dává přednost zejména automatizovanému testování, protože síla regresních testů je v jejich značném množství. [5] [7] [11]

3.2.7.4 Smoke testy

Jako smoke testy se označují testy, jejichž účelem je pouze zjistit, zda je systém dostatečně stabilní a všechny jeho hlavní části fungují správně. Testy se zpravidla provádějí bezprostředně po sestavení nové verze a na základě výsledků testů se pak rozhoduje o tom, zda má vůbec na dané verzi smysl v testování pokračovat. [5] [7]

3.2.7.5 End-to-end testy

Principem end-to-end testů, zkráceně označovaných jako E2E, je sledování určité entity (objekt, data apod.) po celou dobu její životnosti napříč systémem, resp. od jednoho konce (end) až ke druhému, a ověřování správnosti jednotlivých interakcí/přechodů a konečného stavu. Zpravidla se E2E testy provádí jako součást akceptačních testů, v podobě scénářů zahrnujících jednotlivé aktivity uživatelů od prvního až do posledního kroku s cílem potvrdit jejich správnost jako celku. Jedná se o testy zahrnující například přihlášení, získání dat, provedení určité aktivity až po odhlášení. [5]

3.3 Automatizované testování

Jak bylo popsané v předchozí kapitole, testování softwaru je velmi náročná disciplína, jelikož fyzické provedení veškerých zmíněných testů je velmi časově náročné. Nabízí se tedy to, co lidé dělají již spoustu let nejen v softwarovém průmyslu, ale i v každém jiném oboru lidské činnosti – vyvinutí a používání vhodných nástrojů, které danou práci usnadňují a zvyšují tak její efektivitu. [1]

Automatizované testování je jakékoliv testování, k jehož provedení je využíván software, kterým jsou vykonávány aktivity místo lidských testerů a do určité míry je i nahrazuje. Jak již bylo zmíněno v kapitole 3.1, dříve trval životní cyklus softwaru běžně několik měsíců až let. V dnešní době agilního vývoje a CI/CD již tomu tak není, běžně se release dělají každých 14 dní a v tomto tempu již není možné komplexně testovat software manuálně. Automatizovanému testování je věnována stále větší pozornost, neboť s rostoucí

komplexitou a nároky na kvalitu softwaru se také vyvíjí a zlepšují schopnosti automatizačních nástrojů. Mluvit ale o plnohodnotném nahrazení manuálního testování je více než přehnané. Účelem automatizovaného testování není nahrazení manuálního testování, nýbrž jeho doplnění. Aktuálně nejčastějším důvodem pro automatizaci je snaha ušetřit náklady na opakované testování a získat efektivnější způsob, jak zamezit regresním defektům a snížit chyby testera. Jak návrh a provedení, tak analýza výsledků testovacích případů jsou činnosti, jejichž čistě manuální výkon je časově velmi náročný. Navíc opakované provádění testovacího případu může vést k určité nepozornosti a opomenutí náležitostí tohoto případu. Proto se někdy i samotní testeři snaží některé z úloh automatizovat, čímž šetří čas a zároveň mají jistotu, že prověřený automatizovaný test poskytuje správný výsledek. Takovýto framework pro automatizované testování lze tedy chápat jako platformu, skrze níž jsou řízeny procesy, struktury, nástroj pro návrh, realizaci a provádění automatizovaných testů. [1] [5] [11]

Pokud je automatizace dobře navržena a prováděná, má mnoho výhod. Mezi ty nejdůležitější výhody lze uvést:

- Snížení nákladů na testování systému z dlouhodobého hlediska, zejména díky automatizaci regresních testů.
- Vysoké pokrytí regresními testy v důsledku stále se zvyšujícího počtu automatizovaných testovacích případů.
- Zvýšení výkonnosti manuálních testerů a analytiků.
- Dosažení vysoké účinnosti vybraných testů.
- Vyvinutý nástroj pro automatizované testování lze využít i v dalších projektech. [11]

3.3.1 Manuální vs. automatizované testování

Typickým názorem některých manažerů je, zcela nahradit manuální testování automatizovaným, čímž by dosáhli zvýšení efektivity a snížení nákladů. To však u většiny systémů je nemožné, jelikož automatizované testy ani ve své nejlepší a nejsofistikovanější podobě nejsou skutečně inteligentní. Automatizované testy sice vykonávají, třeba i s jistou mírou variability a teoreticky i učenlivostí, nakonec jen to, co tvůrce při jejich návrhu učinil. Nelze tedy o automatizovaných testech uvažovat stejně jako o manuálních, jelikož jejich hodnoty jsou zcela odlišné. Automatizovaný test vykonává opakovaně stejnou

činnost rychle a bezchybně. Oproti tomu manuální testování má své kvality, kterým žádné výkonnostní parametry nemohou konkurovat, a to myšlení, kreativitu či znalost aplikace. Automatizovaný test sice dokáže ověřit ve zlomku vteřiny, zda všechny řetězce na stránce zobrazují správná data, ale nedokáže např. ověřit, že písmo stránky je pro člověka nečitelné, nebo že část některého textu je mimo stránku. Automatizace takové činnosti může být někdy i nemožná nebo velmi náročná v porovnání s prostým provedením manuálního testu. [5] [11]

Hlavní rozdíly mezi manuálním a automatizovaným testováním jsou shrnuté do následující tabulky č. 1.

Automatizované testování	**Manuální testování**
Rychlejší exekuce	Nemá technická omezení
Možná paralelizace testu	Zachycení chyb mimo testovanou funkcionalitu
Časově neomezená doba exekuce	Časová náročnost
Nezachytí chyby, které nesouvisí s testovanou funkcionalitou	Možné přehlédnutí chyb
Riziko velké náročnosti na údržbu testu	

Tabulka 1 – Srovnání manuálního a automatizovaného testování

3.3.2 Vhodné testy pro automatizaci

Automatizované testy lze využít na různých úrovních vyvíjeného softwaru – od samotného kódu až po simulaci uživatelských kroků v uživatelském rozhraní testovaného softwaru – GUI (Graphical User Interface). Prakticky každou úroveň V-modelu lze podpořit automatizovanými testy. K dispozici je také řada různých technických možností, na jakém principu automatizovaný testy vytvořit. O automatizovaném testování se nejčastěji slýchává v souvislosti s prováděním regresního testování front-endových testů. Ve skutečnosti jsou podoby automatizovaného testování mnohem rozmanitější. Při automatizaci jsou využívány zejména následující typy testů:

- Jednotkové testy
- Funkční testy
- Výkonnostní testy a jeho varianty
- Bezpečnostní testy
- Regresní testy
- E2E testy. [5] [11]

Automatizace všech testovacích případů není většinou žádoucí a ani prakticky možná, a tak se pochopitelně nabízí otázka, jaké testy je vhodné automatizovat? Přestože jednoznačně správná odpověď patrně neexistuje, při výběru testů k automatizaci by se mělo uvažovat následovně:

- Stabilita – Jedná se o základní kritérium – testovací případ, který je proměnlivý (je často modifikovaný), nemá příliš smysl automatizovat.
- Četnost provádění – Testy, které jsou prováděny často a pravidelně je vhodné automatizovat. Typickým příkladem jsou smoke, integrační a regresní testy.
- Důležitost (priorita) – Testy, které ověřují kritickou funkcionalitu, jsou prováděny na každém novém buildu systému.
- Obtížnost provádění – Testy jsou různě obtížně proveditelné z mnoha důvodů, jako je vysoká komplexita výpočtů, sekvence velkého množství kroků a operací apod. Pokud je manuální test obtížné provést, mělo by se uvažovat o jeho automatizaci.
- Časová náročnost – Pokud provádění určitého testu vyžaduje značné množství času, může se jeho automatizace vyplatit i tehdy, jestliže není prováděn příliš často.
- Složitost případné automatizace – Některé testy lze automatizovat velice jednoduše, jiné nikoliv. [5]

3.3.3 Jak automatizovat testy

Nejjednodušším způsobem, jak testy automatizovat, je vybrání manuálních testů, které se opakují a ty automatizovat. Na základě již vytvořeného testovací scénáře (případu) se jednotlivé kroky přepíšou do skriptu nebo nahrají v nástroji k tomu určenému. Nyní místo manuálního testování pomocí testera je možné testovat automatizovaně. To ale nemusí být zcela efektivní, a to zejména z následujících důvodů:

1. Testovací scénáře vhodné pro manuální testování nemusí být úplně vhodné pro automat.
2. Nevyužita velká část potenciálu automatizovaných testů. [11]

Potenciál automatizovaných testů je veliký, jelikož pokud se již jednou vytvoří automatizovaný test, získávají se i další možnosti, jak jej dále použít. A to zejména:

- Automatizovaný test lze spouštět podstatně častěji než při manuálním testování, čímž je zmenšené riziko neodhalených regresních defektů.
- Automatizované testy lze spouštět na různých platformách.

- Nejdůležitější části testovaného systému je možné otestovat s mnohem větším počtem kombinací vstupních dat.
- Automatizované testy lze využít pro přípravu testovacích dat pro manuální testování.
- Vybrané automatizované testy lze využít i v monitoringu dostupnosti testovacího nebo někdy i produkčního prostředí.
- Funkčnost jednotlivých prostředí lze vhodně vybranými automatizovanými testy ověřit. [11]

3.3.4 Webové stránky

Jelikož tématem této diplomové práce je automatizované testování webových aplikací, je dobré si uvědomit zvláštnosti webových aplikací (stránek), se kterými je nutné při vývoji a testování počítat. Běžně se setkáváme s různě složitými typy webových aplikací. Podle složitosti webových aplikaci je lze rozdělit na:

- Webové stránky
- Webové aplikace
- Systémy s tenkým klientem. [1] [7]

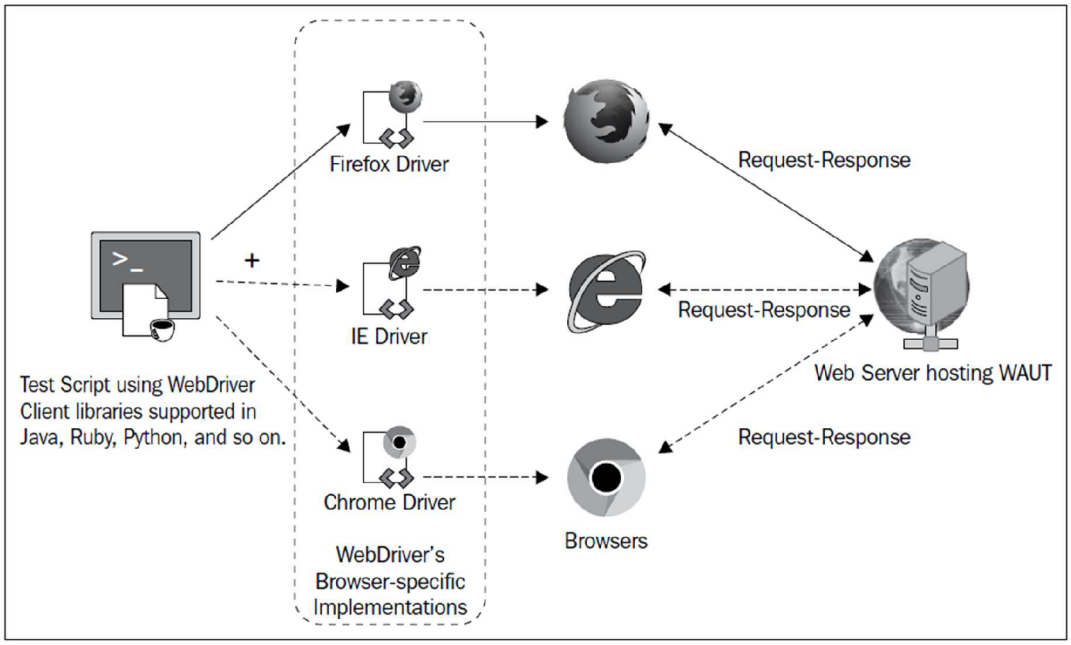
Důležitým faktorem při testování webových aplikací jsou velmi časté změny. S tím je nutno při testování počítat, jelikož není prostor analyzovat příčiny a důsledky faktu časté změny. S těmito častými změnami souvisí jeden velký problém pro vývojáře testů – testy musí neustále aktualizovat. S tím je spojen pojem křehký test, což je test, který v další verzi nefunguje. Nesmírně důležité je i proměnlivá zátěž webové aplikace. Asi každý se setkal se situacemi jako je přihlašování na termín zkoušek, platba internetovým bankovnictvím na poslední chvíli, nebo registrace na očkování, kdy webové aplikace nezvládly zátěž uživatelů. Je tudíž dobré i při běžném provozu aplikace zátěžově testovat. Problém je také množství používaných webových prohlížečů a jejich verzí. U webových aplikací mají uživatelé různé typy konkrétních webových prohlížečů a na nich jsou závislé různé zvyky při práci s internetem. Pro identifikaci ovládacích prvků webové aplikace lze využít metodu šedé skříňky, protože je možné si prohlédnout testovaný HTML (Hypertext Markup Language) kód. Pro automatizované testování je vhodné do HTML kódu přidávat informace, které výrazně zvýší testovatelnost aplikace. Jednotlivé ovládací prvky je možné relativně snadno identifikovat pomocí konkrétních identifikátorů přiřazených jednotlivým

prvkům. Pro zkoumání webové struktury je možné využít nástroje jednotlivých webových prohlížečů. [1] [7]

3.4 Selenium WebDriver

Podle reportu „Prostředí automatizovaného testování 2020“ (Test Automation Landscape in 2020 Report), odpovědělo až 43 % respondentů, že jako nástroj pro automatizované testování GUI svých webových aplikací používá nástroj Selenium WebDriver. [12]

Selenium je jeden z nejznámějších a nejstarších nástrojů pro automatizované testování GUI webových aplikací (stránek). Jedná se o sadu nástrojů pro automatizované testování, jejichž nejpoužívanějším nástrojem je Selenium WebDriver. Jedná se o open-source nástroj pro automatizované testování s webovým prohlížečem. Přibližuje automatizované testování co nejvíce skutečnosti, neboť reálně simuluje práci uživatele. Kromě testování jej lze použít i pro provádění jakýchkoliv činností na webu, které jsou prováděny opakovaně a má smysl je automatizovat. Selenium WebDriver pro simulaci práce uživatele s webovou aplikací využívá speciální aplikační programové rozhraní, díky němuž je aplikaci (prohlížeč) možné nativně ovládat, jak je tomu znázorněno na obrázku č. 11. Veškeré drivery prohlížečů jsou vydávány přímo daným webovým prohlížečem a jsou standardizované W3C (World Wide Web Consortium). Tím je dáno jeho postavení číslo 1 mezi nástroji pro automatizované testování, jelikož se jedná o jediný standardizovaný nástroj pro automatizované testování front-endových (GUI) testů. [13] [14] [15]



Obrázek 11 – Princip fungování Selenium WebDriver [13]

Aby bylo možné vytvářet automatizované testy pomocí Selenium WebDriveru, je zapotřebí několika náležitostí, bez kterých to není možné. Jednotlivé náležitosti jsou shrnuté do následujících kroků:

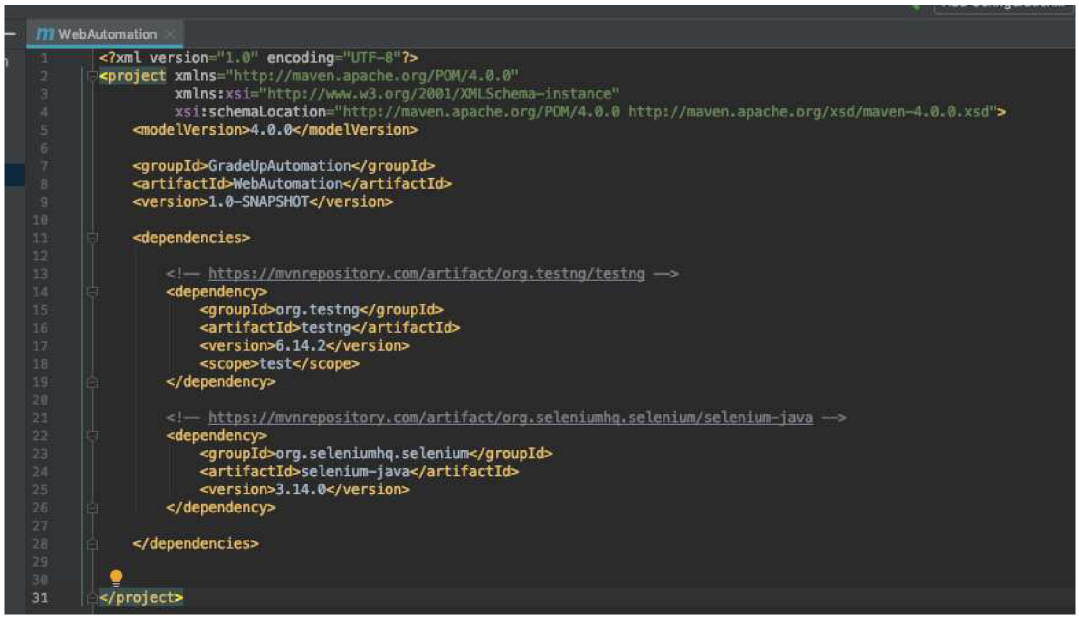
1. Výběr programovací jazyka (Java)
2. Vytvoření projektu (MAVEN)
3. Import externích knihoven
4. Kód testu. [15]

3.4.1 Programovací jazyk a vývojové prostředí

Nejdříve je třeba zvolit programovací jazyk, ve kterém budou automatizované testy vytvářeny. Je možné využít nejpoužívanější programovací jazyky jako Java, C#, Python, Ruby, PHP apod. V rámci této diplomové práce byl zvolen programovací jazyk Java. Na základě vybraného programovacího jazyka je potřebné SDK (Software Development Kit), což je sada nástrojů pro vývoj aplikací. Jelikož byl zvolen jazyk Java, bude se jednat o JDK (Java Development Kit). Pro lepší a přehlednější práci s kódem se standardně využívá některé z vývojových prostředí (IDE) jako IntelliJ IDEA, NetBeans, Eclipse apod. Po spuštění vybraného vývojového prostředí je zapotřebí vytvořit nový projekt, v němž budou prováděny automatizované testy. [13]

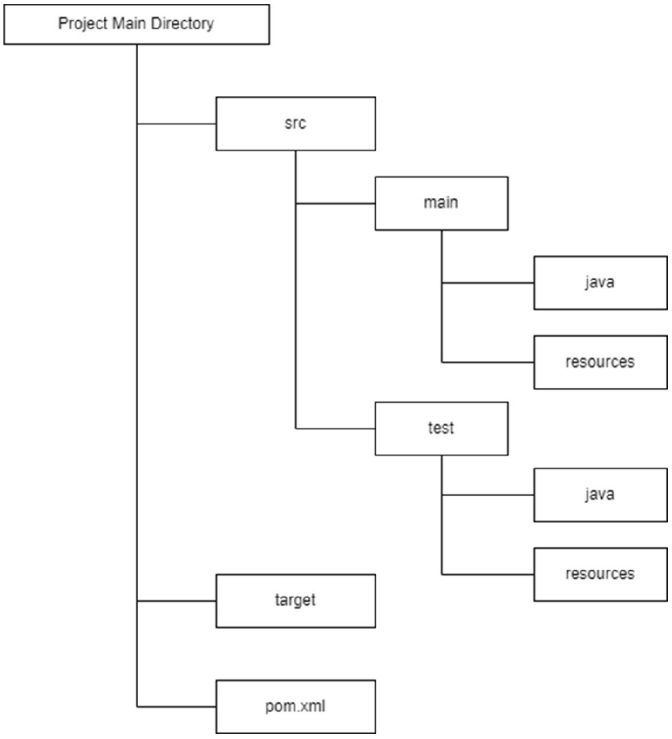
3.4.2 MAVEN projekt

V případě vytváření automatizovaných testů pomocí jazyka Java je vhodné pracovat s tzv. MAVEN projektem. MAVEN (celým názvem Apache Maven) je nástroj (rozšířený systém) pro správu a sestavování aplikací postavených nad platformou Java. Aktuálně je MAVEN plně integrován do všech velkých IDE, takže práce s ním je velmi snadná. Hlavním principem MAVENU je vytvoření objektového modelu nad zdrojovým kódem pomocí souboru pom.xml (project object model), viz příklad na obrázku č. 12. Každý takový soubor pom.xml potom představuje jeden projekt. Obsahem souboru pom.xml jsou informace potřebné ke kompilaci a sestavení programu. Mezi hlavním informacemi MAVEN projektu jsou GroupId, ArtifactId a Version. Jako GroupId se většinou zvolí název webové domény pro kterou jsou testy vytvářeny, a to v obráceném tvaru, jako například cz.czu.automatiozvanetestovani. ArtifactId představuje jméno jar (Java Archive) souboru, který bude vytvořen (zkompilován) v rámci projektu a Version označuje verzi projektu. Dále pomocí elementu <dependencies> a jeho vnořených elementů <dependency> se vytvářejí závislosti na externích knihovnách, které jsou v projektu využity. [16]



Obrázek 12 – Příklad pom.xml [16]

Struktura vytvořeného MAVEN projektu je znázorněna na obrázku č. 13. Vytvořený adresář projekt obsahuje dva adresáře src a target, společně s již zmíněným souborem pom.xml na stejné úrovni.



Obrázek 13 – Struktura MAVEN projektu

Popis jednotlivých adresářů projektu MAVEN je:

- src – kořenový adresář zdrojového kódu a testovacího kódu.
- main – adresář obsahuje zdrojový kód související se samotnou aplikací, nikoli testovací kód.
- test – adresář obsahuje zdrojové kódy testu.
- java – adresář obsahuje kódy Java.
- resources – adresář obsahuje potřebné prostředky projektu.
- target – adresář obsahuje veškeré zkompilované třídy ve výsledném balíčku – .jar (pokud není, vytváří se automaticky při kompilaci). [16]

3.4.3 Externí knihovny

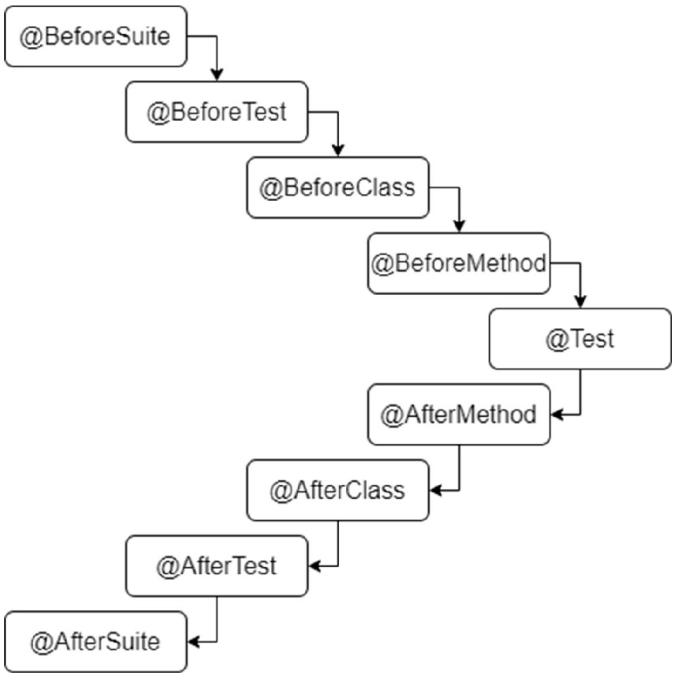
Dalším krokem je import potřebných knihoven k automatizovanému testování. To se provede v souboru pom.xml doplněním elementu <dependencies> s požadovanými vnořenými elementy <dependecy>, ve kterých jsou uvedeny informace o externích knihovnách. Na základě těchto informací MAVEN provede automatické stažení

požadovaných knihoven. První potřebnou knihovnou je samozřejmě knihovna Selenium, která obsahuje veškeré instrukce pro ovládání WebDriveru (prohlížeče). [15]

Jelikož Selenium (knihovna) negeneruje formát výsledku automatizovaného testu, je zapotřebí použít některou z knihoven pro generování výsledků testů. Zároveň je také vhodné použít knihovnu pro tzv. logování neboli zaznamenávání průběhu testu (aplikace).

3.4.3.1 TestNG

Nejpoužívanější knihovnou pro generování výsledků Selenium testů je TestNG. TestNG je framework pro automatizované testování, nebo také jednotkové testy, inspirované frameworkem JUnit. Pomocí TestNG lze generovat report o provedeném testování, v němž uživatel snadno zjistí, které provedené testy byly úspěšné a které nikoliv. Zpřehledňuje také kód automatizovaného testu, a to pomocí tzv. anotací. Anotace v TestNG jsou řádky kódu, které mohou ovládat, jak (kdy) bude provedena metoda pod nimi. Označují se symbolem @ a jejich názvy jsou srozumitelné a již z názvu napovídající co dělají. Hierarchie jednotlivých anotací TestNG je znázorněna na obrázku č. 14 a popis jednotlivých anotací v tabulce č. 2. [17]



Obrázek 14 – Hierarchie anotací TestNG

Anotace	Popis
@BeforeSuite	Metoda bude spuštěna před spuštěním všech testů v této sadě.
@BeforeTest	Metoda bude spuštěna před spuštěním jakékoli testovací metody patřící do tříd uvnitř značky.
@BeforeClass	Metoda bude spuštěna před vyvoláním první testovací metody v aktuální třídě.
@BeforeMethod	Metoda bude spuštěna před každou testovací metodou.
@Test	Testovací metoda je součástí testovacího případu.
@AfterMethod	Metoda bude spuštěna po každé testovací metodě.
@AfterClass	Metoda bude spuštěna po spuštění všech testovacích metod v aktuální třídě.
@AfterTest	Metoda bude spuštěna po spuštění všech testovacích metod patřících do tříd uvnitř značky.
@AfterSuite	Metoda bude spuštěna po spuštění všech testů v této sadě.

Tabulka 2 – Anotace TestNG

Jednotlivé testy lze tak seskupovat a také stanovovat prioritu provedení jednotlivých testů pomocí příkazu priority = <0;∞>, kde 0 reprezentuje nejvyšší prioritu provedení. Kromě toho je možné nastavit opakování testu příkazem invocation count = počet opakování. [17]

Hlavními důvody, proč používat TestNG společně s nástrojem Selenium jsou:

- Generování výstupu testování.
- Zpřehlednění kódu testu pomocí anotací, které jsou srozumitelné.
- Zorganizování testů.
- Spouštění více testů najednou.
- Opakované provedení testu.
- Usnadnění pochopení testů v Seleniu. [17]

3.4.3.2 Log4j

Jedním z nejsnazších způsobů a zároveň nejpoužívanějším nástrojem pro logování neboli zaznamenávání dat (chodu aplikace) v programovacím jazyku Java je open-source Log4j. V rámci aplikací vyvíjených v Java se vyvinul ve faktický standard logování. Log4j dokáže ukládat výstup jak standardním způsobem do konsole, tak do textového souboru, anebo složitějšími způsoby do databáze apod. Aby bylo možné Log4j použít s nástrojem Selenium stačí opět přidat knihovnu v hlavním konfiguračním souboru projektu pom.xml.

Následně v konfiguračním souboru Log4j ve složce src/ se nastaví definice výstupu logovacího procesu. Log4j se skládá ze 3 hlavních komponent: Logger – samotná instance pracující se zprávami, Appenders – tam bude uložen výstup, Layout – určuje, jaký bude formát výstupu.
[18] [19]

3.4.4 Lokalizování elementů

Aby bylo možné ovládat webovou stránku automatizovaně, je potřeba kromě znalosti akcí Selenia umět lokalizovat elementy (prvky) webové stránky, jako je tlačítko přihlásit, nadpis stránky apod. To je prováděno pomocí DOM (Document Object Model). Jedná se o jakési rozhraní umožňující načtení celého XML (Extensible Markup Language) dokumentu stránky do paměti a vytvoření stromové objektové reprezentace. Jednotlivým objektům stromové struktury se potom říká uzly (nody), které jsou reprezentovány různými elementy, tagy, atributy, textovými obsahy, komentáři, vlastnostmi jednotlivých prvků a dalšími částmi dokumentu. Veškeré lokalizování prvků webové stránky na základě DOM je shrnuté do následující tabulky č. 3. [13]

Lokátor	**Popis**
class	Lokalizovaní prvků, jejichž atribut CLASS odpovídá hledané hodnotě.
id	Lokalizování prvků, jejichž atribut ID odpovídá hledané hodnotě.
name	Lokalizovaní prvků, jejichž atribut NAME odpovídá hledané hodnotě.
link text	Lokalizování prvků, jejichž viditelný text odpovídá hledané hodnotě.
tag name	Lokalizování prvků, jejichž název značky odpovídá hledané hodnotě.
partial link text	Lokalizování prvků, jejichž text odkazu odpovídá hledané hodnotě.
css	Lokalizovaní prvků odpovídající selektoru CSS.
xpath	Lokalizování prvků na základě výrazu XPath.

Tabulka 3 – Lokalizování prvků webové stránky pro Selenium WebDriver

Veškeré lokalizování prvků lze získat z HTML kódu webové stránky. Ze zmíněných lokátorů je nejvíce použitelný lokátor XPath, pomocí kterého lze zapsat veškeré prvky webové stránky.

3.4.4.1 XPath

XPath je dotazovací jazyk nad XML dokumentem a společně s XSLT (Extensible Stylesheet Language Transformations) je neoddělitelně spojen. Mimo jiné jej lze použít při

používání DOM, respektive při lokalizování prvků webové stránky. Dotazovací jazyk má dvě základní schopnosti. První je možnost zapisovat výrazy, jejichž výsledkem je nejčastěji množina uzlů, které jsou dále zpracovávány. Tou druhou je, že obsahuje knihovnu s více než 100 funkcemi pro nejrůznější operace, které jsou většinou využity při vytváření výstupu v XSLT. [20]

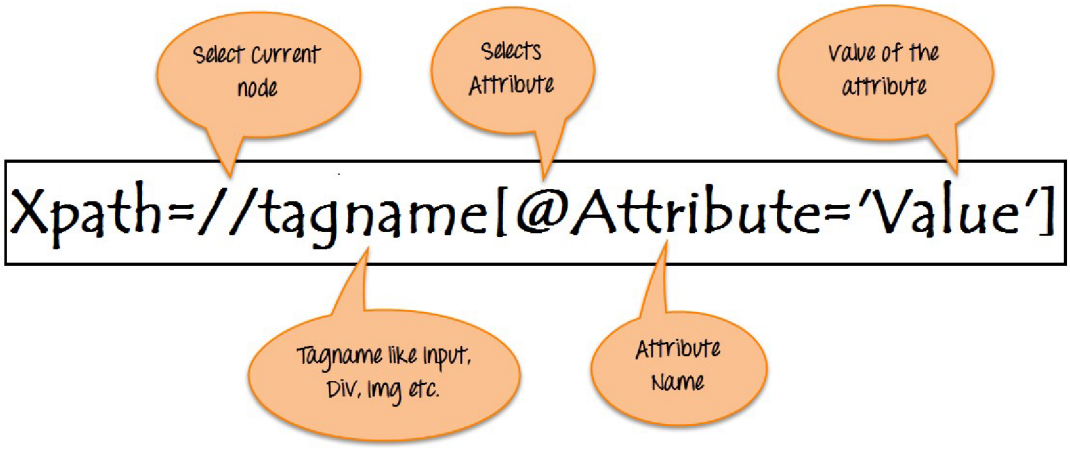
Existují dva hlavní zápisy XPath:

1) Absolutní cesta XPath

   Jedná se o výpis kompletní stromové struktury, hierarchii jednotlivých elementů od kořene dokumentu až po hledaný elementem. Zápis absolutní cestou začíná vždy jedním lomítkem (/), které značí, že první element je zároveň kořen dokumentu. Hlavní nevýhodou tohoto zápisu je, že v případě změn v cestě k prvkům je zápis nefunkční.

2) Relativní cesta XPath

Oproti absolutnímu zápisu obsahuje pouze elementy nutné pro nalezení požadovaného prvku. Zápis začíná dvojitým lomítkem a dokáže vyhledat prvky (uzly) kdekoliv na webové stránce, a to díky struktuře HTML DOM. Tento zápis je vždy preferován, jelikož se nejedná o úplnou cestu z kořenového prvku dokumentu. Příklad zápisu relativní cesty XPath v prostředí webových stránek (DOM) je znázorněna na obrázku č. 15. [20] [21]



Obrázek 15 - Základní formát zápisu XPath [21]

Jak již bylo napsáno, zápis začíná dvojitým lomítkem, čímž se zvolí aktuální uzel (tagname). Tagname reprezentuje daný uzel (element), v případě HTML stránky se jedná

o elementy jako input, div, img, button apod. Jako tagname lze použít *, což znamená výběr veškerých elementů odpovídajících vybranému atributu. Pomocí @ se vybere atribut elementu – Attribute a jeho požadovaná hodnota – Values v uvozovkách. Ne vždy ale tento základní formát zápisu je dostačující. Používají se tedy i zápisy pomocí predikátu číselné hodnoty v hranaté závorce, což se chápe jako pozice uzlu v posloupnosti na dané ose. Nebo je často využíváno hierarchických vztahů mezi uzly neboli jednotlivé osy výběru uzlů. [21]

Možnosti pohybu po osách lze znázornit na následujícím ukázkovém xml dokumentu:

```
<A>
  <AB>
        <ABC1>
        </ABC1>
        <ABC2>
                <ABCD1>
                        <ABCD1E1>
                        </ABCD1E1>
                </ABCD1>
                <ABCD2>
                </ABCD2>
                <ABCD3>
                        <ABCDE1>
                                <ABCDEF1>
                                </ABCDEF1>
                        </ABCDE1>
                        <ABCDE2>
                        </ABCDE2>
                        <ABCDE3>
                        </ABCDE3>
                </ABCD3>
                <ABCD4>
                </ABCD4>
        </ABC2>
        <ABC3>
        </ABC3>
  </AB>
</A>
```

S danou strukturou ukázkového xml dokumentu a výchozího (vybraného) elementu ABCD3 jsou jednolité definované vztahy uzlu k ostatním uzlům a jejich zápis XPath znázorněné v tabulce č. 4. [20] [22]

Vztah	Popis	XPath zápis	Ukázkový XML
Self (aktuální)	Označuje sebe sama.	self::element	ABCD3
Parent (rodič)	Označuje vždy svého jediného rodiče.	parent::element	ABC2
Ancestor (předci)	Označuje všechny předky včetně rodiče v pořadí od rodiče výše.	ancestor::element	ABC2, AB A
Ancestor-or-self (aktuální a předci)	Označuje sebe sama a všechny své předky v pořadí od sebe výše.	ancestor-or-self::element	ABCD3 ABC2 AB A
Child (děti)	Označuje všechny přímo vnořené potomky v pořadí, jak jsou v XM.L.	child::element	ABCDE1, ABCDE2, ABCDE3
Descendant (potomci)	Označuje všechny potomky v pořadí, jak jsou v XML.	descendant::element	ABCDE1 ABCDEF1 ABCDE2 ABCDE3
Descendant-or-self (aktuální a potomci)	Označuje sama sebe a všechny své potomky v pořadí od sebe, jak jsou v XML.	descendant-or-self::element	ABCD3 ABCDE1 ABCDEF1 ABCDE2 ABCDE3
Preceding-sibling (předchozí sourozenci)	Označuje všechny v XML předcházející uzly, které jsou sourozencem aktuálního uzlu, v opačném pořadí než v XML.	preceding-sibling::element	ABCD2 ABCD1
Preceding (předchůdci)	Označuje všechny předcházející uzly s výjimkou předků, v opačném pořadí než v XML.	preceding::element	ABCD2 ABCD1E ABCD1
Following-sibling (následní sourozenci)	Označuje všechny následující uzly, které jsou sourozenci aktuálního uzlu, v pořadí jak jsou v XML.	following-sibling::element	ABCD4
Following (následníci)	Označuje všechny následující uzly s výjimkou potomků v pořadí, jak jsou v XML.	following::element	ABCD4 ABCD4E1

Tabulka 4 – Vztahy mezi uzly XPath [20]

Kromě již zmíněného klasického formátu zápisu XPath a hierarchických vztahů mezi uzly lze použít i některé z funkcí XPath, pomocí kterých jsou zadávány tzv. dynamické zápisy XPath. Tyto funkce bývají využity v případě, že prvky webové stránky nelze lokalizovat klasickým způsobem. [14] [21]

Nejpoužívanějšími funkcemi XPath pro vyhledávání webových prvků jsou:

- contains()

Tato metoda se používá v případě, že se hodnota libovolného atributu dynamicky mění. Účelem funkce je najít prvek s částečným výskytem daného textu kdekoliv v textu. Lze ji použít v kombinaci s jakýmkoliv atributem, nebo ten nejčastější způsob použití pro lokalizování prvku s reálným textem na webové stránce. Příklady zápisu této metody jsou tedy následující:

XPath=//*[contains(@name, 'user')]

XPath=//*[contains(text(), 'Přihlásit se)].

- OR & AND

Také je možné použít logické operátory OR & AND. Jejich zápis bývá nejčastěji v hranatých závorkách predikátu. Logický operátor OR neboli logický součet vrací elementy splňující alespoň jedenu z podmínek. Operátor AND neboli logický součin pak vrací elementy, které splňují veškeré podmínky výrazu. Příklad zápisu jsou tedy následující:

XPath=//*[@type=reset or @name='btnLogin']

XPath=//*[@type='submit' and @name='btnLogin'].

- starts-with()

Oproti metodě contains, je tato metoda k nalezení prvku s daným počátečním textem. Využívá se zejména při nalezení webových prvků, jejichž hodnota se dynamický mění, ale počáteční text je stejný. Příklad zápisu je následující:

XPath=//label[starts-with(@id,'message')]. [14] [21]

3.4.5 Kód automatizovaného testu

Každý automatizovaný test musí začít inicializací (nastavením) a spuštěním WebDriveru webového prohlížeče. Jednotlivé WebDrivery lze stáhnout, jak již bylo napsáno, od samotných výrobců webových prohlížečů. Po stažení pak už jenom stačí na začátku automatizovaného kódu daný WebDriver inicializovat následovně:

```
System.setProperty("webdriver.chrome.driver","/path/to/chromedriver");
ChromeDriver driver = new ChromeDriver();
```

[13] [14] [23]

Díky tomu se otevře stažený WebDriver webového prohlížeče a je možné jej pomocí knihovny selenium-java ovládat. První akcí po otevření zpravidla bývá přesměrování na

požadovanou webovou stránku, nebo otevření dané webové aplikace. To se provede pomocí příkazu:

```
driver.get("https://selenium.dev"); //Zkrácený zápis
driver.navigate().to("https://selenium.dev"); //Delší zápis
```

[13] [14]

Po takto otevřené webové stránce lze přejít k vytváření jednotlivých akcí automatizovaného testu. Většinu ovládacích akcí prohlížeče lze dosáhnout pomocí metod knihovny Selenia. V případě, že neexistuje metoda, nebo v rámci Selenia nefunguje, používají se k ovládání prohlížeče tzv. Javascripty. Javascript je skriptovací jazyk, který umožňuje vytvářet dynamický obsah webových stránek. Jelikož veškeré webové prohlížeče jej umí spouštět bez nutnosti instalace pluginů, jedná se o ideální doplněk pro Selenium, v případě kdy jednotlivé metody Selenia plně nestačí k provedení daného testu. Jedná se zejména o tzv. „vanilla Javascripty" neboli skripty, které umožňuje prohlížeč nativně ovládat, bez nutnosti nějakých knihoven. Pro spuštění daného scriptu se používá metoda executeScript, která spouští skript s danými argumenty. Díky argumentům je pak možné předat informaci javascriptu, o jaký prvek webové stránky se bude jednat. [24]

Na následujícím příkladu je znázorněna akce volba radiobuttonu:

```
WebElement inputElement = driver.findElement(By.xpath(radioXpath));
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("arguments[0].checked = true; arguments[0].blur(); return
true", inputElement);
```

[24]

Na uvedeném příkladu zápis "arguments[0]." znamená druhý parametr v metodě executeScript, tedy definovaný inputElement. Nula v závorce je z důvodu, jelikož metoda executeScript dokáže předat celé pole argumentu. [24]

Kromě provádění jednotlivých kroků v testovacím scénáři (případu), jako kliknutí na tlačítko nebo vyplnění pole, je zapotřebí také jejich kontrola. To se provádí pomocí tzv. asserts. V Seleniu jsou asserts ověření, nebo kontrolní body pro aplikaci, díky čemuž je uváděno, že chování aplikace je podle očekávání. Zjednodušeně se dá říct, že asserty se používají k ověření jednotlivých testovacích případů a pomáhají testerům pochopit, zda byl test úspěšný či nikoliv. Asserty jsou dvojího typu: hard assert a soft assert. Soft assert jsou tzv. měkké testy. Pokud u těchto testů není splněná podmínka, tak se výsledek zaznamená,

ale automatizovaný test pokračuje dál a až po jeho skončení se zobrazí. Opakem toho jsou hard assert, kde, pokud není splněná podmínka testování, tak automatizovaný test nepokračuje dál, zobrazí se chybová hláška a skončí. [25] [26]

Mezi nejpoužívanější metody assert. (hard assert) a softAssert. (soft assert) patří:

- assertEquals()

Tato metoda obsahuje minimálně 2 argumenty a její účelem je porovnání skutečné (obdrženého) hodnoty (výsledku) s očekávanou hodnotou. Pokud se obě hodnoty shodují, kontrola je úspěšně splněna. Pomocí této metody lze porovnávat řetězcem celá čísla, dvojníky a mnoho dalších proměnných.

- assertNotEquals()

Tato metoda dělá přesný opak metody assertEquals, tedy porovnává dvě hodnoty, které pokud nejsou stejné, podmínka kontroly je úspěšně splněna.

- assertTrue

Metoda assertTrue slouží k ověření "booleovské" hodnot, na základě zadané podmínky. Pokud je podmínka splněna, vrátí se true a kontrola je úspěšně splněna.

- assertFalse()

Opět se jedná o metodu, která dělá přesný opak jiné metody, v tomto případě assertFalse.

- assertNull

Tato metoda slouží k ověření objektu, k tomu, zdá má návratovou hodnotu null, neboli je „prázdný". Pokud je výsledek null, podmínka kontroly je úspěšně splněna.

- assertNotNull

Opak metody assertNull. [25] [26]

Shrnutí nejpoužívanějších metod kódu automatizovaného testu v programovacím jazyku Java pro ovládání webového prohlížeče a kontroly testu je uvedené v následující tabulce č. 5.

Popis akce	Java
Inicializace (nastavení) prohlížeče	`System.setProperty("webdriver.chrome.driver","/path/to/chromedriver");` `ChromeDriver driver = new ChromeDriver();`
Přesměrování na webovou stránku	`driver.get("https://selenium.dev");` `driver.navigate().to("https://selenium.dev");`
Kliknutí na tlačítko zpět	`driver.navigate().back();`
Kliknutí na tlačítko dopředu	`driver.navigate().forward();`
Refresh stránky	`driver.navigate().refresh();`
Kliknutí na tlačítko stránky	`WebElement loginButton = driver.findElement(By.xpath("//button(@name='btnSearch'"));` `loginButton.click();`
Vyplnění polí	`driver.findElement(By.xpath("//input(@name='user'")).sendKeys("Selenium");`
Javascript	`JavascriptExecutor js = (JavascriptExecutor) driver;` `js.executeScript();`
Kontroly (ověření) – Asserty	`softAssert.assertTrue(); Assert.asserTrue();` `softAssert.assertEquals(); Assert.assertEquals();`

Tabulka 5 – Seznam instrukcí Selenia

3.5 HOCON

HOCON (Human-Optimized Config Object Notation) je konfigurační jazyk, pomocí kterého jsou vytvářený soubory .conf, případně .hocon, a který je snadno čitelný pro člověka,. Jedná se o nadmnožinu JSON (JavaScript Object Notation) a o jednu z nejlepších možností, jak zapisovat a upravovat konfigurační soubory při programování. Primárním cílem jazyka HOCON je zachovat stejnou sémantiku (stromová struktura; sada typů; kódování/únik) jako je tomu v JSON, a učinit ji pohodlnější jako formát konfigurovatelného souboru upravitelného člověkem. Syntaxe jazyka HOCON vychází z již známe syntaxe JSON [27]. Oproti JSON je HOCON flexibilnější v tom, že existuje několik způsobů, jak zapsat platný zápis. Veškerá konfigurace se zapisuje do tzv. polí, které jsou označeny názvem (klíč) a závorkami (hranaté i {}). Hodnoty k prvkům se přidělují pomocí znaku : nebo =. [28] [29] [30]

Mezi hlavní komponenty jazyka HOCON patří:

- **Klíč** – jedná se o řetězec předcházející hodnotě.
- **Hodnota** – je řetězec, číslo, objekt, pole nebo logická hodnota za klíčem.
- **Separator klíče hodnoty** – odděluje klíče od hodnot, pomocí : nebo =.
- **Komentář** – pomocí prefixu # nebo // se za prefixem zakomentuje celý řádek kódu.
- **Substituce** – odkaz na již vytvořenou konfiguraci pomocí zápis ${klíč}. [28]

Jednotlivé komponenty jazyka HOCON lze předvést na následujícím příkladu souboru example.conf:

```
example {                        //Toto je příklad zápisu HOCON
   hrac1: {
      jmeno : “Frank”,
      vek : 37
   }
   hrac2 {
      jmeno = “John”
      vek = “34”
   }
}
```

Na uvedeném příkladu **jmeno : “Frank”**, je klíč jmeno, hodnota tohoto klíče je Frank a separator klíče hodnoty je =. Hlavním klíčem souboru je ale example ohraničené závorkami – {}. Obsahem pole example jsou dvě vnitřní pole (klíče) hrac1 a hrac2, které obsahují klíče jmeno a vek s uvedenými hodnotami. Na příkladu je vidět, že pole hrac1 a hrac2 mají odlišný zápis, ale vůči HOCONU jsou oba validní, což značí značnou flexibilnost HOCONU oproti JSON. Pokud bychom se chtěli odkazovat na tento zápis v nějakých jiných konfiguračních souborech, je nejdříve nutné naimportovat vytvořený konfigurační soubor pomocí zápisu include "example.conf", čímž se naimportuje veškerý obsah souboru example.conf v novém konfiguračním souboru a pomocí substituce se lze jednoduše odkazovat na již vytvořené hodnoty. Na uvedeném příkladu zápisem ${example.hrac1.jmeno} se získá (odkáže na) hodnotu jména hráče 1 - Frank. [29]

Pro načítání jednotlivých vytvořených konfigurací souborů se využívá knihovna typesafe. I tuto knihovnu lze naimportovat v hlavním konfiguračním souboru projektu MAVEN – pom.xml. Konfigurační soubory se načtou pomocí metody load(), díky čemuž je následně možné přistupovat a pracovat s jednotlivými prvky konfiguračního souboru

pomocí metod jako: getString – v případě textové hodnoty, nebo getList – v případě práce s polem. [31]

Jazyk HOCON lze využít pro jakékoliv zadávání hodnot na základě konfiguračních souborů. V souvislosti s nástrojem pro automatizované testování je možné pomocí HOCON zapisovat jednotlivé akce, které jsou v automatizovaném testování prováděny, zejména lokalizování jednotlivých elementů webové stránky pomocí XPath.

4 Vlastní práce

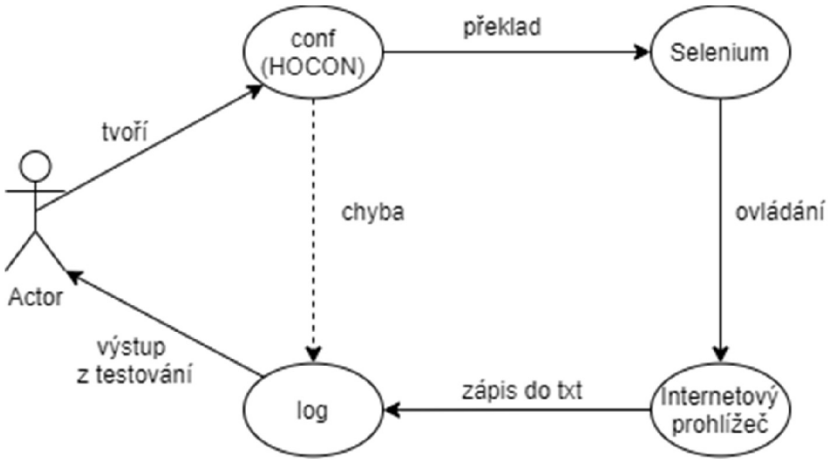
Na základě syntézy získaných poznatků z odborné literatury uvedených v této práci, byl vytvořený překladač HOCON pro nástroj pro automatizovaného testování webových aplikací (stránek) Selenium WebDriver, tak aby bylo umožněné vytvářet a spouštět automatizované testy běžnými uživateli.

Jako první byl vytvořen návrh překladače (viz kapitola 4.1), kde je detailně popsáno, k čemu je překladač určený, jak má fungovat a co vše je zapotřebí pro realizaci tohoto překladače. Následně byla provedena realizace (vývoj) překladače (viz kapitola 4.2), kde je popsáno, co je zapotřebí pro fungování překladače, jak překladač funguje, včetně vytvořeného kódu. Dále byla vytvořena syntaxe akcí, které budou interpretovány Seleniu. Po vytvoření překladače byl překladač ověřen nad vybranou webovou aplikací, pro kterou se vytvořili příkladové testovací případy. Součástí ověření bylo provedení manuálního testování a automatizovaného testování v programovacím jazyku Java.

4.1 Návrh překladače

Hlavní požadavek na HOCON překladače pro Selenium byl, aby bylo běžným uživatelům umožněno vytvářet automatizované testy, případně přímo manuálním testerům aplikace (uživatelským testerům). Vytváření automatizovaných testů pomocí samotného Selenia, není možné pro manuální testery. Automatizované testy jsou psány v různých programovacích jazycích, které běžný uživatelský tester zpravidla neovládá. Je proto zapotřebí zjednodušit tvorbu automatizovaných testů tvůrcům těchto testů.

Překladač funguje na principu konfiguračních souborů v jazyku HOCON. Pomocí jednoduché syntaxe HOCON je možné, aby si sám uživatel vytvářel automatizované testy již po velmi krátké době, než aby se učil vytvářet automatizované testy v jazyku Java. Zároveň zápis automatizovaného testu by měl být kratší než v jazyku Java a hlavně přehlednější. Pro překladač jsou na základě klíčových slov nadefinované jednotlivé možné akce automatizovaného testu, které jsou interpretovány (překládány) nástroji pro automatizované testování Selenium WebDriver. Vytvořený překladač je univerzální a je možné ho použít pro různé webové aplikace. Celý návrh překladače je znázorněn na obrázku č. 16.



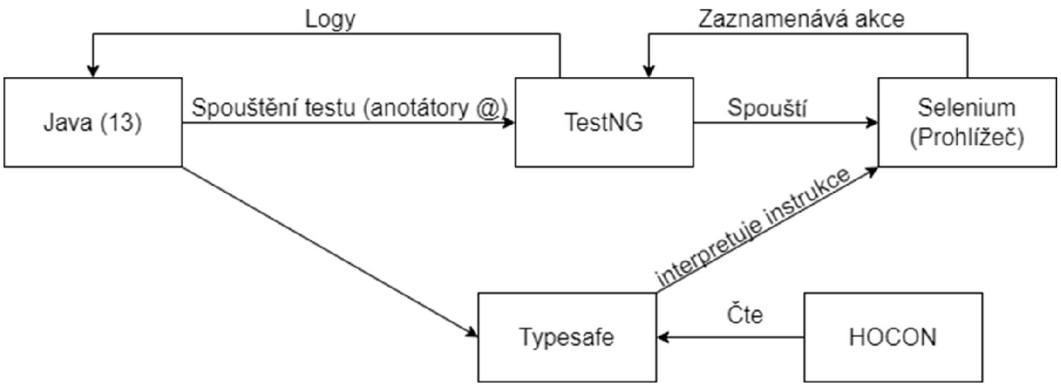
Obrázek 16 – Návrh překladače

4.2 Realizace překladače

Pro realizaci (vývoj) překladače bylo zvoleno prostředí IntelliJ IDEA ULTIMATE, které plně disponuje veškerými náležitostmi projektu. Překladač bude spouštěn ve zvoleném prostředí, případně bude využito zkompilovaného balíčku (jar) ke spuštění pomocí příkazové řádky. V obou případech je zapotřebí JDK verze 13 a novější.

4.2.1 Použité nástroje

jelikož jsem členem vývojového týmu, který vyvíjí software v programovacím jazyku Java, proto i tato aplikace (překladač) byla psána v programovacím jazyku Java. Vývoj a běh aplikace bylo prováděno pomocí JDK 13, který obsahuje soubor základních nástrojů na vývoj aplikace pro platformu Java. Pro zjednodušení buildu aplikace byl zvolen project management nástroj pro řízení buildů aplikací – MAVEN projekt. Pro výstup testování byla využita testovací funkcionalita Javy TestNG a pro načítání konfiguračních souborů v jazyce HOCON byla použita knihovna Typesafe. Veškeré použité nástroje a jejich účel jsou znázorněné na obrázku č. 17.



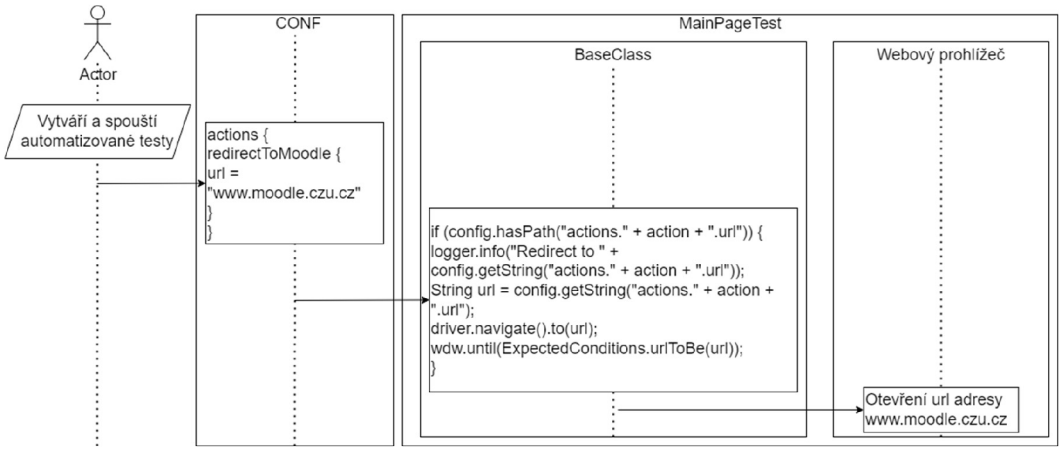
Obrázek 17 – Použité nástroje překladače

4.2.2 Fungování překladače

Po spuštění programu (překladače) se z hlavního konfiguračního souboru překladače načtou konfigurační soubory automatizovaných testů, které program vykoná ve zvoleném webovém prohlížeči. Konfigurační soubory automatizovaného testu obsahují seznam akcí automatizovaného testu (testovacího scénáře), které se mají provést. Překladač prochází tento soubor jako seznam a pomocí klíčových slov interpretuje Seleniu jakou akci má provést. Pokud akci není možné provést, nebo konfigurační soubor obsahuje nějaké chyby, daný automatizovaný test skončí a přejde se k dalšímu automatizovanému testu.

Uživatel (autor automatizovaných testů) pracuje pouze s konfiguračními soubory v jazyku HOCON na základě syntaxe nadefinovaných akcí překladače. Konfigurační soubory mohou být libovolně používány v různých testech a mohou se také libovolně opakovat. Pořadí, v jakém jsou prováděny, je vždy určeno podle toho, jak jdou akce v seznamu akcí v konfiguračním souboru za sebou.

Následující sekvenční diagram na obrázku č. 18 znázorňuje překlad akce přesměrování (otevření) webové stránky.



Obrázek 18 – Sekvenční diagram

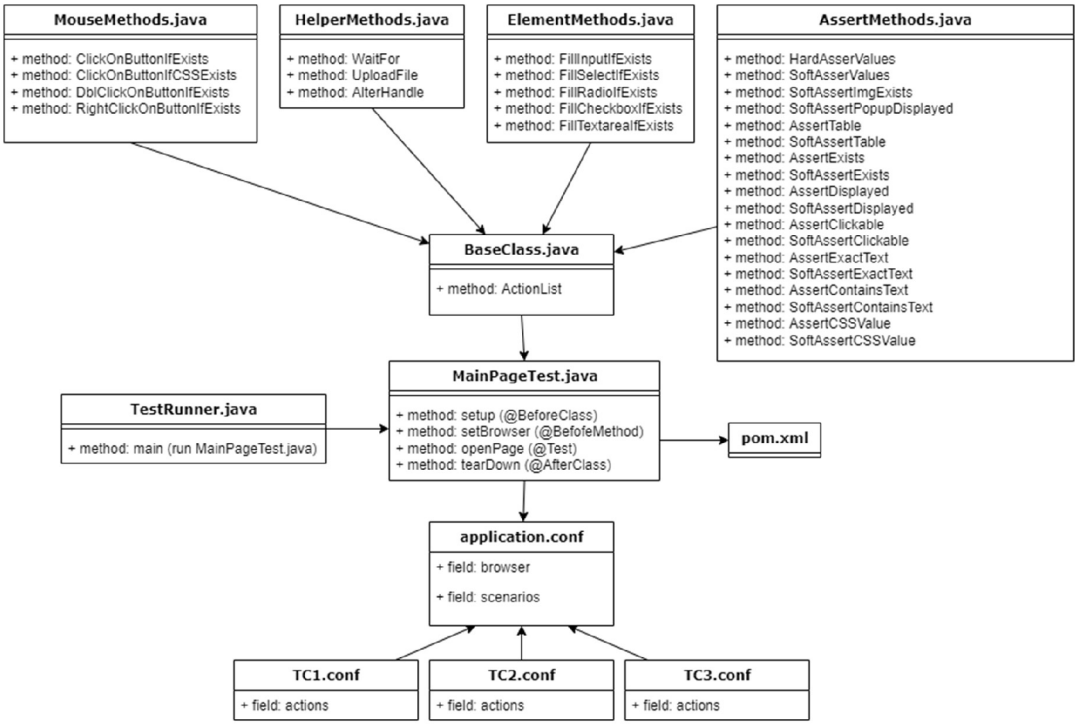
Z diagramu je vidět, že nadefinovaná akce v HOCON redirectMoodle obsahuje klíčové slovo url = "www.moodle.czu.cz", které překladač interpretuje Seleniu na základě definice (BaseClass).

4.2.3 Kód překladače

Tato kapitola se bude věnovat pouze souborům java, se kterými autor testů nepřichází do kontaktu a pomocí kterých jsou nadefinované jednotlivé akce a fungování překladače. Automatizované testy jsou tvořeny pomocí souborů conf, které jsou popsány v následující kapitole 4.2.4. Veškerý vytvořený kód, respektive jednotlivé soubory java, lze naleznout v elektronických přílohách práce – prekladacProjekt/Selenium.

Celý překladač je vytvořený pomocí projektu MAVEN. Jak bylo napsáno v teoretické části, hlavním souborem MAVEN projektu je pom.xml, ve kterém jsou naimportovány potřebné knihovny. Kromě zmíněných elementů v teoretické části se zde vyskytuje i element <build>, který specifikuje, jaké akce budou provedeny během kompilace balíčku. Obsahuje element <plugins> s dvěma vnořenýma elementy <plugin>. První z elementů <plugin> je maven-jar-plugin, což je plugin pro zabalení programu do zkompilovaného balíčku - .jar. V jeho vnořeném elementu <configuration> se nachází jednotlivé konfigurace: přidání externí složky do běhu programu (testConfig) a kde hledat potřebné knihovny pro běh překladače (ve složce lib/). Pomocí druhého elementu <plugin> jsou při zkompilování balíčku potřebné (použité) externí knihovny nakopírované do adresáře target/lib.

Pro lepší přehlednost, jsou veškeré soubory překladače znázorněny na obrázku č. 19.



Obrázek 19 – Class diagram překladače

Kód překladače se nachází v adresáři Selenium/src/main/java (viz elektronické přílohy práce) a je rozdělený do několika souborů, které jsou v této kapitole jednotlivě popsány. Jedná se pouze o popis daných souborů, jak pracují a k čemu jsou určené.

4.2.3.1 TestRunner.java

Z důvodu spouštění překladače samostatně pomocí zkompilovaného balíčku (jar) z příkazové řádky byla vytvořena tato třída (viz elektronické přílohy práce – prekladacProjekt /Selenium/src/main/java). Třída má jediný účel, a to spuštění souboru MainPageTest.java, respektive testovací třídy, která nejde sama o sobě spustit, jelikož soubor neobsahuje main class.

4.2.3.2 MainPageTest.java

Hlavním souborem projektu je MainPageTest.java (viz elektronické přílohy práce – prekladacProjekt/Selenium/src/main/java), který je tzv. spustitelný. V tomto souboru jsou naimportovány zmíněné knihovny potřebné k běhu Selenia a překladače HOCON. Při

testech jsou použity následující anotátory TestNG. Jednotlivé anotace s jejich účelem použití jsou:

- @BeforeClass – načtení konfiguračních souborů a nastavení softAssert.
- @BeforeMethod – nastavení prohlížeče.
- @Test – provedení testovacího scénáře (metoda ActionList).
- @AfterClass – výpis softassertu a zavření prohlížeče.

4.2.3.3 BaseClass.java

Třída BaseClass (viz elektronické přílohy práce – prekladacProjekt/Selenium/src /main/java) je přímo volána z třídy MainPageTest (hlavní třída), obsahuje metodu ActionList. Tato metoda interpretuje Seleniu veškerá předdefinovaná klíčová slova, která je možné použít v konfiguračním souboru HOCON a podle kterých se provádí požadované akce. Na začátku se načte konfigurační soubor automatizovaného testu. K tomuto se používá knihovna Typesafe. Pokud takovýto soubor má seznam úkonů (akcí) – actionList, a velikost tohoto seznamu je větší než 0, znamená to, že daný automatizovaný test obsahuje úkony, které se mají v daném pořadí, tak jak jdou za sebou, provést. V tomto případě se načte seznam pro běh aplikace (testu). Po načtení seznamu actionList překladač prochází všechny akce a podle předefinovaných klíčových slov jednotlivě interpretuje Seleniu to, co má provést. Pro přehlednost kódu jsou metody veškerých akcí rozděleny do tříd podle jejich typu, a to: MouseMethods, HelperMethods, ElementMethods a AssertMethods.

4.2.3.4 ElementMethods.java

Třída ElementMethods (viz elektronické přílohy práce – prekladacProjekt/Selenium /src/main/java) obsahuje nejběžnější elementární metody webové stránky. Jedná se o metody jako vyplnění input, zvolení hodnoty ze selectboxu a zaškrtnutí checkboxu nebo radiobuttonu. Jednotlivé metody jsou:

- FillInputIfExists – vyplnění zvoleného inputu a provedení požadovaného eventu.
- FillSelectIfExists – výběr požadované hodnoty ze zvoleného selectu.
- FillRadioIfExists – volba zvoleného radia.
- FillCheckboxIfExists – zaškrtnutí zvoleného checkbox.
- FillTextareafExists – vyplnění inputu zvolené textarea.

4.2.3.5 MouseMethods.java

Třída MouseMethods (viz elektronické přílohy práce – prekladacProjekt/Selenium /src/main/java) obsahuje ovládací prvky pomocí myši. Jednotlivé metody jsou:

- ClickOnButtonIfExists – kliknutí na html element.
- ClickOnButtonIfCSSExists – kliknutí na html element na základě CSS.
- DblClickOnButtonIfExists – dvojkliknutí na html element.
- RightClickOnButtonIfExists – pravý klik na html element.

4.2.3.6 HelperMethods.java

Třída HelperMethods (viz elektronické přílohy práce – prekladacProjekt/Selenium /src/main/java) obsahuje tzv. pomocné metody, pro hladký chod testu. Jednotlivé metody jsou:

- WaitFor – požadované čekání na prvek.
- UploadFile – nahrání souboru.
- AlterHandle – odkliknuti dialogového okna prohlížeče.

4.2.3.7 AssertMethods.java

Třída AssertMethods (viz elektronické přílohy práce – prekladacProjekt/Selenium /src/main/java) obsahuje veškeré kontrolní metody pro assert testy dané webové stránky. Většina kontrol je obojího typu, soft i hard assert. Jednotlivé metody jsou:

- Soft/HardAssertValues – kontrola hodnot požadovaných prvků.
- Soft/AssertTable – kontrola hodnot tabulky.
- Soft/AssertExists – kontrola existence prvku.
- Soft/AssertDisplayed – kontrola zobrazení prvků.
- Soft/AssertClickable – kontrola možnosti kliknutí na prvek/tlačítko.
- Soft/AssertExactText – kontrola přesné daného výrazu.
- Soft/AssertContainsText – kontrola hledaného výrazu.
- Soft/AssertCSSValue – kontrola CSS prvku.
- SoftAssertImgExists – kontrola existence obrázku.
- SoftAssertPopupDisplayed – kontrola zobrazení dialogového okna.

4.2.4 Tvorba automatizovaných testů

Tvorba automatizovaných testů je prováděna v konfiguračním jazyku HOCON pomocí konfiguračních souborů - .conf. Soubory lze vytvářet v jakémkoliv textovém editoru, ale je vhodné použít editor, který disponuje pluginem HOCON (pro lepší přehlednost kódu), jako například Notepad++ nebo prostředí IntelliJ IDEA.

4.2.4.1 Zápis prvků

Pro lokalizování prvků aplikace (tlačítko, pole, nadpis stránky apod.) je použitý zápis XPath (případně CSS). Pro tvorbu testů je nutné se orientovat v HTML kódu webové stránky a ovládat zápis prvků webové stránky pomocí XPath (případně CSS). Každý webový prohlížeč disponuje nástrojem pro zkoumání struktury webové stránky, pomocí kterého lze jednoduše dosáhnout XPath zápisu požadovaného prvku webové aplikace. Pomocí tlačítka F12 ve zvoleném prohlížeči se otevře nástroj pro zkoumání webové stránky, který zobrazí v určité části obrazovky webového prohlížeče celý HTML kód stránky. Kliknutí na ikonku  (nebo klávesová zkratka CTRL + SHIFT + C) se aktivuje mód pro zkoumání prvků webové stránky. Následným kliknutím na požadovaný prvek webové stránky se v kódu HTML zvýrazní kód prvku. Následně lze zapsat daný prvek pomocí XPath ručně (viz kapitola 3.4.4.1), nebo kliknout pravým tlačítkem myší a kopírovat cestu XPath.

4.2.4.2 Konfigurační soubory překladače

Hlavní konfigurační soubor – application.conf je hlavní soubor překladače, bez kterého nelze překladač spustit.

Obsah hlavního konfiguračního souboru **application.conf** je následující:

```
conf {
     browser = {
        name = "prohlizec" //IE, chorme nebo edge
        version = "verze_prohlizece" //pouze u chrome a edge
    }
    scenarios = [
"nazev_konfiguracniho_souboru_testovaciho_scenare" //např.: "TC.conf"
    ]
}
```

Soubor obsahuje jedno hlavní – conf a dvě vnitřní pole (objekty) – browser a scenarios. Název hlavního pole – conf se nesmí změnit, jelikož je definován v kódu překladače, takže jakákoliv změna by znamenala nefunkčnost překladače. Pomocí parametru browser se volí, ve kterém webovém prohlížeči (a jeho verzi) budou automatizované testy prováděny. Pole scenarios obsahuje jednotlivé automatizované testy (v uvozovkách), respektive jejich konfigurační soubory. Automatizované testy se provádí v pořadí, jak jsou zapsány za sebou.

Hlavní konfigurační soubor automatizovaného testu obsahuje jednotlivé části (soubory) testu. Struktura (šablona) hlavního konfiguračního souboru automatizovaného testu je následující:

TC.conf:

```
include "soubor1.conf"
include "soubor2.conf"

TC {
 defaultSleep = 2500
 actions = [
      soubor1,
      soubor2
 ]
}
```

Název souboru může být libovolný, ale je nutné zachovat stejný název jak pro soubor, tak v souboru označení hlavního pole (v ukázkovém příkladu souboru, název souboru TC.conf, vnitřní obsah souboru TC {}). Pro lepší přehlednost testu a údržbu je dobré si test rozdělit na několik částí (např. přesměrování, přihlášení, odhlášení apod.), aby veškeré akce testu nebyly obsažené v jednom konfiguračním souboru, ačkoliv je takovýto zápis také možný. Každý konfigurační soubor (část) automatizovaného testu je nutné přidat do hlavního konfiguračního souboru testu pomocí příkazu: include “cesta k souboru”. Pomocí parametru defaultSleep (je zadáván v milisekundách) je možné nastavit pauzu mezi akcemi. Parametr actions určuje, které a v jakém pořadí budou prováděny jednotlivé části testu. Jednotlivé akce se mezi sebou oddělují čárkou.

Konfigurační soubor (část) automatizovaného testu obsahuje nadefinované jednotlivé akce, které jsou v rámci automatizovaného testu prováděny. Struktura (šablona) konfiguračního souboru automatizovaného testu je následující:

soubor1.conf (příklad):

```
soubor1 {
actionsList = [
   nazevAkce1,
   nazevAkce2,
   ]
 actions {
        nazevAkce1 {
        }
        nazevAkce2 {
        }
 }
}
```

Název souboru je libovolný, ale stejně jako u hlavního konfiguračního souboru automatizovaného testu, musí být shodný s názvem hlavního pole souboru. Pomocí seznamu actionsList, podobně jako v hlavním konfiguračním souboru testu seznamu actions, jsou prováděny jednotlivé akce v pořadí, jak jdou za sebou. Jednotlivé akce se oddělují čárkou. Akce mohou mít libovolný název a mohou se libovolně opakovat v testu. Pokud není název akce v seznamu actionList, tato akce se neprovede, i přesto, že je definována.

4.2.4.3 Syntaxe akcí překladače

Obsah jednotlivých akcí (pole actions) je tvořený pomocí nadefinovaných klíčových slov, reprezentující danou akci (krok z testovacího scénáře). Jak již bylo vysvětleno v kapitole 4.2.4.1, veškerý zápis prvků se provádí zápisem XPath. Podle typu akce je zapotřebí různý počet klíčových slov a k nim požadované hodnoty uvedené v uvozovkách. Akce pro překladač byly průběžně definovány na základě vybrané aplikace pro ověření překladače a dalších aplikací, které byly testovány v rámci diplomní praxe. Akce lze rozdělit na:

- Ovládací akce
- Pomocné akce
- Kontrolní akce.

1) Ovládací akce

Pomocí ovládacích akcí lze aplikaci požadovaně ovládat. Jedná se o akce typu přesměrování na požadovanou url adresu, vyplnění textového pole, vybrání hodnoty ze selectboxu, zaškrtnutí checkboxu. kliknutí na tlačítko (odkaz) apod. V následující tabulce č. 6 jsou zapsány jednotlivé nadefinované ovládací akce:

#	Popis akce	HOCON syntaxe překladače

1	Vyplnění url adresy	nazevAkce {
		url = "url adresa“
		}

2	Přihlášení (Windows login)	nazevAkce {
		url = "url adresa“
		user = "login name"
		password = "heslo"
		}

3	Vyplnění textového pole (v html <input type="text" name="name_inputu_z_html"/>)	nazevAkce {
		input = "//input[@name='name_inputu_z_html']"
		value = "pozadovana hodnota"
		event = "event_z_hmtl" //nepovinný
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

4	Zvolení hodnoty ze selectu (v html <select name="name_selectu_z_html"></select>)	nazevAkce {
		select = "//select[@name='name_selectu_z_html']"
		value = "value z options uvnitr selectu"
		event = "event_z_hmtl" //není povinný
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

5	Zaškrtnutí checkboxu (v html <input type="checkbox" name="name_checkboxu_z_html"/>)	nazevAkce {
		checkbox = "//input[@name='name_checkboxu_z_html']"
		event = "event_z_hmtl" //není povinný
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

6	Zaškrtnutí radio buttonu (v html <input type="radio" id="id_radia_z_html"/>)	nazevAkce {
		radio = "//input[@id='id_radio_z_html']"
		event = "event_z_hmtl" //není povinný
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

7	Vyplnění textarei (v html <textarea name="name_textarei_z_html"></textarea>)	nazevAkce { textarea = "//textarea[@name='name_textarei_z_html']" value = "pozadovana hodnota" event = "event_z_hmtl" //není povinný

		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

8	Kliknutí na levé tlačítko myši	nazevAkce {
		button = "xpath_prvku"
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

9	Dvojklik tlačítka	nazevAkce {
		dblClick = "xpath_prvku"
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

10	Pravý klik na tlačítko (pravé tlačítko myši)	nazevAkce {
		rightClick = "xpath_prvku"
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

11	Kliknutí na CSS button	nazevAkce {
		CSSButton = "xpath_prvku"
		CSSField = "CSS prvek"
		CSSValue = "pozadovana hodnota"
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

12	Nahrání souboru	nazevAkce {
		fileName = "soubor z adresáře files"
		uploadElement = "//input[@name='name_z_html']"
		wdw = cislo //nepovinný, maximální doba čekání na prvek, standardně 15 sekund
		}

13	Odkliknuti dialogového okna prohlížeče	nazevAkce {
		alertAction = "dismiss" nebo "accept"
		}

Tabulka 6 – Ovládací akce překladače

2) Pomocné akce

Pomocné akce se využívají k hladšímu průběhu testu. Jsou to akce typu zapauzování běhu testu, čekání na požadovaný stav prvku, nebo překliknuti mezi okny. V následující tabulce č. 7 jsou zapsány jednotlivé nadefinované pomocné akce:

#	**Popis akce**	**HOCON syntaxe překladače**

1	Zapauzovaní běhu testu	nazevAkce {
		sleep = cislo v ms (1000ms = 1s)
		}

2	Překliknuti mezi okny	nazevAkce {
		sleep = 5000 - jak dlouho se ma pockat pred prekliknutim
		switchWindows = true
		}

3	Čekání na prvek	nazevAkce { waitElement = "xpath_prvku"

		wdw = cislo //maximalni cekani na nacteni prvku (300 = 5minut)
		waitTo = "akce na kterou se ceka" //invisible, visible, clickable
		}

4	WebDriver wait (společně s jakoukoliv akcí)	nazevAkce {
		button = "xpath_prvku"
		wdw = cislo //maximalni cekani na nacteni prvku (300 = 5minut)
		}

Tabulka 7 – Pomocné akce překladače

3) Kontrolní akce

Kontrolní akce jsou rozdělené na dva typy soft assert a hard assert. V následující tabulce č. 8 jsou zapsány jednotlivé nadefinované kontrolní akce:

#	Popis akce	HOCON syntaxe překladače (soft assert)	HOCON syntaxe překladače (hard assert)

1	Kontrola hodnot	nazevAkce {
		softAssertValuesList = [
		nazevKontroly1,
		nazevKontroly2
		]
		softAssertValues {
		nazevKontroly1 {
		input = ""xpath_prvku""
		expectedValue = "pozadovana hodnota"
		}
		nazevKontroly2 {
		input = ""xpath_prvku""
		expectedValue = "pozadovana hodnota"
		}
		}
		}
			nazevAkce {
			hardAsserValueList = [
			nazevKontroly1,
			nazevKontroly2
			]
			hardAssertValues {
			nazevKontroly1 {
			input = ""xpath_prvku""
			expectedValue = "pozadovana hodnota"
			}
			nazevKontroly2 {
			input = ""xpath_prvku""
			expectedValue = "pozadovana hodnota"
			}
			}
			}

2	Kontrola obrázku	nazevAkce {
		softImg {
		type = ["nazev_obrazku_z_html_1","nazev_obrazku_z_html_2"]
		element = "//xpath_prvku_u_ktereho_je_obrazek"
		}	

3	Kontrola dialogového okna	nazevAkce {
		softPopup = "nazev_popup"
		softPopupText = "text_dialogoveho_okna"
		}	

4	Kontrola tabulky	nazevAkce {
		softAssertTable = {
		element = "//table[@name='name_z_html'] "
		content = {
		1 = ["bunka1_1","bunka1_2","bunka1_3"] //null->bunka se nekontroluje
		2 = ["bunka2_1","bunka2_2","bunka2_3"]
		}
			nazevAkce {
			assertTable = {
			element = "//table[@name='name_z_html'] "
			content = {
			1 = ["bunka1_1","bunka1_2","bunka1_3"] //null->bunka se nekontroluje
			2 = ["bunka2_1","bunka2_2","bunka2_3"]
			}

5	Kontrola existence	nazevAkce {
		softExists = "xpath_prvku"
		}
			nazevAkce {
			exists = "xpath_prvku"
			}

6	Kontrola zobrazeni	nazevAkce {
		softDisplayed = "xpath_prvku"
		}
			nazevAkce {
			displayed = "xpath_prvku"
			}

7	Kontrola dostupnosti kliknutí	nazevAkce {
		softClickable = "xpath_prvku"
		}
			nazevAkce {
			clickable = " xpath_prvku"
			}

8	Obsahuje přesně daný výraz	nazevAkce {
		textElement = "pozadovany text"
		softExactText = "xpath_prvku"
		}
			nazevAkce {
			textElement = "pozadovany text"
			exactText = " xpath_prvku"
			}

9	Obsahuje někde hledaný výraz	nazevAkce {
		textElement = "pozadovany text"
		softContainsText = "xpath_prvku"
		}
			nazevAkce {
			textElement = "pozadovany text"
			containsText = " xpath_prvku"
			}

10	Kontrola CSS prvku	nazevAkce {
		softAssertCSS = "xpath_prvku"
		CSSField = "CSS prvek"
		CSSValue = "pozadovana hodnota"
		}
			nazevAkce {
			assertCSS = "xpath_prvku"
			CSSField = "CSS prvek"
			CSSValue = "pozadovana hodnota"
			}

Tabulka 8 – Kontrolní akce překladače

4.2.5 Průběh a výstup testování – logování

Celý průběh testování – logování je prováděno pomocí Log4j a je v průběhu testu zobrazeno v konzoli a následně uloženo do adresáře log. Konfigurace logu je obsažena v souboru log4j.properties (viz elektronické přílohy práce – prekladacProjekt/Selenium), v adresáři src/main/resource. Název log souboru je: Selenium_datum-čas spuštění testu.log. Každý záznam (řádek) logu obsahuje datum, čas a danou akci. První záznam

automatizovaného testu je název automatizovaného testu (název souboru) a poslední záznam „End of (nazev testu)“. V konzoli IntelliJ IDEA je výstup testování (TestNG) vždy stejný a to následovný:

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

Takovýto výsledek je způsobený na základě kódu překladače, kdy při spouštění více testů musí být výstup TestNG brán vždy jako 1 test, ačkoli je provedeno více testů.

4.3 Ověření překladače

Po navržení a vytvoření překladače bylo provedeno manuální a automatizované testování pomocí nástroje Selenium WebDriver v programovacím jazyku Java, včetně ověření vytvořeného překladače HOCON pro nástroj Selenium WebDriver. Veškeré testy byly prováděny na základě vytvořených testovacích scénářů. Jako první bylo prováděno manuální testování čtyřmi testery. Dále byly vytvořeny automatizované testy pro překladač HOCON a v programovacím jazyku, které byly následně spuštěny. V každé části testování (manuální, Java a HOCON) byl zaznamenáván průběh testu a doba testování pouze pozitivních testů. Zhodnocení jednotlivých testování bude provedeno v kapitole 5.

Překladač bude ověřen nad školní webovou aplikací, e-shopem pro konfiguraci hokejových dresů (viz obrázek č. 20). Aplikace je vyvíjena metodou CI/CD (agilní model) a předpokládá se, že vytvořené testovací případy budou prováděny před každým releasem verze webové aplikace.

Obrázek 20 – Vybraná aplikace pro ověření (testování)

4.3.1 Testovací případy

Nad vybranou webovou aplikací byly vytvořeny 3 testovací případy, podle kterých byla prováděna jednotlivá testování. Navržené testovací případy pokryvají základní funkcionality aplikace a všechny navržené případy jsou typem testu E2E. Zahrnují spuštění aplikace, přihlášení (registrace), práce v aplikaci a končí odhlášením (vypnutím) z aplikace. Testovací případy jsou navrženy tak, aby byly prováděny najednou, v pořadí, jak byly vytvořeny.

V následující tabulce č. 9 jsou shrnuty vytvořené testovací případy, s krátkým popisem daného testovacího případu a jeho pořadí v testování (číslo testovacího případu).

Číslo TC	Název	Popis
1	Registrace	Uživatel provádí registraci pro e-shop.
2	Přihlášení a úprava osobních údajů	Uživatel se přihlašuje do e-shopu a provádí editaci osobních údajů.
3	Konfigurace tréninkových dresů	Uživatel provádí konfiguraci tréninkové sady dresů.

Tabulka 9 –Testovací případy

Vytvořená šablona pro testovací případ obsahuje informace o:

- Názvu testovacího případu.
- Testovací soubory/data.
  - Zadávané vstupní hodnoty.
  - Očekávané výstupní hodnoty.
- Scénář testu (jednotlivé kroky).
  - Číslo kroku (pořadové).
  - Popis kroku.
  - Očekávaný výsledek.
  - Skutečný výsledek.
- Poznámka.

4.3.1.1 TC 1 – Registrace

Jako první testovací případ byl navržen scénář registrace (viz obrázek č. 21). Jedná se o základní funkcionalitu testované aplikace (e-shopu), bez které není možné vytvořit objednávku. Uživatel provádí registraci, po které je přihlášen pod právě vytvořeným účtem do aplikace (e-shopu). Následně je provedeno odhlášení z aplikace a možnost znovu přihlášení do aplikace (tlačítko PŘIHLÁSIT SE) je dostupné.

TC 1	**Registrace**		
Testovací soubory/data			
Zadávané vstupní hodnoty	Registrační údaje (* libovolný název) • Jméno * • Příjmení * • Email = přidělený email • Heslo = přidělené heslo	Očekávané výstupní hodnoty	Úspěšná registrace uživatele
Scénář testu			
Krok	Popis	Očekávaný výsledek	Skutečný výsledek
1.	Uživatel otevře adresu url: http://dresville.utipa.info/	Zobrazí se titulní stránka a tlačítko PŘIHLÁSIT SE je dostupné.	
2.	Uživatel klikne na tlačítko PŘIHLÁSIT SE.	Zobrazí se přihlašovací stránka a tlačítko PŘEPNOUT NA REGISTRACI je dostupné.	
3.	Uživatel klikne na tlačítko PŘEPNOUT NA REGISTRACI.	Zobrazí se registrační formulář.	

TC 1	**Registrace**		
4.	Uživatel vyplní údaje registračního formuláře: • Jméno, • Příjmení • Email, • Heslo.	Registrační údaje jsou vyplněny.	
5.	Uživatel klikne na tlačítko REGISTROVAT SE.	Úspěšná registrace a přihlášení do e-shopu. Zobrazí se hlavní stránka uživatelova profilu se zadaným emailem v registračním formuláři. Místo tlačítka PŘIHLÁSIT SE se zobrazí tlačítko s uživatelovým jménem a vedle něj tlačítko ODHLÁSIT SE.	
6.	Uživatel klikne na tlačítko ODHLÁSIT SE.	Úspěšné odhlášení z e-shopu, zobrazí se hlavní stránka a tlačítko PŘIHLÁSIT SE.	
Poznámka			

Obrázek 21 – TC 1

4.3.1.2 TC 2 – Přihlášení a úprava osobních údajů (nastavení účtu)

Na základě testovacího případu 1 byl navržen testovací případ pro přihlášení a úpravu osobních údajů právě zaregistrovaného uživatele (viz obrázek č. 22). Jedná se o vyplnění osobních údajů v sekci nastavení účtů a o následnou kontrolu uložení údajů. Většina údajů může mít libovolný název, kromě přihlašovacích údajů a IČO, u kterého se kontroluje jeho formát. Po vyplnění a uložení údajů je provedeno znovunačtení stránky a kontrola vkládaných údajů, zda se uložily. Scénář končí opět úspěšným odhlášením z aplikace.

TC 2	**Přihlášení a úprava osobních údajů (nastavení účtu)**		
Testovací soubory/data			
Zadávané vstupní hodnoty	Přihlašovací údaje • Email = přidělený email • Heslo = přidělené heslo Nastavení účtu (* libovolný název) • Jméno * • Příjmení * • Přihlašovací e-mail = tester1@tester.cz • Kontaktní telefon * • Ulice * • Město * • PSČ * • Název firmy * • IČO • DIČ * • Dodací údaje – Jméno * • Dodací údaje – Příjmení * • Dodací údaje – Ulice a číslo popisné * • Dodací údaje – Město * • Dodací údaje – PSČ *	Očekávané výstupní hodnoty	Přihlášení do e-shopu a editace osobních údajů (nastavení účtu).

Scénář testu

Krok	Popis	Očekávaný výsledek	
1.	Uživatel otevře adresu url: http://dresville.utipa.info/	Zobrazí se titulní stránka a tlačítko PŘIHLÁSIT SE je dostupné.	
2.	Uživatel klikne na tlačítko PŘIHLÁSIT SE.	Zobrazí se přihlašovací stránka a tlačítko PŘIHLÁSIT SE je dostupné.	
3.	Uživatel vyplní údaje: • Email • Heslo.	Údaje jsou vyplněny.	
4.	Uživatel klikne na tlačítko PŘIHLÁSIT SE.	Po úspěšném přihlášení do e-shopu se zobrazí hlavní stránka uživatelova profilu se zadaným emailem v levé části obrazovky. Místo tlačítka PŘIHLÁSIT SE se zobrazí tlačítko s uživatelovým jménem a vedle něj tlačítko ODHLÁSIT SE.	
5.	Uživatel klikne na NASTAVENÍ ÚČTU.	Zobrazí se formulář Nastavení účtu.	

TC 2	**Přihlášení a úprava osobních údajů (nastavení účtu)**		
6.	Uživatel vyplní údaje: • Jméno • Příjmení • Přihlašovací e-mail • Kontaktní telefon • Ulice • Město • PSČ • Název firmy • IČO • DIČ • Dodací údaje – Jméno • Dodací údaje – Příjmení • Dodací údaje – Ulice a číslo popisné • Dodací údaje – Město • Dodací údaje – PSČ.	Údaje jsou vyplněny.	
7.	Uživatel klikne na tlačítko ULOŽIT ZMĚNY.	Zobrazí se zelené okno s informační hláškou Změny uloženy.	
8.	Uživatel klikne v levém horním rohu na název stránky.	Zobrazí se titulní stránka a tlačítko s jménem uživatele je dostupné.	
9.	Uživatel klikne na tlačítko s jeho jménem.	Zobrazí se hlavní stránka uživatelova profilu, s jeho emailem v levé části obrazovky.	
10.	Uživatel klikne na NASTAVENÍ ÚČTU.	Zobrazí se formulář Nastavení účtu.	
11.	Uživatel zkontroluje, zda se zadávané údaje v kroku 6 uložily.	Vyplněné údaje z 6. kroku jsou uloženy.	
12.	Uživatel klikne na tlačítko ODHLÁSIT SE.	Úspěšné odhlášení z e-shopu, zobrazí se hlavní stránka a tlačítko PŘIHLÁSIT SE.	
Poznámka			

Obrázek 22 – TC 2

4.3.1.3 TC 3 – Konfigurace tréninkových dresů

Posledním testovacím případem je otestování jedné z hlavních funkcionalit aplikace. Jedná se o konfiguraci tréninkových dresů (viz obrázek č. 23). Scénář začíná přihlášením do aplikace účtem vytvořeným v 1. testovacím případu. Po přihlášení uživatel přejde do sekce konfigurace tréninkových dresů, kde je prvním krokem vyplnění názvu týmu, čímž se vytvoří projekt pod přihlášeným účtem. Dále uživatel provádí kontrolu dostupných barev a jejich zobrazování na dresu, včetně vybraného loga. Následně uživatel provede

kontrolu uložení a znovunačtení projektu, tam zkontroluje, zda se konfigurace uložila. Scénář končí opět úspěšným odhlášením z aplikace.

TC 3	**Konfigurace tréninkových dresů**		
Testovací soubory/data			
Zadávané vstupní hodnoty	Přihlašovací údaje • Email = přidělený email • Heslo = přidělené heslo Tréninkové dresy údaje (* libovolný název) • Název *	Očekávané výstupní hodnoty	Uložený právě vytvořený návrh (projekt) v sekci TVOJE NÁVRHY se zadaným názvem týmu

Scénář testu

Krok	Popis	Očekávaný výsledek	Skutečný výsledek
1.	Uživatel otevře adresu url: http://dresville.utipa.info/	Zobrazí se titulní stránka a tlačítko PŘIHLÁSIT SE je dostupné.	
2.	Uživatel klikne na tlačítko PŘIHLÁSIT SE.	Zobrazí se přihlašovací stránka a tlačítko PŘIHLÁSIT SE je dostupné.	
3.	Uživatel vyplní údaje: • Email • Heslo.	Údaje jsou vyplněny.	
4.	Uživatel klikne na tlačítko PŘIHLÁSIT SE.	Po úspěšném přihlášení do e-shopu se zobrazí hlavní stránka uživatelova profilu se zadaným emailem v levé části obrazovky. Místo tlačítka PŘIHLÁSIT SE se zobrazí tlačítko s uživatelovým jménem a vedle něj tlačítko ODHLÁSIT SE.	
5.	Uživatel klikne v levém horním rohu na název stránky.	Zobrazí se titulní stránka a možnost konfigurovat tréninkové dresy.	
6.	Uživatel klikne na tréninkové dresy.	Zobrazí se stránka s konfigurací tréninkových dresů. V levém menu se je rozbalena sekce 01 NÁZEV a pole JAK SI ŘÍKÁTE? je dostupné.	
7.	Uživatel vyplní název týmu v poli JAK SI ŘÍKÁTE?	Zadaný název týmu se zobrazí na hlavní stránce nad konfigurovaným tréninkovým dresem a tlačítko HOTOVO, DALŠÍ KROK je dostupné.	
8.	Uživatel klikne na tlačítko HOTOVO, DALŠÍ KROK.	V levém menu se rozbalí sekce 02 BARVY.	
9.	Uživatel klikne na tlačítko + PŘIDAT SADU.	Pod Sada 1 se zobrazí Sada 2.	

TC 3	**Konfigurace tréninkových dresů**		
10.	Uživatel klikne na checkbox Sada 2.	Sada 2 se smaže.	
11.	Uživatel klikne na kolečko s barvou u Sada 1.	Pod Sada 1 se rozbalí paleta barev.	
12.	Uživatel postupně zvolí veškeré barvy a kontroluje, zda se výběrem BARVY změnila barva dresu.	Barva dresu se mění dle aktuálního výběru BARVY.	
13.	Uživatel zvolí libovolnou barvu a klikne na tlačítko HOTOVO, DALŠÍ KROK.	V levém menu se rozbalí sekce 03 LOGO.	
14.	Uživatel klikne na tlačítko VYBRAT Z NAŠÍ KNIHOVNY LOG.	Otevře se stránka s logy.	
15.	Uživatel klikne na první logo (Prague Superboys).	Logo se zobrazí na dresu.	
16.	Uživatel klikne na tlačítko HOTOVO, DALŠÍ KROK.	V levém menu se rozbalí sekce 04 Doplňky.	
17.	Uživatel klikne na tlačítko s jeho jménem.	Zobrazí se hlavní stránka uživatelova profilu, s jeho emailem v levé části obrazovky, a právě vytvořený projekt v sekci TVOJE NÁVRHY.	
18.	Uživatel klikne v sekci TVOJE NÁVRHY na tlačítko PŘEJÍT K NÁVRHU u právě vytvořeného projektu (Název týmu = zadávaný název v kroku 6).	Načte se právě vytvořený projekt, dres vypadá stejně jako, když uživatel opouštěl konfiguraci v kroku 17.	
19.	Uživatel klikne na tlačítko ODHLÁSIT SE.	Úspěšné odhlášení z e-shopu, zobrazí se hlavní stránka a tlačítko PŘIHLÁSIT SE.	
Poznámka			

Obrázek 23 – TC 3

4.3.2 Manuální testování

Jako první bylo provedeno manuální testování aplikace pomocí vytvořených testovacích případů. Manuální testování aplikace bylo prováděno dvěma skupiny testerů. Skupina Google Chrome disponovala 2 testery a testování bylo prováděno na prohlížeči Google Chrome. Skupina Microsoft Edge disponovala též 2 testery a testování bylo prováděno na prohlížeči Microsoft Edge. Testování probíhalo najednou ve třech cyklech, vždy po dokončení všech testovacích případů. Každý tester byl seznámen jak s aplikací, tak s testovacími případy u prvního cyklu. V následující tabulce č.10 jsou zaznamenány

jednotlivé časy provedení testovacích případů, včetně celkového času cyklu a průměrné doby cyklu všech testerů v sekundách:

Výsledky manuálního testování (v sekundách)					
Prohlížeč	Tester 1	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
Google Chrome	TC 1	71,38	33,68	20,53	41,86
	TC 2	145,57	93,57	79,42	106,19
	TC 3	128,84	81,15	56,27	88,75
	Celková doba cyklu	345,79	208,4	156,22	236,8
	Tester 2	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
	TC 1	36,58	29,12	19,57	28,42
	TC 2	105,23	86,76	78,23	90,07
	TC 3	79,62	66,49	52,39	66,17
	Celková doba cyklu	221,43	182,37	150,19	184,7
	Průměrné časy za skupiny	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC za skupinu
	TC 1	53,98	31,40	20,05	35,14
	TC 2	125,40	90,17	78,83	98,13
	TC 3	104,23	73,82	54,33	77,46
	Celková průměrná doba cyklu za skupinu	283,6	195,4	153,2	210,73
Microsoft Edge	Tester 3	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
	TC 1	26,27	11,55	12,46	16,76
	TC 2	82,12	77,87	54,37	71,45
	TC 3	49,05	44,43	44,8	46,09
	Celková doba cyklu	157,44	133,85	111,63	134,3
	Tester 4	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
	TC 1	29,53	19,5	15,26	21,43
	TC 2	95,4	79,89	65,67	80,32
	TC 3	89,3	55,43	61,49	68,74
	Celková doba cyklu	214,23	154,82	142,42	170,5
	Průměrné časy za skupiny	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC za skupinu
	TC 1	27,90	15,53	13,86	19,10
	TC 2	88,76	78,88	60,02	75,89
	TC 3	69,18	49,93	53,15	57,42
	Celková průměrná doba cyklu za skupinu	185,84	144,34	127,03	152,40

Tabulka 10 – Výsledky manuálního testování

4.3.3 Automatizované testování – překladač HOCON

Na základě kapitoly 4.2.4 byly vytvořeny automatizované testy pomocí konfiguračních souborů pro vytvořený HOCON překladač Selenia. Při tvorbě testů bylo

využito objektového přístupu, ve smyslu konfiguračních souborů akcí, které byly naimportovány v hlavním konfiguračním souboru testovacího případu (viz elektronické přílohy práce – prekladacJAR/testConfig/DP/scenarios). Například konfigurační soubor openURL.conf (viz elektronické přílohy práce – prekladacJAR/testConfig/DP /scenariosAction) obsahuje akce redirect (otevření/přesměrování na požadovanou webovou stránku) a clickable (kontrola, zda je možné kliknout na požadované tlačítko). Tento konfigurační soubor byl naimportován do všech hlavních konfiguračních souborů automatizovaných testů, jako první akce automatizovaného testu. Dále se pokračovalo obdobně. Pokud to bylo možné, konfigurační soubory se vytvářely pro všechny automatizované testy. Nejkomplikovanější část vytváření testu byla kontrola barev u TC 3, kde kromě zápisu XPath bylo potřebné použít i CSS pro kontrolu barvy dresu. To se v syntaxi dané akce odrazilo na zadávání až 3 hodnot pro provedení akce a její kontrolu. Veškerý vytvořený kód automatizovaných testů překladače v jazyku HOCON lze najít v elektronických přílohách práce – prekladacJAR/testConfig/DP. Po vytvoření automatizovaných testů bylo testování prováděno na obou webových prohlížečích Google Chrome a Microsoft Edge. Testování probíhalo opět ve 3 cyklech. V následující tabulce č. 11 jsou zaznamenány časy: exekuce jednotlivých testů, celkový čas jednotlivých cyklů a průměrná doba všech cyklů.

Výsledky automatizovaného testování překladače HOCON (v sekundách)				
Google Chrome	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
TC 1	12,3	12,4	12,4	12,37
TC 2	17,8	17,7	17,5	17,67
TC 3	63	62	63	62,67
Celková doba cyklu	93,1	92,1	92,9	92,7
Microsoft Edge	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
TC 1	12,7	12,4	12,5	12,5
TC 2	19,3	19,2	19	19,17
TC 3	64	64	64	64
Celková doba cyklu	96	95,6	95,5	95,7

Tabulka 11 – Výsledky automatizovaného testování překladače HOCON

Celý průběh testu byl zaznamenáván do konzole a následně uložen do adresáře log. Logy, testování jednoho cyklu lze najít v elektronických přílohách práce – prekladacJAR/log.

4.3.4 Automatizované testování – Java

Po automatizovaném testování pomocí překladače bylo provedeno automatizované testování v programovacím jazyku Java. Testování bylo prováděno pomocí prostředí Intellij IDEA a na začátku bylo postupováno stejně jako u HOCON překladače. Nejdříve bylo potřeba vytvořit nový MAVEN projekt a nastavit v pom.xml (viz elektronické přílohy práce – javaSelenium) potřebné dependecies Selenium-java a TestNG. Po vytvoření MAVEN projektu byl vytvořen hlavní „spustitelný“ soubor (třída) MainPageTest.java (viz elektronické přílohy práce – javaSelenium/src/test /java), ve kterém byla použita anotace TestNG pro hladký průběh a výstup testů. Byly použity anotace: @BeforeMethod – nastavení prohlížeče (Google Chrome nebo Microsoft Edge) a WebDriverWait (čekání prohlížeče) na 60 sekund, @AfterMethod – zavírání prohlížeče po skončení testu a @Test – automatizovaný test (testovací případ) Java. Při tvorbě automatizovaných testů byly použity stejné funkce (instrukce) Selenia jako jsou použité u jednotlivých akcí překladače HOCON. Nebyl ale použit objektový přístup, takže veškeré instrukce všech testů byly obsaženy v hlavním souboru (MainPageTest.java). Tím je demonstrován nejlehčí možný způsob vytvoření automatizovaného testu v programovacím jazyku Java, pro srovnání s testy pro překladač HOCON. Kromě psaní jednotlivých akcí, jako otevření webové stránky, kliknutí na tlačítko apod., bylo potřeba psát i logy každého testu a jednotlivé podmínky kontrol. Veškerý vytvořený kód automatizovaného testu v programovacím jazyku Java lze najít v elektronických přílohách práce – javaSelenium. Testování bylo prováděnO stejně jako u manuálního testování ve 3 cyklech a bylo prováděno pomocí webových prohlížečích Google Chrome a Microsoft Edge. V následující tabulce č. 12 jsou zaznamenány časy: exekuce jednotlivých testů, celkový čas jednotlivých cyklů a průměrná doba všech cyklů.

Výsledky automatizované testování v Java (v sekundách)				
Google Chrome	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
TC 1	8,5	8,4	8,5	8,47
TC 2	14,7	14,5	14,3	14,50
TC 3	51,4	51,5	51,4	51,43
Celková doba cyklu	74,6	74,4	74,2	74,40
Microsoft Edge	Cyklus 1	Cyklus 2	Cyklus 3	Průměrná doba TC
TC 1	10,3	10,2	10,3	10,27
TC 2	16	15,8	16,1	15,97
TC 3	52,7	53,9	53,2	53,27
Celková doba cyklu	79	79,9	79,6	79,50

Tabulka 12 – Výsledky automatizovaného testování Java

Logy testovacích případů byly zapisovány pouze v konzoli Intellij IDEA, pomocí funkce System.out.println. Zkopírované logy testování v poznámkovém bloku lze najít v elektronických přílohách práce – javaSelenium/log.

V případě úspěšného provedení testu vypadal výsledek (testNG) v konzoli následovně:

```
===============================================
Default Suite
Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
===============================================


Process finished with exit code 0
```

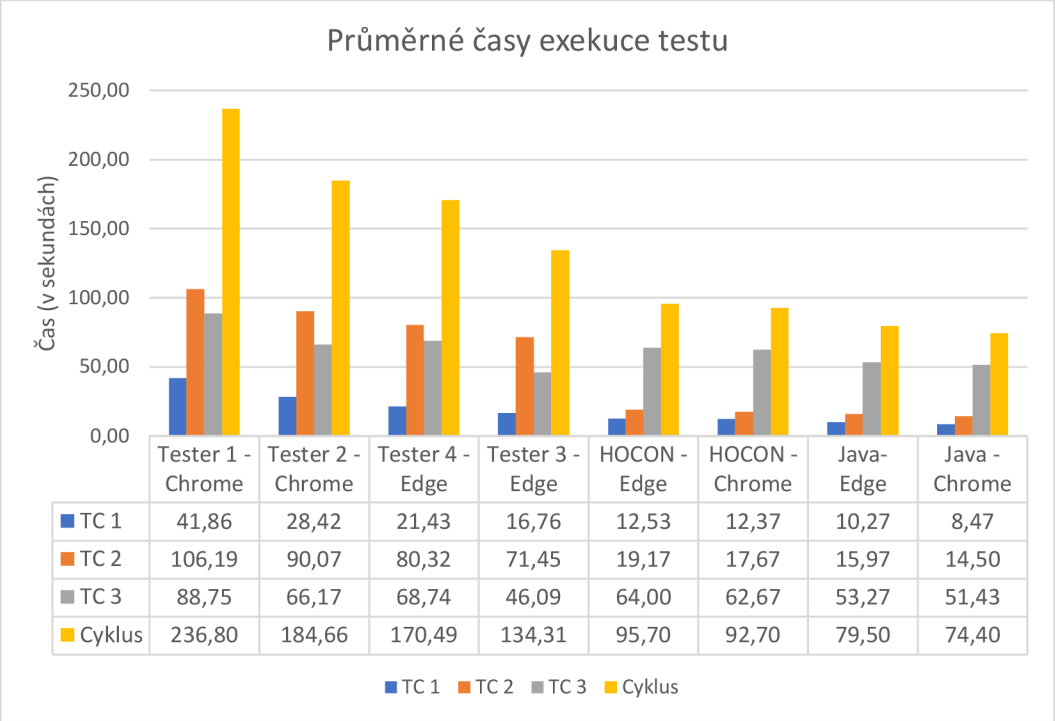
5 Výsledky a diskuse

Po úspěšném vytvoření HOCON překladače nástroje Selenium WebDriver pro automatizované testování webových aplikací a otestování aplikace všemi třemi uvedenými způsoby lze přistoupit k zhodnocení výsledků a porovnání výhod/nevýhod testování pomocí překladače na vybrané webové aplikaci. Zhodnocení je definováno v následujících aspektech:

- Rychlost testování
- Kód automatizovaného testu
- Ekonomika.

5.1 Zhodnocení rychlosti testování

Z hlediska rychlosti testování se budou porovnávat dosažené jednotlivé časy testování aplikace (na základě vytvořených TC) ve webových prohlížečích Google Chrome a Microsoft Edge pomocí manuálního testování (Tester), automatizovaného testování v Seleniu programovacím jazykem Java a automatizovaného testování pomocí navrženého HOCON překladače v programovacím jazyku Java. Veškeré časy jsou zprůměrované za všechny provedené cykly. Veškeré průměrné zaznamenané časy jsou shrnuty do následujícího grafu č. 1 s tabulkou:



	Tester 1 - Chrome	Tester 2 - Chrome	Tester 4 - Edge	Tester 3 - Edge	HOCON - Edge	HOCON - Chrome	Java- Edge	Java - Chrome
TC 1	41,86	28,42	21,43	16,76	12,53	12,37	10,27	8,47
TC 2	106,19	90,07	80,32	71,45	19,17	17,67	15,97	14,50
TC 3	88,75	66,17	68,74	46,09	64,00	62,67	53,27	51,43
Cyklus	236,80	184,66	170,49	134,31	95,70	92,70	79,50	74,40

Graf 1 – Průměrné časy exekuce testu

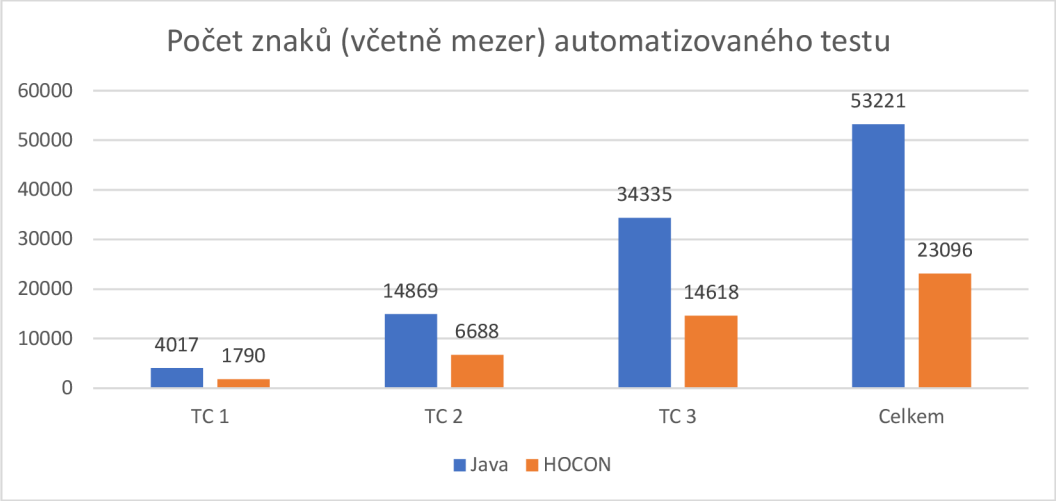
V celkovém srovnání automatizovaného vs. manuálního testování je na první pohled z grafu zřejmé, že nejlepší naměřené časy byly dosaženy pomocí automatizovaného testování, a to jak v samotném Seleniu, tak pomocí překladače HOCON. V celkovém průměru je automatizované testování o 2,13krát rychlejší než manuální testování. Byl tak potvrzen předpoklad, že manuální testování bude oproti automatizovanému delší. Pouze v případě TC 3 bylo manuální testování rychlejší než automatizované. To je ale způsobené speciálním požadavkem na otestování dvaceti čtyř dostupných barev dresů, kde při automatizovaném testování dochází pokaždé po volbě barvy ke kontrole HTML kódu stránky, zda zvolená barva odpovídá barvě dresu na webové stránce, zatímco při manuálním testování je samotným výběrem barvy dresu provedena i automatická kontrola lidským okem.

V automatizovaném testování je rychlejší testování v programovacím jazyku Java. Je to způsobené kódem HOCON překladače, který kromě provádění jednotlivých akcí musí nejdříve přeložit instrukce Seleniu. Celý kód automatizovaného testu je tedy reálně mnohem delší, než se uživateli zdá. Naměřené časy HOCON překladače Selenia jsou ale ve všech případech v průměru pouze o 1,25krát pomalejší, což není úplně špatný výsledek, co se týče rychlosti překladače.

Co se týče porovnání dosažených časů automatizovaného testování ve vybraných webových prohlížečích, nejlepšího času ve všech případech dosahuje prohlížeč Google Chrome, čímž potvrdil svoje postavení nejrychlejšího webového prohlížeče na aktuálním trhu. U manuálního testování nelze soudit, ve kterém prohlížeči bylo testování rychlejší, jelikož doba trvání testu je z největší části ovlivněna manuálním testerem a nikoli rychlostí prohlížeče.

5.2 Zhodnocení kódu automatizovaného testu

Zhodnocení kódu automatizovaného testu bylo provedeno porovnáním délky a přehlednosti kódů automatizovaných testů, kdy u HOCON překladače Selenia bylo nahlíženo pouze na kódy automatizovaných testů (konfigurační soubory). Následující graf č. 2 znázorňuje počet použitých znaků (včetně mezer) pro jednotlivé automatizované testy (testovací případy).



Graf 2 – Počet znaků (včetně mezer) automatizovaného testu

V celkovém součtu počtu znaků jsou testy vytvořené v HOCON syntaxi překladače 2,3krát kratší než v programovém jazyku Java. Nejenže je překladač HOCON kratší, ale také na první pohled přehlednější. Některé části vytvořeného kódu automatizovaného testu bylo možné využít pro všechny tři testovací případy, čímž bylo využito objektového přístupu, které, pokud ho chceme využít v programovacím jazyku Java, vyžaduje značné zkušenosti s programováním. Co se týče kódu automatizovaného testu v programovacím jazyku Java, je zapotřebí znát základy daného programovacího jazyka. V případě testů v překladači HOCON je syntaxe jazyka jasně dána, včetně akcí, které lze provést v rámci automatizovaného testování. Způsobem, jakým je překladač vyvinutý, je možné jednoduše přidávat další akce (metody) do již vytvořených tříd a tím zvyšovat komplexnost a použitelnost překladače pro všechny typy webových aplikací.

5.3 Zhodnocení ekonomické

Aby bylo možné provést určité ekonomické zhodnocení vytvořeného překladače, je zapotřebí zjistit, kolik stojí práce jednotlivých pracovníků. Odhad mezd byl stanoven podle průměrného platu aktuálních nabídek vybraných prací. Dále byla tato mzda přepočítána na hodiny za předpokladu standardní pracovní doby 160 hodin za měsíc. Výsledky jsou uvedeny v následující tabulce č.13:

Pozice	Hodinová sazba (kč)
Manuální tester	200
Automatizovaný tester	350
IT analytik	400

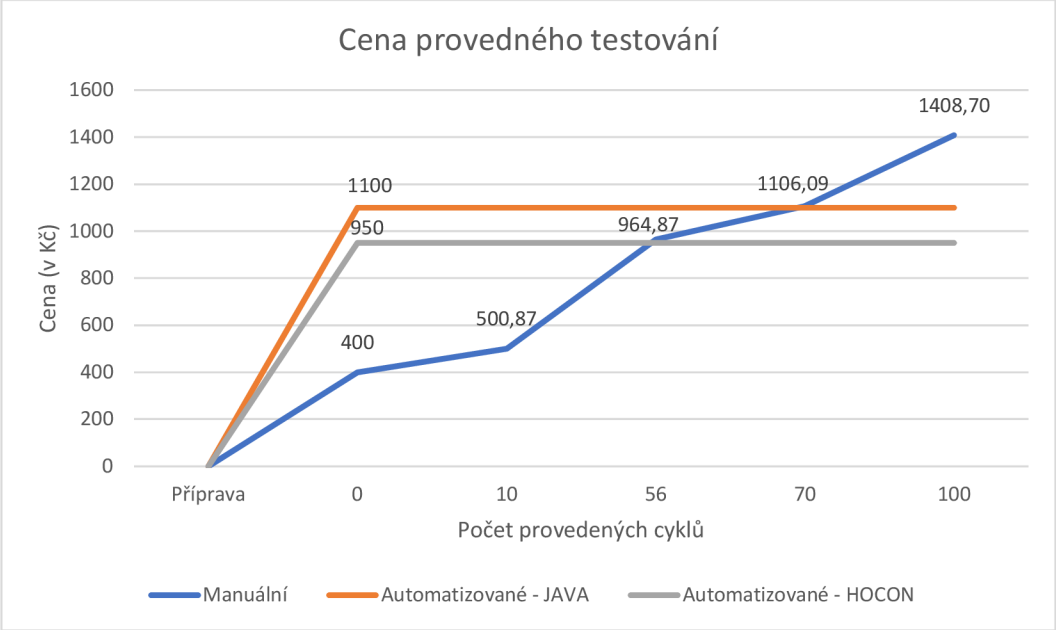
Tabulka 13 – Hodinová sazba vybrané pracovní pozice

Dále byly navržené podmínky pro jednotlivá testování, které jsou obsaženy v následující tabulce č. 14.

Manuální	**Automatizované – JAVA**	**Automatizované – HOCON**
1. Analytik napíše testovací případ – 1 hodina	1. Analytik napíše testovací případ – 1 hodina	1. Analytik napíše testovací případ – 1 hodina
2. Manuální tester otestuje aplikaci	2. Automatizovaný tester napíše a vyladí test v programovacím jazyku Java – 3 hodiny	2. Automatizovaný tester vyladí překladač pro danou aplikaci – 1 hodina
3. Aplikace se nemění	3. Aplikace se nemění	3. Manuální tester napíše automatizovaný test v HOCON – 2 hodiny
		4. Aplikace se nemění

Tabulka 14 – Podmínky ekonomického zhodnocení

Tyto podmínky byly konzultovány s lidmi, kteří pracují na stejných pozicích a provádí podobné, ne-li stejné úkony, aby byla co nejpřesněji demonstrována realita. Následně byla na základě stanovených cen a dosažených výsledků testování spočtena cena provedení jednoho manuálního cyklu na 7,46 Kč, při rychlost 19,83 cyklu/hod. Následující graf č. 3 znázorňuje cenu testování jednotlivých cyklů.



Graf 3 – Cena provedeného testování

Celkové náklady (příprava) pro automatizované testování vychází o něco málo lépe v případě překladače HOCON. Celková doba určena k přípravě (vytvoření) automatizované testu je v obou případech stejná, ale časy jednotlivých prací se liší. V případě automatizovaného testování HOCON je využití automatizovaného testera pouze třetinové, dochází tak k ulehčení jeho práce a možnost využití na jiných projektech. Dále je z grafu č. 3 zřejmé, že automatizované testování se vyplatí (v našem případě) až po provedení 56 cyklů (2 hodiny 49 minut) testování pro vyvinutý překladač, nebo 70 cyklů (3 hodiny 32 minut) v případě automatizovaného testování v programovacím jazyku Java.

6 Závěr

V této diplomové práci byla řešena problematika automatizovaného testování pomocí nástroje Selenium, zejména vytváření automatizovaných testů běžnými uživateli. Jako první byla zpracovaná teoretická část, ve které bylo čerpáno z dostupných knižních a internetových zdrojů. Na základě této části práce bylo získáno spoustu nových poznatků a informací, které byly dále využity v praktické části a při plnění dílčích cílů diplomové práce. V rámci teoretické části byl také řešen jeden z dílčích cílů práce: charakteristika jazyka HOCON a možnosti jeho využití, které bylo definováno v kapitole 3.5.

V praktické části byl nejprve vytvořen návrh překladače s potřebnými nástroji a zdůvodněno jejich zvolení. Při návrhu překladače byl kladen důraz na splnění dílčího cíle práce: zajištění implementace volby prohlížeče v překladači a výstupu testování do textového dokumentu. Volba webového prohlížeče byla splněna na základě hlavního konfiguračního souboru překladače, ve kterém uživatel provádí volbu WebDriveru webového prohlížeče a automatizovaných testů, které se mají provést. Další dílčí cíl práce a zároveň požadavek na překladač bylo zajištění výstupu do textového dokumentu. To bylo splněno pomocí vytvořeného logování Log4j, které je zobrazeno při průběhu automatizované testu a po ukončení testu uloženo do složky log. Po vytvoření překladače bylo možné přejít k jeho ověření, na základě stanoveného dílčího cíle porovnat s manuálním testováním a vytvářením testovacích scénářů (automatizovaných testů) přímo v jazyku Java. To bylo prováděno na vybrané webové aplikace, pro kterou byly vytvořeny tři testovací případy. Veškeré testování, jak manuální, tak automatizované pomocí vytvořeného překladače a klasickým způsobem v programovacím jazyku Java, bylo prováděno ve třech cyklech, kde jeden cyklus znamenal provedení všech vytvořených testovacích případů. Jako první bylo provedeno manuální testování, kde pomocí čtyř manuálních testerů bylo provedeno otestování webové aplikace na základě vytvořených testovacích případů. Během testování byl testerům měřen čas a výsledné časy byly zapsány do tabulky. Po manuálním otestování se přistoupilo již k automatizovanému testování, kde bylo nejprve provedeno automatizované testování pomocí překladače. Na základě vytvořeného popisu fungování a syntaxe překladače byly vytvořeny jednotlivé automatizované testy. Po jejich vytvoření bylo provedeno spuštění vytvořených automatizovaných testů a stejně jako u manuálního testování byl měřen čas ve třech

cyklech. Jako poslední bylo provedeno automatizované testování „klasickým“ způsobem, v programovacím jazyku Java. Automatizované testy v Java byly vytvořeny – odvozovány z vytvořených testů pro překladač, respektive u jednotlivých akcí byl použit překlad akce definované pro překladač v jazyce Java. Bylo tudíž použito stejných funkcí (instrukcí) Selenia jako u překladače, ale všechen kód byl napsaný v jednom souboru, což znamená velmi malou nebo velice komplikovanou možnost modifikaci testů. Zároveň logování bylo prováděno pouze do konzole. To vše bylo dáno z důvodu demonstrace nejsnazšího možného způsobu vytváření automatizovaných testů přímo v jazyce Java, aby bylo možné porovnat oba vybrané způsoby vytváření automatizovaných testů. Po veškerém úspěšném testování bylo možné přejít k zhodnocení výsledku. Překladač byl zhodnocen z hlediska rychlosti testování, kódu automatizovaného testu a z ekonomického hlediska. Z hlediska rychlosti testování byl potvrzen překlad rychlejší exekuce testu automatizovaného testování oproti manuálnímu. V celkovém průměru naměřených časů bylo automatizované testování 2,13krát rychlejší než manuální. V porovnání časů automatizovaného testování bylo rychlejší testování v Java, a to ve všech případech. Nicméně naměřené časy překladače jsou ve všech případech v průměru pouze o 1,25krát pomalejší. Zhodnocení kódu automatizovaného testu bylo provedeno sečtením veškerých znaků vytvořených automatizovaných testů, kde v celkovém součtu má překladač HOCON zápis 2,3krát kratší než v programovacím jazyku Java. Poslední zhodnocení se týkalo ekonomického hlediska (ceny). Výsledkem bylo, že automatizované testování se v obou případech vyplatí až od určitého počtu opakování. V případě vytvořených testovacích případů by se jednalo minimálně 56 cyklů, pokud se použije překladač, nebo 70 cyklů, pokud budou použity testy v Java.

Závěrem práce lze konstatovat, že hlavní cíl práce vytvořit a ověřit překladač nad testovacím nástrojem Selenium WebDriver tak, aby bylo umožněno vytvářet automatizované testy běžným uživatelům bez programátorských znalostí bylo úspěšně provedeno na vybrané aplikaci. Vytvořené automatizované testy na základě jazyka HOCON jsou intuitivní, takže i neznalý uživatel je schopen po krátké době automatizované testy vytvářet. Ideální scénář využití překladače je nechat vytvářet konfigurační soubory automatizovaných testů manuálním testerům a v případě potřeby nechat vyladit překladač zkušeným automatizovaným testerem pro danou webovou aplikaci.

7 Seznam použitých zdrojů

1. **Patton, Ron.** *Testování softwaru.* Praha : Computer Press, 2002. 80-7226-636-5.

2. **Help, Software Testing.** SDLC (Software Development Life Cycle) Phases, Process, Models. *Software Testing Help.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/#Recommended_Reading.

3. **Martin, Matthew.** SDLC (Software Development Life Cycle): What is, Phases & Models. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/software-development-life-cycle-tutorial.html#5.

4. **Trunkett, Oliver.** SDLC Methodologies: From Waterfall to Agile. *Virtasant.* [Online] 2020. [Citace: 22. Únor 2022.] Dostupné z: https://www.virtasant.com/blog/sdlc-methodologies.

5. **Roudenský, Petr a Havlíčková, Anna.** *Řízení kvality softwaru. Průvodce testováním.* Brno : Computer Press, 2013. 978-80-251-3816-8.

6. **25000, ISO.** ISO/IEC 25010. *ISO 25000.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

7. **Herout, Pavel.** *Testování pro programátory.* České Budějovice : Kopp, 2016. 978-80-8232-481-1.

8. **Hamilton, Thomas.** What is Software Testing? Definition, Basics & Types in Software Engineering. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/software-testing-introduction-importance.html#1.

9. **Egilmez, Ismail.** 4 Software Testing Roles. *Blog Thundra.* [Online] 2021. [Citace: 22. Únor 2022.] Dostupné z: https://blog.thundra.io/4-software-testing-roles.

10. **Hamilton, Thomas.** TEST PLAN: What is, How to Create (with Example). *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html.

11. **Bureš, Miroslav, a další.** *Efektivní testováni softwaru.* Praha : Grada Publishing, a.s., 2016. 978-80-247-5594-6.

12. **Yang, Chou.** Test Automation in 2020: Findings from the DevTestOps Landscape Report. *mabl.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.mabl.com/blog/test-automation-in-2020-findings-from-the-devtestops-landscape-report.

13. **Avasarala, Satya.** *Selenium WebDriver Practical Guide.* Birmingham : Pack Publishing Ltd., 2014. 978-1-78216-885-0.

14. **Gundecha, Unmesh.** *Selenium Testing Tools Cookbook.* Birmingham : Pack Publishing Ltd., 2012. 978-1-84951-574-0.

15. **Selenium.** WebDriver. *Selenium.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.selenium.dev/documentation/webdriver/.

16. **Kumari, Suman.** Introduction To Apache Maven | Automation Tool For Projects. *Medium.* [Online] 2019. [Citace: 22. Únor 2022.] Dostupné z: https://medium.com/gradeup/introduction-to-apache-maven-automation-tool-for-projects-ca1533730198.

17. **Smith, John.** TestNG Tutorial: What is Annotations & Framework in Selenium. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/all-about-testng-and-selenium.html.

18. **Rungta, Krishna.** Log4j with Selenium Tutorial: Download, Install, Use & Example. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/tutorial-on-log4j-and-logexpert-with-selenium.html.

19. **Waseem, Mohammad.** How to use Log4j in Selenium. *BrowserStack.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.browserstack.com/guide/log4j-in-selenium.

20. **Herout, Pavel.** *XSLT 2.0 a SVG prakticky.* České Budějovice : Kopp nakladatelství, 2010. 978-80-7232-406-4.

21. **Rungta, Krishna.** XPath in Selenium: How to Find & Write Text, Contains, OR, AND. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/xpath-selenium.html.

22. **W3schools.** Xpath axes. *W3schools.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.w3schools.com/xml/xpath_axes.asp.

23. **Selenium.** Install browser drivers. *Selenium.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.selenium.dev/documentation/webdriver/getting_started/install_drivers/.

24. **Rungta, Krishna.** JavaScriptExecutor in Selenium WebDriver with Example. *Guru99.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.guru99.com/execute-javascript-selenium-webdriver.html.

25. **Pujari, Chaitanya.** Assert and Verify Methods in Selenium. *BrowserStack.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.browserstack.com/guide/verify-and-assert-in-selenium.

26. **Help, Software Testing.** Assertions In Selenium Using Junit And TestNG Frameworks. *Software Testing Help.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.softwaretestinghelp.com/assertions-in-selenium/#2_assertTrue.

27. **JSON.** Introducing JSON. *JSON.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.json.org/json-en.html.

28. **Sponge.** Introduction to HOCON. *Sponge.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://docs.spongepowered.org/stable/en/server/getting-started/configuration/hocon.html.

29. **GitHub.** HOCON. *GitHub.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://github.com/lightbend/config/blob/main/HOCON.md.

30. **W3schools.io.** Hocon - Tutorial. *W3schools.io.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://www.w3schools.io/file/hocon-introduction/#advantages-of-hocon.

31. **Github.** Config. *Github.* [Online] 2022. [Citace: 22. Únor 2022.] Dostupné z: https://github.com/lightbend/config.

7.1 Seznam obrázků

7.2 Seznam tabulek

7.3 Seznam grafů

7.4 Seznam použitých zkratek

CI/CD	Continuous Integration/Continuous Delivery
DOM	Document Object Model
E2E	End-To-End
GUI	Graphical User Interface
HOCON	Human-Optimized Config Object Notation
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JAR	Java Archive
JDK	Java Development Kit
JSON	JavaScript Object Notation
QA	Quality Assurance
SDK	Software Development Kit
SDLC	Software Development Life Cycle
SDS	Software Design Specification
SRS	Software Requirement Specification
TC	Test Case
UAT	User Acceptance Testing
XML	Extensible Markup Language

Přílohy

Součástí diplomové práce je přiložené CD na zadní straně práce, obsahující elektronické přílohy práce v následujících adresářích:

- prekladacProjekt – kód překladače (java), včetně vytvořených konfiguračních souborů automatizovaných testů
- prekladacJAR – zkompilovaný balíček jar překladače pro samostatné spuštění, včetně vytvořených konfiguračních souborů
- javaSelenium – kód automatizovaných testů v programovacím jazyce Java