Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Automatizace testování SW

Pavel Diviš

© 2023 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Pavel Diviš

Informatika

Název práce

Automatizace testování SW

Název anglicky

Software testing automation

Cíle práce

Cílem této práce je navržení a popsání ideálního projektového toolchainu, který je potřeba pro automatizaci testování SW.

Metodika

Teoretická část této práce bude založena na studiu odborných informačních zdrojů zabývajících se testováním softwaru a jeho automatizací. Následovat bude podrobnější popis problematiky testování a jeho využití a popis postupů a metod, které se při testování využívají.

Praktická část spočívá v návrhu ideálního projektového toolchainu, který se dá využít při automatickém testování SW. Poznatky získané během zpracování práce budou shrnuty a zhodnoceny.

Doporučený rozsah práce

35-40 stran

Klíčová slova

Automatizace SW, testování SW

Doporučené zdroje informací

Agile Testing: A Practical Guide for Testers and Agile Teams 1st Edition, Janet Gregory, Lisa Crispin
Efektivní testování softwaru, Bureš Miroslav, Renda Miroslav, Doležel Michal a kolektiv
More Agile Testing: Learning Journeys for the Whole Team, Janet Gregory, Lisa Crispin
Testování softwaru řízené návrhem, Matt Stephens, Doug Rosenberg

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE
1906

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 11. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 15. 03. 2023

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Automatizace testování SW" jsem vypracoval samostatně pod vedením vedoucího bakalářské 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 bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.03.2023 ____________________________

Poděkování

Rád bych touto cestou poděkoval panu Ing. Jiřímu Brožkovi, Ph.D. za ochotu a trpělivost při vedení této bakalářské práce.

Automatizace testování SW

Abstrakt

Tato bakalářská práce se zabývá problematikou vývoje a testování softwaru a procesem automatizace testování.

V teoretické části popisuje a definuje základní principy testování, modely, které se v testování využívají a také jednotlivé testovací úrovně, které se při testování softwaru využívají. Teoretická část zároveň popisuje výhody a nevýhody automatizace testování a nástroje, které se pro automatizaci využívají.

Praktická část se poté zabývá navržením a analýzou automatizovaného testovacího prostředí ve firmě. Prvním dílčím cílem je nastavení automatizovaného testovacího prostředí s využitím nástrojů Jenkins a SVN. Hlavním cílem práce je poté analýza časové efektivity automatizovaného testování ve srovnání s manuálním testováním. Ze zjištěných poznatků a informací jsou poté vyvozeny závěry práce.

Klíčová slova: testování, automatizace, manuální testování, software, Jenkins, SVN, C++, ISTQB, testovací scénář, vývoj softwaru, CI/CD

Software testing automation

Abstract

This bachelor thesis deals with software development and testing and the process of test automation.

In the theoretical part, it describes and defines the basic principles of testing, the models used in testing and the different test levels used in software testing. The theoretical part also describes the advantages and disadvantages of test automation and the tools that are used for automation.

The practical part then deals with the analysis of an automated testing environment in a company. The first sub-objective is to set up an automated testing environment using Jenkins and SVN tools. The main objective of the work is then to analyze the time efficiency of automated testing compared to manual testing. The findings and information are then used to draw conclusions of the thesis.

Keywords: testing, automatization, manual testing, software, Jenkins, SVN, C++, ISTQB, test script, software development, CI/CD

Obsah

1 Úvod **10**

2 Cíl práce a metodika **11**

2.1 Cíl práce 11

2.2 Metodika 11

3 Teoretická východiska **12**

3.1 Testování softwaru 12

3.2 Modely životního cyklu vývoje 14

3.2.1 Vodopádový model (Waterfall) 14

3.2.2 V-Model 14

3.2.3 Agilní model 15

3.3 Techniky testování 17

3.3.1 Statická analýza kódu 17

3.3.2 Testování bílé skříňky 18

3.3.3 Testování černé skříňky 18

3.3.4 Testování šedé skříňky 19

3.4 Standardy a certifikace 20

3.4.1 International Software Testing Qualifications Board 20

3.4.1.1 Principy testování podle ISTQB 20

3.4.2 ASPICE 21

3.4.3 ISO/IEC 29119 21

3.5 Automatizace testování 23

3.5.1 Výhody automatizace testování 24

3.5.2 Nevýhody automatizace testování 24

3.6 Úrovně testování 26

3.6.1 Jednotkové (unit) testy 26

3.6.2 Modulové testy 26

3.6.3 Smoke testy 26

3.6.4 Softwarové integrační testy 27

3.6.5 Systémové testy 27

3.6.6 Akceptační testy 27

3.7 DevOps prostředí 28

3.7.1 Průběžná integrace 29

3.7.2 Průběžné dodávání 29

3.8 Nástroje používané v CI/CD 30

3.8.1 SVN 31

3.8.2 Git 31

3.8.3 Jenkins 32

3.8.4 GitLab CI/CD ... 33

4 Vlastní práce ... 34

4.1 Plánování testovací strategie ... 34

4.1.1 Požadavky ... 34

4.1.2 Testovací úrovně ... 34

4.2 Použité nástroje a technologie ... 37

4.2.1 Nástroje pro automatizaci ... 37

4.2.2 Nástroje pro testování ... 39

4.3 Časová analýza automatického a manuálního testování ... 41

4.4 Problémy při automatizovaném běhu testů ... 44

4.4.1 Vypršení licencí potřebných pro běh testů ... 44

4.4.2 Nedostupnost Jenkins „nodes“ ... 44

5 Výsledky a diskuse ... 45

6 Závěr ... 47

7 Seznam použitých zdrojů ... 48

8 Seznam obrázků, tabulek, grafů a zkratek ... 51

8.1 Seznam obrázků ... 51

8.2 Seznam tabulek ... 51

8.3 Seznam grafů ... 51

1 Úvod

Testování je v dnešní době nepostradatelnou součástí procesu, který má za cíl vyvinout kvalitní softwarový produkt. Kvalita vyvinutého softwarového produktu je často ta vlastnost, která rozhodne o tom, zda bude softwarový produkt konkurenceschopný či nikoliv. Koncoví uživatelé softwarů bývají čím dál tím více nároční a mívají daleko větší očekávání než kdy dříve.

Cílem testování je zajištění požadované kvality a odhalení co největšího množství defektů. Podcenění testování při vývoji softwaru může mít za následek nejen neúspěch na trhu, ale také značné zvýšení nákladů na samotný vývoj. Včasné odhalení defektů má za následek urychlení vývoje, ale také snížení dodatečných nákladů na údržbu po vydání softwaru do produkce.

S rostoucí složitostí aplikací a potřebou častých aktualizací se nicméně stává manuální testování časově náročným a náchylným k chybám. Automatizace testování softwaru se tímto stává stále důležitějším aspektem vývoje moderních softwarových produktů. Automatizace testování může výrazně snížit náklady a zlepšit kvalitu softwaru tím, že umožní rychlejší, spolehlivější a opakovatelnější testování.

Cílem této bakalářské práce je poskytnout přehled o automatizaci testování softwaru, včetně metodiky a nástrojů, které jsou k dispozici pro automatizaci různých typů testů. Práce se také zaměřuje na výhody a nevýhody automatizace testování a na faktory, které je třeba zvážit při rozhodování o tom, zda je vhodné automatizovat testování v konkrétním případě.

Praktická část si klade za cíl rozvinutí autoamatizovaného testování, definování nových testů a následné vyhodnocení, zda se automatizace testování skutečně vyplatí ve srovnání s manuálním testováním.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je navržení, popsání a zanalyzování ideálního projektového toolchainu, který je potřeba pro automatizaci testování SW.

2.2 Metodika

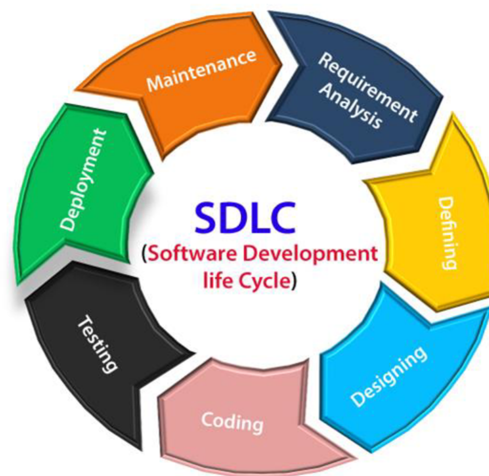
Teoretická část této práce bude založena na studiu odborných informačních zdrojů zabývajících se testováním softwaru a jeho automatizací. Následovat bude podrobnější popis problematiky testování a jeho využití a popis postupů a metod, které se při testování využívají.

Praktická část spočívá v návrhu ideálního projektového toolchainu, který se dá využít při automatickém testování SW. Poznatky získané během zpracování práce budou shrnuty a zhodnoceny.

3 Teoretická východiska

3.1 Testování softwaru

Testování softwaru je proces, při kterém dochází k odhalování defektů v programech, které vyvíjí vývojářský tým. Testování poskytuje informace o kvalitě a správnosti softwaru všem zainteresovaným stranám na projektu. Základním smyslem testování je hledání úspěchu (pozitivní test) a hledání neúspěchu (negativní test). Testovací tým musí ověřit, že zákazníkem požadované funkce fungují tak, jak mají a zároveň, že zákazníkem nežádoucí funkce nejsou součástí softwaru. [1]



Obrázek 1 - Software Development Life Cycle [35]

Všechny fáze vývoje softwaru přehledně popisuje tzv. životní cyklus vývoje softwaru (z angl. „Software Development Life Cycle“). Software Development Life Cycle je strukturovaný proces, který popisuje posloupnost kroků, které jsou spojeny s vývojem softwaru a který umožňuje vytváření kvalitních a levných softwarových produktů v co nejrychlejším čase. Každá fáze tohoto cyklu generuje výstupní data, která jsou použita jako vstupní data pro fázi, která bezprostředně následuje. Moderní SDLC strategie však nejsou striktně lineární a nastávají situace, kdy se projektový tým musí v SDLC vracet o pár kroků zpět, aby provedl opravy či vylepšení.

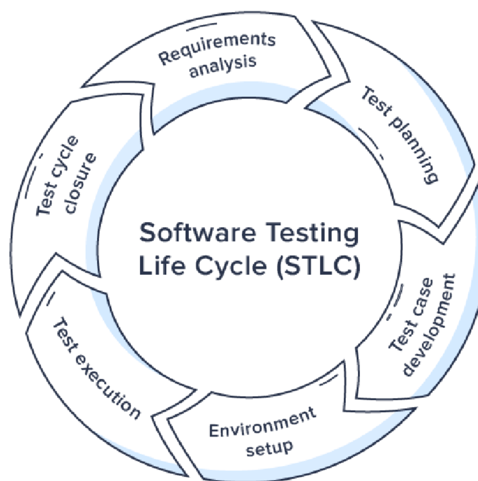
Testování je pouze jednou fází tohoto cyklu a jako výstupní data z této fáze by měl sloužit otestovaný softwarový produkt, který je připraven k nasazení do produkčního prostředí. Testování poskytuje informace o kvalitě softwaru a tyto informace slouží projektovému týmu ke správnému rozhodování a plánování následujících kroků. Testovací fáze je ukončena v momentu, kdy software splňuje kvalitativní požadavky, které byly nadefinovány

v předchozích fázích SDLC. Ke zvýšení a udržení kvality softwaru je potřeba správné, rychlé a včasné testování. [2]

Fázi testování podrobněji popisuje tzv. životní cyklus testování softwaru (z angl. „Software Testing Life Cycle“). Software Testing Life Cycle je strukturovaný proces, který popisuje posloupnost specifických kroků a akcí, které jsou prováděny v průběhu testovacího procesu. Tato posloupnost kroků a akcí napomáhá projektovému týmu ke splnění cílů kvality softwarového produktu.

STLC obsahuje šest fází: analýza požadavků, plánování, vývoj testovacích scénářů, nastavení testovacího prostředí, spuštění testů a reportování výsledků testů. Tyto fáze mohou být v průběhu vývoje softwaru několikrát zopakovány a to až do té doby, kdy je software připraven k nasazení do produkčního prostředí.

STLC a SDLC spolu úzce souvisejí, ale jejich cíle jsou rozdílné. Hlavní rozdíl spočívá v tom, že SDLC je zodpovědné za sběr požadavků a vytvoření vývojového plánu a STLC je zodpovědné za tvorbu testovacích scénářů a za ověření, že všechny naimplementované funkce splňují tyto požadavky. Oba tyto cykly vyžadují vzájemnou spolupráci a komunikaci členů projektového týmu. [3]



Obrázek 2 - Software Testing Life Cycle [36]

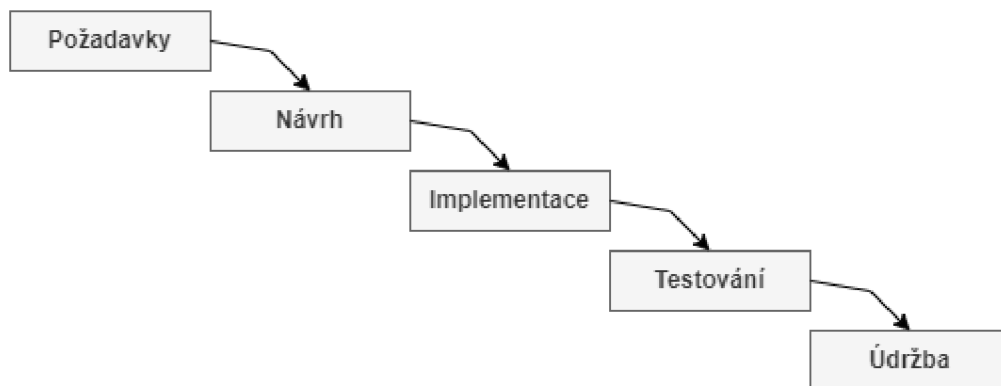
3.2 Modely životního cyklu vývoje

3.2.1 Vodopádový model (Waterfall)

Vodopádový model byl první představený model pro vývoj softwaru. Je označován jako lineárně sekvenční model. To znamená, že jakákoliv fáze vývojového procesu začíná pouze tehdy, je-li předchozí fáze dokončena. Ve vodopádovém modelu se fáze nepřekrývají.

To má za důsledek to, že k testování dochází až po dokočení všech předcházejících vývojových aktivit. Ve vodopádovém modelu neexistuje možnost jakékoliv iterace či revize předchozí fáze. Není tedy možné reagovat na dodatečné změny, které nastaly během dalšího vývoje. V případě nesprávně zadaných požadavků či nesprávného návrhu je nutno celý proces zrušit a opakovat ho znovu od začátku.

Chybějící flexibilita tohoto modelu je předmětem kritiky. V praxi je téměř nemožné dokončit jednu fázi a zahájit další, bez toho aniž by bylo potřeba se k ní v budoucnu vrátit. Původní podoba vodopádového modelu se v praxi již prakticky nevyužívá. [4]

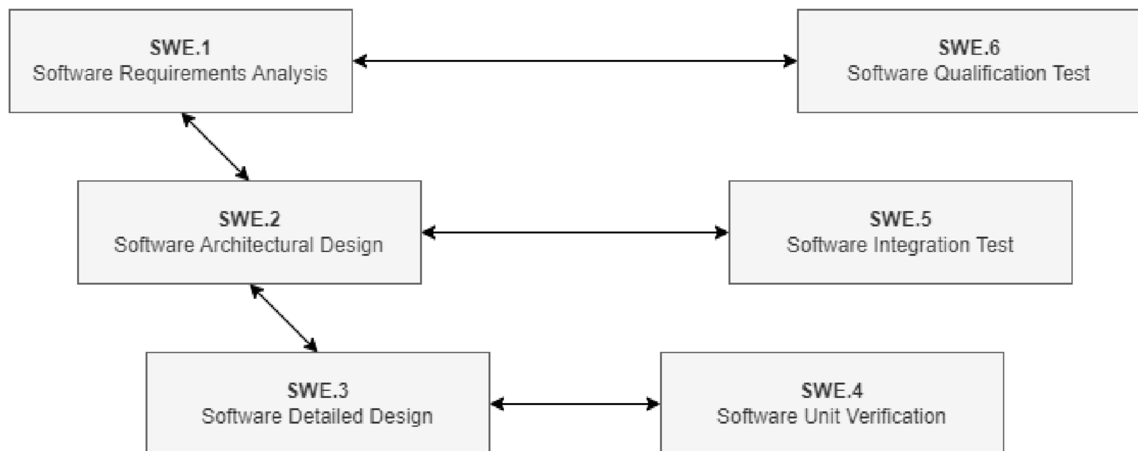


Obrázek 3 - Vodopádový model [37]

3.2.2 V-Model

V-Model je sekvenční SDLC model, který je vyzobrazen do písmena V. Je také označován jako verifikační a validační model. V-Model vychází z vodopádového modelu a představuje jeho rozšířenou verzi, kde ke každé aktivitě na levé straně existuje aktivita na jeho pravé straně. To znamená, že pro každou fázi životního cyklu vývoje softwaru existuje příslušná testovací úroveň. Levá část tohoto modelu představuje činnosti spojené se specifikací požadavků a samotným návrhem a pravá strana představuje již zmíněné testovací úrovně. Toto jednoznačné rozdělení umožňuje jednodušše určit jaké úrovně testování budou potřeba.

Výhodou V-Modelu je, že je velmi jednoduchý a je snadné mu porozumět a aplikovat ho v praxi. Jednoduchost tohoto modelu také znamená, že je snadné vést projekt, která využívá tento model. Nevýhodou je, že stále nenabízí potřebnou flexibilitu, kterou v dnešní době komplexní softwarové produkty vyžadují. Využití V-Modelu je tedy omezené na projekty, kde jsou zákaznické požadavky důkladně a správně definovány a na projekty s krátkou dobou trvání. [5]



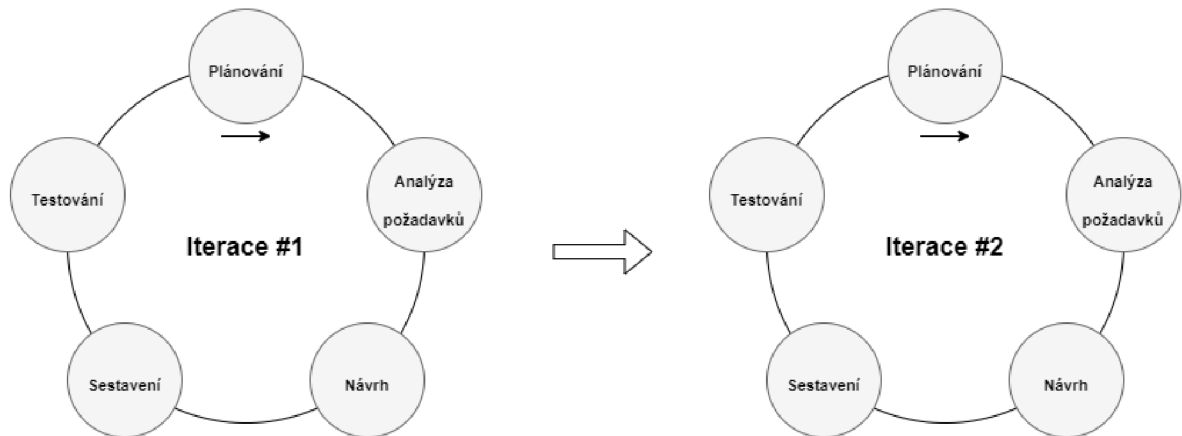
Obrázek 4 - Software Engineering V-Model [38]

3.2.3 Agilní model

Agilní SDLC model je kombinací iterativního a inkrementálního procesního modelu, který má za cíl rychlé dodávky funkčního softwarového produktu. Agilní metody rozdělují softwarový produkt na menší části, které jsou dodávány zákazníkovi v kratších časových intervalech. Tyto intervaly obvykle trvají od jednoho až ke třem týdnům. Určení délky tohoto intervalu ale může být individuální v závislosti na tom v jaké fázi se celý projekt nachází. Na začátku každého intervalu musí proběhnout plánování, následuje analýza požadavků, návrh, implementace, sestavení a testování. Na konci každého intervalu je softwarový produkt prezentován zákazníkovi, který má možnost sdělit projektovému týmu zpětnou vazbu na dodaný software. Poté se celý tento interval opakuje.

Agilní model je v dnešní době velice oblíbený a využívaný, především díky své vysoké přizpůsobovosti a flexibilitě. Nejoblíbenějšími agilními metodami jsou Rational Unified Process, Scrum, Crystal Clear, Extreme Programming, Adaptive Software Development, Feature Driven Development a Dynamic Systems Development Method. Tyto metody jsou blíže popsány v příručce s názvem Agile Manifesto, která byla vydána v roce 2001.

Nevýhodou agilního modelu je vyšší náročnost na vedení projektu v tomto prostředí. Agilní model je velmi závislý na interakci se samotným zákazníkem, takže pokud zákazník nemá jasno v tom, co přesně očekává, tak hrozí, že projekt může být veden špatným směrem. Další nevýhodou je přísnější řízení dodávek softwaru. Zákazník po skončení každé iterace očekává dodaný software s předem dohodnutou fukcionalitou, takže pokud se v průběhu iterace objeví problém a vývoj se zpozdí, je nutno tuto změnu v plánování vykomunikovat se zákazníkem. [6]



Obrázek 5 - Agilní model [39]

3.3 Techniky testování

Techniky testování se obecně rozdělují podle účelu použití. Existují statické a dynamické techniky. Statické techniky testování nepotřebují funkční software a využívají se spíše v počátcích fáze vývoje softwaru. Mezi statické techniky testování se řadí například statická analýza kódu či přezkoumání strategických dokumentů.

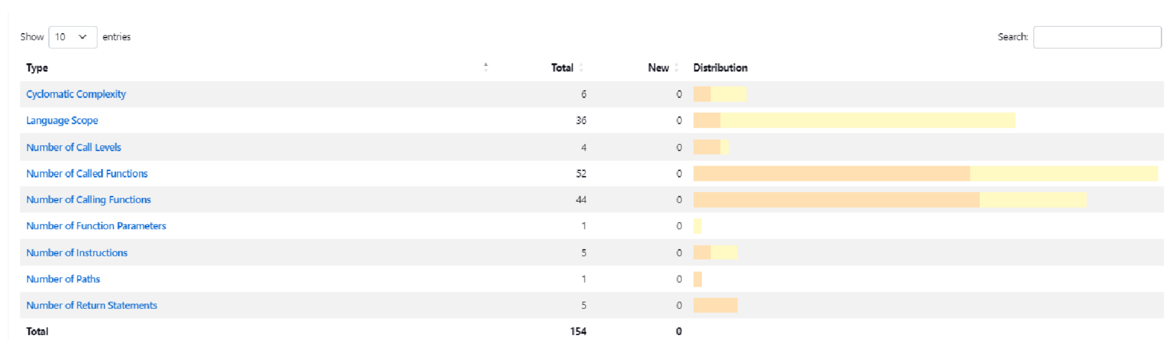
Dynamické techniky testování vyžadují funkční software a dělí se na základě znalosti zdrojového kódu. Pokud je potřeba znalost zdrojového kódu, tak se jedná o testování bílé skříňky. Pokud není potřeba znalost zdrojového kódu, tak se jedná o testování černé skříňky. Existuje také testování šedé skříňky, kde je potřeba pouze omezená znalost zdrojového kódu. Pochopení rozdílu mezi těmito technikami je důležité, protože se k oběma technikám přistupuje rozdílně. Rozdílem také je kdo je za implementaci testů zodpovědný. [7]

3.3.1 Statická analýza kódu

Statická analýza kódu je činnost, která má za cíl analýzu a ověření zdrojového kódu. Statická analýza analyzuje kvalitu, spolehlivost a zabezpečení zdrojového kódu, aniž by bylo potřeba kód spustit. Pomocí statické analýzy se identifikují defekty a slabá místa zabezpečení v kódu, která mohou ohrozit bezpečnost, stabilitu a zabezpečení softwaru.

Statická analýza kódu doplňuje dynamické techniky testování a nabízí několik výhod. Jednou z nich je soulad se standardy programování. Statická analýza kódu ověřuje, zda zdrojový kód vyhovuje standardům programování, jako je například MISRA C++ nebo HIS.

Existuje několik nástrojů, která umožňují statickou analýzu kódu. Mezi tyto nástroje patří například Polyspace či CodeSonar. [8]



Show 10 entries Search:

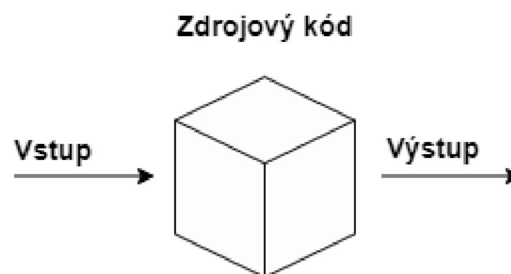
Type	Total	New	Distribution
Cyclomatic Complexity	6	0	
Language Scope	36	0	
Number of Call Levels	4	0	
Number of Called Functions	52	0	
Number of Calling Functions	44	0	
Number of Function Parameters	1	0	
Number of Instructions	5	0	
Number of Paths	1	0	
Number of Return Statements	5	0	
Total	154	0	

Obrázek 6 - HIS Findings report [40]

3.3.2 Testování bílé skříňky

Testování bílé skříňky je testovací technika, která testuje vnitřní struktru softwarové komponenty a má za cíl ověření její použitelnosti a správnosti. K testování bílé skříňky je potřeba přístup ke zdrojovému kódu a znalost jeho chování. Testování bílé skříňky se označuje také jako „code-based“ testování nebo „structure-based“ testování. Z tohoto důvodu jsou testy pro bílou skříňku implementovány vývojáři, kteří tím získávají zpětnou vazbu na jejich vlastní implementaci kódu.

Testování bílé skříňky může být využito na úrovni systémových testů, integračních testů či jednotkových (unit) testů. Dalším cílem testování bílé skříňky je ověření všech rozhodovacích větví, cyklů a příkazů v kódu. K tomuto ověření se využívá dvou metod: pokrytí příkazů (z angl. „Statement Coverage“) a pokrytí větví (z angl. „Branch coverage“). Využití těchto metod má za následek optimalizaci kódu a objevení defektů v ranné fázi vývoje. Nevýhodou testování bílé skříňky je, že jsou testy implementovány těmi stejnými vývojáři, kteří pracovali na implementaci zdrojového kódu. To může mít v některých případech za následek nedostatečnou kvalitu těchto testů. [9]

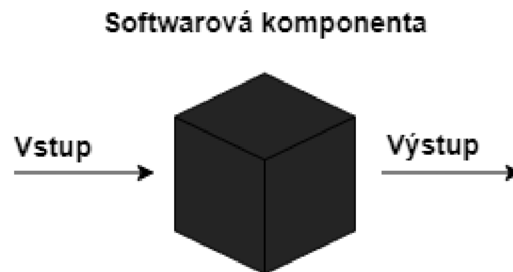


Obrázek 7 - Testování bílé skříňky [41]

3.3.3 Testování černé skříňky

Testování černé skříňky je definováno jako testovací technika, při které tester testuje vnější strukturu softwarové komponenty na základě specifikace. K testování černé skříňky není zapotřebí znalost zdrojového kódu, protože tyto testy berou v potaz pouze celkové vnější chování softwarové komponenty. Tím je simulován uživatelský pohled. [10]

Testování černé skříňky je též označováno jako „specification-based“ testování, kdy tester na základě vstupních dat očekává specifická výstupní data. ISTQB definuje pro černou skříňku tyto techniky testování: rozdělení tříd ekvivalence, analýza hraničních hodnot, testování dle rozhodovací tabulky, testování přechodů stavů, testování případů užití. [11]



Obrázek 8 - Testování černé skříňky [42]

3.3.4 Testování šedé skříňky

Testování šedé skříňky je využíváno především pro testování webových aplikací, kdy je tato technika přínosná při integračním testování, penetračním testování a doménovém testování. Hlavním cílem této testovací techniky je najít problémy, které vznikly v důsledku nesprávně naimplementované vnitřní struktury či v důsledku nesprávného použití samotné aplikace.

Při testování šedé skříňky tester nedisponuje kompletní znalostí zdrojového kódu, nicméně disponuje alespoň dílčími informacemi, jak vypadá a funguje vnitřní struktura softwarové komponenty či aplikace. [12]

3.4 Standardy a certifikace

3.4.1 International Software Testing Qualifications Board

International Software Testing Qualifications Board (zkráceně ISTQB) je certifikační mezinárodní rada pro testování softwaru. Terminologie ISTQB je mnohými uznávána jako hlavní terminologie v oblasti testování softwaru a spojuje mnoho lidí a společností po celém světě. ISTQB nabízí mnoho úrovní certifikací a jejich certifikace bývají často uznávány a vyžadovány velkými společnostmi zabývajícími se testováním softwaru. [13]

3.4.1.1 Principy testování podle ISTQB

Proces testování existuje již několik desítek let a díky tomu bylo zformulováno mnoho principů, které poskytují společný návod pro všechny druhy testování. Mezi tyto principy patří například:

- testování ukazuje přítomnost defektů, nikoli jejich nepřítomnost
- kompletní testování není možné
- včasné testování šetří čas a peníze

Pointou prvního principu je, že proces testování ukazuje přítomnost defektů, ale nedokazuje jejich nepřítomnost. Tato informace je velice důležitá a je potřeba jí míti na paměti po celou dobu běhu procesu testování. Ani kompletně otestovaný software, ve kterém nebyly nalezeny žádné defekty, nelze se stoprocentní jistotou označit za správný. Smyslem testování je snížit pravděpodobnost výskytu neodhalených defektů. Tato pravděpodobnost nicméně nikdy nebude nula.

Je k dispozici několik úrovní testování, které dokáží tuto pravděpodobnost snížit k hodnotě, která bude velice blízko nule, ale vždy se při testování musí počítat i s nedokonalostmi testovacího procesu či s lidským faktorem. [11]

Druhý princip přináši informaci o tom, že kompletní otestování všeho není možné. Tato informace koresponduje s předchozím principem, kdy bylo řečeno, že nepřítomnost defektů v softwaru neznamená jeho stoprocentní správnost. Kompletní otestování všeho není možné, protože žádný testovací tým nikdy nebude schopný obsáhnout všechny možné kombinace všech vstupních parametrů a podmínek. Podle ISTQB je přínosnější se zaměřit na analýzu rizik a prioritizaci, než vyvíjet snahu o kompletní otestování softwaru. [11]

Poslední princip je důležitý především pro manažment, vedení projektového týmu či samotného zákazníka. Tento princip zmiňuje, že včasné testování šetří čas a peníze. Testovací aktivity by se měly ideálně zahájit co nejdříve a ideálně v rámci CI/CD „pipeline". Tento princip prakticky znemožňuje využití vodopádového modelu či klasického V-Modelu, nýbrž v těchto modelech nelze naplno využít výhod včasného testování. [11]

3.4.2 ASPICE

ASPICE (Automotive Software Process Improvement Capability dEtermination) je standard využívaný v automobilovém průmyslu. ASPICE vychází ze standardu SPICE a jeho cílem je definovat procesy při vývoji a testování softwaru v automobilovém odvětví. ASPICE proces je velice komplexní a jde ruku v ruce s mnoha osvědčenými postupy řízení bezpečnosti a kvality.

Při ověřování dodržování ASPICE procesu je možné dosáhnout až pěti úrovní ohodnocení. Dosažení úrovně ASPICE 2 je nicméně ve skutečnosti dostačující pro dokončení auditu standardu funkční bezpečnosti ISO 26262, který mluví především o bezpečnosti elektronických a elektrických systémů v automobilech. [14]

3.4.3 ISO/IEC 29119

ISO/IEC 29119 (Software and systems engineering – Software testing) je sada mezinárodních standardů pro testování softwaru. Skládá se z pěti častí. Každá z těchto částí popisuje jinou fázi testovacího procesu:

- 1. část – Pojmy a definice
- 2. část – Testovací procesy
- 3. část – Testovací dokumentace
- 4. část – Testovací techniky
- 5. část – „Keyword-Driven" testování

Tato norma se stala významnou v oblastni testování softwaru a je široce používána ve velkých organizacích a vývojových týmech. Na následujícím obrázku lze vidět, které kapitoly by neměly chybět v testovacím plánu. [15]

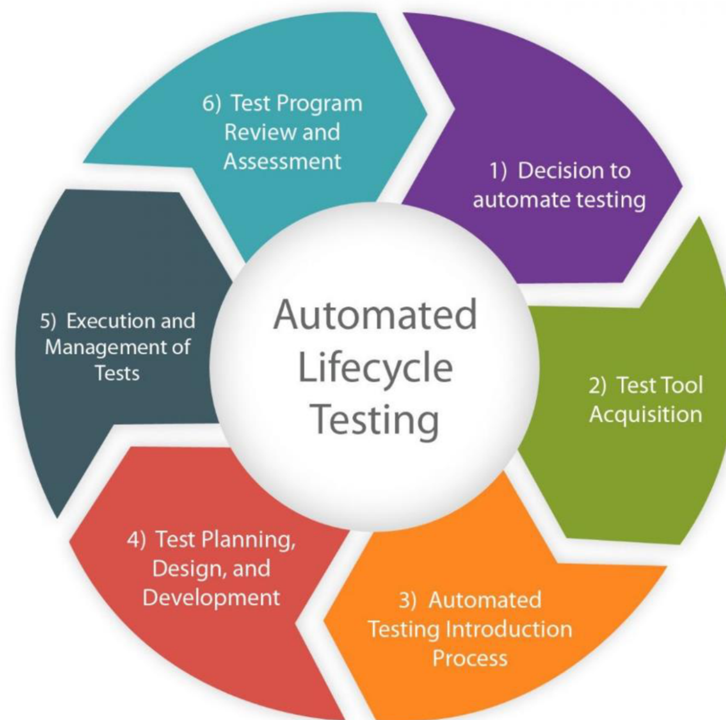
B.1.3 Test Plan	Shall
a) Context of the testing:	Shall
i) Project(s)/Test sub-process(es)	Shall
ii) Test item(s)	Shall
iii) Test scope	Shall
iv) Assumptions and constraints	Should
v) Stakeholders	Should
b) Testing communication	Should
c) Risk register:	Shall
i) Product risks	Shall
ii) Project risks	Shall
d) Test strategy:	Shall
i) Test sub-processes	Shall
ii) Test deliverables	Shall
iii) Test design techniques	Shall
iv) Test completion criteria	Shall
v) Metrics to be collected	Shall
vi) Test data requirements	Shall
vii) Test environment requirements	Shall
viii) Retesting and regression testing	Shall
ix) Suspension and resumption criteria	Shall
x) Deviations from the Organizational Test Strategy	Should
e) Testing activities and estimates	Shall
f) Staffing:	Should
i) Roles, activities, and responsibilities	Should
ii) Hiring needs	Should
iii) Training needs	Should
g) Schedule	Shall

Obrázek 9 - Obsah testovacího plánu podle ISO/IEC 29119-3 [43]

3.5 Automatizace testování

Automatizace testování je činnost, která se v posledních letech stala neoddiskutovatelnou součástí testování softwaru. S přibývající komplexností všech vyvíjených aplikací a systémů je automatizace testování téměř nutnost a bez automatizovaných testů je prakticky nemožné efektivně udržovat software v dostatečné kvalitě. Automatizace testování drasticky snižuje čas a úsilí vynaložené na testování. Ve srovnání s manuálním testováním je při nastavování procesu vyžadováno více úsilí, ale jakmile je tento proces nastaven, lze ušetřit spoustu času a úsilí. [16]

Ne všechny testy však mohou být zautomatizovány a ne u všech testů se to vyplatí. Rozhodnutí kdy a které testy automatizovat je prvním krokem v procesu, který se nazývá cyklus automatického testování (z angl. „Automated Lifecycle Testing“). O vhodnosti automatizace vždy rozhoduje rozsah a velikost benefitů, které nám tato automatizace přinese. [17]



Obrázek 10 - Cyklus automatizovaného testování [44]

3.5.1 Výhody automatizace testování

Cílem automatizovaného testování je časová úspora při jejich spouštění. Obecně lze říci, že automatizace testování má za účel usnadnit, urychlit a zefektivnit provádění postupu testování. Jak již bylo zmíněno, o vhodnosti automatizace rozhoduje rozsah a velikost benefitů, které nám tato automatizace přinese. Mezi benefity automatizace lze zařadit:

- rychlá zpětná vazba
- spolehlivost
- časová úspora
- snížení celkových nákladů
- snadnější pokrytí požadavků

Rychlá zpětná vazba je jeden z nejpřínosnějších benefitů automatizovaných testů. Díky automatizovaným testům je jednoduché odhalovat defekty v ranné fázi vývoje, což snižuje náklady na vývoj a pomáhá snižovat čas strávený na následných opravách těchto defektů.

Dalším benefitem je rozhodně spolehlivost. Pokud je potřeba spouštět každý den ty samé testy, tak existuje vysoká šance, že se projeví lidský faktor a že se člověk v nějakém kroku zmýlí a kvůli této chybě poté dojde k nepravdivému výsledku testu. Na kvalitu lidské práce má vliv mnoho faktorů, které často ani nedokážeme ovlivnit. Toto nám při automatizovaném testování nehrozí, protože nástroje, které se k automatizaci testování používají, jsou proti těmto faktorům imunní.

Neméně důležitým benefitem je také časová úspora. Čas je velice důležitá veličina, které máme při vývoji a testování téměř vždy nedostatek. Pokud se testy spouštějí a vyhodnocují automaticky, ušetříme tím čas pracovníků, kteří by museli tyto testy spouštět a vyhodnocovat manuálně. Je ale důležité míti na paměti to, že časová úspora se projeví především v delším časovém horizontu. Příprava a implementace automatizovaných testů nám v počátku zabere více času, ale s každým následným spuštěním těchto testů už čas pouze šetříme. [18]

3.5.2 Nevýhody automatizace testování

Automatizace testování nemá ale pouze výhody, automatizace má samozřejmě i pár nevýhod. Jednou z nevýhod je rozhodně nutnost údržby a aktualizace testů v případě potřeby. Pokud je do softwaru zavedena nějaká změna, která změní funkcionalitu softwarové komponenty, tak je potřeba zaktualizovat i test, který tuto funkcionalitu testuje. Pokud nedojde k dostatečně rychlé aktualizaci testů, může to mít negativní dopad na běh a výsledky testů.

Testy, které nebyly včas zaktualizované, nemusí odhalit všechny defekty a ty tak teoreticky moho projít přes testovací proces až k zákazníkovi.

Další nevýhodou je nutnost prvotní investice do nákupu všech nástrojů a dalších zdrojů, které budou využity pro automatizaci testování. Tato investice je dalším nákladem se kterým je potřeba při automatizaci počítat. Nákup nástrojů a vytvoření prostředí pro běh automatických testů je nicméně investice, která se při správném nastavení testovacího plánu rychle vrátí. [20]

3.6 Úrovně testování

3.6.1 Jednotkové (unit) testy

Unit testování je první úroveň testování a předchází integračnímu testování. Slouží k ověření, že jsou jednotky (unity) správně naimplementované a poskytují rychlou zpětnou vazbu vývojáři. Unity jsou nejmenší testovatelnou částí každého softwaru. Unit testy spadají pod testování bílé skříňky, tudíž jsou implementovány a spouštěny přímo vývojáři. [21]

Na úrovni unit testování lze uplatnit i tzv. vývoj řízený testy (z angl. „test-driven development“). Podstata tohoto vývoje spočívá v tom, že se nejprve implementují unit testy, které nejprve neprochází a až poté se implementuje samotný zdrojový kód. Cílem poté je naimplementovat kód takovým způsobem aby jednotkové testy procházely. Jakmile jednotkové testy procházejí, přichází na řadu refaktorizace kódu. Tento postup s sebou přináší nevýhodu v tom, že existuje šance, že se vyskytuje chyba v jednotkovém testu a kvůli této chybě tento test neprochází. Jednotkové testy jsou výborným kandidátem na automatizaci. [21] [22]

3.6.2 Modulové testy

Modulové testování je nadstavba jednotkového testování. Cílem modulového testování je ověření, že každý modul pracuje správně a splňuje všechny požadavky. Modulové testování se zaměřuje na otestování jednotlivých modulů, které jsou sestaveny z jednotlivých unit. Modulové testy tedy ověřují, zda tyto unity dohromady tvoří funkční modul. Modulové testování patří pod testování bílé skříňky, tudíž jsou v kompetenci vývojářského týmu. Modulové testy jsou vhodným kandidátem na automatizaci. [23]

3.6.3 Smoke testy

Smoke testování je označováno také jako ověřovací testování sestavení. Cílem smoke testu je ověření, že je sestavení softwaru dostatečně stabilní pro další testování. Smoke testy obvykle testují pouze nejdůležitější funkce softwaru.

Smoke testy jsou vhodným kandidátem na automatizaci, protože je potřeba je spouštět pravidelně (při každém sestavení) a není náročné je udržovat a aktualizovat. Smoke testy bývají v kompetenci testerů, nýbrž obvykle nebývá potřeba znalost zdrojového kódu. [24]

3.6.4 Softwarové integrační testy

Integrační testování má za cíl otestování vnější integrace softwaru, což znamená, že ověřují komunikaci mezi dvěmi softwarovými jednotkami či moduly. Integrační testování se zaměřuje na ověření správnosti komunikačního rozhraní.

V závislosti na tom, jak náročné může být vytvořit automatizované integrační testy, bývá na této testovací úrovni využíváno kombinace automatizovaných i manuálních testů. Pro integrační testování může být v komplikovaných případech potřeba spolupráce s integrátorem či dalšími firmami, které se na vývoji podílí. [25]

3.6.5 Systémové testy

Systémové testování je testování černé skříňky. Jejich cílem je validace softwarové komponenty proti softwarovým požadavkům. Systémové testování je v některých případech označováno také jako softwarové kvalifikační testování. Tyto testy ověřují, že na základě specifických vstupních dat, vrací softwarová komponenta správná výstupní data. Tato úroveň testování je v plné kompetenci testerů, nýbrž není potřeba znalost zdrojového kódu. Systémové testování ověřuje software z pohledu zákazníka.

Tato úroveň testování je využívána pro potřeby regresního testování, protože testuje celou vnější strukturu a tím poskytují informaci o celkové kvalitě softwaru. Tato testovací úroveň je výborným kandidátem na automatizaci. [26]

3.6.6 Akceptační testy

Akceptační testování je poslední fází testování softwaru před jeho převzetím zákazníkem. Toto testování probíhá již přímo na zařízení zákazníka, který má obvykle připravené své vlastní testovací scénáře, které je nutno splnit.

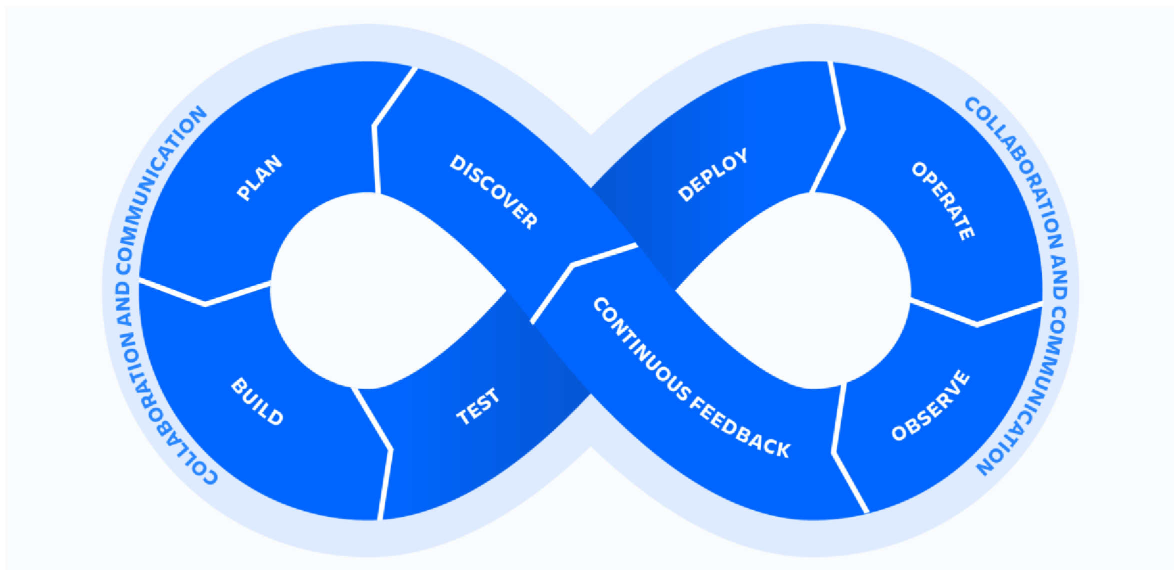
Existuje více typů akceptačního testování, jako příklad si můžeme uvést například uživatelské nebo byznysové. Akceptační testy nejsou vhodnou volbou pro automatizaci, ýbrž často neexistuje možnost jak akceptační požadavky automaticky otestovat a testování probíhá na straně zákazníka. [27]

3.7 DevOps prostředí

DevOps je sada postupů, nástrojů a obecné filozofie, která automatizuje procesy mezi vývojem softwaru a týmy, které se věnují sestavování, testování a vydávání softwaru. Hlavní myšlenkou DevOps je zefektivnění a zrychlení spolupráce všech týmů, které se věnují vývoji a testování softwaru.

DevOps se tedy snaží zefektivnit a zjednodušit způsob komunikace a spolupráce mezi jednotlivými týmy a hlavním nástrojem pro dosažení tohoto cíle je především automatizace, která umožňuje množství potřebné koordinace mezi týmy snížit na minimum.

DevOps týmy zpravidla využívají nástroje, které jim umožňují zautomatizovat a urychlovat procesy, které se neustále opakují, což pomáhá zvyšovat spolehlivost. Tyto nástroje pomáhají týmům s důležitými DevOps postupy. Mezi tyto postupy se z pohledu testování řadí především průběžná integrace (z angl. „Continuous Integration“) a průběžné dodávání (z angl. „Continuous Delivery“). Životní cyklus DevOps obsahuje osm fází, které se neustále opakují a které se na sebe navazují. Průběžná integrace i průběžné dodávání představují pouze dvě z těchto osmi fází. [28]



Obrázek 11 - DevOps životní cyklus [28]

3.7.1 Průběžná integrace

Průběžná integrace je agilní DevOps postup, který umožňuje většímu počtu vývojářů přispívat a spolupracovat na sdíleném zdrojovém kódu. Průběžná integrace je postavena na několika základních pilířích. Mezi tyto pilíře lze zařadit automatizované testování, správa verzí a také automatizované sestavování. Cílem průběžné integrace je především urychlení integrace změn, které vývojáři do softwaru zavádí.

Průběžná integrace umožňuje velice rychlé sestavení a otestování (většinou pomocí jednotkových, modulových či integračních testů, které jsou implementovány přímo vývojáři) při každém zavedení nové změny do sdíleného repozitáře. Následovat může také například statická analýza kódu. Tato rychlá testovací smyčka umožňuje rychlejší detekci defektů. [29]

3.7.2 Průběžné dodávání

Průběžné dodávání je další z agilních DevOps postupů a cílem průběžného dodávání je dostat co nejrychleji novou verzi softwaru do stavu, ve kterém je připravena k otestování v produkčním prostředí nebo v prostředí, které se produkčnímu prostředí podobá či ho nějakým způsobem simuluje.

V tomto prostředí je poté možnost otestovat novou verzi softwaru například pomocí systémových testů či akceptačních testů. Průběžné integrace a průběžné dodávání se navzájem doplňují a obvykle jsou implementovány společně v tzv. „CI/CD pipeline“. [30]

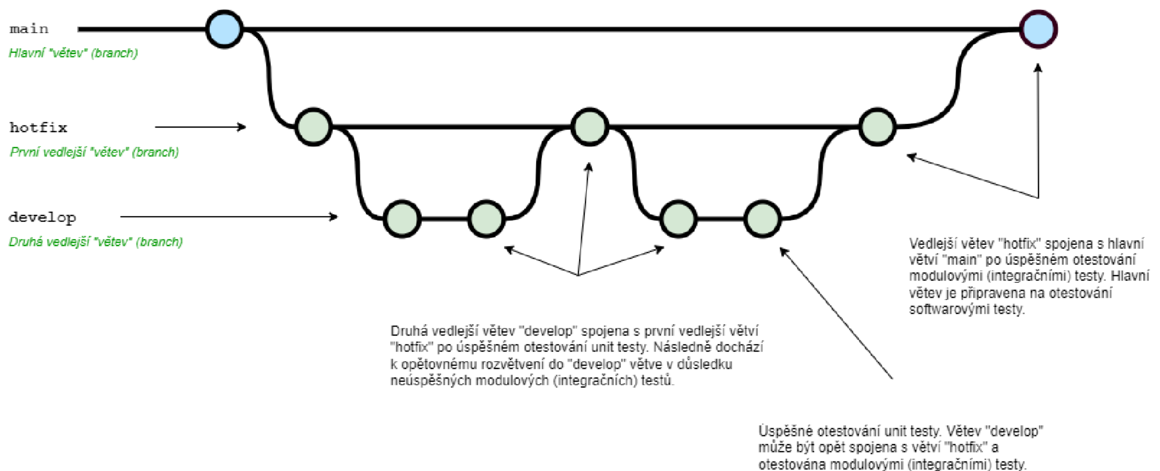
3.8 Nástroje používané v CI/CD

K efektivnímu praktikování DevOps postupů a využívání CI/CD jsou potřeba nástoje, které umožňují automatizaci průběžné integrace a průběžného dodávání.

Základem této automatizace jsou systémy pro správu verzí. Systémy pro správu verzí jsou nepostradatelnou součástí vývojového procesu softwaru. Systémy pro správu verzí umožňují sledovat historii změn v kódu a přehledně evidovat, které změny byly provedeny, a také kým a v jakém konkrétním čase nebo na jaké revizi byly tyto změny provedeny. Tato funkce je využívána i v samotném testování, protože tato možnost umožňuje testovacímu týmu sledovat jakým způsobem se kód mění a vyvíjí a jaký dopad mohou mít tyto změny na testování. Systémy pro správu verzí umožňují také tzv. „větvení“ (branchování) kódu, kdy umožňují vytvořit více „větví“ (branches) s odlišným kódem a tyto větve poté jednotlivě otestovat.

Největším přínosem systémů pro správu verzí je však uchovávání všech potřebných souborů pro sestavení softwaru na jednom místě, které je sdílené pro celý projektový tým. To je prvním krokem k implementaci průběžné integrace a průběžného dodávání, neboť tato funkcionalita umožňuje každému členovi týmu pravidelně sdílet své změny, čímž se obvykle v CI/CD spustí proces automatizovaného sestavení a testování, které poskytne rychlou zpětnou vazbu o naimplementovaných změnách.

Mezi nejznámější systémy pro správu verzí lze zařadit SVN či Git. [31]



Obrázek 12 - Ukázka "větvení" kódu [45]

Dalšími využívanými nástroji jsou nástroje, které slouží k samotné automatizaci procesu vývoje softwaru. Tyto nástroje jsou používané pro zjednodušení a zefektivnění procesu vývoje a testování softwaru. Tyto nástroje napomáhají vytvářet procesy, které poté automaticky provádějí sestavování a testování softwaru. Cílem tohoto procesu je ušetření času a práce, které by vyžadovalo manuální sestavení a testování.

Existuje několik nástrojů, které se k dají k takovým účelům využít. Mezi ně patří například Jenkins či GitLab CI/CD. [31]

3.8.1 SVN

SVN bylo založeno v roce 2000 společností CollabNet a je k dispozici jako open source software. SVN nabízí velmi podobné funkce jako Git, ale i přesto se dá pár hlavních rozdílů najít.

Jedna z věcí ve které SVN s Gitem odlišují, je typ systému. SVN je centralizovaný typ systému, což znamená, že v SVN existuje jeden centrální repozitář a k tomuto centrálnímu repozitáři přistupují jednotliví vývojáři či testeři. Je tedy téměř nemožné aby na jednom souboru pracovalo více vývojářů či testerů.

Druhým rozdílem je možnost práce offline. SVN neumožňuje práci offline a to z důvodu využívání centrálního serveru na kterém je vše uloženo a jakákoliv aktivita musí směřovat právě přes tento centrální server. [32]

3.8.2 Git

Dalším systémem pro správu verzí je Git. Git byl založen v roce 2005 Linusem Torvaldsem, který je znám především zahájením vývoje jádra operačního systému Linux. Git byl postaven na pilířích jako jsou rychlost, jednoduchý design, podpora nelineárního vývoje či schopnost zvládnout velké projekty.

Git využívá distibuovaný systém, což znamená, že v Gitu každý vývojář či tester pracuje s celou kopií repozitáře. Díky distribuovanému charakteru je Git vhodný pro práci offline a nabízí možnost vytvoření nové „větve“ a provádění změn bez připojí k centrálnímu repozitáři.

Git také nabízí lepší integraci do dalších nástrojů, které se mohou využívat k vývoji a testování softwaru. Jako příklad můžeme uvést například nástroj JIRA. Git je také obecně rychlejší než SVN a nabízí flexibilnější možnosti, co se týče větvení a spojování souborů zpět do hlavní větve.

Git je velice oblíbeným nástrojem a v posledních letech se stal více využívaný než zmiňované SVN. Je téměř jisté, že tento trend bude pokračovat i v dalších letech a Git nahradí SVN na většině projektů. [32]

3.8.3 Jenkins

Jenkins je open source nástroj, jehož první verze byla vydána již v roce 2011. Jenkins podporuje mnoho pluginů, které usnadňují další integraci s dalšími nástroji.

Jenkins po propojení s SVN nebo Gitem nabízí možnost spouštění automatického sestavení a testování po každé změně ve zdrojovém kódu (po každém committu). Další možností, která je často využívaná, je spouštění tohoto celého procesu mimo obvyklou pracovní dobu.

Obrázek 13 - Příklad Jenkins "pipeline" [46]

Tento proces obvykle začíná tzv. „checkoutem“, kdy proběhne stažení relevantních souborů a složek z repozitáře a pokračuje sestavením testovacího prostředí a samotných testů. Poté následuje samotné provedení testů a uložení výsledků testů do požadované složky. Užitečnou funkcí, kterou Jenkins také nabízí, je logování, které umožňuje najít a opravit případný problém v zautomatizovaném procesu. Tuto funkci ocení především projektové týmy, které spouští Jenkins v průběhu noci, kdy neexistuje možnost průběžně sledovat pokrok a hlášení, které Jenkins průběžně ukládá.

```
2023-03-02 01:21:26  [htmlpublisher] Archiving HTML reports...
2023-03-02 01:21:26  [htmlpublisher] Archiving at BUILD level C:\jenkins\workspace\ARA-RGL\11\001_cov\Coverage_UT to /JENKI
2023-03-02 01:21:26  ERROR: Specified HTML directory 'C:\jenkins\workspace\ARA-RGL\11\001_cov\Coverage_UT' does not exist.
```

Obrázek 14 - Příklad chybové hlášky [47]

Definice jednotlivých Jenkins jobů jsou psány v tzv. groovy skriptech. Na následujícím obrázku lze vidět příklad groovy skriptu s názvem „publish results“, který na Jenkinsu zveřejní výsledky testů ve formátu HTML. [33]

```
stage('publish results') {
   steps {
      publishHTML([allowMissing: false, alwaysLinkToLastBuild: false,
      keepAll: true, reportDir: ".\\SWT\\Reports_6_1",
      reportFiles:'index.html', reportTitles: 'TR_SW_6_1'])

      publishHTML([allowMissing: false, alwaysLinkToLastBuild: false,
      keepAll: true, reportDir: ".\\SWT\\Reports_6_2", reportFiles:
      'index.html', reportTitles: 'TR_SW_6_2'])
   }
}
```

3.8.4 GitLab CI/CD

GitLab CI/CD je open source nástroj pro vývoj softwaru, který umožňuje implementaci a automatizaci průběžné integrace a průběžného dodávání. GitLab byl vytvořen v roce 2013 Valerym Sizovem a Dmytrem Zaporozhetsem.

Na rozdíl od Jenkinsu je GitLab CI/CD již předpřipravený a využívá vestavěnou funkci pro CI/CD, která umožňuje velice rychlé nastavení kanálů pro CI/CD. Další funkcí je Auto DevOps, díky které je doručování softwaru snadné, efektivní a automatizované.

GitLab CI/CD se stává čím dál tím více oblíbenějším nástrojem. [34]

4 Vlastní práce

Praktická část je zaměřena na navržení, provádění a udržování testovacího prostředí a automatických testů. Testovacím objektem je parkovací asistent spadající do skupiny ADAS (Advanced Driver Assistance Systems). Testovací strategie počítá pouze s testováním na softwarové úrovni. Parkovací asistent bude vyvíjen v programovém jazyce C++ v agilním modelu vývoje.

4.1 Plánování testovací strategie

4.1.1 Požadavky

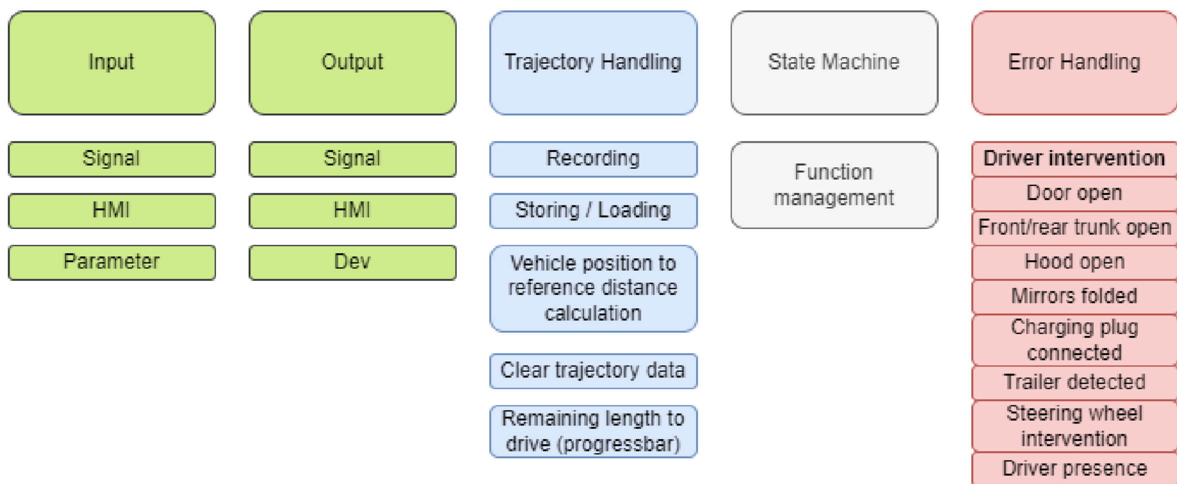
Testovací strategie byla vytvořena především na základě požadavků zákazníka. Z důvodu toho, že se testovací objekt plánuje využívat v osobních automobilech, je potřeba dbát zvýšenou pozornost na bezpečnost softwarové komponenty.

Mezi hlavní požadavky tudíž patří především 100% pokrytí zákaznických a softwarových požadavků testy. Dalším požadavkem je dosáhnutí SOP (Start Of Production) s 0 detekovanými defekty, které by měly za následek bezpečnostní hrozbu pro řidiče.

4.1.2 Testovací úrovně

Z důvodu velkého tlaku na bezpečnost softwaru je potřeba nastavit testovací úrovně tak, aby bylo možno detekovat, co možná nejvíce defektů v softwaru. V teoretické části byly popsány vybrané testovací úrovně, které se běžně při testování softwaru v automotive prostředí využívají. Z tohoto teoretického popisu lze tedy vycházet.

Aby bylo možno správně zvolit potřebné testovací úrovně, které jsou potřeba pro zmíněnou softwarovou komponentu, je potřeba si nejprve navrhnout architekturu. Tato architektura nemusí být do detailu propracovaná, ale měla by poskytnout alespoň tzv. „high-level“ přehled o tom, jak bude softwarová komponenta fungovat a co bude obsahovat.



Obrázek 15 - Ukázka „high-level" architektury [48]

Při pohledu na „high-level" architekturu lze vidět, že se softwarová komponenta skládá z 5 modulů a z 20 malých „unit" (jednotek). Spojením těchto jednotek a modulů vznikne celý softwarová komponenta.

Pro otestování softwarové komponenty budou tedy využity unit testy, které umožní otestování jednotlivých jednotek ihned po naimplementování. Unit testy spadají pod testování bílé skříňky, tudíž je potřeba přístup do zdrojového kódu a tyto testy budou naimplementovány přímo vývojáři. Unit testy dodají vývojářům potřebnou zpětnou vazbu k jejich implementaci zdrojového kódu.

Unit test, který má za úkol otestovat specifickou část kódu, musí být naimplementován jiným vývojářem než kód samotný. Tím se docílí toho, že bude zdrojový kód správně otestován. V následující části textu lze vidět příklad jednoduchého unit testu, který ověřuje, že došlo ke změně hodnoty ve vzdálenosti kamery k překážce.

```
TEST_P(Unit_AssistantToCamera_Distance_TestSuite,
AssistantToCameraDistance)
{
    // Set initial value of AssistantToCamera
    vehicleDisplacement.distance = floatToFlt(std::get<1>(GetParam()));
    assistantOutput.run();

    vehicleDisplacement.distance = floatToFlt(std::get<2>(GetParam()));
    assistantOutput.run();

    // Check if AssistantToCamera.Distance has expected value
    EXPECT_TEST(std::get<0>(GetParam()),
EXPECT_EQ(testOutput.ToCamera.Distance, std::get<3>(GetParam())));
}
```

Na další testovací úrovni se poté nacházejí samotné moduly, které jsou schránkou pro menší jednotky, které se nacházejí uvnitř. Na této testovací úrovni bude využito modulových testů i integračních testů.

Modulové testy umožňují otestování samotné funkcionality modulů. Tyto testy mohou být v některých případech velice podobné samotným unit testům, nicméně z důvodu potřebné kvality a bezpečnosti softwaru, bude využito i této testovací úrovně. V tomto případě je vhodné se řídit principem o včasném testování, který zmiňuje ISTQB. Včasné otestování šetří čas a peníze a tudíž je využití modulových testů dobrou volbou.

Další testovací úrovní jsou integrační testy, které mají za úkol otestovat samotné rozhraní těchto modulů. Toto rozhraní umožňuje modulům komunikovat s ostatními moduly či s aplikacemi třetích stran.

Je důležité mít na paměti, kdo má zodpovědnost za který typ testování. Zatímco modulové testování je testování bílé skříňky a má velice blízko k samotnému unit testování a je potřeba aby se mu věnovali vývojáři, tak integrační testování už je testování černé skříňky. K testování rozhraní a komunikace mezi moduly obvykle nebývá potřeba znát zdrojový kód. Toto testování tedy mohou provádět samotní testeři.

Poslední úrovní testování jsou systémové testy. Tyto testy bývají některými test manažery označovány jako nejdůležitější a nejvíce vypovídající, protože testují celkovou kvalitu softwaru. Systémové testy spadají pod testování černé skříňky, tudíž jsou v plné kompetenci testerů, kteří nemají přístup ke zdrojovému kódu a kteří testují funkcionalitu na základě specifikace. Touto úrovní testů zamezíme tomu, že je naše testování jakkoliv ovlivněno tím jak byl naimplementovaný zdrojový kód.

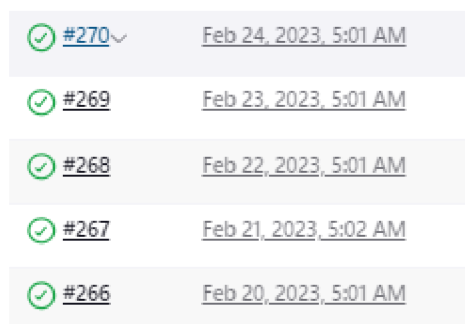
Po celkové analýze „high-level“ architektury je závěr tedy takový, že bude potřeba využití čtyř úrovní testování: unit testy, modulové testy, integrační testy a systémové testy.

4.2 Použité nástroje a technologie

4.2.1 Nástroje pro automatizaci

K automatizaci procesu testování bylo využito služeb nástrojů Jenkins a SVN. Oby tyto nástroje byly zvoleny především z důvodu zkušenosti autora s těmito nástroji. Existují další alternativy, které byly popsány v teoretické části.

Na Jenkinsu bylo využit služeb CI/CD „pipeline“, která umožňuje nastavení specifického času, kdy je potřeba testy spouštět. Pro účely této práce bylo nastavené pravidelné spuštění testů v 5:00 v každý pracovní den. Druhou (nevyužitou) variantou bylo nastavení automatického spouštění testů po každé změně ve zdrojovém kódu. Tato varianta byla vyhodnocena jako nevhodná pro větší projekty z důvodu častých změn v kódu, což poté způsobuje vytváření front na Jenkinsu. Pokud nastane během běhu testů jakýkoliv problém, budou se muset testy spustit manuálně. K dispozici bude také výpis z konzole, který umožní identifikování případných problémů s automatickým během testů.



Obrázek 16 - Automatické spouštění testů na Jenkinsu [49]

Aby bylo možné naplno využít benefitů CI/CD, bude potřeba několik serverových počítačů, kterým se také říká Jenkins „nodes“. Pro účely automatizace se využije pěti Jenkins „nodes“. Existuje více možností jak tyto serverové počítače nakonfigurovat. První možností je tzv. Jenkins „controller“, který funguje jako manažer ostatních Jenkins „nodes“ a určuje kdy a co se na Jenkinsu spustí. Druhou možností je tzv. Jenkins „agent“, který poté slouží jako jako „zaměstnanec“ Jenkins „controlleru“ a vykonává vše, co mu tento Jenkins „controller“ přikáže. Jeden z počítačů tedy bude plnit roli Jenkins „controlleru“ a další čtyři budou plnit roli Jenkins „agentů“. Každý Jenkins „agent“ bude zodpovědný za jednu testovací úroveň. Tímto je možné předejít případným problémům s nedostatkem Jenkins „agentů“ a také případným technickým problémům.

Je důležité si uvědomit, že všechny automatizované testy poběží v tomto serverovém Jenkins prostředí a je tudíž důležité mít k dispozici dostatečný počet licencí, které budou využity pro spouštění nástrojů, které jsou k testování nezbytné.

SVN bude poté využíváno jak jako sdílené uložiště zdrojového kódu, tak také jako uložiště automatizovaných testů. Z tohoto důvodu je důležité Jenkinsu říci, kde na SVN má hledat, které důležité artefakty. Je samozřejmě také nezbytné, aby měl samotný Jenkins přístup ke všem důležitým složkám na SVN. Toho bude dosaženo využitím technického uživatele. V následující části textu lze vidět groovy skript, který provede tzv. „checkout“ a stáhne všechna potřebná data z SVN repozitáře do Jenkinsu.

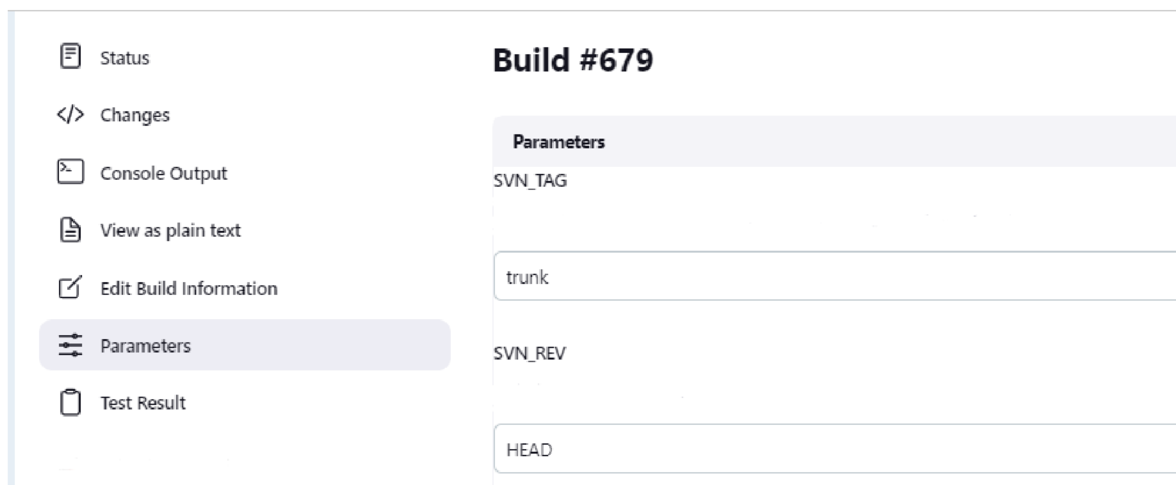
```
stage('checkout') {
    steps {
        checkout(
            [$class: 'SubversionSCM',
            additionalCredentials: [],
            excludedCommitMessages: '',
            excludedRegions: '',
            excludedRevprop: '',
            excludedUsers: '',
            filterChangelog: false,
            ignoreDirPropChanges: false,
            includedRegions: '',
            locations: [
                [
                    credentialsId: "${CREDENTIAL_ID}",
                    depthOption: 'infinity',
                    ignoreExternalsOption: true,
                    local: './020',
                    remote: './$SVN_TAG/020_tools@$SVN_REV'
                ],
                [
                    credentialsId: "${CREDENTIAL_ID}",
                    depthOption: 'infinity',
                    ignoreExternalsOption: true,
                    local: './030',
                    remote: './$SVN_TAG/030_code@$SVN_REV'
                ],
                [
                    credentialsId: "${CREDENTIAL_ID}",
                    depthOption: 'infinity',
                    ignoreExternalsOption: true,
                    local: './040/03_sw-test',
                    remote: './$SVN_TAG/040_test/03_sw-test@$SVN_REV'
                ],
        ],
        quietOperation: true,
        workspaceUpdater: [$class: 'UpdateWithCleanUpdater']]
        )
    }
}
```

Propojení mezi SVN a Jenkinsem nebývá složité a lze ho docílit právě podobným groovy skriptem. Tento skript v tomto konkrétním případě využívá tři proměnné:

- CREDENTIAL_ID
- SVN_TAG
- SVN_REV

CREDENTIAL_ID je parametr, který Jenkins využívá k samotnému přihlášení do SVN. Jak již bylo zmíněno, bylo využito služeb tzv. technického uživatele, který bude využíván pouze Jenkinsem. V mnoha společnostech se využívá k podobným účelům pouze jeden technický uživatel pro všechny typy projektů. Hlavním důvodem je bezpečnost, protože při existenci desítek nebo dokonce stovek technických uživatelů by se snadno mohlo stát, že na některý z nich zapomeneme nebo se ztrat údaje nebo že se tyto údaje dostanou do nesprávných rukou.

Proměnné SVN_TAG a SVN_REV využívají jako vstupní hodnoty parametry z příslušného Jenkins „jobu". Jenkins umožňuje nastavit si jako parametr libovolnou SVN revizi či libovolný SVN tag. Toto je obrovská výhoda, kterou nám systémy pro správu verzí nabízí a umožňuje nám to spouštění automatických testů na více verzích kódu.



Obrázek 17 - Parametry na Jenkinsu [50]

4.2.2 Nástroje pro testování

Pro testování bylo využito služeb Microsoft Visual Studia, které má v sobě již integrované testovací frameworky jako Microsoft Unit Testing Framework for C++ a Google Test for C++. Google Test je framework, která spadá do xUnit rodiny a obě zmíněné funkcionality jsou součástí „Desktop development with C++" balíčku.

Dalším využitým nástrojem bylo TPT 18, které bylo využito pro systémové testování. TPT bylo propojeno s Jenkinsem pomocí následujícího groovy skriptu.

```
stage('run SWT tpt') {
  steps {
    warnError('SW-Tests Failed') {
      bat '''.\\020_tools\\tpt\\tpt.exe --nosplash --run build
      .\\040_test\\03_sw-test\\Test_6_2.tpt "Jenkins (EXE)_%PLATFORM%"'''

      bat '''.\\020_tools\\tpt\\tpt.exe --nosplash --run build
      .\\040_test\\03_sw-test\\Test_6_3.tpt "Jenkins (EXE)_%PLATFORM%"'''

      bat '''.\\020_tools\\tpt\\tpt.exe --nosplash --run build
      .\\040_test\\03_sw-test\\Test_6_4.tpt "Jenkins (EXE)_%PLATFORM%"'''
    }
  }
}
```

Výsledky těchto TPT testů byly poté pomocí následujícího groovy skriptu publikovány v předem definované složce.

```
stage('publish TPT JUNIT') {
    steps {
       catchError(buildResult: 'SUCCESS', stageResult: 'SUCCESS', message:
      'publish of test results failed') {
            junit  allowEmptyResults: true, testResults:
            "results/Test/$PLATFORM/testdata_6_2/*.xml"

            junit  allowEmptyResults: true, testResults:
            "results/Test/$PLATFORM/testdata_6_3/*.xml"

            junit  allowEmptyResults: true, testResults:
            "results/Test/$PLATFORM/testdata_6_4/*.xml"
       }
   }
}
```

4.3 Časová analýza automatického a manuálního testování

Po dokončení všech potřebných prací na automatizovaných testech bylo provedeno pět kontrolních spuštění všech testovacích úrovní, aby se ověřila funkčnost testů a také došlo k zaznamenání času trvání jednotlivých běhů. Všechny časové údaje jsou včetně následného zprocesování výsledků a jejich uložení v předem definované složce. Všechny časové údaje jsou zaznamenány v tabulce níže.

Testovací úroveň	**Trvání běhu - automatizovaný**				
Unit testy	1. běh	2. běh	3. běh	4. běh	5. běh
	5m 47s	5m 31s	6m 29s	4m 58s	5m 02s
	Průměrný čas: 5m 33s				
Modulové testy	1. běh	2. běh	3. běh	4. běh	5. běh
	6m 13s	7m 00s	6m 57s	7m 24s	7m 19s
	Průměrný čas: 6m 58s				
Integrační testy	1. běh	2. běh	3. běh	4. běh	5. běh
	8m 14s	8m 58s	9m 20s	8m 34s	9m 36s
	Průměrný čas: 8m 56s				
Systémové testy	1. běh	2. běh	3. běh	4. běh	5. běh
	34m 11s	36m 28s	30m 58s	34m 42s	32m 26s
	Průměrný čas: 33m 45s				

Tabulka 1 - Čas trvání automatizovaného běhu testů

Následně bylo provedeno pět manuálních spuštění testů na všech testovacích úrovních. Každý manuální běh byl zároveň proveden jiným testerem za účelem dosáhnutí, co nejobejktivnějších výsledků. Všechny časové údaje jsou opět včetně následného zprocesování výsledků a jejich uložení v předem definované složce. Všechny časové údaje jsou opět uvedeny v tabulce níže.

Testovací úroveň	**Trvání běhu - manuální**				
Unit testy	1. běh	2. běh	3. běh	4. běh	5. běh
	8m 18s	9m 23s	8m 48s	7m 58s	8m 59s
	Průměrný čas: 8m 41s				
Modulové testy	1. běh	2. běh	3. běh	4. běh	5. běh
	10m 41s	10m 21s	10m 36s	11m 11s	10m 04s
	Průměrný čas: 10m 34s				
Integrační testy	1. běh	2. běh	3. běh	4. běh	5. běh
	11m 48s	12m 37s	12m 29s	12m 13s	13m 06s
	Průměrný čas: 12m 26s				
Systémové testy	1. běh	2. běh	3. běh	4. běh	5. běh
	53m 01s	51m 34s	50m 21s	52m 34s	51m 44s
	Průměrný čas: 51m 50s				

Tabulka 2 - Čas trvání manuálního běhu testů

Jeden automatizovaný běh testů trval v průměru 55 minut a 12 sekund. Oproti tomu jeden manuální běh testů v průměru trval 83 minut a 31 sekund. Během jednoho běhu všech testovacích úrovní tedy došlo v průměru k ušetření 28 minut a 19 sekund.

V případě zmiňované časové úspory je nicméně důležité mít na paměti, že se testy na všech testovacích úrovních spouštějí pravidelně každý den. To znamená, že časová úspora během jediného pracovního týdne budou již 2 hodiny 21 minut a 35 sekund. Během jednoho měsíce, který obsahuje čtyři pracovní týdny to bude již 9 hodin a 26 minut a 20 sekund.

V případě nutnosti spuštění testů vícekrát za den se tato časová úspora značně zvyšuje. Především v případě potřeby rychlých a častých oprav v softwaru se automatizace běhu testů

začne vyplácet čím dál tím více a rychleji. Časová úspora se také zvyšuje s každým přidáním další testovací úrovně a s nově naimplementovanými testy.

V každé iteraci, která byla nastavena na 5 týdnů, bylo potřeba naimplementovat v průměru 25 nových testů dohromady na všech úrovních. Těchto 25 nových testů v každé iteraci způsobilo zpomalení automatizovaného běhu testů v průměru o 5 minut a 31 sekund. S každou další iterací byl tedy automatizovaný běh testů o přibližně 5 minut pomalejší.

Těchto 25 testů nicméně způsobilo, že se manuální běh testů zpomalil o 12 minut a 31 sekund. Testování manuálního běhu bylo však otestováno pouze jedním testerem, čímž nelze stoprocentně zaručit objektivitu výsledku.

4.4 Problémy při automatizovaném běhu testů

4.4.1 Vypršení licencí potřebných pro běh testů

Ne tak častý problém, ale velmi obtížně řešitelný pro běžného uživatele, který potřebuje spustit testy v automatizovaném prostředí. Všechny potřebné licence mají obvykle platnost na 12 měsíců, nicméně existuje možnost, že se na potřebu obnovení potřebných licencí zapomene a poté nastane problém, který zmenožní spuštění testů.

Licence pro účely automatizovaného testování jsou uloženy buď přímo na samotném Jenkinsu, nebo na sdíleném repozitáři odkud si je Jenkins poté sám stahuje. Pro běžného uživatele je velmi obtížné předejít tomuto problému, protože obvykle nemá dostatečné znalosti a práva, které by mu umožňovali vyčíst informace o tom, kdy a jaká licence vyprší.

Nejjednoduším řešením je vytvoření listu využívaných nástrojů, kde bude jako jeden z parametrů uveden i datum expirace licence.

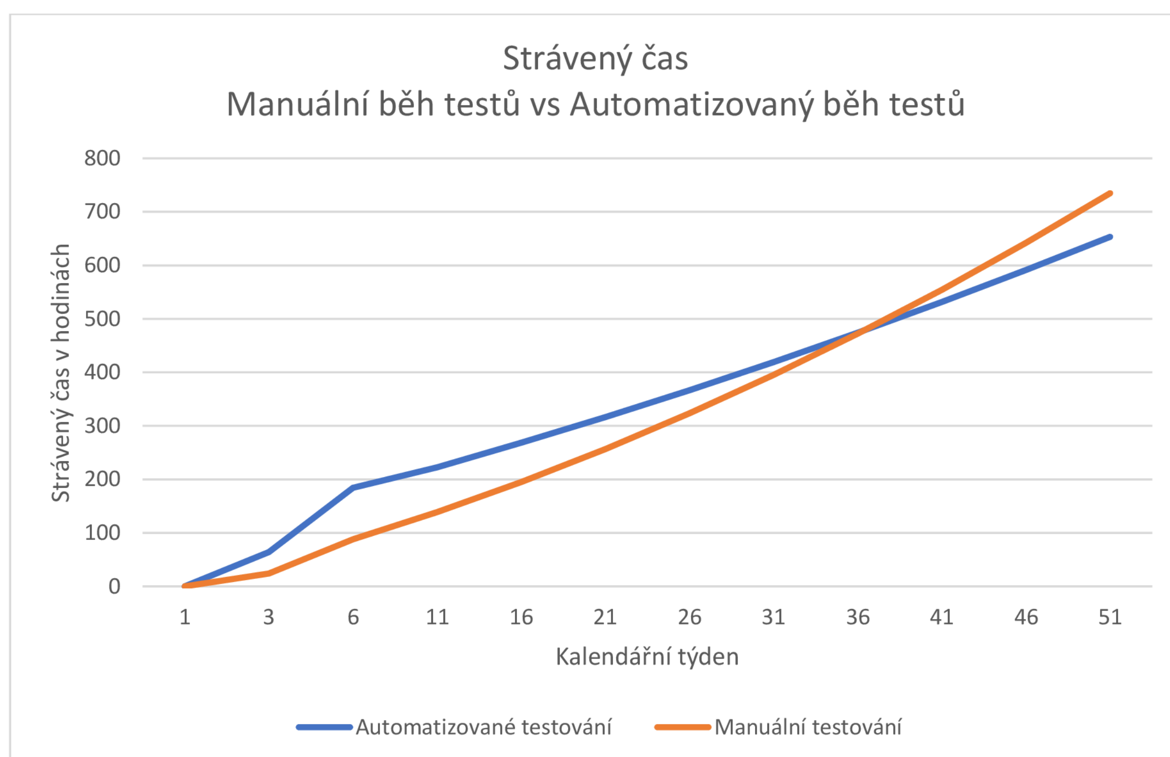
4.4.2 Nedostupnost Jenkins „nodes“

Problém, který nastává v momentu, kdy je ve firmě více aktivních projektů a neúměrný počet Jenkins „nodes“. Tento problém může značně omezit benefity, které s sebou automatizace přináší, neboť se vždy musí počkat na dostupný Jenkins „node“ a to může trvat minuty až hodiny.

Úspora času je jednou z největších výhod automatického běhu testů. Pokud tedy strávíme hodiny čekáním na dostupný Jenkins „node“, tak se časová úspora velice snižuje.

5 Výsledky a diskuse

Na následujícím grafu lze vidět vývoj stráveného času při běhu testů. Tento vývoj v čase počítá s postupným zpomalováním obou běhů z důvodu implementace nových testů. Zpomalení automatizovaného běhu bylo v průměru 10 % v každé iteraci a zpomalení manuálního běhu bylo v průměru 15 %. Jedna iterace trvala 5 týdnů. S přibývajícími testy se tedy zvyšoval rozdíl ve stráveném čase při běhu testů. Graf zobrazuje počáteční investici do nastavení automatizovaného prostředí a do implementace testů a také bere v potaz potřebnou údržbu automatizovaného testovacího prostředí. Touto údržbou se strávilo v průměru 5 hodin během jedné iterace.



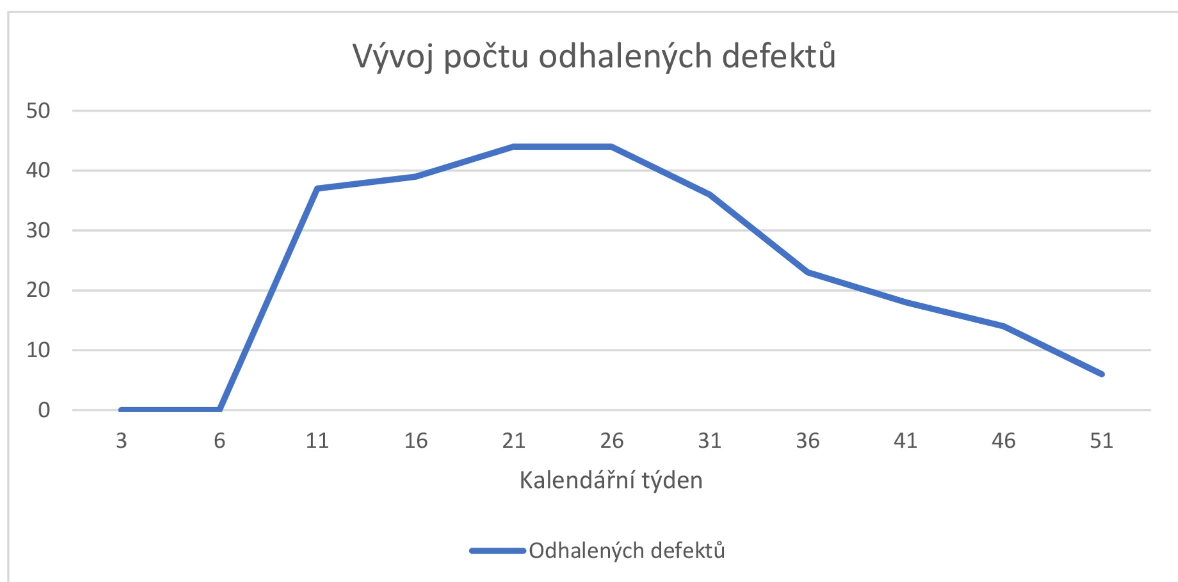
Graf 1 - Časový vývoj běhu testů

Na základě dat z grafu lze vypozorovat, že automatizace testování si sice vyžádala větší počáteční investici, nicméně v průběhu času se tento rozdíl, navzdory potřebné údržbě automatizovaného testovacího prostředí, snižoval. K průniku obou os dochází v kalendářním týdnu 37, kdy se tedy stává automatizace reálně výnosnější a efektivnější než manuální testování. Na konci sledovaného období (po 50 týdnech od začátku měření) je celková časová úspora 82 hodin ve prospěch automatizovaného testování, což je přibližně 10 člověkodnů.

Tato úspora se na první pohled nemusí zdát tak velká a přínosná, nicméně je nutno brát v potaz, že s každým dalším spuštěním testů a s dalšími naimplementovanými testy se tato

časová úspora nadále zvyšuje. Také je potřeba brát v potaz i další benefity, které s sebou tato automatizace přináší. Tato automatizace má využití i na dalších projektech, které se v rámci této firmy plánují, čímž se značně snižuje potřeba stráveného času na vývoji a implementaci automatizovaného testovacího prostředí na příštích projektech.

Kromě těchto benefitů má také automatizované testování výhodu ve spolehlivosti. Automatizace testování eliminuje riziko, které s sebou přináší lidský faktor. Při každodenní potřebě spouštění testů je vysoké riziko toho, že se člověk v nějakém kroku zmýlí a tím znehodnotí celý běh testů a reportované výsledky. Automatizované testování přináší tedy konzistentnější výsledky, ve které mohou mít projektové týmy větší důvěru.



Graf 2 - Vývoj počtu odhalených defektů

Tyto konzistentnější výsledky měly za následek také celkové snížení počtu odhalených defektů. Po naimplementování testů bylo v softwaru odhaleno 37 defektů, z nichž 13 jich mělo vysokou prioritu. Tyto čísla se i díky automatizovanému testování povedlo snížit na 8 odhalených defektů, z nichž pouze 1 byl ohodnocen vysokou prioritou.

6 Závěr

V teoretické části této bakalářské práce proběhlo prozkoumání a zpracování informací o teorii testování softwaru a automatizaci testování, a jejichž využití v rámci moderního vývoje softwaru pomocí DevOps a CI/CD. Zmíněny byly také výhody a nevýhody automatizace testování a kdy je vhodné automatizaci využít. Teoretická část také zmiňuje standardy, které se v dnešní době v testování softwaru vyžadují. V neposlední řadě bylo zmíněno několik nástrojů, které se při automatizaci testování dají využít. Tyto informace a poznatky byly poté využity v praktické části této práce.

V praktické části jsme se poté zaměřili na návrh a implementaci automatizace testování, s cílem zjistit, zda může automatizace testování uspořit čas v porovnání s manuálním testováním. Díky využití Jenkinsu došlo k automatizaci procesu testování, která firmě pomohla v dlouhodobém horizontu ušetřit čas, který by byl jinak stráven procesem manuálního testování, které je zároveň méně spolehlivé než testování automatizované.

Výsledky ukázaly, že automatizace testování je efektivní a může uspořit značné množství času a zdrojů. Po 37. týdnu bylo již dosaženo návratnosti počáteční investice a od tohoto momentu bylo automatizované testování efektivnější a levnější než testování manuální. Je důležité zdůraznit, že konkrétní časový okamžik, kdy se automatizované testování začalo vyplácet, závisí na mnoha faktorech, jako jsou například rozsah testování, použité nástroje, složitost aplikace, ale také na investovaných zdrojích a čase. Proto je nutné pečlivě analyzovat výnosy a náklady pro každý konkrétní projekt a situaci a zhodnotit, zda se automatizace testování vyplatí.

Celkově jsme zjistili, že využití automatizace testování v rámci DevOps a CI/CD procesů je klíčové pro úspěšný a efektivní vývoj softwaru. Využití automatizovaného testování významně napomohlo ke snížení počtu odhalených defektů z počátečních 37 na konečných 8. Na základě výsledků lze doporučit další výzkum v oblasti automatizace testování, zejména v souvislosti s novými technologiemi a nástroji.

7 Seznam použitých zdrojů

[1] KITNER, Radek. Co je testování softwaru? [online]. [cit. 2023-03-09]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-softwaru/

[2] VELIMIROVIC, Andreja. What is SDLC? Understand the Software Development Life Cycle [online]. 2022 [cit. 2023-03-09]. Dostupné z: https://phoenixnap.com/blog/software-development-life-cycle

[3] HAMILTON, Thomas. STLC (Software Testing Life Cycle) Phases, Entry, Exit Criteria [online]. 2022 [cit. 2023-03-09]. Dostupné z: https://www.guru99.com/software-testing-life-cycle.html

[4] What Is SDLC Waterfall Model? [online]. 2023 [cit. 2023-03-09]. Dostupné z: https://www.softwaretestinghelp.com/what-is-sdlc-waterfall-model/

[5] SDLC - V-Model [online]. 2021 [cit. 2023-03-09]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm

[6] SDLC - Agile Model [online]. 2021 [cit. 2023-03-09]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm

[7] KITNER, Radek. Přehled testovacích technik [online]. [cit. 2023-03-09]. Dostupné z: https://kitner.cz/testovani_softwaru/prehled-testovacich-technik/

[8] What Is Static Code Analysis? [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.mathworks.com/discovery/static-code-analysis.html

[9] HAMILTON, Thomas. White Box Testing – What is, Techniques, Example & Types [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.guru99.com/white-box-testing.html

[10] ASHTARI, Hossein. Black Box vs. White Box Testing: Understanding 3 Key Differences [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.spiceworks.com/tech/devops/articles/black-box-vs-white-box-testing/

[11] OLSEN, Klaus, Meile POSTHUMA a Stephanie ULRICH. Učební osnovy – Certifikovaný tester základní úrovně [online]. 2018. [cit. 2023-03-09]. Dostupné z: https://castb.org/wp-content/uploads/2020/05/ISTQB_CTFL_CZ_3_1_1-6.pdf

[12] Grey Box Testing Tutorial With Examples, Tools And Techniques [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.softwaretestinghelp.com/grey-box-testing-tutorial/

[13] Welcome to ISTQB® [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.istqb.org

[14] MOBILITY, Insider. What Is ASPICE? [online]. 2023. [cit. 2023-03-09]. Dostupné z: https://www.aptiv.com/en/insights/article/what-is-aspice

[15] THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION a THE INTERNATIONAL ELECTROTECHNICAL COMMISSION. ISO/IEC/IEEE 29119 [online]. 2021. [cit. 2023-03-10]. Dostupné z: https://www.iso.org

[16] BOSE, Shreya. Automation Testing Tutorial: Getting Started [online]. 2021. [cit. 2023-03-10]. Dostupné z: https://www.browserstack.com/guide/automation-testing-tutorial

[17] SINGH, Pallavi. 6 Stages of Automation Testing Life Cycle (ATLC) [online]. 2020. [cit. 2023-03-10]. Dostupné z: https://qacraft.com/6-stages-of-automation-testing-life-cycle/

[18] BUREŠ, Miroslav, Michal DOLEŽEL a Miroslav RENDA. Efektivní testování softwaru. Praha: Grada Publishing, 2016. ISBN 978-80-271-9389-9.

[19] TEST AUTOMATION: TOP 10 BENEFITS OF AUTOMATION TESTING [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.testrigtechnologies.com/top-10-benefits-of-automation-testing/

[20] JOSEPH, Timothy. What Are the Limitations of Automation Testing? [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://blog.qasource.com/resources/what-are-the-limitations-of-automation-testing

[21] STF. Unit Testing [online]. 2022. [cit. 2023-03-10]. Dostupné z: https://softwaretestingfundamentals.com/unit-testing/

[22] HAMILTON, Thomas. What is Test Driven Development (TDD)? Example [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.guru99.com/test-driven-development.html

[23] KAGUPTA, Kanish. Module Testing [online]. 2022. [cit. 2023-03-10]. Dostupné z: https://www.geeksforgeeks.org/module-testing/

[24] PANKAJ, PP. Smoke Testing | Software Testing [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.geeksforgeeks.org/smoke-testing-software-testing/

[25] SANJOY. Software Engineering | Integration Testing [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.geeksforgeeks.org/software-engineering-integration-testing/

[26] What Is System Testing – A Ultimate Beginner's Guide [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.softwaretestinghelp.com/system-testing/

[27] Acceptance Testing | Software Testing [online]. 2022. [cit. 2023-03-10]. Dostupné z: https://www.geeksforgeeks.org/acceptance-testing-software-testing/

[28] What Is DevOps? [online]. 2020. [cit. 2023-03-10]. Dostupné z: https://www.atlassian.com/devops

[29] REHKOPF, Max. What is continuous integration? [online]. 2020. [cit. 2023-03-10]. Dostupné z: https://www.atlassian.com/continuous-delivery/continuous-integration

[30] What is continuous delivery? [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://www.atlassian.com/continuous-delivery

[31] JACOBS, Mike, Liz CASEY a Ed KAIM. Co je správa verzí? [online]. 2023. [cit. 2023-03-10]. Dostupné z: https://learn.microsoft.com/cs-cz/devops/develop/git/what-is-version-control

[32] LAMB, Tristian. Git vs SVN [online]. 2021. [cit. 2023-03-10]. Dostupné z: https://www.gitkraken.com/blog/git-vs-svn

[33] Jenkins, Build great things at any scale [online]. 2023. [cit. 2023-03-10]. Dostupné z: www.jenkins.io

[34] GitLab CI/CD [online]. 2023 [cit. 2023-03-12]. Dostupné z: https://docs.gitlab.com/ee/ci/

[35] Software Development Life Cycle [online]. [cit. 2023-03-14]. Dostupné z: https://www.javatpoint.com/software-engineering-software-development-life-cycle

[36] Software Testing Life Cycle [online]. [cit. 2023-03-14]. Dostupné z: https://bambooagile.eu/insights/what-is-stlc/

[37] Vodopádový model [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[38] Software Engineering V-Model [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[39] Agilní model [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[40] HIS Findings report [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[41] Testování bílé skříňky [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[42] Testování černé skříňky [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[43] Obsah testovacího plánu podle ISO/IEC 29119-3 [online]. [cit. 2023-03-14]. Dostupné z: https://www.iso.org/standard/79429.html

[44] Cyklus automatizovaného testování [online]. [cit. 2023-03-14]. Dostupné z: https://u-tor.com/topic/test-automation-strategy

[45] Ukázka "větvení" kódu [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[46] Příklad Jenkins „pipeline“ [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[47] Příklad chybové hlášky [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[48] Ukázka „high-level“ architektury [cit. 2023-03-14]. Zdroj: Vlastní zpracování

[49] Automatické spouštění testů na Jenkinsu [cit. 2023-03-15]. Zdroj: Vlastní zpracování

[50] Parametry na Jekinsu [cit. 2023-03-15]. Zdroj: Vlastní zpracování

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

Obrázek 1 - Software Development Life Cycle [35] 12
Obrázek 2 - Software Testing Life Cycle [36] 13
Obrázek 3 - Vodopádový model [37] 14
Obrázek 4 - Software Engineering V-Model [38] 15
Obrázek 5 - Agilní model [39] 16
Obrázek 6 - HIS Findings report [40] 17
Obrázek 7 - Testování bílé skříňky [41] 18
Obrázek 8 - Testování černé skříňky [42] 19
Obrázek 9 - Obsah testovacího plánu podle ISO/IEC 29119-3 [43] 22
Obrázek 10 - Cyklus automatizovaného testování [44] 23
Obrázek 11 - DevOps životní cyklus [28] 28
Obrázek 12 - Ukázka "větvení" kódu [45] 30
Obrázek 13 - Příklad Jenkins "pipeline" [46] 32
Obrázek 14 - Příklad chybové hlášky [47] 32
Obrázek 15 - Ukázka „high-level“ architektury [48] 35
Obrázek 16 - Automatické spouštění testů na Jenkinsu [49] 37
Obrázek 17 - Parametry na Jenkinsu [50] 39

8.2 Seznam tabulek

Tabulka 1 - Čas trvání automatizovaného běhu testů 41
Tabulka 2 - Čas trvání manuálního běhu testů 42

8.3 Seznam grafů

Graf 1 - Časový vývoj běhu testů 45
Graf 2 - Vývoj počtu odhalených defektů 46