Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Automatické testování softwaru

Adam Pechar

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Adam Pechar

Informatika

Název práce

Automatické testování softwaru

Název anglicky

Automated software testing

Cíle práce

Bakalářská práce se věnuje automatickému testování softwaru se zaměřením na webové aplikace. Hlavním cílem bakalářské práce je analyzovat zvolený testovací scénář z hlediska možností zavedení automatizace, navrhnout a implementovat postup automatizace a porovnat oba přístupy.

Dílčí cíle práce jsou:
- analyzovat postupy automatizace testování software na základě studia odborných informačních zdrojů
- definovat testovací scénář a analyzovat ho s ohledem na možnosti vhodného zavedení automatizace
- navrhnout a implementovat postup automatizace testování a ověřit jeho přínosy na základě porovnání s původním testovacím scénářem

Metodika

Metodika teoretické části bakalářské práce je založena na studiu odborných informačních zdrojů v oblasti automatizace testování software. Metodika praktické části spočívá v definování konkrétního testovacího scénáře a určení dílčích částí testovacího postupu vhodných pro zavedení automatizace. Navržená automatizace bude validována na základě srovnání obou způsobu provedení testování. Sjednocením poznatků teoretické části a výsledků praktické části budou formulovány závěry bakalářské práce.

Doporučený rozsah práce

40-50

Klíčová slova

Automatické testování, manuální testování, webová aplikace, software, uživatelské scénáře

---

Doporučené zdroje informací

Adam Roman – Study Guide to the ISTQB (R) Foundation Level 2018 Syllabus. ISBN13: 9783319987392

Arnon Axelrod – Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects, 2018. ISBN-13: 978-1484238318.

Matt Stephens; Doug Rosenberg – Testování softwaru řízené návrhem, 2015. ISBN: 978-80-251-3607-2.

Miroslav Bureš; Miroslav Renda; Michal Doležel; kolektiv – Efektivní testování softwaru, Grada Publishing, a. s, 2016, ISBN: 978-80-247-5594-6

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jan Pavlík

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 16. 8. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 5. 10. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 11. 03. 2022

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Automatické testování softwaru" 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 30.11.2022 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Janu Pavlíkovi za čas, který mi v rámci konzultací věnoval, za jeho cenné rady k vypracování této bakalářské práce a za jeho vstřícný přístup.

Automatické testování softwaru

Abstrakt

Bakalářská práce se věnuje tvorbě automatických testů pro zvolenou webovou aplikaci, kterou vyvíjí softwarová firma. Automatické testy pro webovou aplikaci jsou vytvořené pomocí nástroje Cypress. Jednotlivé testy jsou realizovány na základě vytvořeného testovacího scénáře. Na základě testovacího scénáře bude realizováno manuální a automatické testování. Samotná práce se věnuje jednotlivým typům testování, procesu zavádění automatických testů a jejich přínosu. Jsou zde vysvětlené důležité pojmy, které se vážou k problematice testování. Současně se práce zabývá výhodami a nevýhodami obou přístupů testování a vzájemně je mezi sebou porovnává. Popisují se zde případy, kdy automatizace naopak není přínosná nebo ji nelze zavádět. Dále práce popisuje metriky a návratnost automatického testování. V neposlední řadě je zde popis jednotlivých nástrojů pro automatické testování, jsou popsány jednotlivá kritéria, která slouží pro výběr vhodného nástroje v závislosti na typu projektu. V teoretické části jsou stanovené definice, metody, principy a úskalí pro celou oblast testování. Praktická část se zabývá výběrem vhodného testovacího nástroje, vytvoření testovacích scénářů a následné realizaci obou přístupů testování. Na závěr jsou porovnány jednotlivé rozdíly mezi manuálním a automatickým testováním.

Klíčová slova: Automatické testování, manuální testování, webová aplikace, software, testovací scénáře

Automated software testing

Abstract

The bachelor's thesis focuses on the creation of automatic tests for a selected web application developed by a software company. Automatic tests for the web application are created in Cypress. Individual tests are implemented based on the created test cases. Both manual and automated testing will take place through these test scenarios. The work itself deals with individual types of testing, the process of implementing automatic tests and their benefits. Important concepts related to testing are explained. At the same time, the work handles the advantages and disadvantages of both approaches to testing and compares them with each other. It is necessary to determine whether automation is possible for each case and if the benefits of automation are worth the effort based on measured metrics. Last but not least, there is a description of individual tools for automatic testing and describes the concrete criteria which are used to select the appropriate one depending on the specific project. The theoretical part sets out the definitions, methods, principles and pitfalls of the whole industry of testing. The practical part deals with choosing a suitable testing tool, the creation of test scenarios and the subsequent implementation of both testing approaches. Finally, the individual differences between manual and automatic testing are compared.

Keywords: Automated testing, manual testing, web application, software, test scenarios

Obsah

1 Úvod **13**

2 Cíl práce a metodika **14**

2.1 Cíl práce 14

2.2 Metodika 14

3 Teoretická východiska **15**

3.1 Softwarový vývoj 15

3.1.1 Systémový software 15

3.1.2 Programovací software 15

3.1.3 Aplikační software 15

3.2 Životní cyklus vývoje softwaru 16

3.2.1 Plánování a brainstorming 17

3.2.2 Analýza požadavků a proveditelnosti 17

3.2.3 Design 17

3.2.4 Vývoj 18

3.2.5 Testování 18

3.2.6 Nasazení 18

3.2.7 Údržba 18

3.3 Metody vývoje softwaru 19

3.3.1 Waterfall 19

3.3.2 Agile 20

3.3.3 V model 21

3.4 Testování 22

3.5 Důvody testování 23

3.6 Typy testů podle způsobu provedení 24

3.6.1 Manuální testování 24

3.6.2 Automatické testování 24

3.7 Typy jednotlivých testů 25

3.7.1 Explorativní testování 25

3.7.2 Systémové testy 25

3.7.3 Akceptační testy 26

3.7.4 Funkční testování 26

3.7.5 Nefunkční testování 26

3.7.6 Testování zabezpečení 26

3.7.7 Testovací pyramida 27

3.7.8 Jednotkové testy 28

3.7.9 Integrační testy 29

3.7.10 End-to-End testování 29

3.7.11 Black box 30
3.7.12 White box 30
3.8 Automatické testování 30
3.8.1 Podmínky pro automatizaci 32
3.8.2 Neefektivita automatizace 32
3.8.3 Výhody automatizace 33
3.8.4 Vhodnost použití automatizace 33
3.9 Měřitelnost a metriky automatizace 33
3.9.1 Návratnost automatizace 34
3.9.2 Pokrytí testovacích scénářů automatickými testy 34
3.10 Proces automatického testování 34
3.10.1 Rozsah automatizace 35
3.10.2 Výběr testovacího nástroje 35
3.10.3 Plánování a nastavení infrastruktury 35
3.10.4 Vývoj automatických testů 36
3.10.5 Spuštění a analýza výsledků 36
3.10.6 Údržba a podpora 36
3.11 Kritéria pro výběr automatického nástroje 36
3.11.1 Porozumění požadavků projektu 37
3.11.2 Identifikace klíčových vlastností nástroje pro automatizaci 37
3.12 Konkrétní nástroje pro automatické testování 38
3.12.1 Cypress 38
3.12.2 Puppeteer 39
3.12.3 Selenium 39
3.12.4 Playwright 39

4 Vlastní práce 40
4.1 Popis zvolené aplikace 40
4.2 Výběr testovacího nástroje 41
4.2.1 Testovací software pro automatizaci 42
4.3 Předpoklady testovacích případů pro automatizaci 42
4.4 Testovací scénář 43
4.4.1 Detail testovacího scénáře 44
4.4.2 Jednotlivé kroky testovacího scénáře 45
4.5 Vývoj automatických testů 47
4.5.1 Nastavení počátečních podmínek 48
4.5.2 Test průchodu horní navigace 48
4.5.3 Vytvoření nových metod 49
4.5.4 Cypress Test Runner 50
4.5.5 Využití selektorů 51
4.5.6 Test formuláře 52

4.5.7 Test stránky zadávací dokumentace 53
4.6 Porovnání obou přístupů testování 54
4.6.1 Výhody a nevýhody manuálního testování 55
4.6.2 Průběh manuálního testování 55
4.6.3 Výhody automatického testování 55
4.6.4 Nevýhody a překážky automatického testování 56
4.6.5 Nemožnost automatizovat určité úkony 56
4.6.6 Průběh automatického testování 57

5 Výsledky a diskuze 58
5.1 Výsledky manuálního testování 58
5.2 Výsledky automatického testování 59
5.3 Metriky a návratnost automatizace 60
5.3.1 Pokrytí testovacích scénářů automatizací 60
5.3.2 Rychlost běhu automatických testů 61
5.3.3 Návratnost automatizace 61
5.4 Shrnutí automatizace a zhodnocení výsledků 61

6 Závěr 63

7 Seznam použitých zdrojů 64

Seznam obrázků

Obrázek 1 – Schéma životního cyklu vývoje softwaru, (Josh 2019) 16
Obrázek 2 – Vodopádová metodika, (Trunkett 2020) 19
Obrázek 3 – Proces agilní metodiky, (Trunkett 2020) 20
Obrázek 4– Metodika V model, (Sami 2021) 22
Obrázek 5 - testovací pyramida, (Knott 2015) 28
Obrázek 6 – Black-box, White-box (Kumar 2012) 30
Obrázek 7 – proces automatického testování (Arumughom 2018) 35
Obrázek 8 – Úvodní část testovacího scénáře, zdroj autor 44
Obrázek 9 – Detailní popis testovacího scénáře, zdroj autor 45
Obrázek 10 – Ukázka before hook, zdroj autor 48
Obrázek 11 – Test horní navigace, zdroj autor 49
Obrázek 12 – Zavedení metody, zdroj autor 50
Obrázek 13 – Využití definované metody, zdroj autor 50
Obrázek 14 – Běh testu v Test runner, zdroj autor 51
Obrázek 15 – Best practicies v rámci selektorů, (Cypress.io nedatováno) 52
Obrázek 16 – Test formuláře, práce autora 53
Obrázek 17 – Test zadávací dokumentace, práce autora 54
Obrázek 18 – Výsledek manuálního testování, práce autora 58
Obrázek 19 – Výsledek automatického testování, práce autora 59

Seznam tabulek

Tabulka 1 – Výsledek manuálního testování, zdroj autor 59
Tabulka 2 – Výsledek automatického testování, zdroj autor 60

1 Úvod

V dnešní hektické době je kladen důraz na urychlení veškerých procesů za co nejkratší dobu s vynaložením co nejmenších nákladů. Není tomu výjimkou ani oblast testování softwaru. Zatímco nároky na kvalitu softwaru se zvyšují, tak doba pro testování zůstává stejná nebo se dokonce zmenšuje. Jakýkoliv proces vyžaduje kontrolu, zda se očekávání shodují s reálnou podobou. Celý proces vytváření nového softwarového produktu není vůbec jednoduchá záležitost. Je zapotřebí správné plánování a podrobná analýza. Testování softwarového produktu je nedílnou součástí celého procesu vývoje softwaru, bez které by nebylo možné úspěšně dokončit celý vývoj. Moderní přístup automatizace se týká nejenom odvětví informačních technologií, ale vzniká tendence automatizovat co největší kvantum opakujících se aktivit.

Zavedení automatizace v oblasti testování softwaru je proces, který vyžaduje značné množství času. Z počátku je potřeba investovat čas pro správné nastavení testovacího prostředí, určení rozsahu automatizace, volbu vhodného testovacího nástroje a také pro vytvoření samotných testů. Pokud jsou správně nastavené a dodržené veškeré procesy, tak se investice do automatických testů postupem času začne vyplácet. V opačném případě, kdy automatizace probíhá bez stanoveného procesu, tak existuje větší míra pravděpodobnosti, že se tato investice nikdy nevyplatí. Je nutné podotknout, že prvopočáteční sestavení automatických testů bývá časově nejnáročnější, nicméně je důležité postupovat tak, aby vytvořené testy bylo možné použít znovu a modifikovat již existující řešení podle potřeb, tudíž se časová náročnost pro vytvoření nových testů časem značně zmenší a návratová hodnota se naopak navýší. Důležité je podotknout, že automatické testování nemá zcela nahradit ruční testování, ale snížit testovací případy, které se provádí opakovaně a získat co nejvíce pomocí kombinace obou přístupů.

2 Cíl práce a metodika

2.1 Cíl práce

Bakalářská práce se věnuje automatickému testování softwaru se zaměřením na webové aplikace. Hlavním cílem bakalářské práce je analyzovat zvolený testovací scénář z hlediska možností zavedení automatizace, navrhnout a implementovat postup automatizace a porovnat oba přístupy.

Dílčí cíle práce jsou:

- analyzovat postupy automatizace testování software na základě studia odborných informačních zdrojů
- definovat testovací scénář a analyzovat ho s ohledem na možnosti vhodného zavedení automatizace
- navrhnout a implementovat postup automatizace testování a ověřit jeho přínosy na základě porovnání s původním testovacím scénářem

2.2 Metodika

Metodika teoretické části bakalářské práce je založena na studiu odborných informačních zdrojů v oblasti automatizace testování software. Metodika praktické části spočívá v definování konkrétního testovacího scénáře a určení dílčích částí testovacího postupu vhodných pro zavedení automatizace. Navržená automatizace bude validována na základě srovnání obou způsobu provedení testování. Sjednocením poznatků teoretické části a výsledků praktické části budou formulovány závěry bakalářské práce.

3 Teoretická východiska

3.1 Softwarový vývoj

Softwarový vývoj je proces, který zahrnuje soubor činností v oblasti počítačové vědy. Obsahuje několik procesů, mezi tyto procesy se řadí návrh, specifikace, analýza, dokumentace, programování, testování, nasazení a následná podpora softwaru. Účel vývoje je zhotovit požadovaný software, který uspokojí potřebu skupiny cílových uživatelů. Software jako takový je soubor instrukcí nebo programů, které říkají počítači, co dělat. Dělí se na tři základní typy. (IBM research 2014; IBM nedatováno)

3.1.1 Systémový software

Systémový software je označení pro druh počítačového programu, který se používá pro spuštění hardwarových a aplikačních programů počítače. Slouží jako prostředník mezi hardwarem a ostatními programy počítače. Operační systém je nejznámějším příkladem systémového softwaru, spravuje všechny ostatní programy v počítači. Systémový software se používá ke správě počítače, běží na pozadí a zachovává základní funkce počítače, tudíž uživatelé mohou k provedení určitých úkolů spouštět aplikační software vyšší úrovně. (IBM nedatováno; Lutkevich 2021)

3.1.2 Programovací software

Programovací software pomáhá programátorovi při následném vývoji dalšího softwaru. Příklady programovacího softwaru jsou: kompilátory, assemblery, debuggery, tlumočníky, textové editory atd. Integrovaná vývojářská prostředí (IDE) jsou kombinací všech těchto softwarů. (IBM nedatováno)

3.1.3 Aplikační software

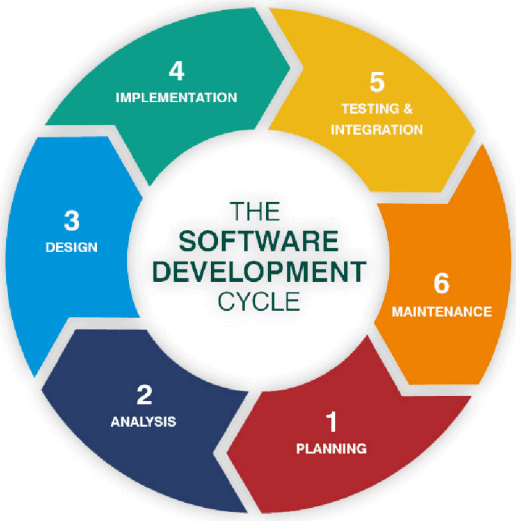
Aplikační software je definovaný jako kterýkoliv program, který je určený pro koncové uživatele a pomáhá jim provádět potřebné úkoly. V tomto smyslu lze libovolný program koncového uživatele nazývat aplikací. Příkladem jsou sady kancelářských aplikací, software pro správu dat, přehrávače médií, bezpečnostní programy, mobilní a webové aplikace. Mezi příklady aplikačního softwaru lze zařadit: Microsoft Word, Microsoft Excel, VLC media player a mnoho dalších. (IBM nedatováno; Lutkevich 2020)

3.2 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru (SDLC, což znamená software development life cycle) je nepostradatelnou součástí každého projektu, jenž je zaměřen na vývoj softwaru. Pro tento pojem je charakteristická návaznost každé etapy v procesu vývoje. Celý vývoj by měl mít předem definované časové a finanční rozpětí, které bývá pro tento vývoj klíčový. Dodržování procesu SDLC vede k systematickému a disciplinovanému vývoji softwaru, který zaručuje jeho hodnotu. Účelem tohoto procesu je dodávat vysoce kvalitní produkt, který odpovídá požadavkům potřeb zákazníka. Jednotlivé etapy si můžeme představit jako části mozaiky. Každá část mozaiky má určité vlastnosti, bez dokončení jedné části mozaiky nelze přejít na další část. Celý tento cyklus se dá rozdělit do několika fází, tyto fáze nejsou striktně definovány, jelikož se mohou lišit v závislosti na potřebách projektu jako takových:

1. Plánování a brainstorming
2. Analýza požadavků a proveditelnosti
3. Design
4. Vývoj
5. Testování
6. Nasazení
7. Údržba

(Řepa 1999; Josh 2019)



Obrázek 1 – Schéma životního cyklu vývoje softwaru, (Josh 2019)

3.2.1 **Plánování a brainstorming**

Brainstorming je první krok procesu ve vývoji softwaru. Všechno to začíná nápadem, ale každý nápad musí být pečlivě promyšlen, aby mohl být implementován. Tato fáze SDLC znamená získání informací od všech zúčastněných stran, včetně zákazníků, odborníků z oboru a vývojářského týmu. Pečlivé plánování je počáteční a jednou z hlavních fází vývoje softwaru, protože předpokládá určení rozsahu a proveditelnost daného projektu. Definuje se způsob, jak nový systém splňuje strategické cíle, dostupnost zdrojů, problémy spojené s náklady a časové možnosti. Teprve po vypracování plánu do podrobných detailů, tak lze přejít k dalšímu kroku. (Dziuba nedatováno; Altvater 2020; Josh: 2019)

3.2.2 **Analýza požadavků a proveditelnosti**

Během této fáze procesu je projekt podrobně definován a provádí se analýza projektu. Celý koncept je představen vývojovému týmu, aby lépe porozuměl cíli projektu a objasnily vzniklé otázky. Vývojářský tým by měl určit proveditelnost projektu a způsob, jakým se bude projekt úspěšně a efektivně implementovat. Analýza proveditelnosti zobrazuje všechny technické a ekonomické aspekty, které ovlivňují proces vývoje projektu: čas, náročnost provedení, náklady, úkoly a odhady zapojení členů týmu v jednotlivých fázích vývoje. Analýza požadavků také pomáhá při identifikaci rizik v počáteční fázi projektu. Strategii snižování rizik lze vypracovat od samého začátku. Přehledně strukturovaná dokumentace zajišťuje lepší spolupráci napříč celým procesem.

3.2.3 **Design**

Tato etapa SDLC začíná přeměnou specifikace na plán návrhu. Vytváří se vlastní konceptualizace řešení, což znamená podrobná softwarová architektura, která splňuje stanovené požadavky. Během této fáze je vytvořena celá struktura projektu s konečným prototypem pro další fáze projektu. Jedná se o druh vizuálního modelování, počínaje funkčností řešení, až po definování základních komponent. Jsou stanovené akceptační kritéria, která jsou dále členěna, aby bylo jasně stanovené chování jednotlivých prvků ve všech situacích. Všechny zúčastněné strany poté tento plán zkontrolují a poskytnou zpětnou vazbu. Nedokončení této fáze způsobí téměř jistě překročení finančních nákladů nebo dokonce konec projektu. (Dziuba nedatováno; Altvater 2020; Josh 2019)

3.2.4 **Vývoj**

V této fázi začíná skutečný vývoj, jedná se o převod projektové dokumentace na skutečný software v rámci procesu vývoje. Tato část celého cyklu SDLC je obecně nejdelší, protože je páteří celého procesu. Programátoři začínají psát kód podle analyzovaných požadavků. Musí dodržovat určité pokyny, aby kód splňoval specifikace, které byly stanovené jejich organizací a programovacími nástroji. Už během této fáze probíhá základní testování, jelikož je nezbytné odchytávat vzniklé bugy již v rané fázi vývoje. (Dziuba nedatováno; Altvater 2020)

3.2.5 **Testování**

Software se předá testovacímu týmu po dokončení vývoje. Testovací tým neboli QA (Quality assurance) se pustí do testování celého softwarového systému. Testování se provádí z důvodu, aby bylo ověřené, že se implementovaly všechny požadavky správně a našla se většina chyb ještě předtím, než se projekt dostane do produkční fáze. QA tým provádí celou řadu testů, které zahrnují například funkční, systémové, designové, průzkumné, bezpečnostní testování a mnoho dalších typů testů. Pokud během tohoto procesu tester objeví nějakou chybu, tak ji nahlásí vývojářskému týmu. Poté, co developerský tým tuto chybu opraví, tester znovu zkontroluje, že chyba byla skutečně opravena a zároveň se neprojevila s touto opravou chyba nová. Tento proces pokračuje do té doby, dokud nejsou implementovány všechny požadavky a software neobsahuje žádné znatelné chyby. (Dziuba nedatováno; Altvater 2020; Josh: 2019)

3.2.6 **Nasazení**

V této fázi je cílem nově vytvořený a otestovaný software nasadit do produkčního prostředí. Některé organizace finální nasazení uskutečňují ve více fázích v závislosti na obchodní strategii. Během této strategie je z počátku dostupná pouze testovací verze produktu, což umožňuje zachytit většinu chyb a zapracovat návrhy na vylepšení ještě před vydáním finální verze produktu. (Altvater 2020; Josh 2019)

3.2.7 **Údržba**

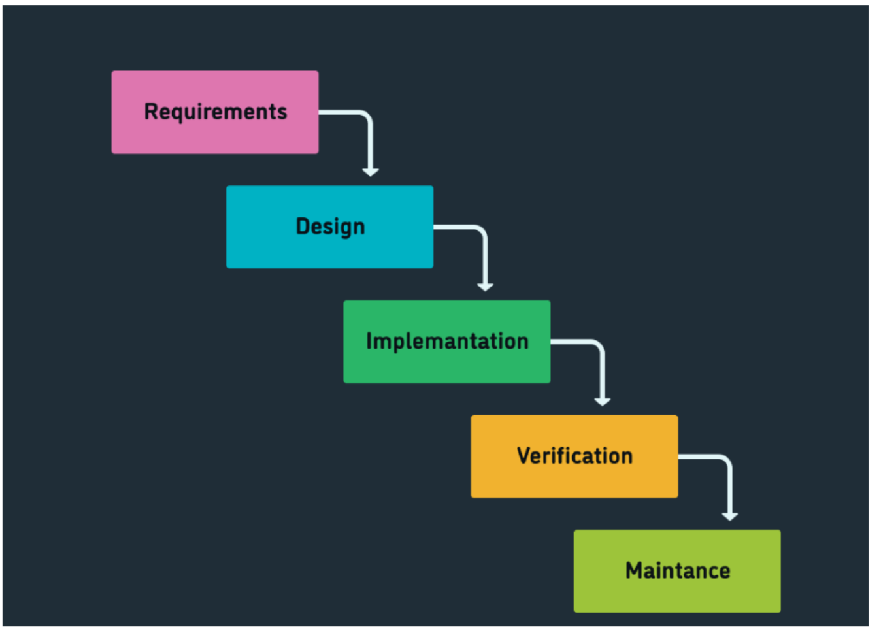
Poslední fáze životního cyklu vývoje softwaru zahrnuje údržbu a pravidelné aktualizace. Této fázi se věnuje maximální pozornost, jelikož se během této fáze získává zpětná vazba od uživatelů. Během této fáze se opravují nalezené chyby a přidávají se nové funkce. Zvýšená pozornost se věnuje zajištění, že systém bude nadále fungovat dle stanovených požadavků, které byly sjednané napříč celým SDLC. (Josh 2019)

3.3 Metody vývoje softwaru

Metody SDLC znázorňují způsob, jakým se bude postupovat během celého procesu vývoje. Každá metoda má své výhody i nevýhody a současně je každá metoda vhodná pro odlišný typ projektu. Zkušený projektový manažer by měl mít dostatečné znalosti, aby byl schopný na základě kontextu a požadavků vybrat vhodný model SDLC, což zajistí efektivní a úspěšný průběh celého procesu vývoje. (Trunkett 2020)

3.3.1 Waterfall

Vodopádový model je způsob vývoje softwaru, který má již na začátku stanovené kompletní zadání, termíny a cenu. Tento model zahrnuje pevnou strukturu, která vyžaduje, aby byly všechny systémové požadavky definovány na samém začátku projektu. Jedná se o lineární sekvenční průběh, který vytváří takzvaný tok (jako vodopád) v postupné fázi implementace. To znamená, že jakákoli fáze procesu vývoje začíná pouze v momentu, je-li předchozí fáze dokončena. Tento přístup nedefinuje proces návratu do předchozí fáze. Klíčovou roli hraje přesná dokumentace a definování všech scénářů, které nastanou během SDLC. Jakmile je vývoj dokončen, produkt je testován podle původních požadavků a až následně je přiřazeno případné přepracování. Vodopádový přístup je nejstarší ze všech metodik. Společnosti v softwarovém průmyslu obvykle potřebují větší flexibilitu, kterou tato metodika bohužel nenabízí, nicméně vodopádová metoda stále zůstává využívaným řešení pro určité typy projektů. (Trunkett 2020; Sami 2021)



Obrázek 2 – Vodopádová metodika, (Trunkett 2020)

Výhody

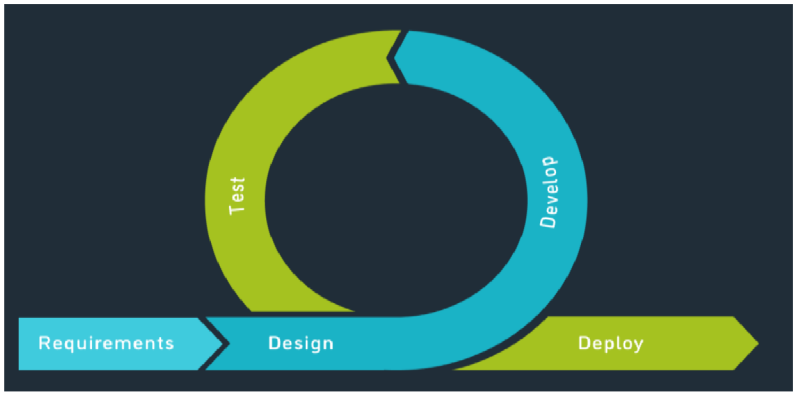
- Každá fáze má specifické výstupy
- Předem stanovený fixní plán
- Snadnější predikce rozsahu projektu, ceny a postupu
- Lépe měřitelné, jestli se vývoj časově zpožďuje, či je v předstihu

Nevýhody

- Pěvně stanovený rozsah před začátkem vývoje přináší riziko, že bude nutné po dokončení vývoje dodělat nalezené nesrovnalosti
- Velice obtížné provádět změny v průběhu
- Absence dostatečné zpětné vazby od všech zúčastněných stran v průběhu
- Je nezbytné mít připravený detailní návrh již na začátku projektu

3.3.2 **Agile**

Agilní metodiky jsou opakem vodopádového přístupu. Flexibilní metody, které umožní projekt přizpůsobit během vývoje a reagovat na změny požadavků v průběhu. Během této metody se software vyvíjí po iteracích, tzv. sprintech, které většinou trvají dva až čtyři týdny, ve kterých týmy řeší hlavní potřeby svých zákazníků a provádějí testování za pochodu. Tato metodika zahrnuje neustálou spolupráci mezi zúčastněnými stranami a dochází ke kontinuálnímu zlepšování v každé fázi. Pomocí rychlé iterace v jednotlivých sprintech lze dosáhnout funkční verze software ještě předtím, než je dokončený celý vývoj. Agilní neboli iterativní metodiky nesou označení pro celou skupinu metodik, do které patří například Scrum nebo Kanban. Scrum se řadí mezi nejpopulárnější agilní metodiku, díky tomu se většinou používá jako synonymum pro celé uskupení agilních metod. (Sami 2021; Arnon 2018)



Obrázek 3 – Proces agilní metodiky, (Trunkett 2020)

Výhody

- Schopnost provést rychlé změny v zadání na základě vzniklé situace
- Snižuje se riziko na selhání celého projektu díky flexibilitě daného přístupu
- Softwarový produkt maximalizuje přínos pro uživatele
- Konečný výsledek je kvalitní software v co nejkratším možném čase
- Komunikace a neustálé vstupy od zákazníků, které předcházejí potencionálním problémům v dalším vývoji

Nevýhody

- Dokumentace se dokončuje v pozdější fázi
- Nutná spolupráce ze strany zákazníka
- Zákazník musí mít základní technické znalosti a také rozumět této metodice vývoje
- Složitější odhad, kdy bude softwarový produkt dokončen a kolik bude potřeba financí (Sami 2021)

3.3.3 V model

Jedná se o evoluci klasické metodiky vodopádu. Kroky procesu SDLC jsou po fázi kódování převráceny nahoru. Tento model se označuje jako verifikace a validace. V model má velmi přísný přístup, přičemž další fáze začíná až po dokončení předchozí fáze. Namísto lineárního pohybu dolů se kroky procesu po fázi implementace a kódování ohnou nahoru, aby vytvořil typický tvar V. Zásadní rozdíl oproti klasickému vodopádu spočívá v raném plánování testů. Testování začíná společně s vývojem, každá fáze vývoje má odpovídající fázi testování. Díky tomu lze najít chyby ve správnou chvíli, což eliminuje nadbytečné náklady.

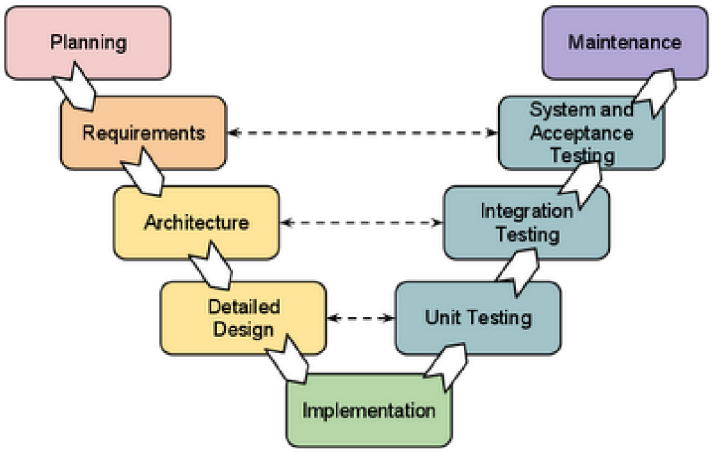
Výhody

- Jednoduché a snadné použití
- Každá fáze má specifické výstupy
- Vyšší šance na úspěch než u vodopádového modelu (díky vývoji testování již na začátku životního cyklu)
- Ověření a validace produktu v raných fázích vývoje

Nevýhody

- Není flexibilní, stejně jako vodopádová metoda
- Nastavení rozsahu je obtížné a nákladné
- Software se vyvíjí během fáze implementace, tudíž nevznikají žádné prototypy softwaru
- Model neposkytuje jasnou cestu k problémům zjištěným během fází testování

(Sami 2021)



Obrázek 4– Metodika V model, (Sami 2021)

3.4 Testování

Testování softwaru lze považovat za disciplínu, která je bohatá svou rozmanitostí. Jedno ze základních pravidel ISTQB (mezinárodní certifikační rada potvrzující znalosti z oblasti testování software) říká, že testování je závislé na kontextu. Rozdílný software budeme testovat odlišným způsobem, pokud se jedná o bankovní systém, mobilní aplikaci či řídící software dopravního letadla. (Bureš 2016)

Nutné zdůraznit, že ve své podstatě téměř každý IT projekt je nezbytné testovat. Testování má ověřit, zda softwarový produkt splňuje zadané požadavky. Vede se diskuze mezi stakeholdery (zainteresovaná strana) o tom, do jaké míry a detailů bude software testován.

Softwarové systémy jsou každodenní součástí našeho života. Od obchodních aplikací (např. bankovnictví, investice) až po spotřebitelské produkty (např. automobily). Drtivá většina lidí má zkušenost se softwarem, který nefungoval podle jejich očekávání. Software, který nepracuje správně může způsobit spoustu problémům, včetně ztráty peněz, času, firemní pověsti nebo dokonce zranění, či v krajním případě i smrti. (Roman 2018)

Celý tento proces lze připodobnit ke stavbě domu. Je zapotřebí architekt, zedníci (vývojáři), stavební dozor (testeři). Architekt sestaví plán, jak bude dům ve finální podobě vypadat. Zedníci začnou podle tohoto plánu stavět a stavební dozor provádí kontrolu, zda stavba odpovídá požadavkům a zadání od architekta, aby se dodržely původní kritéria podle návrhu. (Kitner nezadáno)

Testování softwaru je způsob, jak posoudit kvalitu softwaru a snížit riziko softwarového selhání při provozu. Běžný, ačkoliv nesprávný způsob vnímání testování je, že se skládá výhradně ze spouštění testů a kontroly výsledků. Testování softwaru je proces, který zahrnuje velké kvantum činností. Provedení testů (včetně kontroly výsledků) je pouze jednou z mnoha činností. Proces testování také zahrnuje činnosti, jako je plánování testů, testovací a riziková analýza, návrh a implementace testů, bezpečnostní testování, hlášení průběhu a výsledky testů, hodnocení kvality testovaného objektu.

Dále existují různé způsoby testování, které lze rozdělit například na dynamické a statické testování. Testování jednotlivých součástí a následně celého systému se říká dynamické testování, které se dělí dle kritéria, zda je ověření funkčnosti na struktuře White box nebo na specifikaci Black box (tester nezná vnitřní strukturu). Opakem dynamického testování je statické, které nevyžadují běh softwaru a využívají se zejména v počáteční fázi SDLC, kdy se čeká na funkční prototyp. (Roman 2018)

Testování zahrnuje kontrolu, zda systém splňuje stanovené požadavky, ale také ověření, zda daný systém ve svém prostředí splňuje potřeby uživatelů a dalších zúčastněných stran. Získané výsledky z provedených testů jsou reportovány a následně poskytnuty všem potřebným skupinám v rámci vývoje softwarového produktu. Testovací proces napřímo ovlivňuje vývojářský tým, který musí komunikovat důležité informace. (Roman 2018)

3.5 Důvody testování

Je důležité objasnit, proč je testování důležité. V historii výpočetní techniky se několikrát stalo, že softwarový systém byl uveden do provozu s přítomností nejedné vady, které způsobily značné potíže. Pro testování existuje několik důvodů:

1) Použití vhodných testovacích technik může snížit frekvenci těchto problematických dodávek, pokud jsou tyto techniky aplikovány s odpovídající úrovní testovacích

znalostí, které odpovídají příslušným testovacím úrovním v náležité fázi životního cyklu softwarového vývoje.

2) Ovlivnění pozice firmy dodávající software. Jakmile firma dodává software, který obsahuje chyby, tak se její pozice na trhu znatelně zhorší.
3) Identifikace defektů v počáteční fázi vývoje snižuje celkové náklady na odstranění této chyby. Testovací tým při návrhu systému úzce spolupracuje s návrháři systémů, což vede ke zlepšení porozumění daného návrhu a jeho testování, díky čemuž se snižuje riziko zásadních konstrukčních vad.
4) V případě, že se nalezne chyba až v produkční verzi, náklady na její opravu jsou mnohonásobně vyšší. Oproti tomu, když je chyba nalezena již během vývoje, tak její oprava probíhá rychleji a za mnohem menší náklady, než by tomu bylo v produkci.
5) Jakmile se provede důkladně celý testovací proces již během vývoj SDLC, tak se snižuje pravděpodobnost, že by dodávaný software obsahoval fatální chyby, které by mohly způsobit pozastavení chodu softwarového produktu.

(Roman 2018; Sharma nezadáno)

3.6 Typy testů podle způsobu provedení

Existuje mnoho způsobů, podle kterých lze testy členit. Jeden ze základních způsobů je dělení na základě realizace jednotlivých testů.

3.6.1 Manuální testování

Manuální testování je považováno za nejpraktičtější typ testování a v určitém okamžiku ho používá každý tým. Jedná se o testování, které provádí přímo tester krok za krokem, na základě stanovených testovacích případů (testcase), bez použití testovacího skriptu. V současné době je tendence velkou část těchto manuálních testů nahradit automatickými testy, přesto má manuální testování své nepostradatelné místo v určitých oblastech. Nevýhoda je v tom, že manuální způsob testování je pomalé a časově náročné. (Kinsbruner 2019; Kitner nezadáno)

3.6.2 Automatické testování

Automatizace veškerých procesů je trend moderní doby, stejně tomu je i v testování, kde za posledních několik let nastává enormní nárůst. Automatické testování využívá programovací jazyky, skripty a specializované nástroje k automatizaci procesu testování softwaru. Jeho hlavní

cíl je automatizovat zaběhlé testovací procesy, které se opakují dokola, čímž se ušetřit náklady i čas. Pomáhá testerům provést více testování a zlepšit celkové pokrytí testů. Na druhou stranu dosáhnout automatizace manuálních testů není vždy jednoduché. Zavedení automatizace do běhu může být časově náročné, vyžaduje to údržbu a současně jsou potřeba určité znalosti pro zavedení automatických testů. (Kinsbruner 2019; Kitner nezadáno)

3.7 Typy jednotlivých testů

Každý typ testování má své důvody, vlastnosti, výhody a nevýhody. Testování lze rozdělit do několika úrovní. Každá úroveň testu představuje instanci testovacího procesu, sestávající se ze specifických postupů v návaznosti na úrovni vývoje softwaru. Pro každou úroveň testování je vyžadováno vhodné testovací prostředí. Např. při akceptačním testování je ideální testovací prostředí podobné tomu produkčnímu, zatímco při testování jednotlivých komponent využívají vývojáři obvykle vývojové prostředí, které se může od toho produkčního lišit.

3.7.1 Explorativní testování

Jedná se o druh manuálního testování, které je do určité míry náhodné a nedrží se stanoveného testovacího scénáře. Rozdíl je ve způsobu a záměru provedení. Explorativní testování je neformální testování, při kterém testovací tým navrhuje a provádí testy na základě svých zkušeností a výsledků přechozích testů. Cílem tohoto testování je prozkoumat aplikaci a vyhledat co největší množství chyb, testy se neprovádějí podle předem stanoveného scénáře, ale mají prozkoumat, jak se daná aplikace chová v nahodilých situacích. Někdy se může stát, že během tohoto testování může být odhalená hlavní závada nebo selhání celého systému. Během tohoto testování je potřeba sledovat, jakým způsobem se postupovalo, aby se daná chyba podařila znovu navodit. (Roman 2018)

3.7.2 Systémové testy

Systémové testy se zaměřují na chování a možnosti celého testovacího systému, často s ohledem na úkoly typu end-to-end, které softwarový systém může provádět. Současně se zaměřuje na nefunkční chování, které při provádění těchto úkonů může systém vykazovat. Systém se testuje jako celek, nikoliv jako jednotlivé dílčí části, tento druh testování se provádí v pozdější fázi SDLC, jelikož je zapotřebí, aby sytém byl funkční jako celek. Pro tento druh testování je předem vytvořený testovací scénář. Hlavní úkol je prověřit správnost fungování dle požadovaných kritérií, očekávání zákazníka a stanovených specifikací projektu. (Roman 2018)

3.7.3 **Akceptační testy**

K tomuto druhu testování dochází v případě, kdy se během předchozích fází testování nadále nevyskytují žádné chyby. Akceptační testování pomáhá prověřit kvalitu aplikace, která je klíčová k plnohodnotnému užívání. Umožní jasně stanovit, jaké jsou požadavky na aplikaci, co všechno se od ní očekává a jaké jsou cíle uživatelů, kteří s danou aplikací pracují. Tento druh testu většinou provádí samotný klient a ověřuje, zda průchod systému odpovídá obchodním požadavkům. Prověřuje se, zda se vše nastavilo podle stanovené dokumentace, která splňuje potřeby pro koncového uživatele. Jedná se o nejvyšší úroveň testování. Po provedení všech testů se reportují nalezené nesrovnalosti mezi specifikací a samotnou aplikací dodavateli. Poté, co jsou chyby opraveny, tak proběhne znovu akceptační testování. Zákazník přijímá software pouze tehdy, pokud softwarový produkt funguje podle očekávání. (Roman 2018)

3.7.4 **Funkční testování**

Funkční testování systému zahrnuje testy, které vyhodnocují funkce, jenž by měl systém provádět. Funkční testy by měly být prováděny na všech testovacích úrovních (např. testy pro jednotlivé komponenty, celé sekce, systémy). Testují se jednotlivé vstupy a výstupy. Funkční testování zohledňuje chování samotného softwaru.

3.7.5 **Nefunkční testování**

Nefunkční testování vyhodnocuje charakteristiky softwarového systému jako je použitelnost, účinnost výkonu nebo zabezpečení. Tento způsob testování zkoumá různé vlastnosti sytému, které nesouvisejí s jeho funkcionalitou, ale naopak se testují nefunkční parametry aplikace. Nefunkční testování je testování "jak dobře" se systém chová, jak zvládá zátěžové situace, aby se ověřilo, zda je stabilní, bezpečný a dlouhodobě udržitelný. Na rozdíl od ostatních způsobů testování, tak lze tento způsob provádět na všech úrovních, a to co nejdříve. Odkazuje na aspekty softwaru, které nemusí souviset s konkrétní funkcí nebo akcí uživatele, jako je škálovatelnost nebo zabezpečení. Pozdní odhalení nefunkční chyby může způsobit velké problémy, které se často složitě opravují. (Roman 2018)

3.7.6 **Testování zabezpečení**

Testování zabezpečení je druh testování softwaru, který se řadí pod nefunkční testování. Má za úkol odhalit zranitelnost, hrozby a rizika softwaru, které zabrání škodlivým útokům ze strany vetřelců. Hlavní cíl bezpečnostních testů spočívá v identifikaci všech možných mezer a

slabin softwarového systému, odhalit potenciální hrozby a zranitelnosti, aby systém nepřestal fungovat nebo jej nebylo možné zneužít. V případě nesprávného bezpečnostního testování, případně neprovedení tohoto typu testování může dojít ke ztrátě informací, finančních příjmů, správné funkčnosti a pověsti celé organizace. Existuje sedm hlavních typů v rámci testování zabezpečení podle metodické příručky Open Source Security Testing.

- Skenování zranitelnosti: Tento druh testování se provádí pomocí automatizovaného softwaru pro skenování systému proti známým signaturám zranitelnosti.
- Bezpečnostní skenování: Zahrnuje identifikaci slabých míst sítě, případně systému a poskytuje řešení pro snížení těchto hrozeb. Skenování tímto způsobem lze provést ručním i automatickým skenováním.
- Penetrační testování: Testování tohoto typu simuluje útok škodlivého hackera. V rámci penetračních testů je součástí analýza konkrétního systému pro kontrolu potenciálních zranitelností vůči externímu pokusu o hackování.
- Posouzení rizik: Toto testování zahrnuje analýzu bezpečnostních rizik pozorovaných v organizaci. Jednotlivá rizika jsou rozdělena na základě závažnosti na nízká, střední a vysoká. V rámci tohoto testování dochází k doporučení pro kontrolu a poskytnutí opatření pro snížení rizik.
- Bezpečnostní audit: Jedná se o interní kontrolu aplikací a operačních systémů z hlediska bezpečnostních chyb. Způsob tohoto auditu lze současně provést kontrolou programovacího kódu.
- Etické hackování: Jedná se o cílené hackování samotných systémů. Na rozdíl od zlomyslného hackování, které krade pro své vlastní zisky, tak etické hackování slouží k odhalení bezpečnostní chyby v systému.
- Posouzení postoje: Kombinace bezpečnostního skenování, etického hackování a hodnocení rizik, aby byl ukázán celkový stav zabezpečení dané organizace. (Hamilton 2022)

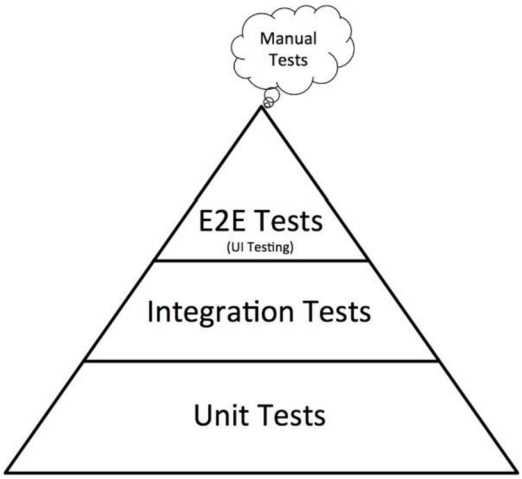
3.7.7 **Testovací pyramida**

Testovací pyramida je koncept, který může pomoci vývojářům a QA týmu vytvořit velice kvalitní software. Napomáhá vývojářům zkrátit dobu potřebnou pro identifikaci, zda zavedení příslušné změny porušuje stávající kód. Její přínos spočívá také při vytváření spolehlivější testovací sady. Testovací pyramida, označovaná také jako pyramida automatizace testování je

koncept, který strategicky seskupuje softwarové testy do různých kategorií, které by měly být zařazené do testovací sady. Zpravidla se softwarové testy v rámci tohoto přístupu řadí do tří různých úrovní, což přispívá k zajištění vyšší kvalitě, zkrácení času potřebného k nalezení hlavní příčiny chyb a vytvoření spolehlivějších testů. (Knott 2015)

Tento princip testování je rozdělen na tři následující úrovně:

- Jednotkové testy
- Integrační testy
- End-to-End testy



Obrázek 5 - testovací pyramida, (Knott 2015)

Vrchol pyramidy je určen pro pomalejší a více četné manuální testovací scénáře, které testují samotnou aplikaci na úrovni uživatelského rozhraní. Podle tohoto principu by mělo být v rámci nižšího patra větší pokrytí testů. Jednotkové testy neboli základ testovací pyramidy se provádí pouze pomocí automatických testů. Tato část nevyžaduje interakci s uživatelským rozhraním, proto tyto testy bývají zpravidla nejrychlejší. (Knott 2015)

3.7.8 **Jednotkové testy**

Testování jednotlivých softwarových komponent se nazývá Unit Testing neboli jednotkové testy, což je označení pro první testování, které se provádí po dokončení vývoje a tvoří základ testovací pyramidy. Tento druh testování obvykle neprovádí tester, ale programátor, jelikož se testování provádí na úrovni samotného kódu, tudíž je vyžadována podrobná znalost návrhu interního programu a kódu samotného. Důraz se klade na nejmenší jednotku softwarového dílku, ověření kódu a jeho funkčnosti, kde se vývojář soustředí na ukázkový vstup a jeho následný výstup. V této fázi je potřeba ověřit, že v izolovaných

podmínkách funguje vše podle očekávání. Testování tohoto druhu dokáže odhalit chyby v počáteční fázi vývoje a jejich náprava je mnohem levnější. Ověřuje se tzv jednotka (unita). Tyto testy se neprovádějí manuálně, ale jsou automatizované a jejich běh by měl být co nejrychlejší. (Roman 2018)

3.7.9 **Integrační testy**

Tuto fázi testování provádí samotný testovací tým, na rozdíl od jednotkového testování, které většinou provádí vývojáři. Integrační testování, které zahrnuje testování API (Aplikační Programovací Rozhraní), což je podmnožinou integračního testování, které zprostředkovává komunikaci a výměnu dat mezi samotnými softwarovými systémy. Testování se zaměřuje na prověření, že všechny integrované moduly/komponenty fungují dohromady a jsou schopné navzájem komunikovat. Tyto testy zahrnují i kontrolu, zda probíhá komunikace s potřebným hardwarem nebo operačním systémem podle stanovených předpokladů. Ověřuje se integrita systému a rozhraní, které interaguje s externími komponentami. U tohoto druhu testování se místo standardního uživatelského vstupu (klávesnice) a výstupu používá software, pomocí kterého se provádí volání API, při kterém se získává odpověď a odezvu systému. Testování API se poměrně liší od testování GUI (grafického uživatelského rozhraní), jelikož se v tomto způsobu testování nesoustředí na vzhled a dojem z aplikace, ale soustředí se na obchodní logiku a softwarovou architekturu. Tento druh testu předchází úniku defektů na vyšší úrovně testování. Integrační testování může probíhat buď manuálním nebo automatizovaným způsobem. (Roman 2018)

3.7.10 **End-to-End testování**

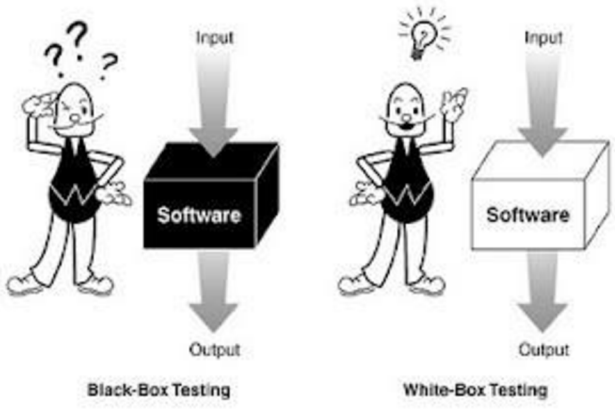
Na vrcholu pyramidy se nachází tzv. end-to-end testy neboli testy od začátku do konce. Hlavní úkol těchto testů je zajištění, že celá aplikace funguje podle předem stanovených předpokladů. End-to-end testování probíhá přesně tak, jak název napovídá, probíhá kontrola funkčnosti aplikace od začátku až do samotného konce a poskytuje největší jistotu, že software funguje podle očekávání. Při provádění těchto testů je důležité se přenést do role uživatele a představit si jeho perspektivu. Tento způsob testování se nachází na samotném vrcholu pyramidy, jelikož obvykle trvá nejdéle. V rámci tohoto testování se musí prověřit veliké množství testovacích scénářů. Stejně jako integrační testy, tak mohou vyžadovat, aby aplikace komunikovala s externími komponentami. (Roman 2018)

3.7.11 **Black box**

Testovací technika černé skříňky (označovaná jako behaviorální) je založena na analýze vhodné testovací základny. Tato technika se využívá pro funkční i nefunkční testování. Black-box technika se soustředí na vstupy a výstupy testovaného objektu, bez odkazu na jeho vnitřní strukturu. Testovací metoda černé skříňky je založena na předpokladu, že tester je obeznámen pouze s tím, co má daný software dělat. Tester nezná samotný kód aplikace a její vnitřní uspořádání, tudíž se nemůže podívat do vnitřku oné schránky. Tento způsob se často využívá u testování klientem, jelikož nepotřebuje znát vnitřní chování, ale testuje simulaci běžné používání aplikace. (Roman 2018; Kumar 2012)

3.7.12 **White box**

Testovací technika bílé skříňky (označovaná jako strukturální) je založena na analýze architektury, podrobném návrhu, vnitřní struktuře a kódu testovaného objektu. Provádí se během vlastního vývoje aplikace. Na rozdíl od testovacích technik černé skříňky, tak se testovací techniky bílé skříňky soustředí na strukturu a zpracování v testovaném objektu. U této techniky je nezbytná znalost kódu, tudíž tento test většinou provádí samotný vývojář. Úskalí tohoto testu bývá v tom, že jedinec, který tento test provádí může být ovlivněn tím, že zná vnitřní fungování aplikace, což může zapříčinit neobjektivní testování. (Kumar 2012)



Obrázek 6 – Black-box, White-box (Kumar 2012)

3.8 **Automatické testování**

Při vývoji softwarového produktu nelze zaručit stoprocentní stabilitu. Během vývoje se vyskytne velké množství chyb. Aplikace se musí otestovat několikrát během vývoje a také při samotném dokončení. Klíčové je zajistit, že se podařilo většinu chyb odhalit a opravit, ještě

před vydáním produkční verze koncovým uživatelům. V případě, že se jedná o rozsáhlý softwarový vývoj, tak je důležité mít stanovené metodické testování, které se provádí napříč celým SDLC. Toho lze dosáhnout vytvořením testovacích scénářů, které určují testovací postup krok za krokem, aby bylo zajištěno, že se funkce softwarového projektu chová podle stanoveného očekávání.

Tento způsob testování znamená stále se opakující procesy. V případě, že se do aplikace přidává nová funkcionalita, je zapotřebí protestovat celý systém, jelikož je zde riziko, že zavedené změny ovlivnily jiné komponenty, a tudíž se objevily nové chyby. Každé opravení nalezené chyby přináší riziko vzniku chyby nové. Potvrzující a regresní testování přináší velkou časovou náročnost, jelikož se musí znovu otestovat části které se již testovaly, což může mít nepříznivé následky, jelikož se kvůli tomu nestíhají testovat nové části aplikace. Potvrzující testování má za úkol ověřit, že oprava nalezené chyby byla úspěšně provedena. Regresní testy mají za úkol ověřit, že s opravou tohoto původního defektu nevznikla nová chyba v jiné části aplikace. Tato situace přináší další hrozbu v podobě vzniku chyby ze strany testovacího týmu, jelikož se jedná o rutinní záležitost, tudíž je náročné si zachovat maximální pozornost a nic nepřehlédnout. Nejenom z těchto důvodů se vytvářejí automatické testy, které tyto problémy do jisté míry eliminují. (Hamilton 2012; Axelrod 2018; Pekař 2017)

Testování lze rozdělit na základě několika kritérií, jedno ze základních rozdělení je na manuální a automatické, na základě toho, zda testování provádí člověk nebo testovací skript, respektive software. Účelů pro použití automatického testování je celá řada. Zavedením automatických testů přináší časové a finanční úspory, zvyšuje se účinnost testování a pokrytí celé aplikace jednotlivými testy. Zároveň je optimalizovaná rychlost a způsob testování dílčích částí a systému jako celku. Ve své podstatě se jedná o téměř stejné druhy testů, které by prováděl člověk v rámci manuálního testování. Nesmí se opomenout fakt, že vývoj automatických testů stojí značné množství času, které musí testovací tým vynaložit na jeho vytvoření a odladění, což v některých případech může stát více času, než bylo původně plánováno.

Zároveň je nutné předem stanovit požadavky, které mají automatické testy splnit. Určení rozsahu automatizace, které je spojené s konkrétní částí aplikace a zda dává automatizace v tomhle případě smysl je klíčové. Současně se musí brát na vědomí, že je potřeba investovat čas do sestavení samotných automatických testů, jejich odladění a následná údržba. Pokud se určitá sekce aplikace bude v průběhu vývoje měnit, tak je potřeba zvážit, zda se investice do

automatizace za těchto podmínek vyplatí, jelikož bude testy nutné upravit, aby byly znovu funkční.

Automatické testy musí splňovat stanovené požadavky, aby bylo možné se stoprocentně spolehnout na výstupy z těchto testů. V případě, že by testy nevykazovaly stabilní chování, což se projeví tím, že v určitém počtu spuštění projdou jen v některých případech, tak je nezbytné tyto testy stabilizovat, jinak by nebylo možné se na dané testy stoprocentně spolehnout a celý tento proces by ve finále byl kontraproduktivní. Cílem automatizace je snížit počet testovacích případů, které mají být provedené ručně. Automatické testy mohou běžet bez dozoru lidské přítomnosti, nehledě na denní dobu, tudíž mohou běžet i přes noc. (Hamilton 2012; Rehkopf nezadáno; Pekař 2017)

3.8.1 **Podmínky pro automatizaci**

QA týmy neustále zlepšují svou strategii směrem k inkluzivnějšímu testovacímu přístupu, aby zvýšili efektivitu a pokrytí celého testovacího procesu. Automatizace je nedílnou součástí vývojového cyklu, tudíž je nezbytné určit čeho je potřeba dosáhnout. Test musí splňovat určitá kritéria, aby mohl být automatizován. Samotný automatický test musí být stabilní, nezávislý na výsledcích ostatních testů, znovupoužitelný, vypovídající a rychlý. Zavedení automatických testů neznamená stoprocentní nahrazení manuálních testů za automatické, jelikož všechny testovací scénáře nelze automatizovat. Úspěch automatizace testování projektu spočívá ve výběru vhodných testovacích případů, které je potřeba otestovat ručně a naopak určení testovacích případů, které se automatizovat vyplatí.

3.8.2 **Neefektivita automatizace**

Automatické testování je považované za efektivní, ale má to svá úskalí. Každopádně ne všechny části softwarového produktu je možné automatizovat, u některých to může být prakticky nemožné nebo dokonce nevýhodné. V případě, že se část aplikace neustále mění, tak nedává smysl automatizace, jelikož by bylo nutné testy neustále udržovat podle zavedených změn. V případě, že se projekt nachází v pozdní fázi vývoje nebo je dokončen vývoj, tak se nevyplatí zavádět automatické testy, jelikož úsilí, které se do vytváření těchto testů vloží se s velkou pravděpodobností nemusí vrátit.

3.8.3 **Výhody automatizace**

Případů, kdy je zavedení automatických testů výhodné je celá řada. Manuální testování od jednotlivých komponent, po dílčí části, až k protestování celé aplikace je časově i finančně náročné. Automatizace ušetří úsilí, finance a čas. Manuální testování se časem může stát nudné, obzvlášť pokud je potřeba neustále provádět regresivní testy dokola, což může být příčina pro vznik chyby při testování. Širší pokrytí pomocí automatických testů a současná úspora času přináší prostor pro začlenění dalších způsobů testování, které by nebylo možné bez automatizace provádět. Po skončení běhu testů je k dispozici okamžitá zpětná vazba v podobě výsledků, které lépe umožní analýzu celkového stavu testovaného softwaru. Vytvořené automatické testy pomocí správných přístupů lze znovupoužít, tudíž na dalších projektech nebude nutné vše programovat od začátku. Ke spuštění testů není nutná přítomnost člověka, ale testy mohou běžet samovolně bez dohledu. (Testim 2019; Axelrod 2018)

3.8.4 **Vhodnost použití automatizace**

Automatické testy je vhodné použít v několika případech:

- Opakující se testovací případy
- Vícejazyčné aplikace
- Rozsáhlé softwarové projekty
- Businessově důležité části aplikace
- Testovací případy, které se těžko testují ručně
  (Testim 2019)

3.9 **Měřitelnost a metriky automatizace**

Ještě předtím, než začne samotný proces automatizace, tak je nezbytné mít ujasněné, proč je důležité měřit kvalitu, efektivitu a návratnost automatizace. Automatizace je beze sporu investice, pokud se neměří její přínos, tak není známé, zda se tato investice vyplatila. Zavedení automatizace a její následná údržba je poměrně časově náročná činnost. Základní klíč k úspěchu spočívá v tom, co platí pro všechny procesy a inovace. Množství úsilí, které se do automatizace vloží by se mělo vrátit, jinak se jedná o neefektivní činnost, která nedává smysl. Metrik, které lze v rámci automatizace sledovat je celá řada. Např.: návratnost, efektivita automatických testů, stabilita, počet odhalených bug atd. (Bose 2021; Chaturvedi 2022)

3.9.1 **Návratnost automatizace**

Návratnost automatizace je základní metrika, která udává kdy se investice a vynaložené úsilí začne vyplácet. Tato metrika přímo souvisí s časem a také náklady. Návratnost se počítá jako získané úspory, tudíž nahrazení manuálního testování automatickým. Tudíž návratnost automatizace = úspora / náklady.

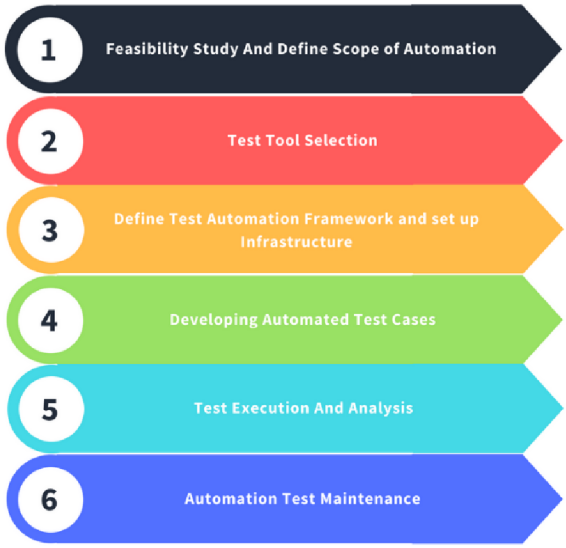
- Úspora je rozdíl mezi dobou běhu automatického testu oproti času, který je potřeba na manuální testování: (provedení manuálních testů – běh automatických testů) * (počet běhů)
- Náklady se rozumí čas, který je potřeba na vytvoření automatických testů + čas na nezbytnou úpravu a údržbu. (Bose 2021; Chaturvedi 2022)

3.9.2 **Pokrytí testovacích scénářů automatickými testy**

Pomocí této informace lze určit, do jaké míry je aplikace pokrytá automatickými testy. Díky tomu lze také určit, jaké testovací scénáře je potřeba testovat manuálně a zda některé další části nestojí za pokrytí automatickými testy. %Pokrytí automatizace = (počet automatizovaných testů / celkový počet testů) * 100. (Bose 2021)

3.10 **Proces automatického testování**

Existuje několik přístupů k samotnému procesu automatického testování. Základ je většinou stejný, rozdíl je naopak v rozsahu, pořadí a členění jednotlivých kroků. Bez dodržení předdefinovaného postupu mohou nastat zásadní problémy. Vynechání zásadních kroků může vést v krajním případě i k neúspěchu celého testovacího procesu. Pokud není správně proveden proces pro automatické testování, tak nelze dosáhnout maximálních výsledků ani plného potenciálu. (Cummings-John nedatováno; Arumughom: 2018; Smriti 2019)



Obrázek 7 – proces automatického testování (Arumughom 2018)

3.10.1 **Rozsah automatizace**

Každý proces začíná definicí. Před implementací samotné automatizace je nutné určit její rozsah. Rozsah automatizace znamená určit části aplikace, které budou automatizovány a jakým způsobem. Definice priorit jednotlivých testů a zohlednění znalostí testovacího týmu hraje důležitou roli.

3.10.2 **Výběr testovacího nástroje**

Automatizace je velice závislá na výběru vhodného nástroje, což je důvod, proč je nutné tomuto kroku zajistit vysokou pozornost. Testovací nástroj musí podporovat všechny technologie, které se vyskytují v dané aplikaci. Důležitá je podpora reportování výsledků testování pro všechny zainteresované strany. V neposlední řadě zde hraje roli cena samotného nástroje a zkušenosti testovacího týmu s tímto nástrojem.

3.10.3 **Plánování a nastavení infrastruktury**

Plánovací fáze znamená vytvoření testovací strategie. Její nezbytnou součástí je rozhodnutí, jak budou využity testovací nástroje, případné frameworky a potřebné knihovny. Definuje se časové rozpětí pro automatizaci. Testovací tým se rozhoduje ohledně testovacích procedur a standardů, které budou nejvhodnější pro testovaný software, příprava testovacích dat a způsob zaznamenání jednotlivých bugů. Zakončení této fáze spočívá ve schválení od stakeholderů. (Cummings-John nedatováno; Arumughom 2018; Smriti 2019)

3.10.4 **Vývoj automatických testů**

Je důležité zachovat veškeré konvence, které platí při programování v rámci pojmenování, organizace a využití metod v samotném testovacím skriptu. Verzovací systém by měl být součástí vývoje. Postupně se pokryjí všechny testovacích případy, které byly definovány v předchozí fázi procesu. V průběhu programování je vhodné použít tzv. "best practices" pro zachování celkové kvality samotných testů. Nedílnou součástí je zachovat princip znuvupoužitelnosti, aby již vytvořené části kódu bylo možné použít znovu.

3.10.5 **Spuštění a analýza výsledků**

Po dokončení všech předchozích kroků je na řadě spuštění testů, pokud se vyskytne chyba, je zapotřebí identifikovat z jakého důvodu chyba nastala, zda je chyba na straně testovacího skriptu nebo na straně testovaného softwaru. Jakmile doběhnou všechny testy je zapotřebí vytvořit srozumitelný výstup, který lze analyzovat a vyvodit závěry. Správně zpracované a analyzované výsledky jsou stěžejní. (Arumughom 2018)

3.10.6 **Údržba a podpora**

Průběžná kontrola by měla být součástí každého procesu automatického testování. Toto je obzvlášť nutné, pokud je naplánovaná v budoucnosti další spouštění existujících testů. I když existuje skript připravený k použití, tak je stále nutné jej udržovat. Průběžná údržba také poskytuje jistotu, že existující kód je možné znovu využít. Podpora těchto testů také zajistí, že testy budou i nadále vykazovat relevantní výsledky.

(Cummings-John nedatováno; Smriti 2019)

3.11 **Kritéria pro výběr automatického nástroje**

Běžné typy testování softwaru, jako je regresní testování, funkční testování, jednotkové testování, integrační testování atd., jsou nahrazovány systematickými testovacími programy využívající nástroje pro automatické testování. Zavedení automatizace je efektivní způsob, jak zvýšit pokrytí testů testovaného softwaru a zvýšit jeho kvalitu. Výběr správného nástroje může být složitý úkol, při kterém je potřeba zvážit mnoho kritérií. Dostupných nástrojů pro automatizaci je celá řada, ale ne každý nástroj splňuje dané předpoklady pro specifický projekt, tudíž je nezbytné zohlednit několik klíčových kritérií, podle kterých vybrat konkrétní nástroj.

3.11.1 Porozumění požadavků projektu

Pro udržení kvality aplikace je nezbytné dodání produktu bez zásadních chyb. Automatizované testování pomáhá zlepšit kvalitu produktu a zvýšit rozsah a hloubku testů. V potaz se musí brát typ projektu (webový/desktopový/mobilní), rozsah samotného projektu a také testovacího týmu.

3.11.2 Identifikace klíčových vlastností nástroje pro automatizaci

Ještě před samotným výběrem konkrétního nástroje pro automatizaci je potřeba vzít v potaz několik klíčových bodů, které by měl z větší části splňovat.

- Jednoduchý vývoj a údržba skriptů: Vývoj a údržba testovacích skriptů by měla být co nejjednodušší, aby se snížilo úsilí a náklady spojené s vývojem automatických testů.
- Snadné spuštění testů pro netechnické uživatele: Provedení testovací sady by mělo být snadné pro všechny členy projektu, aby bylo možné testy spustit podle potřeby. V případě, že manuální tester, který nemá technické znalosti pro vytváření těchto testů, tak by měl být přesto schopný tyto testy spouštět.
- Poskytnout podporu pro testovací prostředí.
- Možnost využít více typů testů: Testovací nástroj by měl poskytovat možnost vytvořit co nejvíc druhů testování, aby bylo možné provádět jednotkové, integrační a end-to-end testování.
- Jednoduché debugování: Pokud z nějakého důvodu přestaly procházet vytvořené automatické testy, mělo by být jednoduše dohledatelné z jakého důvodu nefungují podle představy.
- Stabilita: Pro spolehlivý nástroj je klíčové, aby poskytoval stabilní prostředí pro běh testů, tudíž by se nemělo stát, že neprochází vytvořené testy kvůli samotnému nástroji.
- Možnost reportování výsledků: Výsledky a jejich správná interpretace je klíčová, a proto musí být výsledky z jednotlivých běhů intuitivní, podrobné a srozumitelné.
- Výkonnost: Jeden ze základních požadavků na testovací nástroj je, aby spuštěné testy běžely co možná nejkratší dobu.
- Testování napříč několika prostředí: Podpora pro testování napříč různými prostředí je nezbytná, pokud existuje více koncových uživatelů a neexistuje konkrétní omezení pro specifické prostředí. Například webovou aplikaci je možné spustit na několika internetových prohlížečích, tudíž je zapotřebí větší pokrytí.

- Cena: V závislosti na potřebných kvalitách a odhadech nákladů na projekt je nutné zvážit, zda požadovaná cena daného nástroje odpovídá kvalitě a zda neexistuje levnější obdoba, která poskytuje srovnatelné možnosti za méně nákladů.
- Technická podpora a asistence: Testovací tým, který vytváří automatické testy jednoznačně potřebuje mít k dispozici pomoc, aby bylo možné vyřešit závažné problémy během projektu.
  (Shaikh nedatováno)

3.12 Konkrétní nástroje pro automatické testování

Pro výběr vhodného testovacího nástroje je nutné se řídit specifickými podmínkami konkrétního projektu, což je součástí procesu zavedení automatizace. Každý testovací nástroj má své výhody a nevýhody, které je potřeba důkladně zvážit.

3.12.1 Cypress

Cypress se řadí mezi přední testovací nástroj nové generace vytvořené pro moderní web.

Tento testovací nástroj je postavený na Node.js, představuje komplexní testovací rámec, dodává se jako modul npm (Node.js package manager je správce balíčků pro Javascript). Cypress využívá Mocha – JavaScriptový testovací framework, což umožňuje snadné asynchronní testování. Na rozdíl od většiny nástrojů, které běží mimo prohlížeč a spouští vzdálené příkazy po síti (jako Selenium), tak Cypress funguje opačně, jelikož se spouští ve stejné smyčce běhu jako testovaná aplikace, tudíž je možné upravovat chování prohlížeče za běhu tím, že manipuluje s DOM a mění síťové požadavky. Zaměřuje se na snahu poskytnout dobrou vývojářskou zkušenost a integrované prostředí. Cypress umožňuje vytvářet jednotkové, integrační a end-to-end testy. Skládá se z bezplatného open source, lokálně nainstalovaného Test Runneru a Dashboard servisu pro záznam jednotlivých testů. Jednou z klíčových vlastností Cypressu je automatické čekání, což znamená, že čeká na zviditelnění prvků, dokončení animace, načtení DOM (data object model) a dokončení volání. Není tedy potřeba definovat implicitní a explicitní čekání. Cypress pořizuje snímky během spuštění testů. Lze umístit kurzor na každý příkaz v protokolu příkazů, aby bylo jasné, co se stalo v každém kroku. Cypress poskytuje rychlé, konzistentní a spolehlivé provádění testů. Spojuje se také s jQuery a dědí mnoho metod pro identifikaci komponent uživatelského rozhraní. Cypress podporuje spuštění testů na několika prohlížečích, jako je Chrome, Electron, Firefox a Microsoft Edge. (Atlanta team 2022; Kassandra 2020)

3.12.2 **Puppeteer**

Puppeteer je oblíbený nástroj pro automatizaci testování webových aplikací spravovaný společností Google. Puppeteer je knihovna Node.js, která poskytuje rozhraní API na vysoké úrovní pro ovládání prohlížečů Chrome nebo Chromium přes protokol DevTools. Jeho výhodou je stabilita a poměrná jednoduchost. Puppeteer umožňuje provádět většinu věcí, které lze dělat ručně v prohlížeči. Lze vytvořit snímek obrazovky a soubor stránek v PDF, procházet jednostránkové aplikace s před vykresleným obsahem. Testy se spouští přímo v nejnovější verzi prohlížeče pomocí nejnovějšího JavaScriptu a funkcí prohlížeče. Automatizace odesílání formulářů, testování uživatelského rozhraní, zadávání vstupu z klávesnice, otestování rozšíření Chrome a spoustu dalších funkcí. (Vaidya 2021)

3.12.3 **Selenium**

Selenium je běžně spojené s pojmem automatické testování, tento nástroj je považován za průmyslový standard pro automatické testování uživatelského rozhraní webových aplikací. Selenium je sada nástrojů pro automatizaci testů, jedná se o open source. Testovací skripty lze vytvářet v mnoha různých programovacích jazycích (např. Java, Python, C#, PHP, Ruby a Perl), které běží na více systémových prostředích, mnoha platformách a prohlížečích (Chrome, Firefox, IE). (Vaidya 2021)

3.12.4 **Playwright**

Playwright je Node.js knihovna, umožňuje spouštět automatické testy v několika prohlížečích – Chromu, Firefoxu, Microsfot Edge a Safari. Playwright je z části vytvořen stejnými lidmi, kteří jsou autory Puppeteeru a je spravován společností Microsoft. Používá stejné API, má podobnou syntaxi a jazyk, ale existuje několik rozdílů. Pomocí Playwrightu lze otestovat nativní mobilní emulace Google Chrome pro Android. Klíčová vlastnost je automatické čekání před provedení akcí, než budou požadované prvky viditelné a použitelné, čímž omezuje potřebu používat umělé časové zpoždění, které jsou často zdrojem nespolehlivosti testů. (Kassandra 2020)

4 Vlastní práce

Praktická část bakalářské práce se zabývá realizací procesu automatického testování, zhotovením testovacích scénářů, určení rozsahu automatizace, vytvořením automatických testů, provedení manuálního a automatického testování, porovnání obou přístupů a vyvození výsledků. První krok spočívá ve zvolení vhodného nástroje pro automatizaci webové aplikace, následně se zhodnotí požadavky testovacích případů, načež bude vytvořen testovací scénář. Na základě tohoto scénáře vzniknou automatické testy, poté proběhne ruční a automatické testování. Výsledky obou způsobů testování se navzájem porovnají.

4.1 Popis zvolené aplikace

V první řadě je potřeba představit zvolenou webovou aplikaci pro testování, která se nazývá ASWA. ASWA neboli Asociace softwarových agentur je webová aplikace, která pomáhá s orientací v softwarovém světě. Podporuje férové prostředí, pomáhá zadavatelům i tvůrcům softwaru a zlepšuje IT prostředí. Poskytuje metodiky a kritéria pro tvorbu zadávací dokumentace včetně nutných informací pro dodavatele. Pomáhá zadavatelům porozumět, jak probíhá vývoj a co agentury potřebují k tomu, aby vznikl kvalitní produkt.

Celý vývoj SDLC tohoto projektu probíhá agilní metodikou, tzn. že postupně vzniká celý produkt po dílčích částech. Tím pádem je důležité, aby testování začalo již v počáteční fázi vývoje, tudíž bude možné co nejdříve identifikovat a opravit vzniklé chyby. Čím dříve se detekuje chyba v průběhu celého vývoje, tím méně nákladná je její následná oprava. V první řadě je potřeba zajistit systémové testování celé aplikace, které má za úkol ověřit funkčnost a stabilitu pro nasazení nové verze. Výsledkem tohoto testování je report, na jehož základě se opraví nalezené chyby a uskuteční další kroky. Tento způsob testování je potřeba provádět s každou novou verzí nebo zavedení větší funkcionality, což je velice časově náročné. Toto testování musí proběhnout nejenom na mobilní a desktopové verzi, ale také na více druhů prohlížečů, jelikož je nezbytné mít pokryté co největší spektrum. Testování webové aplikace v rámci více prohlížečů a desktopové i mobilní verze má za následek, že se mnohonásobně navýší potřebný čas k provedení manuálního testování. Současně roste riziko pro vznik chyby ze strany lidského faktoru, jelikož se jedná o opakující se činnost. Zároveň je potřeba provádět testování nových funkcionalit, které by nemělo být blokováno v rámci časových kapacit pro systémové, potažmo regresní a konfirmační testování. Kombinací těchto faktorů má za následek, že nezbývá dostatečný prostor pro jiné druhy testování, tudíž se zmenšuje pokrytí

jednotlivých druhů testování celé aplikace, což může způsobit neodhalené chyby produktu a celkové snížení kvality. V rámci asociace působí několik členů, kteří mají klíčové slovo napříč celým projektem, tudíž je kladen veliký důraz na srozumitelné a podrobné výsledky z testování. Pokud se během testování nepodaří odchytit všechny zásadní bugy a neodhalená chyba se dostane do produkční verze, tak následná oprava je několikrát nákladnější. Nehledě na situaci, kdy může trvat delší dobu, než se vývojářskému týmu dostane informace o dané chybě na produkčním prostředí.

Cíl automatizace je snížit testovací případy, které se musí provádět ručně a pokrýt velkou část testovacích scénářů automatickými testy, aby došlo k úspoře času a vznikl prostor pro další druhy testování, které se omezují z nedostatku času. Jednotlivé běhy testů srozumitelně interpretovat a poskytnout vizuální záznam jednotlivých testů a následně vyvodit potřebné výsledky, které budou podrobně interpretovány. Jelikož se jedná o testování více druhů prohlížečů a mobilní i desktopové verze, tak je časová úspora v tomto případě značná. Současně se předchází vzniku chyby z lidské nepozornosti, jelikož automatické testy provedou sadu úkonů pokaždé stejně, bez ohledu na vnější okolnosti. Při selhání běhu testů funguje okamžitá zpětná vazba, tudíž známe přesnou část, ve které se vyskytuje chyba a následná oprava zabere méně času.

4.2 Výběr testovacího nástroje

Pro zvolení testovacího nástroje je nezbytné porovnat několik klíčových vlastností, které jsou zásadní pro tento projekt. Testovací nástroj musí podporovat integrační a end-to-end testování, jelikož právě tyto testy budou mít velké zastoupení při automatizaci pro zvolenou aplikaci. Důležitá je podrobná interpretace výsledků, jejich analýza a následné vyhodnocení, které musí být srozumitelné. Nutná je také podpora pro spouštění testů v různých prohlížečích, jelikož tato část konzumuje nejvíce času z celého testování a uživatelé na web přistupují z více prohlížečů. Klíčová je také možnost zvolit druh zařízení, na kterém testy poběží, tudíž musí být proveditelné spouštět testy pro mobilní i desktopovou verzi prohlížeče a současně zajištěná podpora pro upravení tohoto zobrazení. Důležité je potřeba brát v potaz cenu, jelikož náklady spojené s testováním musí být co nejnižší. Vývoj samotných testů a následná údržba by měla být jednoduchá, aby se předcházelo vysokým nákladům na realizaci celého procesu automatizace.

4.2.1 Testovací software pro automatizaci

Z těchto důvodů jsem se rozhodl zvolit open-source nástroj Cypress, který nejlépe splňuje předchozí požadavky. Cypress podporuje jednotkové, integrační a end-to-end testování. Výsledky jednotlivých běhů jsou detailně interpretovány skrze dashboard, který poskytuje velké množství informací z jednotlivých běhů testů. V rámci tohoto dashboardu je možné procházet jednotlivé kroky testu a sledovat, co přesně konkrétní krok provedl. Současně je zde možnost zpětně procházet jednotlivé kroky pomocí kurzoru, tudíž jsou známé veškeré informace, pokud některý krok selže nebo neprojde, tím pádem je debugování mnohem jednodušší. Velký benefit je vizuální ukázka běhu jednotlivých testů, které se mohou demonstrovat všem stakeholderům při schvalování do další fáze projektu. Cypress nabízí možnost jednoduše upravit viewport, tudíž se snadno testuje mobilní i desktopová verze v různých zobrazení, které se specifikují před spuštěním testů. Cypress poskytuje jednoduchou integraci přes CI, tudíž je možné spouštět všechny testy s novou verzí aplikace. Díky tomu se automaticky s novou verzí aplikace získá okamžitá zpětná vazba a odhalí se vzniklé chyby. Cypress podporuje programovací jazyk JavaScript a s tím spojené testovací frameworky, pro tento případ bude využit JavaScriptový testovací framework Mocha. Cypress je desktopová aplikace, která podporuje operační systémy macOS, Linux a Windows. Zvolil jsem vývojové prostředí Visual Studio Code, což je taktéž open-source.

4.3 Předpoklady testovacích případů pro automatizaci

Každý proces začíná jasně stanovenou definicí. Před implementací automatických testů je zapotřebí stanovit rozsah pro automatické testy, což znamená, že je nutné určit jednotlivé části, které budou pod kontrolou automatických testů. V tomto případě se jedná o automatizaci webové aplikace. To je nutné brát v potaz, protože jak již bylo zmíněné, tak testy budou spuštěné v rámci několika prohlížečů a různých zařízení, které mají nepatrně odlišné chování, což ve finále může hrát klíčovou roli. Zásadní krok je stanovení priorit pro jednotlivé testy, vyšší prioritu budou mít rizikové části aplikace a samozřejmě důležité části z businessového pohledu. Priorita se stanový na základě důležitosti do čtyř úrovní. Pohybujeme se na stupnici od případů s nejmenší prioritou “malá”, “střední”, “vysoká” a “kritická”, což určí závažnost a preferenci pro testování jednotlivých testovacích případu prioritně před ostatními. Testovací případy s nejmenší prioritou (malá) nedává smysl automatizovat, jelikož jejich nefunkčnost téměř neovlivní fungování aplikace jako celku. Návratnost v tomto případě by byla velice malá, navíc není zapotřebí tyto případy procházet pokaždý co se spustí běh automatických testů.

Pro udržení kvality a eliminování vzniku chyb je důležité primárně automatizovat regresní testy. Každopádně se zde vyskytují i testovací případy, které mají vysokou prioritu a jejich důležitost v rámci businessového hlediska je také vysoká, nicméně jejich automatizace nedává smysl, protože řešení tohoto testu by bylo velice složité a vzhledem k responzivitě prostředí také nestálé, což okamžitě vylučuje přínos vytvoření automatických testů pro tyto případy, jelikož by nebylo možné se na tyto testy stoprocentně spolehnout. Nehledě na fakt, že některé případy ani automatizovat jednoduše nelze. Jako příklad lze uvést testování videí, které se nachází na podstránce Zaměřeno na vývoj. Video lze samozřejmě spustit pomocí kliknutí na tlačítko přehrát video, což není problém pomocí testovacího skriptu, nicméně kontrola jednotlivých funkcionalit, kterými samostatný přehrávač disponuje už možné není. Kontrola zvuku a možnost manipulace hlasitosti videa není možná, funkce pro zvětšení na celou obrazovku a následné zmenšení také působí potíže. Možnost přeskakovat a vracet se v čase videa také nelze a v neposlední řadě zrychlení videa a následné ověření také působí značné potíže. Další případ, který činí automatickému testování potíže je otevírání odkazů v novém okně. Konkrétně se na ASWA webu nachází odkaz, který otevře na nové kartě dotazník pro získání členství. Tento případ není nemožné otestovat, ale jeho realizace je značně problematická z důvodu, že otevření nové lišty prohlížeče a následné vrácení na původní působí nástrojům pro automatické testování velké problémy.

4.4 Testovací scénář

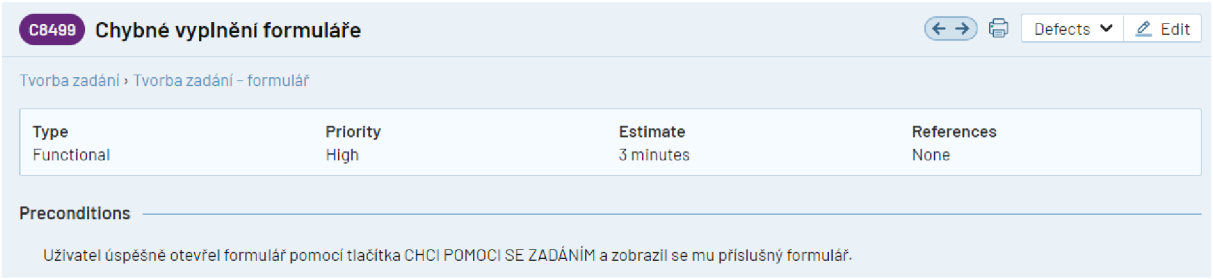
Před začátkem programování samotných automatických testů je nezbytné definovat testovací scénář. Na základě tohoto scénáře budou sestavené automatické testy a současně provedené manuální testování, což je nezbytný krok pro relevantní srovnání obou přístupů testování. Samotné testovací scénáře budou obsahovat prioritu, předpoklady pro možnost uskutečnění daného testovacího scénáře a jednotlivé kroky, které mají jasně nadefinovaný výsledek.

Jednotlivé testovací případy obsahují několik náležitostí pro správné otestování daného scénáře. Testovací scénář je vytvořen v nástroji TestRail, který slouží k zaznamenání testovacích scénářů a následné vyhodnocení výsledků z tohoto testování. Tento nástroj napomáhá zefektivnit testovací procesy, poskytuje přehled o celém stavu testování a umožňuje integraci s ostatními nástroji. Zásadní nevýhodou je, že tento nástroj není open-source, ale je placený, nicméně mám dostupný přístup přes firemní účet. V rámci tohoto nástroje lze spravovat testovací scénář a zaznamenávat následující informace:

- Detail scénáře obsahuje informace o typu testu, který se bude v rámci tohoto scénáře realizovat, tj. jestli se jedná o funkční, regresivní, bezpečnostní, smoke, akceptační nebo jiný druh testu.
- Nachází se zde priorita, která určuje důležitost úspěchu pro konkrétní testovací případ.
- Předpoklady, které je potřeba navodit pro specifický testovací případ, aby bylo možné test uskutečnit a nedošlo k selhání, které bylo způsobené na základě nesprávných podmínek pro daný test.
- Jednotlivé kroky, které je potřeba provést ve stanoveném pořadí, pokud je to vyžadováno. Alternativu představuje textový popis za předpokladu, že pořadí uskutečnění jednotlivých kroků nehraje roli.
- Předpokládané výsledky chování softwaru, ke kterým vedou jednotlivé kroky testování.
- Časová estimace neboli předpokládaný čas, který bude potřeba pro realizaci testování daného testovacího scénáře. Tento čas se stanoví na základě složitosti testovacího případu.
- V neposlední řadě report výsledků zaznamenání celého testovacího cyklu.

4.4.1 **Detail testovacího scénáře**

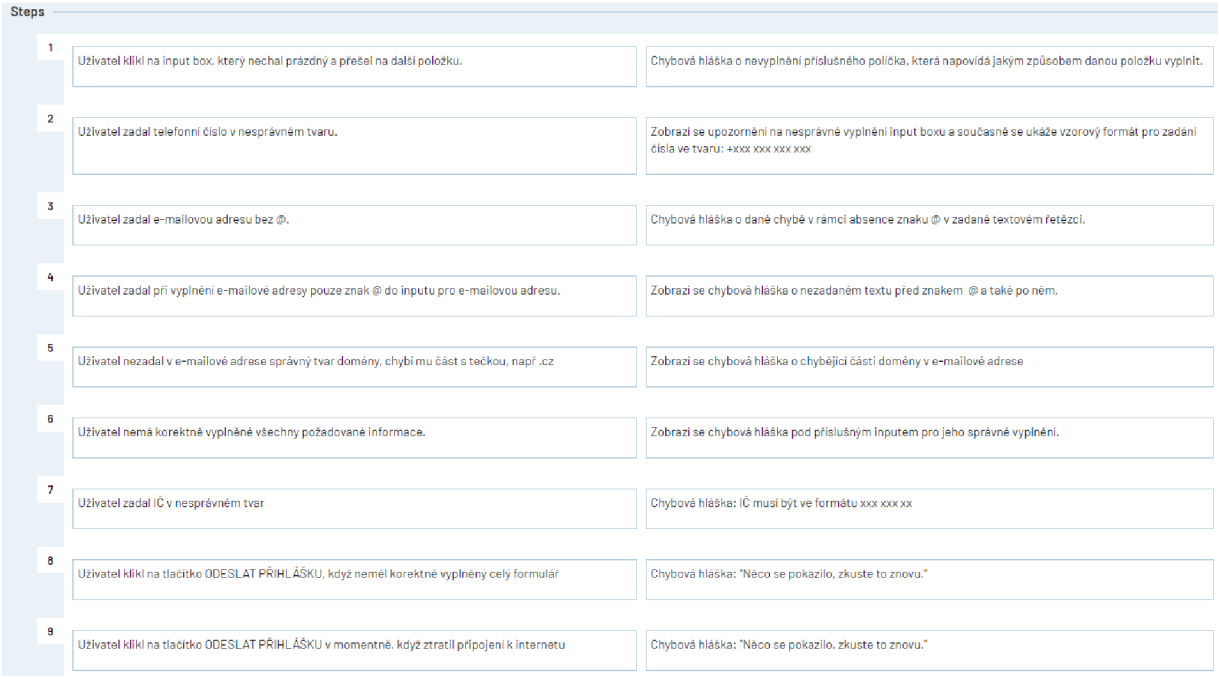
Samotný testovací scénář se v TestRailu skládá ze dvou částí. V detailu první části se nachází předdefinované informace o samotném testovacím scénáři, které se nachází na následujícím obrázku. Tato část zahrnuje název samotného testovacího scénáře, v tomto případě „Chybné vyplnění formuláře“. Podsekci, do které je tento scénář zařazen, konkrétně se jedná o sekci s tvorbou zadání, ve které se nachází podsekce s formulářem. Dále se zde nachází informace o typu testu, priorita a časová estimace.



Obrázek 8 – Úvodní část testovacího scénáře, zdroj autor

Druhá část obsahuje jednotlivé kroky testovacího scénáře. Je zde uvedené pořadí, podle kterého se postupuje v rámci jednotlivých kroků testování. V pravé části se vždy nachází

očekávaný výsledek po provedení daného kroku. Pokud pořadí jednotlivých kroků nehraje roli, tak se zde nachází pouze textový popis konkrétního kroku a následně očekávaný výsledek, bez určeného pořadí.



Steps		
1	Uživatel klikl na input box, který nechal prázdný a přešel na další položku.	Chybová hláška o nevyplnění příslušného políčka, která napovídá jakým způsobem danou položku vyplnit.
2	Uživatel zadal telefonní číslo v nesprávném tvaru.	Zobrazí se upozornění na nesprávné vyplnění input boxu a současně se ukáže vzorový formát pro zadání čísla ve tvaru: +xxx xxx xxx xxx
3	Uživatel zadal e-mailovou adresu bez @.	Chybová hláška o dané chybě v rámci absence znaku @ v zadané textovém řetězci.
4	Uživatel zadal při vyplnění e-mailové adresy pouze znak @ do inputu pro e-mailovou adresu.	Zobrazí se chybová hláška o nezadaném textu před znakem @ a také po něm.
5	Uživatel nezadal v e-mailové adrese správný tvar domény, chybí mu část s tečkou, např .cz	Zobrazí se chybová hláška o chybějící části domény v e-mailové adrese
6	Uživatel nemá korektně vyplněné všechny požadované informace.	Zobrazí se chybová hláška pod příslušným inputem pro jeho správné vyplnění.
7	Uživatel zadal IČ v nesprávném tvar	Chybová hláška: IČ musí být ve formátu xxx xxx xx
8	Uživatel klikl na tlačítko ODESLAT PŘIHLÁŠKU, když neměl korektně vyplněný celý formulář	Chybová hláška: "Něco se pokazilo, zkuste to znovu."
9	Uživatel klikl na tlačítko ODESLAT PŘIHLÁŠKU v momentně, když ztratil připojení k internetu	Chybová hláška: "Něco se pokazilo, zkuste to znovu."

Obrázek 9 – Detailní popis testovacího scénáře, zdroj autor

4.4.2 **Jednotlivé kroky testovacího scénáře**

Detailní popis jednotlivých kroků je velice obsáhlý, tudíž byla zvolena pouze zkrácená verze popisu pro jednotlivé sekce testovacího scénáře.

- **Přístup na webovou stránku:** Tento testovací scénář se zabývá možnostmi přístupů na samotný web. Existují dva hlavní přístupy, první spočívá v zadání celé URL adresy a druhý v prokliku přes konkrétní odkaz na stránku. Následně bude ověřena URL adresa webové aplikace.
- **Testy na hlavní stránce:** Tyto testy spočívají v kompletním otestování hlavní stránky aplikace, jedná se primárně o testování od začátku do konce, tudíž budou otestovány jednotlivé komponenty na této stránce. Je zde kladen důraz na vlídnost prostředí, zda všechny odkazy a tlačítka fungují správným způsobem, který uživatel očekává.
- **Testy formulářů a newsletteru.** Aplikace obsahuje formulář, který se nachází na několika místech. Současně se zde nachází dotazník, který se rovněž vyskytuje na více místech a po kliknutí na příslušné tlačítko se otevře dotazník

v novém okně. Dotazník i formulář mají povinná některá pole a validaci správně zadané e-mailové adresy, telefonního čísla a adresy. Je zapotřebí otestovat, zda je tato validace funkční a zobrazují se správné chybové hlášky v případě zadání nesprávných údajů v rámci formuláře. Dále se zde nachází možnost odebírat nenovější informace v podobě newsletteru, která obsahuje notifikaci v případě úspěšného i neúspěšného pokusu o odebírání novinek.

- **Test stránky s tvorbou zadání.** Na této stránce bude prověřeno, že jednotlivá tlačítka fungují podle očekávání a otevírají se příslušné odkazy. Proběhne kontrola nad korektním zobrazení obrázků na stránce. Současně se zde nachází dva důležité odkazy, které vedou na podsekce s detailním popisem Vodopádového a Agilního přístupu. Na této podstránce se nachází postranní navigace, která scrolluje současně s obsahem a mění se podbarvení na základě části, ve které se uživatel nachází. Na této podstránce v postranním menu je také možné překliknout na druhou metodiku, tudíž není nutné se vracet na předchozí stránku pomoc s tvorbou zadání.
- **Stránka zaměřeno na vývoj.** V rámci toho testovacího scénáře se zkontroluje správné zobrazení všech komponent a odkazů na stránce. Tato stránka je specifická, jelikož obsahuje videa. Proběhne kontrola, zda jde video spustit, zvuk videa, funkce roztažení na celou obrazovku a následné zmenšení, možnost změnit kvalitu videa a rychlost přehrávání. Po kliknutí na detail videa se roztáhne přes část obrazovky a současně se pod tímto videem seřadí ostatní dostupná videa. V případě spuštění jednoho videa lze spustit i další, obě videa se přehrávají zároveň.
- **Test jednotlivých členů.** Tento testovací scénář se zabývá jednotlivými členy asociace. V rámci asociace existuje několik členů, kteří se podílí na fungování celé asociace. Každý člen zde má krátký medailonek, logo a odkaz na vlastní webové stránky konkrétní firmy. Bude zkontrolováno, zda popisy, loga a příslušné odkazy odpovídají jednotlivým členům. V pravé části se nachází sekce Staňte se členem ASWA, který odkazuje na podstránku členství, celá tato část je “sticky“, což znamená, že scrolluje společně s obsahem.
- **Pro média.** Tento testovací scénář ověřuje celou informativní část určenou pro média. Na stránce se nachází jednotlivé články, které obsahují název, náhled

obrázku, datum, krátký popis a tlačítko pro stažení celého článku. Dále se otestuje možnost stažení loga ASWA a fotogalerie ve formátu .zip.

- **Kontakt.** V rámci tohoto testovacího scénáře bude prověřeno správní zobrazení jednotlivých předsedů správní rady a tajemníka asociace. Každý předseda má uvedené jméno, firmu, mobilní a e-mailový kontakt. V rámci tohoto scénáře se otestují možnosti interakce po kliknutí na specifické tlačítko, např. po kliknutí na e-mailovou podstránka pro kontakt, včetně možnosti volání skrze telefonní číslo, odeslání e-mailu na příslušnou adresu se otevře e-mailová adresa. Budou ověřeno, že uvedené informace o asociace jsou správně uvedené.
- **Členství ASWA.** Tento testovací scénář má za účel prověřit, zda jsou dostupné všechny potřebné podmínky pro vstup do členství a zda jsou funkční veškeré odkazy na tuto sekci.
- **Navigace a patička.** Tento testovací scénář má za účel prověřit veškeré odkazy, které se nacházejí v rámci horní navigace a patičky. Bude ověřeno, zda se správně propisuje požadovaná adresa, jestli jednotlivé odkazy přesměrovávají na příslušené odkazy a současně, zda se správně zobrazuje odkaz, na kterém se uživatel nachází.
- **Mobilní a desktopová verze.** Všechny tyto testy proběhnou v rámci dvou zobrazení. První fáze proběhne v rámci mobilní verze a bude následovat desktopová.
- **Webový prohlížeč.** V rámci tohoto testu budou jednotlivé testy provedeny na různých webových prohlížečích, jelikož zvolený testovací nástroj podporuje více webových prohlížečů a koncový uživatelé přistupují na web z vícero různých prohlížečů.
- **Performance a ztráta připojení.** Na závěr proběhne také test výkonu a ověří se, jak se webová aplikace chová v případě, kdy dojde ke ztrátě k připojení.

4.5 Vývoj automatických testů

Jako první je nutné nastavit správné prostředí, které zajistí nutné podmínky pro spuštění jednotlivých testů. Testování může probíhat ve více prostředích podle potřeby, první varianta spočívá v testování produkční verze a druhá na lokálním buildu prostředí. Ideální je využít kombinaci obou možností, aby se ověřilo, že se v závislosti na prostředí neprojeví nové bugy.

4.5.1 **Nastavení počátečních podmínek**

Pro nastavení potřebných podmínek se využívají tzv. "hooks", které si Cypress vypůjčuje z Mocha, což je testovací framework JavaScriptu pro programy Node.js. Jejich využití spočívá v nastavení podmínek, které se spustí před sadou testů nebo za každým testem. Slouží také jako možnost vyčistit podmínky po proběhlé sadě testů. Celkem existují čtyři varianty. Tyto tzv. „hooks" mohou proběhnout jednou před všemi testy, pokaždé před každým testem, po každém testu nebo po všech testech.

```
before(() => {
  cy.viewport(1920,1080);
  cy.visit("https://aswa.cz/");
  cy.url().should("include", "aswa.cz/")
})
```

Obrázek 10 – Ukázka before hook, zdroj autor

Výše uvedený kód znamená využití tzv. „before hook". Pro příkazy v Cypressu je klíčové spojení „cy", pomocí tohoto příkazu se volají jednotlivé metody, které se mohou řetězit s využitím tečkové notace. Tato část kódu proběhne před spuštěním prvního testu, nastaví se viewport (náhled) pomocí příkazu cy.viewport(1920,1080) na klasické zobrazení pro desktopovou verzi. Cypress umožňuje nastavení viewportu pro konkrétní zařízení podle potřeb, např. zobrazení pro iPhone X se provede pomocí příkazu cy.viewport('iphone-x'). Následující příkaz cy.visit() slouží pro navštívení uvedené URL adresy, v tomto případě konkrétně adresy webu ASWA. Po načtení stránky se zkontroluje URL adresa, která by měla obsahovat textový řetězec "https://aswa.cz/". V případě potřeby lze nakonfigurovat v této části podmínky pro počáteční spuštění jednotlivých testovacích skriptů.

4.5.2 **Test průchodu horní navigace**

Následující kód automatického testu provádí testovací scénář pro kontrolu funkčnosti jednotlivých odkazů v horní navigaci. Výchozí pozice je na domovské stránce, zkontroluje se URL adresa, zda je výchozí pozice opravdu na domovské stránce. Dále se použije příkaz cy.get() pro zacílení selektoru, v tomto případě se jedná konkrétně o element pro nadpis h1, který by měl být viditelný a současně obsahovat textový řetězec "Asociace softwarových agentur". Tomuto příkazu odpovídá řádek číslo 11 zdrojového kódu. Další krok vyhledá prvek s textovým řetězcem "Pomoc s tvorbou zadání" a následně na tento prvek klikne. V rámci

tohoto příkazu bylo použito tzv “vynucování”. V některých případech není žádané se chovat jako běžný uživatel, ale je zapotřebí si určité kroky vynutit. Tento případ je naprosto ukázkový pro použití vynucení, jelikož se jedná o vnořenou navigační strukturu, kde uživatel musí umístit kurzor a pohybovat myší ve specifickém vzoru, aby dosáhl na požadovaný odkaz. Ve výchozí pozici daný prvek na stránce není vidět, to ale neznamená, že se na stránce nevyskytuje. Pomocí vynucení lze dosáhnout interakce s prvky, které nejsou vidět nebo jsou dočasně skryté. Princip těchto kroků se opakuje do té doby, než jsou otestované všechny položky v horní navigaci.

```
8   it('Checking the home page', () => {
9     cy.visit("https://aswa.cz/")
10    cy.url().should("include", "aswa.cz/")
11    cy.get("h1").should('be.visible').and('contain', "Asociace softwarových agentur");
12    cy.contains("Pomoc s tvorbou zadání").click({ force: true })
13  })
14
15  it('Checking Creation of tender documentation page', () => {
16    cy.url().should("include", "aswa.cz/tvorba-zadani")
17    cy.get("h2").should('be.visible').and('contain', "Tvorba zadávací dokumentace");
18    cy.contains("Zaměřeno na vývoj").click({ force: true })
19  })
20
21  it('Checking Focused on development page', () => {
22    cy.url().should("include", "aswa.cz/zamereno-na-vyvoj")
23    cy.get("h2").should('be.visible').and('contain', "Zaměřeno na vývoj");
24    cy.contains("Členové ASWA").click({ force: true })
25  })
26
27  it('Checking Members page', () => {
28    cy.url().should("include", "aswa.cz/clenove")
29    cy.get("h1").should('be.visible').and('contain', "Členové ASWA");
30    cy.contains("Pro média").click({ force: true })
31  })
32
33  it('Checking the Documentation page', () => {
34    cy.url().should("include", "aswa.cz/pro-media");
35    cy.get("h1").should('be.visible').and('contain', "Pro média");
36    cy.contains("Kontakty").click({ force: true });
37  })
```

Obrázek 11 – Test horní navigace, zdroj autor

4.5.3 **Vytvoření nových metod**

Je evidentní, že se v testu vyskytuje velké množství kódu, které se opakuje na několika místech, což je signál pro vytvoření nové funkce. V projektu Cypressu existuje speciální složka, která obsahuje soubor určený přesně pro tyto případy, která se automaticky vytvoří s instalací Cypressu. Ve své podstatě je možné definovat nový příkaz, který Cypress automaticky importuje do adresáře s existujícími testy. Při vytváření těchto funkcí je potřeba myslet na nezávislost a znovu použitelnost. V rámci tohoto přístupu jsem zadal potřebné proměnné jako

parametry metody, která se bude volat na několika místech s potřebnými vstupy, tudíž se může použít i pro ostatní testy. Samotná metoda vypadá následujícím způsobem.

```
Cypress.Commands.add("checkUrlElementAndClickLink",
(URL, selector, expectedText, linkInNavigation) => {
    cy.url().should("include", URL)
    cy.get(selector).should('be.visible').and('contain', expectedText);
    cy.contains(linkInNavigation).click({ force: true })
})
```

Obrázek 12 – Zavedení metody, zdroj autor

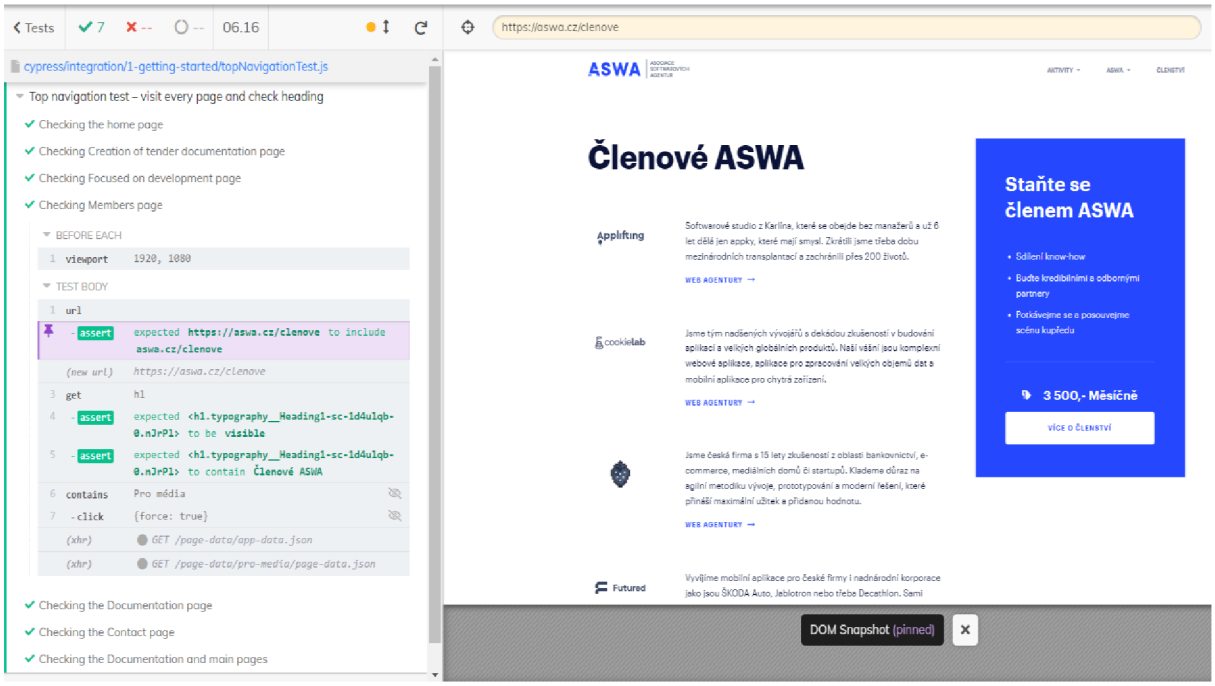
Pomocí tohoto přístupu, kdy se nepoužívají striktně předem dané údaje, ale využívají se proměnné jako parametry metody, tak lze dosáhnout znovu použitelnosti, udržitelnosti a v neposlední řadě zmenšení řádků kódu. Nyní je možné aplikovat nový příkaz na již existující testovací skript. Po úpravě vypadají přepsané testy tímto způsobem.

```
it('Checking Focused on development page', () => {
  cy.checkUrlElementAndClickLink("aswa.cz/zamereno-na-vyvoj",
   "h2", "Zaměřeno na vývoj", "Členové ASWA")
})
```

Obrázek 13 – Využití definované metody, zdroj autor

4.5.4 **Cypress Test Runner**

Cypress spouští testy v interaktivním běhu, což umožňuje vidět příkazy při jejich provádění a zároveň si prohlížet testovanou aplikaci. Testy lze také spouštět v tzv. “headless” módu, což znamená, že se testy spustí na pozadí a v konzoli se zobrazují informace o tomto běhu. Druhá varianta je pomocí vizuálního zobrazení, přesně podle vykonávání jednotlivých kroků testu za pomocí Cypress Test Runner. Na následujícím snímku je zobrazen běh jednotlivých testů z testovací sady, právě ve zmíněném Cypress Test Runner. V levé horní části se nachází celkový počet jednotlivých testů, z toho počet úspěšných a neúspěšných průchodů a také celkový čas běhu těchto testů. Dále se v levé části nachází konkrétní kroky, které test prováděl. Je zde možnost testy procházet krok za krokem, ale také zpětně. U každého kroku vidíme v pravé části přesný stav, který byl před vykonáním jednotlivého kroku a bezprostředně poté. V případě, že testovací krok provádí výběr elementu, tak se zobrazí červená tečka na místě, kam aktuálně testovací nástroj ukazuje (dalo by se to přirovnat ke kurzoru myši). Současně se zde zobrazují všechny volání, které aplikace vykonává v průběhu testů. V pravé části je vizuálně znázorněn každý krok vykonaného příkazu v prohlížeči.



Obrázek 14 – Běh testu v Test runner, zdroj autor

4.5.5 **Využití selektorů**

V tomto případě bylo použito targetování (zacílení) pomocí obsahu textového řetězce z důvodu, že položky v horní navigaci neobsahují vhodnější způsob pro zacílení. Aby bylo možné interagovat s daným elementem, tak je zapotřebí tento prvek zacílit. V ideálním případě prvek obsahuje speciálně vyhrazený selektor, který slouží pouze a jenom k testování. Zacílení pomocí názvu, respektive obsahu textového řetězce je vhodná alternativa, pokud se daný textový řetězec na stránce vyskytuje pouze jednou a tento stav se nebude měnit. Naopak nejhorší možný přístup pro zacílení selektoru je pomocí CSS třídy, která není určená pro testování, případně samotného html elementu, u kterého není stanovená podmínka, která specifikuje obsah textového řetězce, který musí obsahovat. V případě použití nevhodných selektorů se riskuje narušení udržitelnosti a funkčnosti automatických testů. Tyto selektory primárně nebyly navržené pro testování, ale kvůli zobrazení prvků na stránce z hlediska vizuálního zobrazení. S menší změnou aplikace přestanou testy procházet, jelikož je úprava třídního selektoru spojená s funkčností aplikace, nikoliv s jejím testováním. Hrozí vážné riziko, že příkaz pro zacílení potřebného selektoru ukazuje na již neexistují selektor, jelikož původní bylo potřeba modifikovat. Na následujícím obrázku jsou zobrazené jednotlivé způsoby zacílení selektorů a jejich doporučené využití, podle originální dokumentace Cypressu. Jak bylo již řečeno, nejlepší přístup je vytvořit selektory přímo určené pro testování, jelikož je zde jistota,

že se tyto selektory nebudou měnit. V tomto případě je však nutné tyto selektory vytvořit přímo ve zdrojovém kódu samotné aplikace.

Selector	Recommended	Notes
`cy.get('button').click()`	⚠ Never	Worst - too generic, no context.
`cy.get('.btn.btn-large').click()`	⚠ Never	Bad. Coupled to styling. Highly subject to change.
`cy.get('#main').click()`	⚠ Sparingly	Better. But still coupled to styling or JS event listeners.
`cy.get('[name=submission]').click()`	⚠ Sparingly	Coupled to the `name` attribute which has HTML semantics.
`cy.contains('Submit').click()`	✓ Depends	Much better. But still coupled to text content that may change.
`cy.get('[data-cy=submit]').click()`	✓ Always	Best. Isolated from all changes.

Obrázek 15 – Best practicies v rámci selektorů, (Cypress.io nedatováno)

4.5.6 **Test formuláře**

Následující skript testuje funkčnost formuláře v rámci několika kritérií, které jsou popsané v detailu testovacího scénáře. Předpoklad pro tento testovací případ je podmínka, že se uživatel musí nacházet na stránce, která obsahuje tlačítko pro otevření příslušného formuláře, což zajistí řádek zdrojového kódu 9-10. Následně se na stránce vyhledá tlačítko, jež obsahuje text specifikovaný v podmínkové části a následně se na tlačítko klikne. Proběhne kontrola, zda se správně načetl potřebný formulář. Jelikož se jedná o testování formuláře, tak je vhodné použít selektory na základě rozlišení typů pro jednotlivé inputy. Dílčí testy probíhají na podobném principu. Pomocí specifického selektoru se zacílí konkrétní input, proběhne kontrola, zda je viditelný a následně se na něj klikne. V rámci tohoto formuláře jsou povinná všechna pole. Pokud uživatel nevyplní vybraný input box a přesune se na další položku, např. pomocí stisknutí klávesy enter, tak se zkontroluje, že se zobrazí příslušná chybová hláška u dané položky. Specifický případ nastává u pole pro telefonní číslo a e-mailovou adresu, jelikož zde probíhá kontrola na zadání správného formátu pro daný typ textového řetězce.

```
8   it("Visit home page", () => {
9     cy.visit("https://aswa.cz/tvorba-zadani");
10    cy.url().should("include", "aswa.cz/tvorba-zadani");
11  });
12  it("Open form via button", () => {
13    cy.contains("Chci pomoci se zadáním").click({ force: true });
14    cy.get("p").should("be.visible").and("contain", "Kontaktní údaje");
15  });
16  it("Leave input box without any value", () => {
17    cy.get('[type="tel"]').should("be.visible").click().type("{enter}");
18    cy.get("p").should("be.visible").and("contain", "Zadejte prosím telefon");
19  });
20
21  it("Fill in the name", () => {
22    cy.get('[name="name"]').should("be.visible").click()
23      .type("Adam").type("{enter}");
24  });
25  it("Fill in the surname", () => {
26    cy.get('[name="surname"]').should("be.visible").click().type("Pechar");
27  });
28  it("Fill in the surname", () => {
29    const phone = "[type='tel']";
30    cy.get(phone).should("be.visible").click()
31    .type("6478954").type("{enter}");
32    cy.get(phone).clear();
33    cy.get(phone).should("be.visible").click()
34    .type("+420 758 987 888").type("{enter}");
35  });
36  it("Fill in the surname", () => {
37    const email = "[type='email']";
38    cy.get(email).should("be.visible").click().type("test").type("{enter}");
39    cy.get(email).clear();
40    cy.get(email).should("be.visible").click().type("ada.pechar@gmail.com").type("{enter}");
41  });
```

Obrázek 16 – Test formuláře, práce autora

4.5.7 **Test stránky zadávací dokumentace**

Další skript se zabývá testováním stránky s tvorbou zadávací dokumentace a příslušných podstránek o metodikách Waterfall a Agile, které jsou s touto stránkou spojené. V prvním kroku se provede kontrola, zda je správná výchozí poloha v aplikaci. V dalším kroku se zavolá vytvořená funkce, které najde všechny nadpisy úrovně h5 na stránce. Obdržená hodnota z předchozího volání se porovná vůči zadané hodnotě, zda se tyto čísla navzájem rovnají. V případě, že jsou předchozí kroky úspěšné, tak se provede porovnání obdržených hodnot textových řetězců z tohoto volání, v tomto případě nadpisů h5 se zadanými hodnotami v argumentech metody. Dále test klikne na tlačítko s detailem metodiky Waterfall. Proběhne kontrola, zda se zobrazil správný obsah, vrátí se zpátky na výchozí stránku s tvorbou dokumentace a provede totožný krok, tentokrát pro Agile. Následuje krok scrollování obsahu, který bývá často problematický ve spojitosti s automatickým testováním. Provede se scrollování na konec stránky, do úrovně, kde se nachází patička. V závislosti na tomto kroku se

ověří, že v postranní navigaci, která by měla detekovat vizuálně úroveň, ve které se uživatel nachází. Konkrétně se zacílí poslední prvek v navigaci, který obsahuje příslušný text a současně by měl mít aktivní třídu, která signalizuje pozici v dané části aplikace.

```
9     it("Check current location", () => {
10        cy.url().should("include", "aswa.cz/tvorba-zadani");
11        cy.checkUrl("aswa.cz/tvorba-zadani");
12    });
13    it("Check number of headings", () => {
14        //This method search all elements on the page.
15        //Compare entered numbers and check all headings.
16        cy.checkNumberOfElements("h5", 5, "Waterfall", "Agile",
17        "Specifikace", "Kvalita zadání", "Pomoc v neznámém prostředí");
18    });
19    it("Go to Waterfall and Agile methodology pages", () => {
20        cy.contains("Více o waterfall metodice").click();
21        cy.checkUrl("co-je-waterfall");
22        cy.go("back");
23        cy.checkHeading("h2", "Tvorba zadávací dokumentace");
24        cy.contains("Více o Agile").click();
25        cy.checkUrl("co-je-agile");
26    });
27    it("Scroll whole content of the page", () => {
28        cy.get('footer').scrollIntoView().should('be.visible');
29        cy.get("a").should("be.visible")
30        .and("contain", "Berte software jako provozní výdaj")
31        .and("have.class", "active");
32    });
```

Obrázek 17 – Test zadávací dokumentace, práce autora

4.6 Porovnání obou přístupů testování

Po zvolení nástroje pro automatizaci, zhodnocení požadavků testování, vytvoření testovacího scénáře a naprogramování automatických testů, tak přichází na řadu manuální testování a spuštění všech automatických testů. Je nutné podoktnout, že již v průběhu vytváření automatických testů bylo nezbytné provádět částečné manuální testování, jelikož nelze vytvořit automatické testy bez toho, aniž by nejdříve proběhlo manuální testování z důvodu, že je nutné ověřit výchozí stav aplikace, který se musí shodovat s testovacím scénářem. V této části jsou popsané jednotlivé rozdíly obou přístupů, které vyplynuly během testování.

Testování softwaru je rozsáhlá oblast, kterou lze rozdělit do dvou oblastí na manuální a automatické testování. Porovnávat automatické testováním s manuálním není jednoduché. Každý z těchto přístupů má své výhody a nevýhody. Pomocí výsledků z provedení obou přístupů to lze do určité míry objektivně porovnat. Hlavní dva faktory, na které se kladou

nejvyšší nároky je časové, respektive finanční hledisko a kvalita. Cílem každého úspěšného projektu je snížit náklady a čas potřebný k úspěšnému dokončení při zachování kvality.

4.6.1 Výhody a nevýhody manuálního testování

Manuální testování je bezesporu základ, bez kterého se vývoj softwaru neobejde. Tento způsob tetování lze provádět na všech typech zařízení a prohlížečů. V případě, že se jedná o menší projekt nebo se už nachází v pozdní fázi, tak se jedná rozhodně o výhodnější přístup. V případech, kdy se často mění uživatelské rozhraní, tak je tento způsob upřednostňován. Je levnější z pohledu počáteční investice a nevyžaduje znalost programování ani nástroje pro automatické testování.

Na druhou stranu je testování časově náročné, pokud se jedná o regresního a akceptační testování nebo vícejazyčnou aplikaci, případně pokud je nutné testovat na více zařízeních, tak čas potřebný pro otestování mnohonásobně roste. Po několikátém průchodu aplikací jsem začínal ztrácet koncentraci a musel jsem se kvůli zhoršené pozornosti vracet. Čím více průchodů testovacího scénáře jsem provedl, tím se zvyšovalo riziko vzniku chyby, jelikož se začínalo jednat o rutinní záležitost.

4.6.2 Průběh manuálního testování

Manuální testování probíhalo podle testovacího scénáře krok za krokem, celkově se uskutečnilo pět cyklů testování. První manuální testovací cyklus probíhal v rámci desktopové verze, který se uskutečnil skrz webový prohlížeč Chrome. Další dva cykly proběhly rovněž v desktopové verzi, ale tentokrát na prohlížečích Firefox a Microsoft Edge. Následoval testovací cyklus na mobilním zařízení Android s prohlížečem Chrome. Poslední testovací cyklus se uskutečnil na mobilní zařízení iPhone, který má operační systém iOS na prohlížeči Safari. Se zahájením manuálního testování bylo současně nutné ručně zadávat výsledky, což zabralo samo o sobě určité množství času.

4.6.3 Výhody automatického testování

Základní výhodou automatického testování je úspora času a rychlost testů oproti manuálním testů. Další klíčová výhoda je fakt, že není vyžadována přítomnost testera, ale testy se mohou spustit společně s novou verzí aplikace a současně je poskytnutá okamžitá zpětná vazba. Při automatickém testování není nutné automatizovat všechny testovací případy, což

kolikrát ani není možní. Přínos spočívá i v částečném pokrytí, které může zahrnout časově náročné úkony pomocí manuálního testování. Obrovskou výhodou automatických testů v Cypressu je jednoznačně Test Runner Dashboard. Poskytuje přívětivé grafické rozhraní a umožňuje přímo před samotným spuštěním testů zvolit konkrétní prohlížeč, ve kterém se mají testy spustit. Oprava jednotlivých testů, které neprocházejí je mnohem jednodušší. Pomocí vizuálního znázornění každého kroku, který testovací sada obsahuje je znán přesný stav před konkrétním krokem, který selhal. V případě potřeby je k dispozici report z konzole s chybovou hláškou. V momentu, kdy je vytvořené určité množství nových metod, tak se celý proces několikanásobně zrychlí. Jednotlivé kroky se často opakují, tudíž vytvořená metoda se na začátku znovu používá na několika místech celého projektu, pouze s rozdílnými vstupními údaji jako parametry této metody. Automatické testy jsou investicí do budoucna a zajistí lepší celkové pokrytí testů pro daný software. V neposlední řadě se jedná i o „odpočinek" v určité fázi, jelikož testy provedou práci za nás a po neustálém manuálním testování se jedná o příjemnou změnu, která ve finále ulehčí i manuální testování.

4.6.4 **Nevýhody a překážky automatického testování**

Automatické testování přináší celou řadu výhod, ale tím pádem i určité nevýhody. Jedna ze základních nevýhod automatických testů je nutnost upravit testy kvůli změně v aplikaci. V jeden moment mi již existující a úspěšně procházející testy začaly padat. Po dlouhém debuggování a vyhledávání na internetu jsem se stále nedopracoval k jejich opravě. Po pár hodinách jsem si všiml, že určitá část aplikace byla změněná, tudíž ani již dříve funkční testy procházet nemohly. Bylo nutné zjistit provedené změny, přepsat testovací scénáře, provést manuální testování, aby se potvrdila funkčnost a požadavky z testovacího scénáře a nakonec se musela provést úprava v kódu. Nevýhoda testovacího nástroje Cypress je, že přímo nepodporuje více karet a oken v prohlížeči, tudíž je nutné používat náhradní řešení. Další nevýhoda spočívá v občasném neočekáváném chování asynchronních operací, které se ne vždycky chovají tak, jak by měly. V nejnovější verzi Cypressu pořád chybí podpora prohlížeč od Applu Safari. V některých případech se také stávalo, že se pár testů nechovalo vždy stabilně, tudíž jsem musel hledat způsoby, jak tuto stabilitu zajistit.

4.6.5 **Nemožnost automatizovat určité úkony**

Ne všechny testovací případy se dají automatizovat. U některých případů to není možné a u jiných to zkrátka nejde. Problematická část je scrollování do určité části stránky, tudíž

„single page" aplikace je náročnější automaticky testovat. V tomto případě jsem využíval možnost scrollování v závislosti do úrovně specifického elementu na stránce. Interakce s videem je omezená pouze na zapnutí a vypnutí, které většinou prochází "antipattern", kterým je čekání na akci, což se rozhodně nedoporučuje, ale v tomto případě nemusí být vždy jasné, že se dané video vůbec spustilo. Další funckionality videa otestovat nelze. Tento případ jsem omezil na zapnutí a vypnutí videa, ale zbylé funkce jsem otestoval pomocí manuálního testování. Interaktivní a responzivní části aplikace také představují problém pro automatické testy, jelikož se tyto eventy špatně podchycují. Notifikace v rámci aplikace, které se zobrazí po určitém úkonu bývá také složité automaticky otestovat, jelikož většinou po chvilce zmizí a testovací nástroj je nestihne zacílit. V případě, že neexistují vhodné selektory pro zacílení určité komponenty, tak bývá velice složité a řešení nemusí být přívětivé, jelikož s menší změnou můžou přestat procházet testy s tímto spojené. Vizuální testování také není stoprocentně spolehlivé, jelikož prostředí bývá často responzivní a nemusí se vždy přesně vykreslit veškeré prvky včas a malá odchylka znamená selhání testu. V případě, že se nastaví odchylka s větší tolerancí, tak se ztrácí přínos samotného testu, jelikož připouštíme rozdíl od požadovaného řešení.

4.6.6 **Průběh automatického testování**

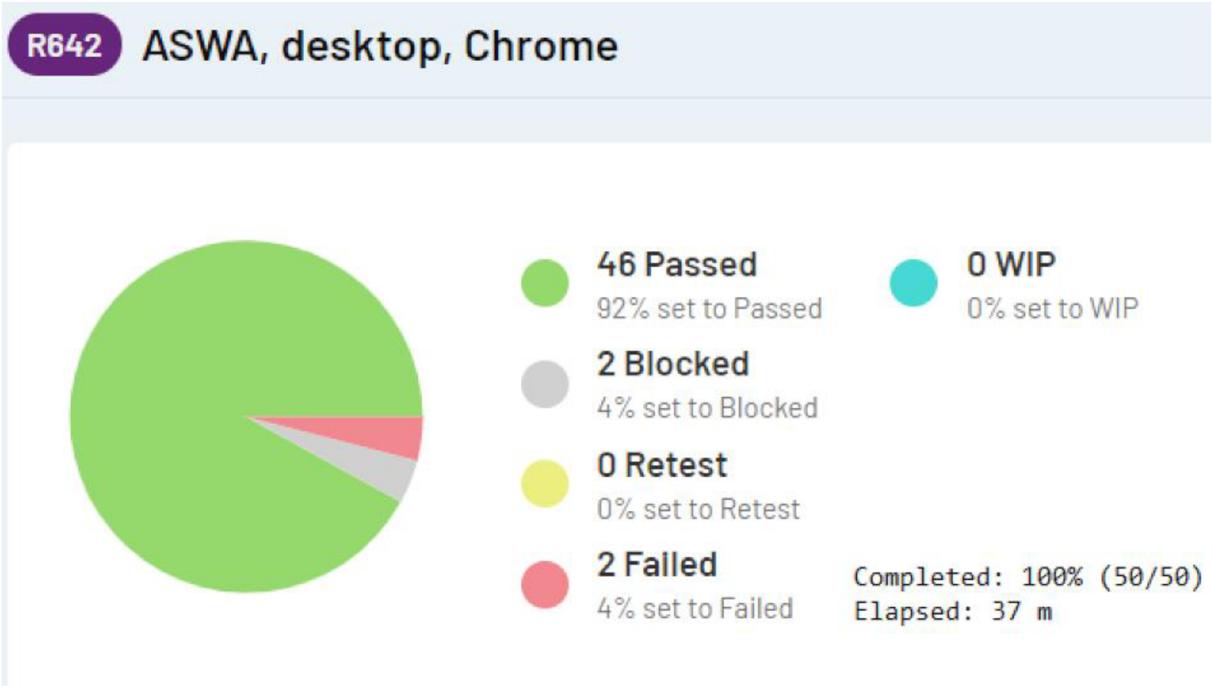
Automatické testy byly realizovány na základě testovacího scénáře. Je nutné podotknout, že kompletně celý testovací scénář nebyl pokryt automatickými testy, což nebyl ani původní záměr. Automatické testy se spouštěli v prohlížečích Chrome, Firefox a Microsoft Edge. Nejdříve tyto testy proběhly v desktopovém a následně mobilním zobrazení s tím rozdílem, že v rámci mobilní verze byly spuštěné pouze dva cykly, aby se celkový počet tetovacích cyklů v rámci manuálního a automatického shodoval. První spouštění se potýkalo s řadou problému, jelikož se vyskytla chyba při implementaci a ve dvou případech na sobě testy byly závislé. Další problém spočíval v tom, že bylo nutné odladit testy, aby vykazovaly konzistentní výsledky a testy procházely ve všech případech.

5 Výsledky a diskuze

V rámci této kapitoly se uvedou výsledky z praktické části a následně budou navzájem porovnány.

5.1 Výsledky manuálního testování

Počátek manuálního testování začal s tvorbou testovacích scénářů. S politováním nelze se stoprocentní přesností určit přesný čas, který byl potřebný pro vytvoření všech testovacích scénářů, jelikož se v průběhu muselo část těchto scénářů modifikovat podle vzniklých změn v aplikaci. Nejedná se však o zásadní odchylku, která by zásadně ovlivnila celkové měření. V rámci sestavení těchto scénářů jsem současně vytvářel automatické testy a průběžně prováděl manuální testování. První měření manuálního testování se uskutečnilo pro desktopovou verzi prohlížeče chrome, které zabralo 37 minut. Také je potřeba zmínit, že při prvním běhu se objevily chyby v aplikaci, tudíž výsledek prvního testu neprošel se stoprocentní úspěšností.



Obrázek 18 – Výsledek manuálního testování, práce autora

Výsledky dalších cyklů manuálního testování jsou uvedené v následující tabulce.

Běh číslo	Způsob testování	Zařízení	Typ prohlížeče	Čas testování
1	manuální	notebook Dell inspiron 15	Chrome	37 min
2	manuální	notebook Dell inspiron 15	Firefox	29 min
3	manuální	notebook Dell inspiron 15	Microsoft Edge	40 min
4	manuální	Honor 10	Chrome	34 min
5	manuální	iPhone 6S+	Safari	27 min
			Průměrný čas	32,4 min
			Celkový čas	167 min

Tabulka 1 – Výsledek manuálního testování, zdroj autor

Z této tabulky vidíme, že průměrný čas, který byl potřeba k otestování je 32,4 minut. Je důležité zmínit, že objevení chyby při testování mělo za následek zvýšení času, který byl potřeba pro celkové dokončení testování, jelikož se nalezená chyba musela zaznamenat. Jak z tabulky vyplývá, tak celková doba, která je potřeba na otestování celé aplikace je poměrně veliká. Otestovat aplikaci v rámci několika prohlížečů a zařízení trvalo 167 minut.

5.2 Výsledky automatického testování

Pokrytí testovacích scénářů automatickými testy vyžadovalo jejich pečlivou kontrolu. Bohužel se nepodařilo pokrýt všechny testovací scénáře, jelikož to nebylo z již zmíněných důvodů možné, nicméně se velkou část testovacích případů pokrýt podařilo. Jednotlivé testovací podmínky byly stanovené stejně, jako tomu je v testovacím scénáři.

Obrázek 19 – Výsledek automatického testování, práce autora

Na předchozím obrázku je vidět podrobný report z prvního běhu, který trval 2 minuty a 14 vteřin. Existující testy se mi podařilo napojit přes CI, tudíž s novou verzí aplikace se automaticky spustila pipelina, která zajistila spuštění všech testů. Tento čas byl delší, jelikož se jednalo o běh skrze CI přes napojení na Cypress dashboard. Cypress Dashboard v tomto případě

také ukazuje možnost zlepšení jednotlivých běhů, pokud by byly testy spuštěné paralelně. Součástí výstupu je i detailní analýza pro konkrétní běhy testů. Ruční spuštění, které probíhá lokálně trvalo kratší dobu.

Běh číslo	Způsob testování	Zařízený	Typ prohlížeče	Čas testování
1	automatický	viewport 1920x1080	Chrome	71 s
2	automatický	Mackbook 15	Firefox	95 s
3	automatický	Mackbook 11	Microsoft Edge	63 s
4	automatický	Samsung S10	Chrome	82 s
5	automatický	iPhone X	Electron	86 s
			Průměrný čas	79,4 s
			Celkový čas	397 s

Tabulka 2 – Výsledek automatického testování, zdroj autor

Průměrný čas činí 79,4 vteřin a celkový čas činil 397 vteřin, což je 6 minut a 37 vteřin. I za předpokladu, že nebyly pokryté komplet veškeré testovací případy, tak je rozdíl evidentní. Úkony, které trvají člověku několik minut, tak zvládne počítač během několika vteřin. Celkový čas, který byl nutný pro vytvoření automatických testů a jejich následná úprava ve výsledku dává 84 hodin, což je zhruba 3,5 dne. Pokud se to přepočítá na pracovní dny neboli man-day(MD), kdy 1MD odpovídá 8 hodinám práce, tak to vychází na 11,75MD. S ohledem na to, kolik času zaberou manuální testy, tak lze prohlásit, že je to velice pěkné číslo. Tento čas také zásadně ovlivnil fakt, že se v průběhu měnila testovaná aplikace, tudíž bylo nutné již existující automatické testy modifikovat. Velká výhoda spočívá v rozdělení testů, jelikož jsou jednotlivé testy na sobě navzájem nezávislé. V případě, že jeden test neprojde, tak to nebude mít vliv na ostatní testy.

5.3 Metriky a návratnost automatizace

5.3.1 Pokrytí testovacích scénářů automatizací

Z této metriky nám vyplývá, že celkový počet testovacích scénářů činí 66 a z toho se podařilo automatizovat 50. Jak již bylo zmíněno, tak některé části aplikace bylo buď nemožné nebo nevýhodné automatizovat, tudíž ani nebylo cílem mít stoprocentní pokrytí všech testovacích scénářů. Pokrytí automatizace = (počet automatizovaných testů / celkový počet testů) * 100. Výsledné pokrytí tudíž činí (50/66) * 100 = 75,76 %.

5.3.2 Rychlost běhu automatických testů

V tomto ohledu dost záleží na typu testů a také nástroje, který se k automatizaci používá. Dříve byla rychlost běhu automatických testů problém, jelikož trvalo opravu dlouho, než všechny spuštěné testy doběhly. Tohle se rozhodně netýká testovacího nástroje Cypress. Pro porovnání automatického a manuálního testování slouží následující výpočet. Rychlost automatických testů = Doba manuálního testování – celková doba trvání běhu automatických testů. Pokud je výsledek kladný, tak dostaneme číslo, které nám udává o kolik jsou automatické testy rychlejší. V opačném případě zjistíme o kolik jsou naopak automatické testy pomalejší. 167 – (397/60) = 160,38. Z čehož vyplývá, že po dokončení běhu celého testovacího cyklu, tak automatické testování proběhne o 160 minut rychleji, než testování manuálním způsobem.

5.3.3 Návratnost automatizace

Návratnost automatizace je základní metrika, která určuje, jestli se investice do procesu automatizace vyplatila, respektive za jak dlouho se začne vracet. Tímto výpočtem zjistíme, kolik se ušetří času pro testování pomocí automatizace. Bez této metriky nelze objektivně stanovit, zda se investice do automatizace vyplatila a v jakém bodě se začíná vracet. Návratnost investice se vypočítá následujícím způsobem. Návratnost = úspory / investice. Z čehož dostáváme návratnost automatizace = [167-(397/60)] / (84*60) * 100 = 3,182 %. Tudíž časová úspora v rámci jednoho testovacího cyklu vychází na 3,182 %.

K výpočtu lze přistupovat opačným způsobem, tudíž výpočet pro návratnost v rámci testovacích cyklů vypadá následovně. Návratnost = Náklady na vytvoření automatických testů / Časová úspora. Návratnost = (84*60) / [167 - (397/60)] = 31,42. Pro správnost výpočtu můžeme ověřit 100 / 3,182 = 31,42. Tímto jsme si ověřili správnost předchozího výpočtu. Z toho nám ve finále vychází, že návratnost automatických testů nastane po provedení 31 testovacích cyklů.

5.4 Shrnutí automatizace a zhodnocení výsledků

Z počátku se může manuální testování zdát jako výhodnější než automatické, jelikož se jedná o velkou počáteční investici. Nastavení a odladění všech testů bývá náročná záležitost. Nehledě na to, že se v průběhu vytváření automatických testů musí občas upravit již existující testy, kvůli změně v samotné aplikaci nebo nové specifikaci, což ostatně byl i tento případ. Zde je ovšem důležité mít na paměti, že automatické testování vyžaduje největší investici na samotném počátku. Čím déle projekt trvá a čím více proběhne testovacích cyklů, tím více se

zavedení automatického testování vyplácí. Ukončením vývoje a nasazením finální verze na produkční prostředí práce nekončí. Následuje údržba a podpora aplikace, tudíž i v této fázi mají automatické testy své opodstatněné místo. Jelikož se jedná o dlouhodobí projekt, který bude potřeba udržovat i do budoucna, tak je přínos automatických testů v tomto případě jednoznačný. Návratnost automatizace se začne vracet po uplynutí 31 testovacích cyklů a podle stanoveného plánu, tak bude vyžadováno několikanásobně více těchto cyklů provést. V tomto případě se jednalo o webovou aplikaci a šlo primárně o testování GUI, potažmo částečného integrační testování. Automatické testy si konzistentně drží svou kvalitu a poskytují relevantní výsledky, na které se lze spolehnout v případě, že je zajištěna jejich stabilita. Již existující testy a části kódu lze znovu použít a tím se zmenšují náklady na vytvoření nových testů. Nehledě na to, že tester může spustit automatické testy a během toho se věnovat jiné aktivitě, což je ve finále velice efektivní přístup. Existují případy, kdy automatizace smysl nedává, např. pokud se jedná o pozdní fázi vývoje a není v plánu navazující vývoj nebo naopak, pokud testovací tým nemá dostatek zkušeností, aby se investovaný čas do tohoto procesu navrátil. Pomocí částečného pokrytí testovacích scénářů skrze automatické testy se docílí toho, že vznikne prostor pro jiné druhy testování, na které nezbýval čas, a tudíž se zvětší celkové pokrytí testů, což vede k celkově větší kvalitě produktu. Stále se zde nachází prostor pro zlepšení, jelikož některé testovací případy nebylo možné automatizovat. Buď automatizace daného scénáře byla příliš složitá nebo to nebylo technicky možné. To se může brzy změnit, poněvadž posun v této oblasti jde skokově dopředu.

6 Závěr

Tato bakalářská práce v rámci teoretické části poskytla metody pro vývoj softwarového životního cyklu. Dále vymezila problematiku testování, konkrétní přístupy a jednotlivé způsoby testování. Následně byla vymezena oblast automatického testování, metriky a návratnost automatizace, proces pro zavedení automatizace, kritéria při výběru nástroje automatického testování a byly popsány konkrétní nástroje.

V praktické části byl popsán důvod pro zavedení automatizace pro konkrétní webovou aplikaci a současně odůvodněn výběr zvoleného testovacího nástroje. Dále byly vytyčené případy, které jsou vhodné pro automatizaci. Následně byl vytvořen testovací scénář, podle kterého byly naprogramovány automatické testy. Byla popsána vhodnost použití jednotlivých druhů selektorů a také využití metod pro opakující se části kódu. Následně proběhlo manuální a automatické testování a byly diskutovány poznatky v rámci průběhu celého testování.

Oba tyto přístupy byly porovnány, byla vypočítána návratnost a byly zhodnoceny výsledky, ze kterých byl stanoven závěr. Postupně docházelo k naplnění hlavních i dílčích cílů celé práce.

7 Seznam použitých zdrojů

- IBM research. Software Development. [online]. 01.07.2014 [cit. 2021-8-28]. Dostupné z: https://researcher.watson.ibm.com/researcher/view_group.php?id=5227

- IBM. *Software Development: What is software development?* [online]. [cit. 2021-8-29]. Dostupné z: https://www.ibm.com/topics/software-development

- Řepa, V. Analýza a návrh informačních systémů. Ekopress, Praha 1999, ISBN 80-86119-13-0. str. 17-19

- Dziuba, Anna. *The Software Development Process We Used to Build 200+ Products* [online]. [cit. 2021-8-29]. Dostupné z: https://relevant.software/blog/7-steps-for-effective-software-product-development/

- ALTVATER, Alexandra. What Is SDLC? Understand the Software Development Life Cycle [online]. APRIL 8, 2020. [cit. 2021-8-29]. Dostupné z: https://stackify.com/what-is-sdlc/

- Innovative Architects. 8 Models of the System Development Life Cycle: What's the best SDLC methodology: Waterfall, Agile, Lean, Iterative, Prototyping, DevOps, Spiral or V-model? [online]. APRIL 8, 2020. [cit. 2021-8-29]. Dostupné z: https://www.innovativearchitects.com/KnowledgeCenter/basic-IT-systems/8-SDLC-models.aspx

- SAMI, Mohamed. Software Development Life Cycle Models and Methodologies [online]. 15.03.2021. [cit. 2021-8-29]. Dostupné z: https://medium.com/@melsatar/software-development-life-cycle-models-and-methodologies-297cfe616a3a

- Arnon Axelrod – Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects, 2018. ISBN-13: 978-1484238318.

- Roman Adam – Study Guide to the ISTQB (R) Foundation Level 2018 Syllabus. ISBN13: 9783319987392

- Kitner – Co je testování softwaru? [online]. [cit. 2021-8-29]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-softwaru/

- SHARMA, Lakshay. Why is Testing Necessary? [online]. [cit. 2021-8-29]. Dostupné z: https://www.toolsqa.com/software-testing/istqb/why-is-testing-necessary/

- KINSBRUNER, Eran. Manual Testing vs Automated Testing [online]. 13.08.2019. [cit. 2021-8-29]. Dostupné z: https://www.perfecto.io/blog/automated-testing-vs-manual-testing-vs-continuous-testing

- Kitner. *Typy testování software (třídění testů)* [online]. [cit. 2021-8-29]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/

- KUMAR, Maruthi. Major testing types -- Black Box, White Box, Grey Box and Visual testing [online]. 2012. [cit. 2021-8-29]. Dostupné z: http://maruthisqa.blogspot.com/2012/11/major-testing-types-black-box-white-box.html

- HAMILTON, Thomas. *Major testing types -- Black Box, White Box, Grey Box and Visual testing* [online]. 06.11.2012. [cit. 2021-8-29]. Dostupné z: https://www.guru99.com/automation-testing.html

- REHKOPF, Max. *Automated software testing* [online]. [cit. 2021-8-29]. Dostupné z: https://www.atlassian.com/continuous-delivery/software-testing/automated-testing

- PEKAŘ, Lukáš. AUTOMATICKÉ TESTOVÁNÍ SOFTWARU – NEBOLI KÓD, CO KONTROLUJE KÓD [online]. 2017. [cit. 2021-8-29]. Dostupné z: https://bonsai-development.cz/clanek/automaticke-testovani-softwaru

- LUTKEVICH, Ben. System software [online]. 2020. [cit. 2021-8-29]. Dostupné z: https://www.techopedia.com/definition/4224/application-software

- LUTKEVICH, Ben. LutkevichSystem software. Whatls [online]. 2021, únor 2021 [cit. 2022-02-28]. Dostupné z: https://whatis.techtarget.com/definition/system-software

- A, Josh. The Circle of Life...(SDLC). *Medium* [online]. 2019, 17.01.2019 [cit. 2022-02-28]. Dostupné z: https://medium.com/@joshalphonse/the-circle-of-life-sdlc-b1b14686d683

- BUREŠ, MIROSLAV, Miroslav RENDA, et al. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. 2016. Praha: Grada. ISBN 978-80-247-5594–6.

- *Testim: Test Automation Benefits: 12 Reasons to Automate in 2020* [online]. 06.11.2019 [cit. 2022-02-28]. Dostupné z: https://www.testim.io/blog/test-automation-benefits/

- CUMMINGS-JOHN, Ronald. *What is automation testing?* [online]. [cit. 2022-02-28]. Dostupné z: https://www.globalapptesting.com/blog/what-is-automation-testing

- ARUMUGHOM, Dhanya. *Automation Testing Explained* [online]. 25.06.2018 [cit. 2022-02-28]. Dostupné z: https://browsee.io/blog/automation-testing-explained/

- SMRITI, Arya. *All You Need To Know About Automation Testing Life Cycle* [online]. 01.04.2019 [cit. 2022-02-28]. Dostupné z: https://www.lambdatest.com/blog/all-you-need-to-know-about-automation-testing-life-cycle/

- SHAIKH, Faiy. *4 Simple Steps to Select the Right Test Automation tool for your Project* [online]. [cit. 2022-02-28]. Dostupné z: https://www.saviantconsulting.com/blog/4-steps-select-test-automation-tool.aspx

- HAMILTON, Thomas. *What is Security Testing?* [online]. 12.02.2022 [cit. 2022-03-12]. Dostupné z: https://www.guru99.com/what-is-security-testing.html

- *Atlanta TEAMCypress* [online]. 2022 [cit. 2022-03-15]. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress

- VAIDYA, Neha. *Puppeteer vs Selenium: Core Differences* [online]. 26.03.2021 [cit. 2022-03-11]. Dostupné z: https://www.browserstack.com/guide/puppeteer-vs-selenium

- KASSANDRA, Helene. *Playwright vs Cypress* [online]. 06.11.2020 [cit. 2022-03-11]. Dostupné z: https://medium.com/sparebank1-digital/playwright-vs-cypress-1e127d9157bd

- BOSE, Shreya. Calculating Test Automation ROI: A Guide. *BrowserStack* [online]. 21.09.2021 [cit. 2022-02-28]. Dostupné z: https://www.browserstack.com/guide/calculate-test-automation-roi

- CHATURVEDI, Varun. Test Automation ROI: How Do We Calculate It?. *QAonCloud* [online]. 2022, 04.01.2022 [cit. 2022-02-28]. Dostupné z: https://www.qaoncloud.com/test-automation-roi-how-do-we-calculate-it/

- TRUNKETT, Oliver. SDLC Methodologies: From Waterfall to Agile: A guidebook for software development life cycle (SDLC) methodologies written by a software delivery expert. *Virtasant* [online]. 27.08.2020 [cit. 2022-02-28]. Dostupné z: https://www.virtasant.com/blog/sdlc-methodologies

- KNOTT, Daniel. The Mobile Test Pyramid. *Ministry of testing* [online]. 31.12.2015 [cit. 2022-02-28]. Dostupné z: https://www.ministryoftesting.com/dojo/lessons/the-mobile-test-pyramid

- CYPRESS.IO. Best practices: Selecting Elements. *Cypress.io* [online]. [cit. 2022-02-28]. Dostupné z: https://docs.cypress.io/guides/references/best-practices