Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Porovnání manuálního a automatizovaného testování softwaru

Ondřej Hofmann

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Ondřej Hofmann

Informatika

Název práce

Porovnání manuálního a automatizovaného testování softwaru

Název anglicky

Comparison of manual and automated software testing

Cíle práce

Hlavním cílem je porovnat efektivitu, kvalitu a pokrytí manuálního a automatizovaného testování softwaru. Mezi vedlejší cíle patří zhodnotit náklady a časovou náročnost obou metod, identifikovat vhodné situace pro použití každé metody a formulovat doporučení pro praxi.

Metodika

Metodika řešení práce je v teoretické části založena na studiu odborných zdrojů a analýze současných metod a nástrojů používaných v manuálním a automatizovaném testování. V praktické části budou použity kvantitativní metody – implementace testovacích scénářů pro obě metody na vybrané aplikaci, měření času potřebného k provedení testů, analýza nalezených chyb a pokrytí testů. Data získaná z experimentů budou analyzována a na základě zjištění budou formulovány závěry a doporučení.

Doporučený rozsah práce

40-60 stran

Klíčová slova

automatizované testování, manuální testování, testování softwaru, Test execution, Bug detection, test case, software quality assurance, regresní testování

Doporučené zdroje informací

Beizer, Boris. Software Testing Techniques. New York: Van Nostrand Reinhold, 1990.
Fewster, Mark; Graham, Dorothy. Software Test Automation: Effective Use of Test Execution Tools. Reading: Addison-Wesley, 1999.
Ministry of Testing. [online]. Dostupné z: https://www.ministryoftesting.com/
Myers, Glenford J.; Sandler, Corey; Badgett, Tom. The Art of Software Testing. Hoboken: Wiley, 2011.
Selenium Documentation. [online]. Dostupné z: https://www.selenium.dev/documentation/
Software Testing Help. [online]. Dostupné z: https://www.softwaretestinghelp.com/

Předběžný termín obhajoby

2024/25 LS – PEF

Vedoucí práce

doc. Ing. Michal Stočes, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 14. 11. 2024

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 21. 11. 2024

prof. PhDr. Michal Lošťák, Ph.D.

Pověřený vedením

V Praze dne 29. 11. 2025

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Porovnání manuálního a automatizovaného 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 29.11.2025 _____

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Michalu Stočesovi, Ph.D. za cenné rady, odborný nadhled a vstřícný přístup během konzultací a při vypracování této bakalářské práce.

Porovnání manuálního a automatizovaného testování softwaru

Abstrakt

Bakalářská práce se zabývá porovnáním manuálního a automatizovaného testování softwaru. Cílem této práce je analyzovat a zhodnotit vybrané metody na konkrétní aplikaci. Práce se skládá z teoretické a praktické části. V první, teoretické části je představeno testování softwaru. Jsou vysvětleny základní pojmy a principy testovacího procesu, jednotlivé fáze testování, role v testovacím týmu a typy testovacích technik. Dále jsou zde popsány úrovně testování softwaru a definovány nástroje pro automatizované testování. Závěrem se teoretická část zaměřuje na vícekriteriální analýzu, která je použita v praktické části.

Ve druhé, praktické části práce je popsán výběr softwaru pro testování, výběr vhodných scénářů pro vybranou aplikaci a následně byly tyto scénáře implementovány do vybraných metod testování. Poté byl stanoven postup a kritéria hodnocení. Bylo provedeno samotné testování a jednotlivé metody byly ohodnoceny na základě stanovených kritérií. V závěru práce jsou identifikovány vhodné situace pro použití každé metody a formulována doporučení pro praxi.

Klíčová slova: testování softwaru, manuální testování, automatizované testování, testovací scénář, Selenium, Cypress, Playwright, zajištění kvality softwaru, Vícekriteriální analýza variant

Comparison of manual and automated testing

Abstract

This bachelor's thesis deals with the comparison of manual and automated software testing. The aim of this thesis is to analyze and evaluate selected testing methods on a specific application. The thesis consists of a theoretical and a practical part. The first, theoretical part introduces software testing. It explains the basic concepts and principles of the testing process, the individual phases of testing, the roles in the testing team, and the main testing techniques. It also describes the levels of software testing and presents tools for automated testing. Finally, the theoretical part focuses on multi-criteria analysis, which is used in the practical part.

The second, practical part of the thesis describes the selection of software for testing, the selection of suitable scenarios for the chosen application, and the subsequent implementation of these scenarios using the selected testing methods. The procedure and evaluation criteria were then established. The testing itself was carried out and the individual methods were evaluated based on the defined criteria. The conclusion of the thesis identifies suitable situations for the use of each method and formulates recommendations for practice.

Keywords: software testing, manual testing, automated testing, test case, Selenium, Cypress, Playwright, software quality assurance, Multiple-Criteria decision analysis

Obsah

1 Úvod

Testování softwaru je důležitou součástí vývoje softwaru, která může zabírat významnou část celkového času projektu. Při vývoji aplikace je poměrně snadné přehlédnout chybu v kódu nebo v návrhu. Tyto chyby mohou způsobit velké ekonomické škody a také vést ke ztrátě důvěry uživatelů. Proto je testování softwaru nedílnou součástí vývoje, jehož cílem je tyto chyby včas odhalit a vyhnout se tak následným ztrátám.

Software lze testovat různými způsoby. Tradičním přístupem je manuální testování, při kterém tester ručně provádí kroky podle testovacího scénáře a ověřuje, zda aplikace funguje správně. Se zvyšujícím se rozsahem vyvíjených aplikací však přestává být manuální testování dostatečně efektivní. Z tohoto důvodu se tak stále častěji zavádí automatizované testování, které umožňuje opakované a rychlé provádění testovacích scénářů.

Bakalářská práce se zabývá problematikou výběru vhodné metody pro testování softwaru se zaměřením na porovnání manuálního a automatizovaného testování. Cílem je porovnat efektivitu, kvalitu a pokrytí vybraných testovacích metod pomocí testovacích scénářů a dále zhodnotit jejich časovou náročnost a náklady.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem je porovnat efektivitu, kvalitu a pokrytí manuálního a automatizovaného testování softwaru. Mezi vedlejší cíle patří zhodnotit náklady a časovou náročnost obou metod, identifikovat vhodné situace pro použití každé metody a formulovat doporučení pro praxi.

2.2 Metodika

Metodika řešení práce je v teoretické části založena na studiu odborných zdrojů a analýze současných metod a nástrojů používaných v manuálním a automatizovaném testování. V praktické části budou použity kvantitativní metody – implementace testovacích scénářů pro obě metody na vybrané aplikaci, měření času potřebného k provedení testů, analýza nalezených chyb a pokrytí testů. Data získaná z experimentů budou analyzována a na základě zjištění budou formulovány závěry a doporučení.

3 Teoretická východiska

3.1 Úvod do testování softwaru

Software se stal nedílnou součástí každodenního života. Ať už se jedná o software v mobilním telefonu, domácích spotřebičích nebo dopravních prostředcích, jeho správné fungování je klíčové. Pokud by software nefungoval správně, mohlo by dojít k velkým škodám nejen na majetku.

K vadám u tohoto softwaru dochází z více důvodů. Ve většině případů se jedná o chybu způsobenou lidským faktorem. Objevují se zde také chyby, které jsou způsobené interakcí mezi testovaným objektem a jeho prostředím. [1]

Oprava těchto škod je levnější v průběhu vývoje než při spuštění. Z tohoto důvodu vzniklo odvětví testování softwaru, které se odhalováním těchto chyb v průběhu vývoje zabývá. [1], [2]

Testování softwaru tedy může být definováno jako činnost, která má za cíl ukázat na rozdíly mezi očekávaným a opravdovým chováním softwaru. Dále může poukázat na nedostatky nebo nekonzistenci v požadavcích již na počátku. [3]

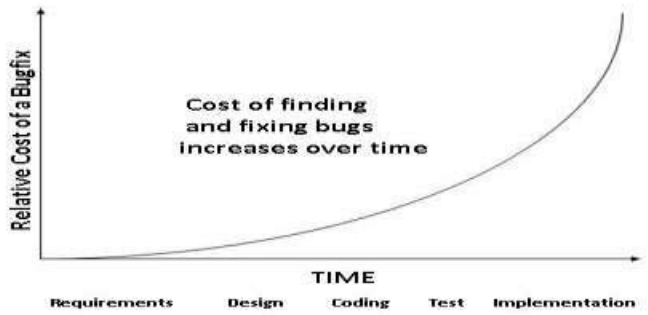
V následující sekci bude představeno testování softwaru.

3.1.1 Cíle a smysl testování

Testování má mnoho cílů. Ať už cíl hlavní, na který se testování zaměřuje, nebo cíle vedlejší, které vyplývají z činnosti testování hlavního cíle. Pokud se konkrétně podíváme na jednotlivé cíle testování, tak zde najdeme na prvním místě ověření kvality softwaru. Není správné říci, že hlavním cílem testování je odhalování chyb. Chyby mohou být odhaleny kterýmkoli uživatelem softwaru. Testování zajišťuje kontrolu kvality softwaru. Testování tak vydává zprávu o stavu softwaru oproti požadovanému stavu. To se následně používá pro určení, zda produkt funguje podle očekávání. Testovací proces tak určí, zda software splňuje podmínky pro vydání. Z tohoto důvodu se testování občas nazývá quality assurance (zajištění kvality). [1], [4]

Pokud se detailněji podíváme i na vedlejší cíle, najdeme zde snahu o snížení nákladů na vývoj softwaru. Jak již bylo řečeno v předchozí kapitole, oprava chyb je levnější na začátku vývoje aplikace než na jejím konci. A tato cena za opravu roste s postupem do každé

další vývojové fáze exponenciálně. Pokud by byla chyba objevena až po nasazení pro uživatele, opravy mohou být několikasetnásobně vyšší. Když následně porovnáme výdaje na opravu těchto chyb oproti ceně za testovací tým, vyplatí se jeho využití. [4], [5]



Obrázek 1 - Oprava chyb v čase; Zdroj: [6]

V ideálním světě, kde je na vývoj neomezeně času a prostředků, je možné dostatečně otestovat celý software a odhalit všechny jeho nedostatky. V reálném světe tomu tak není, a tak se při testování musí rozumně rozdělit úsilí a správně zvolit priority. Obecně se uznává, že by každá funkce softwaru měla být otestována aspoň jedním pozitivním vstupem. Dále by na hlavní součásti softwaru mělo být využito více úsilí než na ty méně důležité. Mezi vedlejší cíle testování tak patří i určování priorit. [4]

Během testovacího procesu je důležité zaznamenávat výsledky. Ať ty chybové nebo naopak výsledky, které dopadly dobře. To pomáhá zjednodušit testovací proces. Tím se eliminuje možnost duplicitních procesů během testování. Zároveň to ulehčuje práci s ověřováním chování softwaru během oprav chyb. Dalším cílem je tak možnost sledovatelnosti a dohledatelnosti testovací procesu.

Mezi další důležité cíle lze zařadit vyváženost a determinističnost. Testování musí balancovat mezi napsanými požadavky, požadavky uživatelů a možnostmi reálného světa. Během testování se také objeví spousta nepsaných nebo nepřímých požadavků, se kterými pak musí testovací tým pracovat. Je důležité, aby znalosti jednotlivých částí softwaru nebyly pouze na jednom členovi, ale byly řádně rozděleny mezi více testerů. Testování by mělo být jasné a přímočaré. Nesmí být náhodné a jednotlivé výsledky musí být opakovatelné. [4]

3.1.2 **Základní pojmy a definice**

- **Testovací proces** – strukturovaný proces, který je složený z jednotlivých činností, které souvisejí s testováním softwaru a které byly předem

naplánované. Testovací procesy musí být aplikovány, aby bylo možné správně posoudit všechny aspekty testovaného softwaru. [6], [7]

- **Testovací tým** – skupina odborníků, která se zabývá testováním konkrétního softwaru. Tento tým zodpovídá za správné provedení testovacího procesu. [8]
- **Testovací scénář** – série instrukcí, které testera navádějí, jak ověřit určitý cíl. Testovací scénář obsahuje součásti, které popisují vstup, akci a očekávanou odezvu. Při dodržení postupu zodpovídá testovací scénář na otázku, zda je dané chování systému dle očekávání. [9]
- **Testovací sada** – strukturovaný soubor testovacích scénářů. Tyto scénáře jsou spolu pospojovány na základě podobností. Testy, které testují jednu určitou komponentu softwaru jsou tak spojeny do jedné sady. [10], [11]
- **Use case** – neboli scénář použití přesně popisuje, jak může uživatel interagovat se systémem k dosažení určitého cíle. Jednotlivé kroky jsou přesně popsány a možné výsledky těchto akcí jasně definovány. Používá se při designování softwaru, oproti testovacímu scénáři, který je používán při testování. [12]
- **Chyba** – jedná se o odchylku od požadavku zákazníka. Zjednodušeně můžeme říct, že se jedná o rozdíl mezi očekávaným chováním komponenty a jejím opravdovým chováním. Chyby vznikají na základě lidského faktoru. Tedy pokud dojde softwarovým inženýrem k omylu nebo neporozumění během vývojové části. [13]
- **Defekt** – principem se jedná o stejnou věc jako chyba. Rozdíl je zde v tom, že jako defekt jsou označeny ty chyby, které byly nalezeny v rámci fáze testování systému (viz kapitola 3.3.3). [13]
- **Selhání** – projev defektu. Jako selhání se označuje moment, kdy je software dodán klientovi a ten v něm následně objeví chybu/defekt. [13]

3.2 Fáze životního cyklu testování

Fáze životního cyklu testování, občas označované zkratkou jako STLC (Software testing life cycle) je sekvence aktivit, které jsou prováděny testovacími inženýry během testovacího procesu. Tento proces má za úkol ověřit, že se software chová podle očekávání.

Použitím rozumného životního cyklu může organizace dosáhnout lepších výsledků. [14], [15], [16]

Typicky životní proces testování rozlišuje šest částí, a to analýzu požadavků, plánování testů, návrh testovacích scénářů, nastavení testovacího prostředí, provedení testů a uzavření testování. Každá z těchto fází má svůj specifický postup, cíl a výsledky. V následujících podkapitolách jsou jednotlivé fáze podrobněji vysvětleny.[15], [16]

3.2.1 **Analýza požadavků**

Analýza požadavků je první fází životního cyklu testování, ve které se testovací tým seznámí s tím, co je potřeba testovat. Cílem této fáze je identifikace, analýza a dokumentace specifikací a požadavků. Tímto procesem se dosáhne, aby požadavky byly úplné a testovatelné. [15], [16], [17]

Mezi činnosti, které se v této fázi provádějí, patří:

1. Kontrola dokumentů s požadavky i dalších souvisejících dokumentů
2. Shromažďování dalších informací o požadavku
3. Identifikace mezer a nejasností v požadavcích či úplně chybějící požadavky
4. Identifikace jakýchkoliv rizik, které by mohly mít vliv na testování
5. Určování priorit testování
6. Kontrola proveditelnosti automatizace [15], [16], [17]

V této fázi je zároveň vhodné udělat matici sledovatelnosti požadavků. Ta se používá pro schopnost dosledovat artefakty k jejich požadavkům. V procesu vývoje softwaru to znamená, že každá změna kódu má být dohledatelná k původnímu požadavku. Z pohledu testování softwaru to znamená, že každá testovací aktivita má být zpětně dohledatelná k původnímu požadavku. Tím zabráníme, aby se testovaly věci, které nesouvisí s požadavky zákazníka. [15]

3.2.2 **Plánování testů**

Plánování testování je druhou fází, kterou testovací tým provádí po analýze a pochopení všech požadavků. Během této fáze se vytváří testovací plán a celková strategie testování, podle kterého se pak celé testování bude řídit.

Činnosti, které jsou v této fázi prováděny, zahrnují identifikaci cílů a rozsahu testování, analýzu souvisejících rizik, definování časového plánu či výběr testovacího prostředí. V neposlední řadě pak také vypracování strategie testování, ve které se vybere, jakou testovací technikou se bude testovat (viz kapitola 3.4) a na jaké úrovni (viz kapitola 3.5.)

Nakonec vedení rozdělí role a přiřadí odpovědnosti. Výsledkem celé této fáze je vytvoření testovacího plánu. To je dokument, který popisuje podrobnosti a motivaci daného testování. Jeho cílem je minimalizaci rizika a podporování důkladnosti a konzistentnosti. [15][16][17]

3.2.3 **Návrh testovacích scénářů**

Při této fázi je potřeba více kreativního myšlení. Na základě testovacího plánu testovací tým navrhuje testy, vytváří testovací scénáře a provádí jejich kontrolu. Testovací tým se tak snaží přijít na všechny možnosti otestování dané funkcionality. Testy by měly být rozsáhlé a obsahovat různé kombinace a permutace.[15], [16], [17]

Správný testovací scénář by měl být jasný, srozumitelný a neumožňovat žádné pochybnosti ohledně obsahu testu. Základem testovacího scénáře je výstižný název. Dále krátké shrnutí, o čem testovací scénář je a podmínky, za kterých je možné scénář otestovat. Hlavní tělo scénáře obsahuje jednotlivé kroky. Ty popisují, co má tester provést, jaká jsou testovací data, která k tomu má použít a jaký výsledek lze očekávat. [18], [19]

Takto vytvořené scénáře se následně rozdělí do testovacích sad a rozdělí se jim priority testování. Propojí se k požadavkům, podle kterých vznikly a doplní se matice sledovatelnosti požadavků. V poslední části pak dochází ke kontrole a autorizaci testovacích scénářů ostatními členy testovacího týmu.[15], [16], [17]

3.2.4 **Nastavení testovacího prostředí**

Nový software nebo nové části softwaru není možné testovat v produkci mezi reálnými uživateli. Je důležité tyto funkčnosti nejdříve zkontrolovat a až pak je do produkčního prostředí nasadit. Z toho důvodu se vytvářejí testovací prostředí, která mají simulovat produkční prostředí. Testovací prostředí by tak mělo být nastaveno stejně, jako prostředí produkční. To zahrnuje jeho nastavení softwaru a hardwaru, ale také síť, velikost uložišť nebo výkon serverů. [15], [16], [17]

Rozlišujeme čtyři základní druhy prostředí, a to:

1. Vývojové prostředí – toto prostředí je pro vývojáře, zde dochází k vývoji softwaru, nových funkcionalit a jejich ladění. [20], [21]
2. Testovací prostředí – hlavní prostředí pro testovací tým. V tomto prostředí probíhá testování, nálezy chyb a nasazení jejich oprav. Testovací prostředí se dá dále rozdělit na UAT (User Acceptance Testing – uživatelské akceptační testování) a SIT (System Acceptance Testing – systém integračního testování). [20], [21], [22]
3. Předprodukční prostředí – téměř přesná replika produkčního prostředí. Zde jsou všechny části z vývojové fáze spojeny dohromady a naposledy se kontrolují před přidáním do produkčního prostředí. [20], [21]
4. Produkční prostředí – zde software běží na produkčním serveru. Software je v tuto chvíli vypublikován koncovým uživatelům. [20], [21]

Důležitou části je také zajištění testovacího týmu potřebným vybavením. Může se stát, že funkcionalita, která funguje v prohlížeči Google Chrom nemusí fungovat v prohlížeči Safari a opačně. Nebo mohou některé funkcionality přestat fungovat na novějších verzích operačního systému. Z tohoto důvodu je důležité, aby testování pokrývalo co nejvíce prostředí. [15], [16], [17]

3.2.5 **Provedení testů**

Po dokončení všech předchozích fází se může přistoupit k testování softwaru. Testovací tým podle testovacího plánu provádí testovací scénáře v testovacím prostředí. Zároveň tato fáze zahrnuje dokumentaci výsledků testování pro analýzu a identifikaci nebo detekci a zaznamenávání vad.

Testující tým je zodpovědný za předání nalezené chyby vývojářům k opravě. V okamžiku, kdy vývojový tým nasadí opravu na testovací prostředí, provádí testovací tým znovu testování softwaru. Na základě času a vážnosti opravené chyby se vybere rozsah testování. Může se totiž stát, že oprava chyby může způsobit další chybu kdekoli v softwaru.

Z tohoto důvodu, kdy je nutné provádět některé testy i několikrát, se doporučuje používání automatizovaných testů (viz kapitola 3.4.2). Lze shrnout, že v této fázi má testovací tým za úkol provádění manuálních a automatizovaných testů a kontrolu pokroku

podle testovacího plánu. Dále pak hlášení chyb v softwaru a následnou kontrolu, že jsou všechny chyby řádně řešeny. [15], [16], [17]

3.2.6 **Uzavření testování**

Uzavření testování znamená ukončení testování a dodání finální verze softwaru zákazníkovi nebo zákazníkům. V této fázi musí být dokončeny všechny činnosti z předchozích fází. Následně pak vedoucí testovacího týmu sestaví závěrečnou zprávu.

Tato závěrečná zpráva podrobně popisuje testovací proces. Jsou v ní uvedeny cíle, rozsah, průběh testování, vykonané činnosti a výsledek testovacího procesu. Jsou zde popsány i problémy, které se během testovací procesu vyskytly, a hodnocení jednotlivých členů týmu.

Ke zhodnocení nedochází pouze formou závěrečné zprávy, ale také v jednotlivých týmech. Týmy zde mají možnost diskutovat o celém testovacím procesu. Nejčastěji jsou však řešeny problémy, se kterými se testeři během testování setkali. Jsou tak například hodnoceny nedostatky ve strategiích nebo testovacích procesech. [15], [16], [17]

3.3 **Role v testovacím týmu**

Testovací tým hraje důležitou roli při testování softwaru. Jejich složení se liší v každém podniku. Jedno by však měly mít všechny testovací týmy společné. Týmy by měly být sestaveny z testerů, jejichž znalosti a zkušenosti se navzájem doplňují a dohromady tvoří celek, který je schopný otestovat celý software. Vedle znalostí a zkušeností, jsou zde i osobnostní vlastnosti, které jsou vhodné, aby tester měl pro efektivnější testování. [23], [24]

Podle Studie [25] existuje 7 vlastností, které jsou pro efektivnost testera důležité. Patří mezi ně organizovanost, zaměření na detail a kreativita. Testeři musí být schopní se zorientovat ve velkém množství testovacích scénářů a rozsáhlém softwaru, ve kterém se snaží najít i ty nejmenší chyby. Dále pak schopnost vidět celý obraz a dobré komunikační schopnosti. Podle studie mohou testeři strávit až 50% času komunikací s ostatními. Studie nakonec zmiňuje zvědavost a adaptibilitu jako poslední důležité vlastnosti.

Před zahájením jakéhokoliv testování je potřeba si v týmu stanovit přesnou hierarchii. Jednotlivé role a úkoly by měly být jasně definovány a správně rozděleny mezi

členy týmu. Každý člen týmu by měl být srozuměn se svojí rolí a úkoly, které zastává. Všechna tato rozdělení by pak měla být sepsána v dokumentu, aby v případě problémů bylo zřejmé, na koho se obrátit.[24]

Celková velikost testovacího týmu je různá, většinou závisí na velikosti firmy a rozsahu testovaného softwaru. Uvádí se však, že ideální velikost je 1 tester na 3 vývojáře. V praxi ale není neobvyklé, že bývá i jeden tester na 10 vývojářů. Mezi hlavní role v testovacím týmu patří: [24], [26], [27]

- **Manažer testování** – osoba, která plně zodpovídá za testovací proces. Vytváří testovací strategii, vybírá priority jednotlivých aktivit, kontroluje jejich plnění a reportuje jejich výsledky zainteresovaným stranám. [1], [28]
- **UX tester** – osoba, která testuje software z pohledu koncového uživatele. Testuje použitelnost softwaru, jako je například použití neoptimálních funkcí, obtěžující přerušení při práci se softwarem nebo nejasné zprávy. [28]
- **Specialista na testování infrastruktury** – osoba, která vytváří a spravuje nástroje potřebné k testování. Je hlavním správcem testovacího prostředí (viz kapitola 3.2.4) a odpovědný za inovace v testovacích nástrojích. [28]
- **Specialista na automatizované testování** – osoba, která vytváří a spravuje automatizované testy. [23], [28]
- **Odborný tester pro konkrétní doménu** – osoba s velkou znalostí v oboru. Převádí obchodní požadavky na konkrétní systémové specifikace a kontroluje jejich plnění. [28]

3.4 Typy testovacích technik

Testování softwaru je srdcem vývoje aplikace, které pravidelně zabírá 40-50% vývojového úsilí. Z důvodu takto rozsáhlých prací je testování softwaru rozděleno na několik typů testování. Základními typy testování jsou manuální testování (statické testování) a automatizované testování (dynamické testování). Ty můžeme dále rozdělit na testování bílé, černé a šedé skříňky. Rozhodnutí o tom, který typ se vybere pro testování, záleží hlavně na úrovni znalosti testera a o vnitřních strukturách a algoritmech testovaného softwarového produktu. [2], [29]

3.4.1 **Manuální testování**

Manuální testování nebo také statické testování je testování za pomocí vlastních sil testera, tedy bez použití jakýchkoliv automatizovaných nástrojů, skriptů apod. Tester se při tomto typu testování chová jako koncový uživatel, prochází software a snaží se odhalit chybu nebo neočekávané chování. [29] Proto je manuální testování občas nazýváno jako průchod, neformální revize, technická revize nebo inspekce. [30], [31]

Hlavní výhodou manuální testování je, že nevyžaduje použití plně funkčního systému nebo softwaru. Testování je možné započít již v raných fázích vývoje, kdy jsou k dispozici pouze dokumenty nebo části softwaru. Díky tomu je možné ho provádět dřív než automatizované testy, které plně funkční software potřebují.

Další velká výhoda manuálního testování je jejich cena. Manuální testy jsou často levnější než testy automatizované. Zároveň bývá návratnost jejich investic vyšší díky dřívější možnosti odhalení chyb, a tedy i dřívější možnosti opravení chyby. [1]

Manuální testování pokrývá dvě hlavní oblasti aktivit: [1]

- formální a neformální revize
- statickou analýzu s použitím nástrojů i bez nich

3.4.2 **Automatizované testování**

Automatizované testování nebo také dynamické testování je typ testování, při kterém tester vytváří a využívá skripty a různé softwary k otestování produktu. [29] Do tohoto procesu patří spravování vytvořených automatizovaných testů, kontrola správného provedení a využívání výsledků testování ke zlepšování kvality softwaru. [2]

Mezi hlavní důvody, proč zvolit automatizované testování patří: [32], [33]

- **Zkrácení doby provedení** – automatizované testy výrazně zkracují dobu testování, a to hlavně v případech, kdy se jedná o opakované a rozsáhlé testy. Testy, které mohou testerům trvat hodiny, zvládne automatizovaný test během pár minut, ne-li sekund. Rychlé otestování softwaru umožňuje rychlejší zpětnou vazbu vývojovým týmům. Rychlá zpětná pomáhá při určování prvotním rozhodování o kvalitě a fungování kódu.
- **Minimalizace lidské chyby** – manuální testování je náchylné na lidské chyby. Přehlédnutí, chybný výklad výsledku testu nebo přeskočení kroku

testu jsou chyby, které mohou při manuálním testování nastat. Automatizací testovacího procesu se možnost výskytu těchto chyb minimalizuje. Při automatizovaném testování je každá část podrobena stejným podmínkám a testovacím sekvencím. To zajišťuje, že je testování konzistentní a umožňuje opakovatelné výsledky.

- **Snížení nákladů** – automatizované testy jsou velkou finanční zátěží při jejich vytváření. Po této fázi ale snižují potřebu lidských zdrojů na provádění jednotlivých testů. V dlouhodobém měřítku tak automatizované testy snižují náklady na testování softwaru.
- **Možnost častého regresního testování** – jednou z metod testování je regresní testování. To zaručuje, že při úpravách softwaru nedošlo k poškození u některé, z již funkčních částí. Při manuálním testování může být regresní testování značně časově náročné a s přibývajícími funkcemi softwaru mohou přibývat i další regresní testy. Automatizací regresních testů dochází ke zkrácení času, potřebného pro provedení testů. Toto zkrácení umožňuje časté provádění regresních testů a zaručuje tak vysokou kvalitu softwaru během všech fází vývoje.

Jednotlivé nástroje automatizovaného testování budou probrány v samostatné kapitole 3.6.

3.4.3 Testování bíle skříňky

White-box testing nebo také testování bílé, strukturální nebo kódové skříňky, je typ testování, při kterém má tester kompletní znalosti o aplikaci. To zahrnuje přístup k zdrojovému kódu a logice aplikace či možnost zkoumání toku dat. Tester tak při vytváření testovacích scénářů využívá vnitřní informace aplikace jako základní zdroj informací. [2], [34]

Strategie testování bílé skříňky vychází z hloubkové analýzy zdrojového kódu. Z tohoto důvodu je tato metoda vhodná pro matematickou analýzu a přesné měření či definice. Testování bílé skříňky je tedy schopno poskytnout přesné údaje o kvalitě testovací sady. Kromě vysoké přesnosti patří mezi výhody testování bílé skříňky například:

- **Brzké odhalení chyb** – testeři se zapojují do procesu vývoje aplikace mnohem dříve. Přístup ke zdrojovému kódu umožňuje testerům započít testování již na počátku vývoje, tedy v době, kdy ještě neexistuje celá aplikace. Existují pouze části kódu, které se v pozdější části vývoje propojí do jedné aplikace.
- **Optimalizace kódu** – během testování může dojít k nalezení části kódu, která by mohla být optimalizována.
- **Bezpečnost** – přístupem k zdrojovému kódu aplikace mohou testeři zkontrolovat správné postupy osvědčených metod pro zabezpečení aplikace.
- **Vyšší míra protestování aplikace** – testeři mají přístup k celému zdrojovému kódu aplikace. Díky tomu není možné přehlédnout menší části aplikace.

Testování bílé skříňky má i své nevýhody. Tato metoda vyžaduje po testerech hlubokou znalost technik implementování softwaru a programovacího jazyka. Často se tak stává, že se testovací a vývojový tým spojí do jednoho. Většinou tím ale dochází k přehlédnutí chyb, které by jinak nezávislé oko testera, který aplikaci nevyvíjel, nepřehlédlo. [2], [34]

3.4.4 Testování černé skříňky

Testování černé skříňky nebo také black-box testing je opačný přístup k testování, než je testování bílé skříňky. V této technice nemá tester přístup k vnitřnímu kódu aplikace a ve většině případech ani znalost o jejím vnitřním fungování. Tento koncept, kdy vnitřní fungování aplikace jsou testerovi skryté nebo uzavřené, dal vzniknout názvu černá skříňka. [2]

Tester v této technice pracuje s dokumentací a specifikacemi aplikace, na základě kterých vytváří testy. Specifikace aplikace popisuje výstupy, které mají nastat pro jakékoliv hodnoty vstupu. Tyto specifikace ale nezmiňují, jak se k tomuto výstupu má aplikace dostat. Testeři se tak primárně zaměřují na vnější chování aplikace. [2], [34]

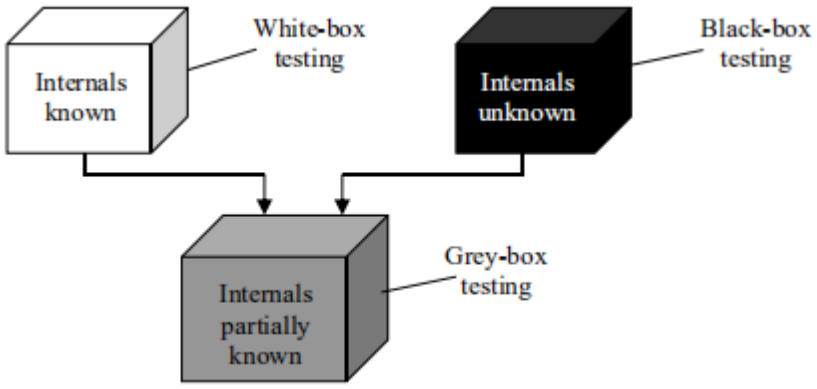
Jak již bylo řečeno, testování černé skříňky je opakem testování bílé skříňky. Díky tomu jsou nevýhody bílé skříňky výhodami černé skříňky a opačně. Jako další výhody pak lze označit například:

- **Zaměření na uživatelskou přívětivost** – testováním již funkční aplikace se může tester více zaměřit na přívětivost aplikace pro uživatele.
- **Efektivní testování integrací** – umožňuje efektivnější testování integrací s ostatními aplikacemi
- **Lepší pro velké projekty** – tato technika je lépe škálovatelná pro velké projekty

Na opačné straně pak stojí hlavní nevýhoda, kdy tato technika oproti bílé skříňce nikdy neumožní otestování veškerých částí kódu. [2], [34]

3.4.5 Testování šedé skříňky

Testování šedé skříňky nebo také grey-box testing je technikou testování, která kombinuje bílou a černou skříňku. Tedy vnitřní chování aplikace je známé jen částečně. Tester využívá tyto znalosti k vytváření testů, které následně provádí z pohledu uživatele. Díky těmto znalostem je tato metoda nejlepší pro testování integrací mezi aplikacemi a ověření správného fungování databáze produktu. [35], [36]



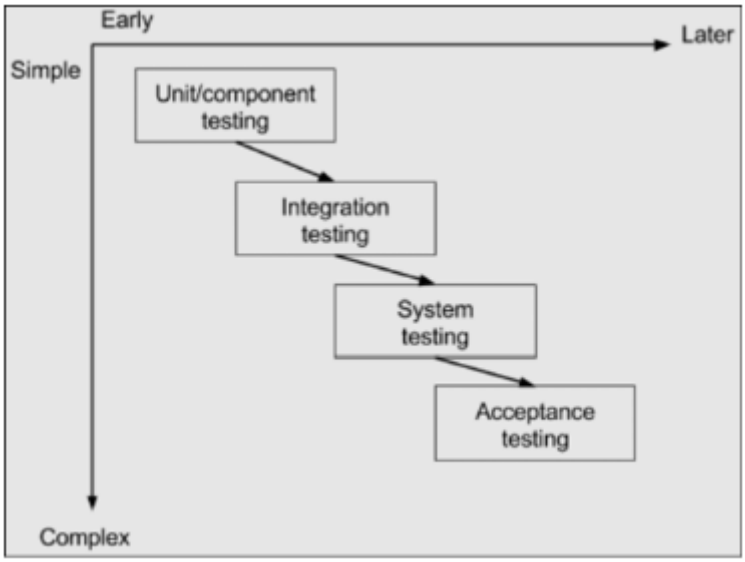
Obrázek 2- Definice Šedé skříňky; Zdroj [36]

3.5 Úrovně testování softwaru

Testování softwaru je velmi náročná činnost. Z tohoto důvodu se zavádí úrovně testování softwaru, které pomáhají tento proces zjednodušit. Každá úroveň testování se používá pro trochu jiné účely a zaměřuje se na odlišné aspekty fungování softwaru. To

umožňuje snadnější odhalení a následné opravení chyby. Každá úroveň testování softwaru vhodná pro jinou část vývoje aplikace.

V dřívějších fázích vývoje se začíná s jednoduššími testy na úrovni testování komponent. Postupně se pak přechází na testování na vyšší úrovni, kdy tyto testy již bývají komplexnější. Sem můžeme zařadit testy integrací a následné testování systému. Celé testování je pak zakončené akceptačními testy, které jsou z hlediska testování nejkomplexnější. [1], [2], [29]



Obrázek 3- Úrovně testování softwaru; Zdroj [37]

3.5.1 **Testování komponent**

Komponentní testy nebo také jednotkové testy jsou nejnižší úrovní testování softwaru. Tato úroveň je specifická testováním komponent samotných. To probíhá tak, že jednotlivé komponenty jsou odpojené od ostatních částí systému a zkoumá se chování jednotlivé komponenty bez vnějších vlivů. Častěji jsou tak tyto testy prováděny na začátku vývoje, kdy ještě nejsou jednotlivé komponenty propojeny. V pozdějších fázích vývoje jsou k odříznutí od systému používány ovladače, simulátory a emulátory. Tyto testy bývají prováděny za pomocí metody bílé skříňky a často tak kvůli nutnosti znalosti kódu bývají testovány samotným vývojářem. Nalezené chyby často bývají rovnou opraveny. [1], [2], [38], [39]

3.5.2 **Testování integrací**

Po testování komponent nastává fáze testování integrací. Hlavní myšlenkou této úrovně je posoudit, jak spolu jednotlivé komponenty komunikují. Ve vývojovém procesu tedy musí nejprve dojít k vývoji a vytvoření integrací mezi komponentami. Až poté dochází k testování integrací, které jsou kontrolovány vůči architektonickému návrhu. Na rozdíl od ostatních úrovní je zde více dbáno na kontrolu efektivity komponent a integrací. Práce se zdroji, výkon nebo robustnost patří mezi hlavní kontrolované vlastnosti. [1], [2], [38]

Existuje mnoho způsobů, jak jednotlivé komponenty integrovat. Je důležité, aby tester těmto integracím rozuměl, protože každá metoda má své výhody i nevýhody. Je také vhodné, aby k integracím komponent docházelo postupně. Při velkém množství integrovaných komponent najednou může být obtížné nalezení příčiny problému. [1], [2], [38]

Základní druhy integrací jsou následující:

- **Big-bang integrace** – je typ integrace, kdy jsou všechny (nebo většina) komponenty propojeny dohromady najednou. Výhodou je, že nejsou potřeba ovladače a zástupné modely pro otestování jednotlivých integrací. Na druhou stranu tato metoda často skončí s velkým počtem chyb, u kterých je následně těžké odhalit jejich původ vzniku. [1], [38]
- **Bottom-up integrace** – neboli integrace zdola nahoru. Tato integrace začíná propojením komponent nižší úrovně a postupně napojuje komponenty úrovně vyšší. Komponenta nejvyšší úrovně je poslední napojenou komponentou. Tato metoda však vyžaduje využití ovladačů pro otestování jednotlivých úrovní. [1], [38]
- **Top-down integrace** – nebo také integrace shora dolů. U této integrace se začíná s komponentou nejvyšší úrovně a na ní jsou postupně napojovány komponenty úrovně nižší. Nevýhodou této metody je potřeba velkého množství zástupných modulů, které v rané fázi integrace nahrazují komponenty nižší úrovně. Zástupné moduly jsou pak v rámci integrace postupně nahrazovány reálnými komponentami. [1], [38]

3.5.3 Testování systému

Po úspěšném integrování všech komponent dohromady přichází na řadu testování systému. V této fázi se snaží testovací prostředí co nejvíce přiblížit reálnému prostředí. Testy jsou zde prováděny za pomocí metody černé skříňky a testeři se zde snaží chovat jako uživatelé systému. Bez znalostí vnitřního fungování aplikace procházejí software tak, jak by ho procházeli samotní uživatelé. Při těchto průchodech kontrolují chování softwaru oproti softwarovým požadavkům. Jako příklad testu, který zde bude tester provádět lze uvést např. přihlášení do aplikace nebo přidání zboží do košíku a následná kontrola, že se zboží v košíku opravdu nachází. [1], [2], [38]

3.5.4 Akceptační testy

Akceptační testy jsou poslední úrovní testování softwaru, kde koncový uživatel kontroluje kvalitu softwaru. V této fázi se software představí zákazníkovi nebo zástupcům uživatelů, kteří následně kontrolují kvalitu softwaru. Software je tak vybraným uživatelům předběžně zpřístupněn k možnosti vlastního testování. Cílem je zde kontrola spokojenosti a získání důvěry od zákazníka či uživatelů. Hledání chyb je zde pouze vedlejší činnost, přestože se zde chyby občas vyskytují. [1], [38]

3.6 Nástroje pro automatizované testování

Označované též jako automatizační frameworky, jsou strukturované kombinace nástrojů, standardů kódování, pokynů a postupů určených k usnadnění automatizovaného testování softwarových aplikací. Zjednodušeně se jedná o aplikaci, která zefektivňuje psaní automatizovaných testů. Tyto nástroje simulují chování uživatelů v testovaném softwaru. Tím ověřují jeho výkon a správnou funkčnost. Jednotlivé výsledky testů jsou zároveň kontrolovány oproti očekávaným výsledkům testů. [33], [40], [41]

Automatizované testování rozlišuje šest základních typů frameworků: [33], [42]

- **Linear Scripting Framework** (Lineární skriptovací rámec) – základní typ frameworku, který funguje na principu „record and playback“ (zaznamenání a přehrávání). Tester provede jednotlivé kroky testu, které jsou zároveň nahrávány do frameworku. Následně spuštěný automatizovaný test bude opakovat kroky vykonané testerem a kontrolovat stejné chování softwaru.

Tento Framework je snadno implementovatelný, ale špatně škálovatelný pro větší projekty. [33], [43]

- **Modular Testing Framework** (modulární testovací rámec) – při tomto typu je testovací skript rozdělen na menší, znovupoužitelné části. Každá část reprezentuje jednu funkcionalitu aplikace. Jednotlivé testy jsou pak skládány z těchto částí. Tento typ vyžaduje velké počáteční úsilí pro vytvoření skriptů. V dlouhodobém měřítku se však tento typ vyplatí. [33], [43]
- **Data-Driven Framework** (datově řízený rámec) – v tomto typu je testovací skript oddělen od testovacích dat. Framework si dotahuje data z externích zdrojů jako je např. excel nebo CSV. Tento typ je ideální pro testování velkého množství vstupních dat (např. přihlašovací údaje). [33], [43]
- **Keyword-Driven Framework** (rámec řízený klíčovými slovy) – k běhu testu jsou využívána klíčová slova, která řídí jednotlivé akce. Jednotlivá předdefinovaná slova mohou být „click“, „enter text“ nebo „verify“. Při vytváření píše tester spíše slovní pokyny než části kódu. Z tohoto důvodu se tento typ využívá pro testery s malou znalostí programovacího jazyka. Popřípadě pro software, ve kterém bývají časté změny. [33], [43]
- **Hybrid Framework** (hybridní rámec) – tento framework kombinuje klíčové vlastnosti několika výše zmíněných frameworků. Tento framework je tak vhodný pro projekty s rozsáhlým testováním a rozmanitými testovacími podmínkami. [33], [43]
- **Behavior-Driven Development** (BDD) Framework (rámec pro vývoj řízený chováním) – v tomto frameworku je hlavní zaměření na spolupráci mezi vývojáři, testery a dalšími zainteresovanými stranami. Toho je dosaženo pomocí syntaxe, která je psána v jednoduchém jazyce. Hlavní snahou je zde tedy propojení členů týmu s technickými znalostmi se členy bez nich. [33], [43]

Typický framework obsahuje několik základních komponent. První klíčovou komponentou je centralizované uložiště, ve kterém jsou uloženy veškeré testovací skripty. Díky centralizovanému uložišti umožňují frameworky správu těchto testovacích dat. Další důležitou komponentou je vykonávací modul, který umožňuje spouštět jednotlivé testovací

skripty. Po doběhnutí testů nabízí frameworky velkou řadu detailních zpráv o běhu testu. Za zmínku stojí také velká řada knihoven, které ulehčují proces vytváření testovacích skriptů nebo možnost integrace na další systémy. [33]

3.6.1 **Selenium**

Jedním z nejpoužívanějších nástrojů pro automatizované testování webových aplikací je Selenium. Jedná se o open-source framework postavený Paulem Hammantem v roce 2006. Je primárně postavený v Javě, podporuje však velkou škálu programovacích jazyků např. PHP, C# nebo Python. Hlavní motivací pro použití Selenia je také kompatibilita s velkým množstvím prohlížečů a operačních systémů. [44], [45], [46]

Architektura selenia byla navržena tak, aby co nejlépe umožňovala automatizaci webových prohlížečů. Obsahuje proto komponentu WebDriver, která umožňuje testování prohlížečů tak, jako kdyby ji používal reálný uživatel. Dále obsahuje komponentu IDE (Integrated Development Enviroment), která funguje na principu „record and playback“ pro usnadnění vytváření testovacích skriptů. Zároveň zde také nalezneme komponentu Grid, která umožňuje běh jednoho scénáře v různých prohlížečích a na různých operačních systémech. [47]

Selenium získalo na popularitě z mnoha důvodů. Je zdarma a díky open-source kódu komunita stále vytváří nové funkcionality. Je robustní, flexibilní a umožňuje běh několika testů najednou. Jak již bylo zmíněno výše, umožňuje využití více programovacích jazyků a použití v různých operačních systémech. [46], [48]

I přes to není Selenium perfektní framework. Navzdory svojí robustnosti může být nastavování Selenium prostředí náročné. Neobsahuje možnost na porovnání obrázků, které už modernější frameworky obsahují nebo ve svém základu neobsahuje komponentu na vytváření zpráv o výsledku testování. [45], [48]

3.6.2 **Cypress**

Dalším z populárních frameworků je Cypress. V posledních letech se zvedá komplexita jednotlivých aplikací a s tím často i potřeba po častějších vydání nových verzí. V návaznosti na tento požadavek vznikl modernější framework Cypress, který reflektuje novější přístup k frameworkům. Cypress je open-source framework, který oproti Seleniu běží přímo v prohlížečí, což zlepšuje celkovou efektivitu testování. [45], [49], [50]

Architektura Cypressu si zakládá na rychlosti a spolehlivosti. Díky běhu v prohlížeči umožňuje okamžitou zpětnou vazbu. Mezi nástroje Cypressu patří automatické čekání, kdy framework sám počká, dokud nebude element na stránce viditelný nebo možnost snadného ladění, kdy je při běhu testu možné tento test zastavit a zkontrolovat si stav aplikace. [49], [50]

Mezi další výhody Cypressu lze zařadit snadné nastavení a instalaci, kdy oproti Seleniu stačí stáhnout pouze jeden balíček, který obsahuje celý framework. Cypress má také rozsáhlou dokumentaci a rostoucí komunitu uživatelů. To umožňuje lepší pomoc pro uživatele Cypressu. Na druhé straně je ale Cypress limitován pouze na programovací jazyky JavaScript a TypeScript. Pokud tedy nový uživatel s těmito jazyky neumí, je zde potřeba přeučení. Zároveň není jeho kompatibilita s různými prohlížeči taková, jako u Selenia. [42], [45], [50]

3.6.3 **Playwright**

Playwright je také open-source framework od společnosti Microsoft, který stejně jako Cypress vznikl jako modernější alternativa pro Selenium. Podporuje skriptování ve více programovacích jazycích a umožňuje testování ve více prohlížečích za pomocí jediného API (Application Programming Interface), které umožňuje propojení mezi jednotlivými částmi systému. [51], [52]

Playwright je průkopníkem bezhlavého testování webových aplikací. Tato inovace umožňuje spouštění testů bez nutnosti grafického rozhraní prohlížeče. Dále zde nalezneme nástroje pro automatické čekání, record and playback nebo důvěryhodné události. Důvěryhodné události jsou situace, kdy framework simuluje chování uživatele. Toto napodobení je natolik přesvědčivé, že je prohlížeč následně interpretuje jako autentické akce uživatele. [51], [52]

Hlavní výhodou Playwrightu je díky modernímu přístupu jeho rychlost. Dalšími výhodami jsou integrace na velké množství různorodých knihoven, stabilita a velká škála funkcí. Za zmínku určitě stojí i možnost testování prohlížeče Safari (nativní prohlížeč pro operační systémy iOS) v systémech windows za použití nástroje WebKit. Výraznějších nevýhod Plawright moc nemá. Pokud bychom však měli nějaké zmínit, jednalo by se o nepodporování nativních mobilních aplikací a vyžadování pokročilých konfiguračních znalostí pro integraci nestandartních knihoven. [42], [51]

3.7 Vícekriteriální hodnocení variant

Vícekriteriální hodnocení patří mezi disciplíny operačního výzkumu a zabývá se situacemi, ve kterých jsou posuzovány varianty podle často konfliktních kritérií. Jednotlivá vícekriteriální hodnocení jsou popsána množinou variant a množinou hodnotících kritérií. Vícekriteriální hodnocení variant je typ vícekriteriálního hodnocení, které má konečnou množinu přípustných variant. Cílem může být výběr jediné varianty, určení pořadí variant nebo třídění variant. [53], [54]

Jednotlivá kritéria mohou být ohodnocena za použití různých stupnic a škál. Příkladem je binární nominální stupnice, která nabývá pouze hodnot 0 pro neshodu nebo 1 pro shodu. Dále ordinální stupnice, která uspořádává hodnoty od nejvíce důležitého po nejméně důležité. A například Likertova stupnice, která se používá pro případy, kdy nelze kritéria kvantifikovat. Důležité je, aby při konečném hodnocení byla všechna kritéria transformována na srovnatelné jednotky. [53]

Aby mohla být jednoznačně vybrána kompromisní varianta, je důležité ohodnotit jednotlivá kritéria preferenční hodnotou, takzvanou váhou. Pro rozhodovatele však bývá často obtížné přesně své preference rovnou kvantifikovat. Na řadu tak přicházejí metody pro určení vah jednotlivých kritérií, například:

- **Metoda pořadí** – rozhodovatel uspořádá kritéria sestupně podle jejich důležitosti. Kritéria následně ohodnotí číselně od nejdůležitější tak, že každému dalšímu kritériu je přiřazena hodnota o 1 nižší. Nejméně důležité hodnotě je přiřazena váha 1.
- **Bodovací metoda** – rozhodovatel si zvolí bodovací stupnici důležitosti (např. 1-10) a následně ohodnotí všechna kritéria podle důležitosti, přičemž nejvyšší hodnota odpovídá nejvyšší důležitosti kritéria.

Výsledné odhady vah jsou získány znormováním čísel, která byla jednotlivým kritériím přiřazena. [54]

Následně lze varianty a kritéria uspořádat do rozhodovací matice. Po určení vah a ohodnocení jednotlivých kritérií se použije zvolená metoda vícekriteriálního ohodnocení (např. metoda váženého součtu), která umožní určit umožní určit celkové hodnocení a pořadí variant. [54]

4 Vlastní práce

Praktická část porovnává manuální testování s automatizovanými testy realizovanými ve frameworcích Selenium, Cypress a Playwright na vybrané webové aplikaci. Toho je dosaženo zkoumáním časové náročnosti testů, nákladů na implementaci, nákladů na běh testu a stability testů. Výsledkem je zhodnocení, kdy dává která metoda smysl v praxi, a formulace doporučení pro jejich použití.

Jako webová aplikace pro testování v praktické části byl vybrán Moodle LMS. Pro něj byly identifikovány tři vhodné testovací scénáře s postupně zvětšující se obtížností provedení. Tyto scénáře byly následně implementovány do všech vybraných frameworků. Všechny scénáře byly realizovány N-krát pro každou metodu a jednotlivé výsledky testů byly zaznamenány. Získaná data byla dále analyzována a zpracována v tabulkách a grafech.

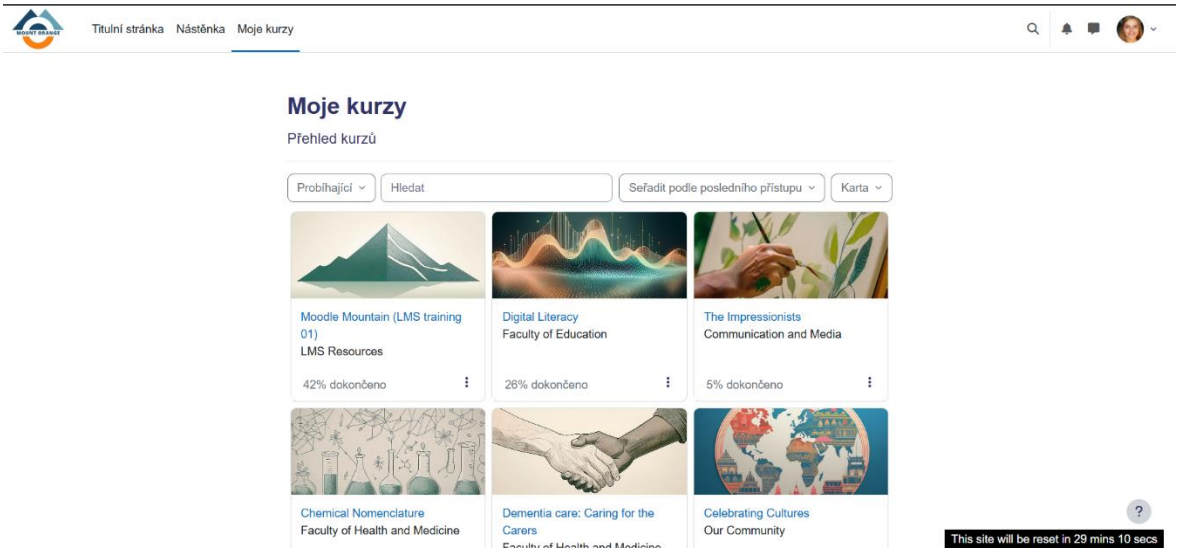
Na základě získaných výsledků je provedena vícekriteriální analýza metodou váženého součtu. Z již získaných výsledků a z poznatků teoretické části byla vybrána kritéria hodnocení, kterým byly přiřazeny váhy. Výstupem této vícekriteriální analýzy je pořadí metod pro vybrané testovací scénáře a doporučení, ve kterých situacích preferovat manuální testování či konkrétní automatizační framework.

4.1 Výběr webové aplikace pro testování

Pro tuto bakalářskou práci byla vybrána open-source webová aplikace Moodle LMS (dostupná na school.moodledemo.net). Moodle se zaměřuje na eLearning a v poslední době i na využití AI při výuce. [55] Důvody pro vybrání aplikace jsou osobní zkušenost s používáním aplikace a přítomnost veřejné demo verze pro testování. Ta obsahuje předdefinované testovací uživatele, na kterých je možné testovat jednotlivé funkcionality. Testovací uživatelé jsou nastavení tak, aby reflektovali situace využití reálnými uživateli. Najdeme zde tak uživatelské profily představující studenta, učitele, rodiče nebo třeba manažera. Zároveň je aplikace každou hodinu resetována do úvodního stavu. Zbývající čas do resetování stránky vidí tester po celou dobu užívání aplikace v pravém dolním rohu. Pokud by se tak testerovi nebo automatizovanému testu podařilo aplikaci rozbít nebo změnit její nastavení, bude tak po resetování aplikaci opět možné testovat. Tento reset byl během testování brán v potaz a testy byly realizovány tak, aby neprobíhaly během resetu aplikace.

Testovaná verze Moodle LMS je 5.1. Jako prohlížeč pro provedení testů byl vybrán Google Chrome. Tento prohlížeč je světově nejpoužívanější a je přívětivý pro realizaci testů. K testům byla využita jeho verze 142.0.7444.60. V aplikaci byl zvolen jazyk UI: čeština.

K testům byly využity profily „student“ a „teacher“. Uživatelský profil „student“ simuluje běžného účastníka kurzů. Profil má předem nastavené kurzy, které může procházet a zobrazovat si jejich obsah (viz obrázek 4). Nemá umožněno jednotlivé kurzy měnit, nahrávat na ně materiál nebo upravovat seznam účastníků. Uživatelský profil „teacher“ simuluje učitelský účet. Tento profil je vlastníkem jednotlivých kurzů, má tak možnost jejich zobrazení, editace, nahrávání materiálů a editování účastníků kurzu. Situace, ve kterých jsou tyto uživatelské profily využity, jsou popsány u jednotlivých testovacích scénářů.



Obrázek 4 - Ukázka aplikace Moodle z pohledu profilu „student“ (Moje kurzy); Zdroj: vlastní

4.2 Identifikace vhodných scénářů pro testování

Tato kapitola se zabývá identifikací vhodných scénářů pro testování. V kapitole jsou popsána kritéria pro výběr vhodných scénářů, následně je stanovena jednotná šablona pro testovací scénáře. Podle této šablony jsou nakonec vytvořeny a uvedeny konkrétní testovací scénáře.

Testovací scénáře byly vybrány tak, aby reflektovaly nejčastěji prováděné akce uživatelů. Jedná se o základní akce, které provádí většina uživatelů při návštěvě aplikace

(klíčové oblasti UI). Náročnost jednotlivých scénářů se postupně mírně zvyšuje, aby bylo možné otestovat různé úrovně obtížnosti. Tím, že se jedná o elementární akce, které se často opakují, pravděpodobnost výskytu chyby je nízká. To testování umožní více se zaměřit na stránku efektivity testů a na náklady spojené s implementací a údržbou testů. Zároveň se ale jedná o dostatečně reprezentativní scénáře pro možnost odhalení případné chyby v aplikaci.

Kritéria výběru scénářů:

- uživatelská relevance (často prováděné akce),
- pokrytí klíčových oblastí UI,
- vzestupná obtížnost testů.

4.2.1 Šablona pro testovací scénáře

Pro vytvoření šablony byla využita norma ISO 29119-3. Tato norma umožňuje tailoring (přizpůsobení). Vytvořená šablona proto byla přizpůsobena pro účely této bakalářské práce. Navrhnutá šablona je zobrazena v tabulce 1.

ID testu					
Název					
Popis					
Předpoklady pro provedení testu					
Krok	**Akce**	**Vstupní data**	**Očekávaný výsledek**	**Stav**	**Skutečný výsledek**
1.					
2.					
3.					
			Celkový výsledek		

Tabulka 1 - Šablona pro testovací scénáře; Zdroj: vlastní

Vysvětlení použití jednotlivých polí:

- **ID testu** – unikátní označení testu, podle kterého je možné testovací scénář vyhledat (např. test-1).

- **Název** – stručný název testovacího scénáře. Název by měl být srozumitelný a dostatečně vypovídající o obsahu testu. Již po jeho přečtení by měl tester vědět, co bude předmětem testu.
- **Popis** – zde je shrnut cíl testu. Popis obsahuje určení, co test ověřuje a případně je zmíněný postup, jak k tomu dojít.
- **Předpoklady pro provedení testu** – popisuje podmínky, které musí splněny pro provedení tohoto testovacího scénáře (např. uživatel s rolí student)
- **Krok** – číselné označení jednotlivých kroků.
- **Akce** – popis toho, co má tester provést. V těchto polích jsou popsány kroky, které má tester provést k provedení testu. Jedno pole = jeden elementární krok, který má tester v aplikaci provést.
- **Vstupní data** – konkrétní hodnoty pro daný krok. Pole je vyplněno v návaznosti na pole akce ve stejném kroku. Pokud se provádí akce, pro kterou nejsou potřeba vstupní data, zůstává toto pole prázdné.
- **Očekávaný výsledek** – popisuje, co má tester od aplikace očekávat po provedení akce. Očekávané výsledky vycházejí ze specifikací a požadavků projektu.
- **Stav** – Jak testovací krok dopadl. „Passed“ – prošel, „Failed“ – neprošel, „Not run“ – netestován.
- **Skutečný výsledek** – vyplňuje se po provedení testu. Tester zde může doplnit poznámky k běhu testu.
- **Celkový výsledek** – výsledek celého testu. Určuje, jaký je finální výsledek testu („Passed“, „Failed“). Pokud byl stav u jakéhokoliv kroku „Failed“, výsledkem celého testu je pak tento stav.

V následujících podkapitolách (4.2.2-4.2.4) jsou uvedené testovací scénáře, které byly vytvořeny podle této šablony pro webovou aplikaci moodle.

4.2.2 Test 1 – Přihlášení do aplikace

Jako první testovací scénář byl zvolen proces přihlášení uživatele do aplikace. Jedná se o základní akci, kterou musí provést každý uživatel při používání aplikace. Pokud by

přihlášení nefungovalo, nebylo by možné otestovat zbylé testy. Z tohoto důvodu byl tento proces vybrán jako základní testovací scénář této práce.

Pro tento testovací scénář byl vybrán testovací profil „student“. Cílem tohoto testu je kontrola správného přihlášení. Je tak možné na tento test použít jakýkoli předdefinovaný profil. Zde byl vybrán profil „student“ z důvodu reprezentace největšího množství reálných uživatelů používajících tuto aplikaci.

Tester začíná tento testovací scénář otevřením přihlašovací stránky aplikace Moodle. Zde provádí tester první kontrolu, že se stránka správně načetla. Do polí Uživatelské jméno a Heslo vepíše přihlašovací údaje profilu „student“ a klikne na tlačítko „Přihlášení“. Následně tester kontroluje, že došlo k přihlášení na správného uživatele, a že se nachází na stránce Moje kurzy. Tímto postupem ověří tester, že přihlášení na daného uživatele funguje správně. V tabulce 2 je tento testovací scénář napsán podle navrhnuté šablony z kapitoly 4.2.1.

ID testu	Test-1				
Název	Přihlášení do aplikace				
Popis	Ověřit, že se uživatel s platnými přihlašovacími údaji úspěšně přihlásí				
Předpoklady pro provedení testu	Dostupný uživatel s platnými přihlašovacími údaji, nastavené české rozhraní aplikace				
Krok	**Akce**	**Vstupní data**	**Očekávaný výsledek**	**Stav**	**Skutečný výsledek**
1.	Otevřete stránku https://school.moodledemo.net/login/		Zobrazí se stránka „Přihlášení do Mount Orange“		
2.	Do pole „Uživatelské jméno“ zadejte platné uživatelské jméno studenta	student	Pole vyplněnou danou hodnotou		
3.	Do pole „Heslo“ zadejte platné heslo studenta	moodle25	Pole vyplněnou danou hodnotou		
4.	Klikněte na tlačítko „Přihlášení“		Uživatel byl úspěšně přihlášen. Zobrazena stránka „Moje kurzy“		
			Celkový výsledek		

Tabulka 2 – Testovací scénář přihlášení do aplikace; Zdroj: vlastní

4.2.3 **Test 2 – Zobrazení kurzu**

Pro druhý testovací scénář byl zvolen proces zobrazení zapsaného kurzu. K hlavním důvodům, proč uživatelé používají aplikaci Moodle, patří vzdělávání. To je v této aplikaci zprostředkováno pomocí kurzů. Do těchto kurzů nahrávají učitelé materiály, které následně slouží studentům k vzdělávacím účelům. Cílem tohoto testu tak je kontrola správného zobrazení kurzů, do kterých je uživatel zapsán.

Pro účely tohoto testu byl vybrán profil „student“. Tento profil má zapsané kurzy a je proto vhodný pro použití v tomto testu. Pro tento testovací scénář by bylo možné použít i profil „teacher“, avšak tím by pak bylo testováno zobrazení kurzu z pohledu vlastníka. Tento testovací scénář je zaměřen na zobrazení z pohledu studenta zapsaného do kurzu.

Tento testovací scénář začíná stejně jako první testovací scénář. Zde už ale není tolik dbáno na detaily přihlášení. Tester tedy otevře webovou aplikaci a přihlásí se na uživatele „student“. Na stránce „Moje kurzy“ zkontroluje, že má student zapsané kurzy a že jsou viditelné. Následně vybere jeden kurz a klikne na něj. Tester zkontroluje, že se zobrazí stránka s detailem kurzu. Zde zkontroluje, že je stránka zobrazená správně. V tabulce 3 je tento testovací scénář napsán podle navrhnuté šablony z kapitoly 4.2.1.

ID testu	Test-2				
Název	Zobrazení kurzu				
Popis	Ověřit, že si uživatel může zobrazit detail zapsaného kurzu				
Předpoklady pro provedení testu	Dostupný uživatel s platnými přihlašovacími údaji, který je zapsán v aspoň jednom kurzu, nastavené české rozhraní aplikace				
Krok	**Akce**	**Vstupní data**	**Očekávaný výsledek**	**Stav**	**Skutečný výsledek**
1.	Otevřete stránku https://school.moodlemo.net/login/ a přihlaste se	student moodle25	Uživatel úspěšně přihlášen, zobrazena stránka „Moje kurzy“		
2.	Na stránce „Moje kurzy“ zkontrolujte, že student vidí zapsané kurzy		Uživatel vidí zapsané kurzy		
3.	Vyberte jeden kurz a kliknutím na něj přejděte na jeho detail		Uživatel byl přesměrován na stránku s detailem kurzu		
4.	Zkontrolujte, že je stránka správně zobrazena		Stránka se zobrazuje správně		
			Celkový výsledek		

Tabulka 3- Testovací scénář Zobrazení kurzu; Zdroj: vlastní

4.2.4 **Test 3 – Zapsaní studenti v kurzu**

Ve třetím testovacím scénáři byl zvolen proces kontroly zapsaných studentů v kurzu. Učitel má možnost vidět, kdo je zapsán do jeho kurzu. V kurzu kromě studia materiálů probíhá také například testování studentů nebo zadávání domácích úkolů. Učitele následně zajímá plnění těchto povinností studentů. To ale není možné zkontrolovat, pokud učitel nevidí zapsané studenty do kurzu. Cílem tohoto testu je tak kontrola zobrazení zapsaných studentů do kurzu.

V tomto testovacím scénáři byl použit profil „teacher“. Ostatní profily nemohou být pro tento testovací scénář použity. Nemají totiž nastavené kurzy, u kterých by byly sami vedeni jako vlastníci. Z vybraných scénářů je tento scénář nejnáročnější na provedení. Je zde potřeba vykonat nejvíce kroků k ověření cíle testovacího scénáře.

Testovací scénář začíná otevřením přihlašovací stránky aplikace a následným úspěšným přihlášením na profil „teacher“. Tester dále na stránce „Moje kurzy“ vybere jeden kurz a klikne na něj. Po přesměrování na detail kurzu tester vybere záložku „Účastníci“ a klikne na ni. Zobrazí se stránka zapsaných uživatelů do kurzu. Tester zde provede kontrolu, že existují studenti zapsaní do kurzu. V tabulce 4 je tento testovací scénář napsán podle navrhnuté šablony z kapitoly 4.2.1.

ID testu	Test-3				
Název	Zapsaní studenti v kurzu				
Popis	Ověřit, že vlastník kurzu vidí studenty zapsané do kurzu				
Předpoklady pro provedení testu	Dostupný profil s platnými přihlašovacími údaji, který je vlastníkem kurzu, zapsaní studenti v kurzu, nastavené české rozhraní aplikace				
Krok	**Akce**	**Vstupní data**	**Očekávaný výsledek**	**Stav**	**Skutečný výsledek**
1.	Otevřete stránku https://school.moodlademo.net/login/ a přihlaste se	teacher moodle25	Uživatel úspěšně přihlášen, zobrazena stránka „Moje kurzy“		
2.	Na stránce „Moje kurzy“ vyberte jeden kurz a klikněte na něj		Uživatel byl přesměrován na stránku s detailem kurzu		
3.	V horní záložce klikněte na „Účastníci“		Zobrazena záložka „Zapsaní uživatelé“		
4.	Zkontrolujte, že jsou do kurzu zapsaní studenti		Je zobrazen alespoň jeden zapsaný student		
			Celkový výsledek		

Tabulka 4 - Testovací scénář: Zapsaní studenti v kurzu; Zdroj: vlastní

4.3 Implementace scénářů na automatizované scénáře

Kapitola se zabývá implementací tří vybraných scénářů v rámci frameworků Selenium, Cypress a Playwright. Nejprve jsou popsány obecné principy implementace a zásady, díky kterým je možné testy mezi sebou porovnat. Následně se kapitola detailně zabývá implementováním vybraných scénářů do jednotlivých frameworků.

Za účelem převedení manuálních testovacích scénářů na automatizované byly vybrány frameworky Selenium, Cypress a Playwright. Tyto frameworky patří mezi nejpoužívanější a zároveň fungují na podobných principech. Z těchto důvodů byly zvoleny právě tyto tři pro praktickou část.

Během implementace bylo dbáno na to, aby měly všechny frameworky stejné podmínky pro běh testů. Pro vývoj a spouštění testovacích scénářů tak bylo vybráno jednotné vývojové prostředí IntelliJ IDEA Community Edition. Využití tohoto prostředí usnadnilo konfiguraci projektu a běh testů. Současně jsou všechny testy prováděné v prohlížeči Google Chrome.

Jednotlivé testy byly rozděleny do samostatných tříd. Zkráceně jedna třída odpovídá jednomu testovacímu scénáři. Tento způsob byl vybrán pro větší přehlednost v testech a lepší měřitelnost časového provedení. Aby bylo dosaženo co nejlepšího porovnání, byla struktura všech automatizovaných testů navržena tak, aby se mezi sebou co nejvíce podobala. Frameworky tak mají stejnou posloupnost kroků. Jsou kontrolovány stejné atributy a je odkazováno na ty totožné elementy na stránce.

Jednotlivé testy byly v IntelliJ spouštěny ručně. Ověření správnosti výsledku testu je realizováno za pomocí asertací nástroje TestNG. Pokud by se stalo, že se jakýkoliv krok nepodaří vykonat, bude celý test označen jako „Failed“ a v reportu popsána nalezená chyba. Naopak, pokud se všechny kroky testovacího scénáře podaří dokončit, test je označen jako „Passed“.

4.3.1 Selenium

Selenium není samo o sobě plnohodnotný testovací framework. Musel tak být doplněn dalšími nástroji. Jako programovací jazyk byla zvolena Java, proto bylo potřeba použít nástroje, které s ní pracují. Z tohoto důvodu byly pro doplnění Selenia vybrány nástroje TestNG pro psaní a strukturování testů a Maven pro build (překládání zdrojového

kódu) a správu závislostí v Javě. Důležitou součástí projektu je soubor *pom.xml*, který zastává v projektu roli konfiguračního souboru pro Maven.

Pro spuštění prohlížeče Google Chrome byla použita knihovna Selenium WebDriver. Ta umožňuje programově ovládat webový prohlížeč tak, jako by ho používal reálný uživatel.[47] V kódu 1 je zobrazeno, jak byla tato knihovna naimplementována. Při některých běžích vyskakovalo varovné okno Chromu ohledně úniku hesla, které blokovalo některé testy. Proto byl pro účely testu využit inkognito režim prohlížeče.

```
ChromeOptions options = new ChromeOptions();
options.addArguments(new String[]{"--incognito"});
WebDriver driver = new ChromeDriver(options);
```

Kód 1- Selenium: inicializace Chromu; Zdroj: vlastní

Test-1 – Přihlášení do aplikace

Test-1 je realizován v třídě *Login*, která je umístěná v balíčku *Seleniumtests.* Cílem testu je ověřit, že se uživatel úspěšně přihlásí do aplikace. Selenium spustí prohlížeč, v něm otevře stránky Moodle a ověří, že se nachází na správné stránce. Posléze, jak je možné vidět v Kódu 2, nalezne Selenium na stránce za pomocí ID elementů formulář pro zadání přihlašovacích údajů, vyplní jej a odešle. Při implementaci se objevil problém s validací formuláře, kdy při moc rychlém vyplnění obou polí docházelo k odstranění hodnot z pole pro heslo.

Pro stabilizaci byla přidána krátká časová prodleva pomocí metody *Thread.sleep();* (viz Kód 2), která vloží krátký časový interval mezi vyplnění přihlašovacího jména a hesla. Metoda byla implementována s vědomím, že se jedná o funkční, i když méně vhodné řešení.

```
WebElement usernameInput = driver.findElement(By.id("username"));
usernameInput.sendKeys("student");
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
WebElement passwordInput = driver.findElement(By.id("password"));
passwordInput.sendKeys("moodle25");
WebElement submitButton = driver.findElement(By.id("loginbtn"));
submitButton.click();
```

Kód 2 - Selenium: Vyplnění přihlašovacího formuláře; Zdroj: vlastní

Po úspěšném přihlášení dochází k přesměrování na stránku „Moje kurzy“. Test je zakončen tím, že Selenium počká na přesměrování a následně prověří URL nově zobrazené stránky a zkontroluje, že se na stránce zobrazuje text „Moje kurzy“. Posledním krokem testu je zavření prohlížeče za pomocí příkazu *driver.quit();*.

Test-2 – Zobrazení kurzu

Test-2 je realizován ve třídě *CourseMaterialVisibilityCheckStudent* v balíčku *Seleniumtests*. Cílem testu je ověřit, že student vidí kurz a může si ho zobrazit. Test začíná přihlášením na profil studenta a zobrazením stránky „Moje kurzy“, přičemž pro tuto část je znovu využit kód z Testu-1 (přihlášení do aplikace). Následně test ověřuje, že student vidí alespoň jeden kurz a přechází na jeho detail (viz Kód 3). K nalezení prvního kurzu byl použit selektor XPath. Protože není znám kompletní seznam kurzů, test ověřuje pouze to, že se načte alespoň jeden kurz a jeho detail.

```
WebElement course =
wait.until(ExpectedConditions.elementToBeClickable(By.xpath("(//div[conta
ins(@id,'course-info-container')]/div/div/a/span[3]/span[2])[1]")));
course.click();
```

Kód 3- Selenium: přechod na detail kurzu; Zdroj: vlastní

Následně Selenium ověří, že došlo k přesměrování na správnou stránku pomocí kontroly URL a názvu stránky. K ověření je použit předpoklad, že pokud se zobrazí název stránky, tak se stránka načetla správně. Celý test je opět zakončen pomocí příkazu *driver.quit();*, kterým je prohlížeč zavřen. Těmito kroky došlo k ověření, že si student může zobrazit detail kurzu a že se kurz správně načte.

Test-3 – Zapsaní studenti v kurzu

Test-3 je realizován ve třídě *ClassStudents* v balíčku *Seleniumtests*. Cílem testu je ověřit, že učitel vidí studenty přihlášené do kurzu. Test tak začíná přihlášením na profil „teacher“ a následuje přechodem na detail kurzu. Kroky pro přihlášení a zobrazení detailu kurzu jsou s lehkou úpravou kontroly převzaty z Testu 2 (zobrazení kurzu).

Nový kód navazuje tam, kde Test-2 (zobrazení kurzu) končí. Po zobrazení detailu stránky Selenium najde záložku „Účastníci“ a klikne na ni. Jak je uvedeno v kódu 4, Selenium počká na načtení stránky a následně zkontroluje seznam zapsaných účastníků. Pokud se nepodaří list načíst nebo neexistuje nultý záznam studenta, skončí test chybou.

Tím, že není volný přístup k databázi aplikace, bylo za splnění podmínek považováno zobrazení alespoň jednoho studenta.

```
wait.until(ExpectedConditions.urlContains("/user/index.php"));
List<WebElement> participants =
driver.findElements(By.cssSelector("[id*='user-index-participants-
'][id$='_r0']"));
Assert.assertTrue(!participants.isEmpty(), "V kurzu není zapsaný ani
jeden student");
```

Kód 4- Selenium: kontrola zapsaných studentů; Zdroj: vlastní

4.3.2 Cypress

Kapitola se zabývá implementací stejných tří testovacích scénářů, které byly již implementovány a popsány pro Selenium v kapitole 4.3.1. Cílem kapitoly není znovu detailně popsat jednotlivé scénáře, ale spíše poukázat na rozdílnou implementaci frameworku Cypress.

Obecná charakteristika frameworku byla uvedena v kapitole 3.6.2. V této kapitole je popsáno, jak byl framework konkrétně použit. Oproti Seleniu je Cypress plnohodnotným testovacím frameworkem. K spouštění jednotlivých testů a sledování jejich průběhu byl použit grafický nástroj Cypress Test Runner, který je součástí Cypressu. Dále Cypress obsahuje vestavěnou asserční knihovnu a nástroje pro ladění testů. To umožňuje zpětně analyzovat stav stránky během jednotlivých kroků testu. Na rozdíl od Selenia tak nepotřebuje být doplňován dalšími nástroji. Cypress nenabízí možnost výběru programovacího jazyka, pro testy tak musel být použit jeho nativní jazyk JavaScript.

Obdobnou roli jako má soubor *pom.xml* pro Selenium, plní i soubor *package.json* pro Cypress. Daný soubor funguje jako konfigurační soubor projektu, ve kterém jsou definovány zejména použité závislosti. Jednotlivé scénáře jsou kvůli lepší přehlednosti uspořádány stejně jako v Seleniu, kdy jeden soubor obsahuje jeden test (v Cypressu se místo tříd používají soubory). Všechny prováděné testy pak lze nalézt ve složce *Cypress-test/Cypress/e2e*.

Oproti Seleniu není v Cypressu potřeba ručně inicializovat webový prohlížeč. Testy se v Cypressu na rozdíl od ostatních použitých frameworků spouští přes příkazovou řádku. Pro zachování konzistentnosti mezi testy tak byl vybrán prohlížeč Google Chrome.

Během testů v Cypressu se neobjevil problém s vyskakovacím oknem Chromu upozorňujícím na únik hesla. Cypress spouští testy v odděleném prostředí, proto nemusel být použit anonymní (inkognito) režim prohlížeče.

Test-1 – Přihlášení do aplikace

Test-1 je realizován v souboru *login.cy.js* ve složce *e2e.* Cílem testu je ověřit, že se uživatel může přihlásit. V kódu 6 je ukázán první krok testu, kterým je přechod na stránku Moodle LMS. Zde je lehký rozdíl v použité URL oproti Seleniu. Tím, že jsou testy spouštěny v odděleném prostředí, automaticky se načte anglická verze stránky. Musela, proto být do URL přidána specifikace jazyka stránky. Cypress obsahuje zabudované automatické čekání. Není zde potřeba psát podmínky pro načtení stránky, Cypress sám počká, až se stránka načte. Tato funkcionalita je jedním z hlavních rozdílů oproti Seleniu. Po načtení stránky proběhne kontrola pomocí nadpisu stránky. Pro účely tohoto testu stačí předpoklad, že pokud se načetl nadpis stránky, načetla se celá stránka.

```
cy.visit('https://school.moodledemo.net/login/index.php?lang=cs')
cy.contains('Přihlášení do Mount Orange').should('be.visible')
```

Kód 5 – Cypress: zobrazení stránky Moodle LMS; Zdroj: vlastní

Následně, jak je možné vidět v kódu 7, Cypress na stránce podle ID prvku nalezne pole pro zadání přihlašovacího jména a hesla, vyplní je přihlašovacími údaji profilu „student“ a přihlásí se. Díky zabudovanému automatickému čekání zde nedochází k problému s mazáním hodnoty z pole hesla, který se vyskytoval u Selenia (viz kapitola 4.3.1). Tato část kódu se tak obejde bez nuceného uspání.

```
cy.get('#username').type('student')
cy.get('#password').type('moodle25')
cy.get('#loginbtn').click()
```

Kód 6 - Cypress: vyplnění přihlašovacích údajů; Zdroj: vlastní

Po přihlášení Cypress automaticky počká na načtení stránky. Na načtené stránce pak, stejně jako u testu v Seleniu, dojde k ověření URL. Posledním krokem je pak kontrola nadpisu stránky. Pokud nadpis odpovídá předpokládanému řetězci, celý test je označen jako „Passed“. Rozdíl oproti Seleniu je zde v tom, že není potřeba psát kód pro zavření prohlížeče. Cypress ho zavře sám.

Test-2 – Zobrazení kurzu

Test-2 se nachází v souboru *CourseMaterialVisibilityCheckStudent.cy.js* ve složce *e2e*. Cílem testu je ověřit, že student vidí kurz a může si ho zobrazit. Začátek testu probíhá stejně jako Testu-1 (přihlášení do aplikace). Otevře se stránka Moodle, vyplní se přihlašovací údaje a dojde k přihlášení. Z tohoto důvodu byl kód pro přihlášení převzat z Testu-1. Test pokračuje vybráním prvního kurzu na stránce. Důvod, proč je vybírán první kurz, je vysvětlen v kapitole 4.3.1. Cypress na stránce vybere první kurz a na ten klikne. Po načtení stránky zkontroluje URL a správnost nadpisu stránky, což je demonstrováno v Kódu 7. Opět se zde vychází z předpokladu z kapitoly 4.3.1, že pokud se načetl nadpis stránky, načetla se i stránka.

```
cy.contains('.visually-hidden', 'Course name').click({ force: true })
cy.url().should('include', '/course/view')
cy.get('.page-header-headings').should('be.visible')
```

Kód 7 - Cypress: zobrazení detailu kurzu; Zdroj: vlastní

Test-3 – Zapsaní studenti v kurzu

Test-3 je realizován v souboru *ClassStudents.cy.js*, který je ve složce *e2e*. Cílem testu je ověřit, že učitel vidí studenty přihlášené do kurzu. Na začátku testu se uživatel přihlašuje na profil „teacher“ a přejde na detail kurzu. S úpravou přihlašovacích údajů je tato část kódu převzata z Testu-2 (zobrazení kurzu). Následně Cypress přepne na záložku „Účastníci“ a zkontroluje, že existuje alespoň jeden záznam se studentem (viz kód 8). Jak již bylo popsáno v kapitole 4.3.1, předpokládáme, že pokud se načte aspoň jeden student, došlo ke správnému načtení stránky.

```
cy.get('[data-key="participants"]').click()
cy.get('[id*="user-index-participants-"][id$="_r0"]')
```

Kód 8 - Cypress: kontrola zapsaných studentů; Zdroj: vlastní

Naprogramování všech tří testů v Cypressu bylo značně jednodušší než v Seleniu. Zabudované automatické čekání a lepší práce s elementy na stránce testování v Cypressu značně ulehčují programování. Pro napsání testů v Seleniu bylo potřeba zhruba třikrát více řádků kódu než v Cypressu.

4.3.3 **Playwright**

Kapitola se zabývá implementací tří testovacích scénářů, které byly již implementovány a popsány pro Selenium a Cypress v kapitolách 4.3.1 a 4.3.2. Cílem kapitoly není znovu detailně popsat jednotlivé scénáře, ale spíše poukázat na rozdílnou implementaci frameworku Playwright.

Playwright stejně jako Selenium není sám o sobě plnohodnotný testovací framework. Neobsahuje všechny části, které jsou k automatizovanému testování potřebné, a proto potřebuje být doplněn dalšími nástroji. Jako programovací jazyk projektu byla zvolena Java, která byla již využita u Selenia. Z tohoto důvodu byl pro Playwright zvolen nástroj pro build a správu závislostí Maven, který pracuje s Javou. K němu byl připojen nástroj pro psaní a strukturování testů JUnit, který se v kombinaci s Mavenem často používá. Použití podobných nástrojů umožňuje zachovat obdobnou projektovou strukturu jako u Selenia. Stejně jako u Selenia i zde je soubor *pom.xml* klíčovou součástí projektu, který zastává funkci konfiguračního souboru pro Maven. Definuje tak důležité součásti projektu, jako jsou použitá verze Javy, seznam použitých knihoven nebo nastavení buildu.

Pro lepší přehlednost byla zvolena logika, kdy jedna třída odpovídá jednomu testu. Playwright testy lze nalézt ve složce *AutomatedTestsMoodle/src/test/java/playwrighttest*. Všechny testy začínají explicitní inicializací webového prohlížeče v kódu (viz Kód 9). Objekt *BrowserContext* vytvoří v otevřeném prohlížeči izolované prostředí, díky kterému oproti Seleniu není potřeba využít anonymní režim prohlížeče.

```
Browser browser = playwright.chromium().launch(new
BrowserType.LaunchOptions().setHeadless(false));
BrowserContext context = browser.newContext();
Page page = context.newPage();
```

Kód 9 – Playwright: inicializace prohlížeče; Zdroj: vlastní

Test-1 – Přihlášení do aplikace

Cílem testu je ověření možnosti přihlášení uživatele a správné načtení stránky po přihlášení. Test je implementován ve třídě *Login* ve složce *playwrighttests*. Začíná inicializací prohlížeče, což již bylo popsáno výše. V otevřeném prohlížeči otevře Playwright za pomocí metody *page.navigate()* URL adresu Moodle a zkontroluje nadpis stránky. V Playwrightu mají některé příkazy zabudované automatické čekání, například přechod na novou stránku nebo kliknutí na prvek na stránce. Není proto potřeba pro ně psát explicitní

čekací podmínky. Díky této funkcionalitě nemusela být vložena pro Playwright pevná časová prodleva při vyplňování přihlašovacího formuláře. Kód, který byl použit k jeho vyplnění, je demonstrován v kódu 10. Po přihlášení zkontroluje Playwright nadpis stránky, že odpovídá předpokládanému řetězci „Moje kurzy“. Pokud jsou všechny předchozí kroky úspěšně splněny, test je považován za „Passed“. Posledním krokem je zavření webového prohlížeče příkazem *browser.close();*

```
page.fill("#username", "student");
page.fill("#password", "moodle25");
page.click("#loginbtn");
```

Kód 10 – Playwright: vyplnění přihlašovacího formuláře; Zdroj: vlastní

Test-2 – Zobrazení kurzu

Cílem testu je ověřit, že student vidí kurz a může si ho zobrazit. Celý test je realizován ve třídě *CourseMaterialVisibilityCheckStudent*, která se nachází ve složce *playwrighttests*.

Začátek testu je shodný s Testem-1 (přihlášení do aplikace), otevře se stránka Moodle LMS, vyplní se přihlašovací údaje a dojde k přihlášení uživatele „student“. Pro začátek Testu-2 byl proto použit kód z prvního testu. Nový kód začíná na stránce „Moje kurzy“. Zde pomocí selektoru najde Playwright první kurz a klikne na něj. Kód 11 zobrazuje následnou kontrolu detailu kurzu. Playwright počká na načtení stránky, na které zkontroluje adresu URL a nadpis stránky. K jednotlivým kontrolám jsou přidány chybové hlášky. Ty se zobrazí, pokud se daný krok nepodaří vykonat. Při výskytu chyb tyto hlášky umožňují snazší nalezení kroku, který se nepodařilo provést. Detailnější vysvětlení výběru těchto kontrol se nachází v kapitole 4.3.1.

```
assertTrue(page.url().contains("/course/view"), "Url kurzu neodkázalo na
správnou stránku");
assertTrue(page.isVisible(".page-header-headings"), "Nebyl načten název
kurzu/nejsme na správné stránce");
```

Kód 11 – Playwright: kontrola zobrazení kurzu; Zdroj: vlastní

Test-3 – Zapsaní studenti v kurzu

Cílem testu je ověřit, že učitel vidí studenty přihlášené do kurzu. Test se nachází ve třídě *ClassStudents* uvnitř složky *playwrighttests*. Pro tento test je využit profil „teacher“. Začátek testu je shodný s Testem-2 (zobrazení kurzu), proto byl pro začátek Testu-3 použit kód z druhého testu s obměnou přihlašovacích údajů. Nový kód pokračuje na stránce detailu

kurzu. V kódu 14 je možné vidět nalezení záložky „Účastníci“ za pomocí atributu „Data-key“ a následné kliknutí. Na základě úvahy popsané v kapitole 4.3.1 ověří Playwright, že existuje alespoň jeden záznam se zapsaným studentem do kurzu. Test je ukončen příkazem pro zavření prohlížeče.

```
page.click("[data-key='participants']");
page.waitForURL("**/user/index.php**");
assertTrue(page.isVisible("[id*='user-index-participants-'][id$='_r0']"),
"V kurzu není zapsaný ani jeden student");
```

Kód 12- Playwright kontrola zapsaných studentů

Z hlediska náročnosti konfigurace a naprogramování testovaných scénářů pro vybrané frameworky se Playwright nachází uprostřed. Náročnost konfigurace prostředí byla zhruba na stejné úrovni jako u Selenia. Psaní scénářů však bylo značně jednodušší. Svou jednoduchostí se ale nevyrovnal Cypressu, ve kterém bylo vytváření projektu ještě znatelně snazší. Automatické čekání u některých příkazů a efektivní práce s elementy stránky jsou velkou výhodou Playwrightu.

4.4 Vyhodnocení testů

Kapitola se věnuje postupu měření výkonu a spolehlivosti manuálního testování a jednotlivých frameworků (Selenium, Cypress, Playwright) při testování aplikace Moodle. Cílem kapitoly je popsat použitou metodiku tak, aby bylo možné reprodukovat naměřené výsledky a správně je interpretovat.

Měření bylo provedeno na notebooku s procesorem Intel Core i5-7200 s 8GB RAM v operačním systému Windows 10 Enterprise 64bit. Testy proběhly v prohlížeči Google Chrome na verzi 142.0.7444.60. Dále byly využity frameworky pro automatizaci testování Selenium 4.27.0, Cypress 15.6.0, Playwright 1.56.0. Testovanou aplikací byla demo verze aplikace Moodle LMS (school.moodledemo.net). Během testování nebyly na počítači prováděny žádné další náročnější aktivity, které by mohli ovlivnit běh testů.

Pro měření byly využity testovací scénáře popsané v kapitolách 4.2.2-4.2.4 (přihlášení do aplikace, zobrazení kurzu, zapsaní studenti v kurzu). Tyto scénáře byly implementovány pro všechny vybrané metody testování. U jednotlivých metod byly

prováděny kroky se stejným logickým postupem. Liší se pouze jejich technická implementace daná možnostmi konkrétního frameworku.

4.4.1 Postup měření

Pro každou kombinaci vybrané metody a testovaného scénáře bylo provedeno 11 běhů. První běh byl vždy považován za zkušební (tzv. warm-up) a nebyl započítán do statistik. Do statistické analýzy tak vstupuje 10 platných běhů pro každou kombinaci metody a scénáře. Měření probíhala až nad finální, stabilní verzí testů. Případné chyby v implementaci testů byly odstraněny ve vývojové fázi a nepromítly se do měřených výsledků.

Manuální testy byly prováděny jedním testerem. Tím je simulováno pracovní prostředí, kde jeden tester vykonává svou testovací sadu stále dokola. Čas běhu byl měřen ručně na stopky. Čas se začal počítat od momentu otevření testovacího scénáře a skončil dokončením posledního kroku testu. Do testu tak byl započítán celý čas, který tester potřebuje pro nastudování testu, což nejlépe simuluje pracovní prostředí.

Pro určení doby běhu automatizovaných testů byl využit reportovací nástroj jednotlivých frameworků, který zaznamenává čas jednotlivých běhů testu. Čas se začíná počítat otevřením webového prohlížeče a končí dokončením posledního kroku testu. Po dokončení testu byl vždy zaznamenán jeho výsledek a doba běhu.

Nad výsledky měření byla provedena statistická analýza naměřených hodnot. Ze získaných dat byl pro každou metodu a každý scénář vypočten aritmetický průměr, který nadále slouží jako hlavní ukazatel doby běhu. Mezi další sledované metriky patřily minimální a maximální naměřená hodnota a směrodatná odchylka. Tyto ukazatele byly zvoleny pro lepší posouzení stability běhů. Naměřená data byla následně zobrazena v grafu v kapitole 5.1.

4.5 Postup výpočtu odhadované ceny jednotlivých metod

Cílem kapitoly je navrhnout jednoduchý model, pomocí kterého bude možné z naměřených časů odhadnout cenu jednotlivých metod testování. Model vychází z předpokladu, že čas jsou peníze (čas * peníze).

4.5.1 Vstupní data a předpoklady

Jako vstupní data pro výpočet nákladů byla využita data z měření popsaného v kapitole 4.4.1 jejichž výsledky jsou v kapitole 5.1. Pro každou metodu (manuální testování, Selenium, Cypress, Playwright) a každý testovací scénář byl vypočten aritmetický průměr doby běhu a zaznamenán celkový čas implementace. Směrodatná odchylka a minimální a maximální naměřené hodnoty nebyly pro tento výpočet využity. Slouží pouze pro posouzení stability jednotlivých metod.

Dále bylo potřeba určit hodinové mzdy pro účely výpočtu odhadovaných nákladů. Hodinovou mzdu byla rozdělena na mzdu manuálního testera a vývojáře automatizovaných testů a je vypočtena jako průměrná mzda vydělená počtem odpracovaných hodin měsíčně (160 hodin). Pro odhad průměrného platu byla využita veřejně dostupná data z portálu *platy.cz*. V tabulce 5 je možné vidět odhadnuté hodinové mzdy na základě průměrné mzdy.

Stanovení hodinové mzdy práce		
Pozice	Průměrná mzda [Kč/měsíc]	hodinová mzda [Kč/hodina]
Manuální tester	58500	365,6
Vývojář automatizovaných testů	74000	462,5

Tabulka 5 - Stanovení hodinové mzdy práce; Zdroj: vlastní

4.5.2 Výpočet odhadovaných nákladů

Na základě vstupních dat popsaných v kapitole 4.5.1 je v této práci využit univerzální vzorec pro odhad nákladů jednotlivých metod, používaný například Hoffmanem a Ramlerem. [56], [57] Vzorec rozlišuje fixní náklady (implementace testů) a variabilní náklady (provedení zvoleného počtu testů).

$$A = V + n \cdot \mathrm{D}$$

Kde V představuje náklady na implementaci testů, D náklady za jedno provedení testu a n počet běhů testu. V této bakalářské práci byl vzorec dále rozlišen na náklady manuálního a automatizovaného testování.

V případě manuálního testování představuje V_m celkové náklady na implementaci testovacích scénářů. Tyto náklady jsou vypočítány jako součin celkového času potřebného pro vytvoření testovacích scénářů a hodinové sazby manuálního testera. Variabilní složka D_m odpovídá nákladům spojeným s během manuálních testů. Je určena jako čas potřebný pro provedení testů vynásobený hodinovou mzdou manuálního testera.

U automatizované testování představuje $V_a(m)$ celkové náklady spojené s implementací scénářů do zvoleného frameworku. Stejně jako u manuální metody vychází z času potřebného pro implementaci testovacích scénářů vynásobeného hodinovou mzdou automatizačního testera. Náklady na jednotlivé běhy automatizovaných testů $D_a(m, s)$ jsou v této práci zanedbány. Testy mají krátkou dobu běhu a jsou spouštěny na existující infrastruktuře, takže automatizační tester během jejich běhu nepracuje (testy mohou být spouštěny bez jeho přítomnosti). Náklady na jeden běh testu jsou tedy obtížné stanovit, přičemž je tato cena v porovnání s náklady za implementaci automatizovaných testů zanedbatelná. Toto zjednodušení vede k mírnému podhodnocení nákladů automatizovaného testování, což je potřeba zohlednit při implementaci výsledků.

Veškeré časy implementace a běhu testů byly převedeny na hodiny a násobeny hodinovou sazbou. Výsledné hodnoty slouží k porovnání ekonomické výkonosti jednotlivých metod. Dále jsou použity při stanovení hodnot kritéria „náklady testů“ ve vícekriteriální analýze v kapitole 5.3.

Navržený postup výpočtu nákladů jednotlivých metod představuje zjednodušený pohled na ekonomiku testování. Do výpočtu nákladů nebyly zahrnuty důležité faktory spojené s implementací a údržbou testovacích metod jako například licence testovacích nástrojů, odvody zaměstnavatele za zaměstnance, odpisy investic, ani nepřímé přínosy automatizovaného testování (např. častější testování nebo vyšší kvalita softwaru v produkci). Cílem této bakalářské práce je spíše stanovení odhadu nákladů vybraných testovacích scénářů do jednotlivých metod testování a na jejich provedení než jejich kompletní ekonomická analýza. Odhady jsou následně použity pro stanovení hodnot kritéria „Náklady testu“ ve vícekriteriální analýze prvků v kapitole 5.3.

4.6 Vícekriteriální analýza variant

Kapitola se věnuje porovnání manuálního testování a nástrojů pro automatizované testování (Selenium, Cypress, Playwright) pomocí vícekriteriální analýzy variant. Metodologicky tato kapitola vychází z poznatků získaných v kapitole 3.7. Pro analýzu byla vybrána kritéria, která hodnotí různé aspekty testování softwaru. Hodnoty kritérií vycházejí z praktické části bakalářské práce a teoretických poznatků získaných v předchozích kapitolách.

Nejprve jsou definována kritéria vícekriteriální analýzy variant (kapitola 4.5.1) a stanovena jejich důležitost ve formě relativních vah (kapitola 4.5.2). Následně je popsána metoda hodnocení výsledků jednotlivých variant. Na základě ohodnocení je vypočteno celkové skóre každé varianty a určeno výsledné pořadí metod.

4.6.1 Definování kritérií

Pro vícekriteriální analýzu prvků byla zvolena tato kritéria:

- **Náklady testů** – finanční náročnost implementace a provedení testů. Hodnoty vycházejí z odhadu ceny jednotlivých metod vyjádřené v kapitole 5.2.
- **Rychlost testů** – jak dlouho v průměru trvalo provedení vybraných scénářů. Hodnoty vycházejí z běhu testů v kapitole 5.1.
- **Srozumitelnost výstupů** – jak snadno čitelné a přehledné jsou výsledky testů pro testera. Hodnoty jsou stanoveny subjektivně autorem práce.
- **Snadnost implementace a údržby testů** – jak obtížné je testy vytvořit a provádět v nich změny. Hodnoty jsou stanoveny subjektivně autorem práce.
- **Pokrytí testů** – jak široký rozsah funkcionalit je možné danou metodou otestovat. Hodnoty vycházejí ze znalostí získaných v teoretické i praktické části bakalářské práce.
- **Flexibilita** – vyjadřuje schopnost nástroje přizpůsobit se různým prostředím a požadavkům. Hodnoty vycházejí ze znalostí získaných v teoretické i praktické části bakalářské práce.

Kritéria byla zvolena na základě konzultace s odborníky z praxe. Odráží tak důležitá témata při výběru metody testování softwaru. Zároveň byla vybírána tak, aby je bylo možné hodnotit na základě dat získaných z předchozích kapitol. Vybraná kritéria pokrývají technické, ekonomické a uživatelské aspekty výběru testovací metody.

Kritéria nákladů testů a rychlosti testů spolu částečně souvisí. Náklady testů v použitém modelu vycházejí především z času běhu testů. U automatizovaných testů se tak jejich hodnota liší jen minimálně. Přesto je vhodné tato kritéria pro účely bakalářské práce ponechat. Čas hraje primární roli při výběru testovací metody. Náklady testů navíc zohledňují rozdílnou hodinovou sazbu manuálního a automatizovaného testování.

4.6.2 Stanovení vah kritérií

Pro stanovení vah kritérií byla použita bodovací metoda. Při této metodě je ze zvolené stupnice jednotlivým kritériím přiřazován určitý počet bodů. Čím lépe je dané kritérium hodnocené, tím větší má počet bodů.[53] Nejdůležitějším kritériím je přiřazováno nejvíce bodů. Pro účely bakalářské práce byla zvolena stobodová škála. Součet bodů všech kritérií je 100. Váhy jsou vypočítány jako *bodové ohodnocení / 100*, viz Tabulka 5

Stanovení vah bodovou metodou		
Kritérium	**Bodové ohodnocení**	**Váha**
Náklady testů	25	0,25
Rychlost testů	20	0,20
Srozumitelnost výstupů	10	0,10
Snadnost implementace	15	0,15
Pokrytí testů	20	0,20
Flexibilita	10	0,10
Suma	100	1

Tabulka 6 - Stanovení vah bodovou metodou; Zdroj: vlastní

Zdroj: vlastní zpracování

Váhy kritérií byly navrženy autorem a následně konzultovány s odborníky z praxe. Největší váhy byly přiděleny nákladům testů (25 %), následně rychlosti testů (20 %) a pokrytí testů (20 %). V praxi firem nejvíce zajímají ekonomické a následně technické

aspekty výběru testovací metody. Proto bylo nejlépe ohodnoceno kritérium nákladů testů. Rychlost testů a pokrytí testů mají primárně technický aspekt, ale s velkým ekonomickým dopadem. Menší váhy byly přiděleny srozumitelnosti výstupů (10 %), snadnosti implementace (15 %) a flexibilitě (10 %). Kritéria související spíše s uživatelskou přívětivostí byla hodnocena jako méně významná, než aspekty ekonomické a technické.

4.6.3 **Metoda hodnocení výsledků**

Na základě výsledků praktické části a teoretických poznatků byla všechna kritéria ohodnocena bodovou škálou 1-10, kde vyšší hodnota vyjadřuje lepší výsledek dané varianty. Kritérium náklady testů má minimalizační povahu. Bylo jej tak nutné přepočítat s inverzní logikou – čím nižší náklady, tím vyšší skóre. Následně byla pro vyhodnocení variant využita metoda váženého součtu pomocí vzorce:

$$\boldsymbol{h_i = \sum_{j=1}^{k} v_j \cdot y_{ij}}$$

kde h_i je ohodnocení i-té varianty, k je počet kritérií, v_j je normovaná hodnota váhy j-tého kritéria a y_{ij} jsou hodnoty i-té varianty v j-tém kritériu. Na základě výsledných hodnot je určeno pořadí, varianta s nejvyšší výslednou hodnotou je ohodnocena jako ta nejvhodnější.

5 Výsledky a diskuse

Kapitola se zabývá analýzou a zhodnocením výsledků testů pro vybrané metody testování (manuální testování, Selenium, Cypress, Playwright). Výsledky jsou prezentovány pomocí grafických zobrazení a tabulkových srovnání. Na jejich základě jsou formulovány závěry a doporučení pro jednotlivé metody.

5.1 Běhy testů

Porovnání vybraných metod probíhalo na třech testovacích scénářích. Každý scénář byl pro každou metodu spuštěn 11krát, přičemž první běh sloužil jako warm-up a nebyl započítán do statistického hodnocení. Do statistického hodnocení tak vstupuje 10 platných běhů testů pro každou kombinaci metody a scénáře. Všechny platné běhy testů skončily úspěšně, což znamená, že během testů nebyla v aplikaci Moodle nalezena žádná chyba.

U jednotlivých metod jsou sledovány tyto statistiky: doba implementace[min], průměrná doba běhu[s], minimální a maximální doba běhu a směrodatná odchylka. Na základě těchto statistik jsou mezi sebou metody porovnány.

V tabulkách 7-10 jsou uvedeny výsledky běhů jednotlivých testů pro všechny vybrané metody testování.

Manuální testování					
Test	**Doba implementace[min]**	**Průměrná doba běhu [s]**	**Minimální**	**Maximální**	**Směrodatná odchylka**
Test č.1	5	37	33	46	3,8297084
Test č.2	6	60	55	68	3,2998316
Test č.3	5	91,8	87	100	3,8528489
Součet	16	188,8			

Tabulka 7 - Výsledky manuálního běhu testů; Zdroj: vlastní

Selenium testování					
Test	**Doba implementace[min]**	**Průměrná doba běhu [s]**	**Minimální**	**Maximální**	**Směrodatná odchylka**
Test č.1	120	9,2666	8,784	9,667	0,2819299
Test č.2	40	11,671	10,882	13,304	0,7638001
Test č.3	60	14,0769	12,604	16,137	0,955769
Součet	220	35,0145			

Tabulka 8 - Výsledky Selenium běhu testů; Zdroj: vlastní

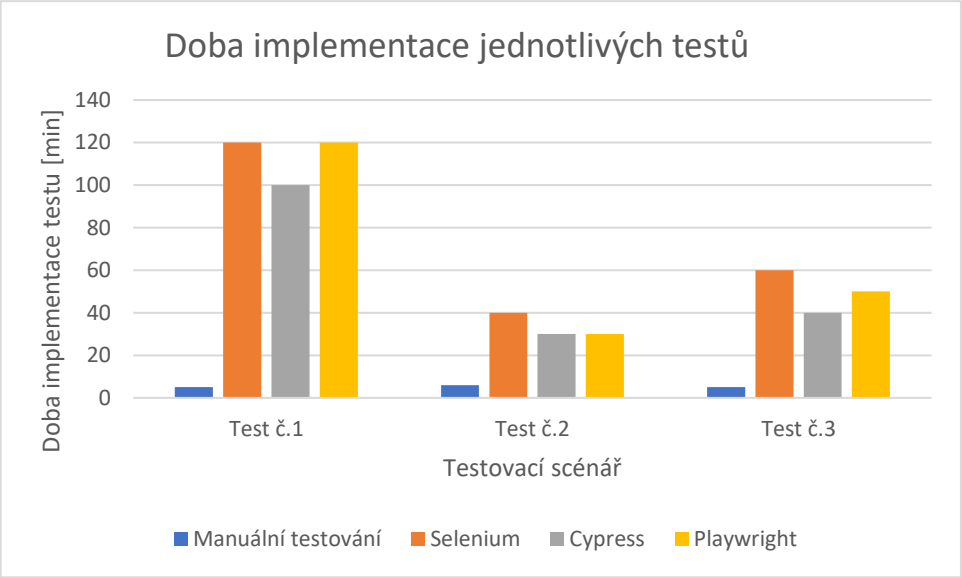
Cypress testování					
Test	**Doba implementace [min]**	**Průměrná doba běhu [s]**	**Minimální**	**Maximální**	**Směrodatná odchylka**
Test č.1	100	2,6897	2,518	2,853	0,1095283
Test č.2	30	5,3675	4,058	6,486	0,941463
Test č.3	40	7,2716	5,566	10,231	1,7590877
Součet	170	15,3288			

Tabulka 9 - Výsledky Cypress běhu testů; Zdroj: vlastní

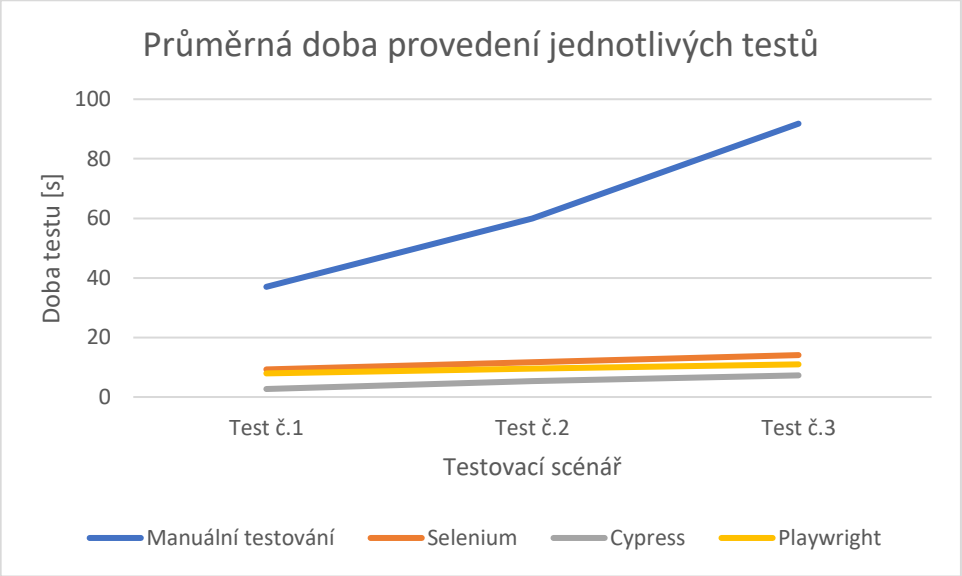
Playwright testování					
Test	**Doba implementace[min]**	**Průměrná doba běhu [s]**	**Minimální**	**Maximální**	**Směrodatná odchylka**
Test č.1	120	7,9335	6,699	9,867	1,1280374
Test č.2	30	9,4962	8,483	10,212	0,5915591
Test č.3	50	10,9695	9,39	13,119	1,1137323
Součet	200	28,3992			

Tabulka 10 - Výsledky Playwright běhu testů; Zdroj: vlastní

V následujících grafech jsou pak porovnány hodnoty doby implementace a průměrné doby běhu testovacích scénářů mezi jednotlivými metodami.



Graf 1 - Doba implementace jednotlivých testů; Zdroj: vlastní



Graf 2 - Průměrná doba provedení jednotlivých testů; Zdroj: vlastní

Z grafu 1 je patrné, že nejkratší dobu implementace vykazuje manuální testování. Celkový čas implementace všech tří scénářů byl 16 minut. To je více než desetinásobně méně než u druhé nejrychlejší metody, Cypressu. Z hlediska času potřebného pro implementaci jsou automatizované metody relativně vyrovnané. Celková doba implementace se u nich pohybuje v rozmezí přibližně 170 až 220 minut.

Odlišný obraz poskytuje graf 2, který porovnává průměrnou dobu provedení testů. S celkovým časem 188,8 sekund trvalo nejdéle manuální testování, což je výrazně více než u ostatních metod. Zároveň je u manuálního testování patrný strmější nárůst doby provedení, což oproti automatizovaným testům naznačuje vyšší citlivost na složitost scénářů.

Automatizované testy jsou z hlediska celkové doby provedení poměrně vyrovnané. Nejrychleji testy provedl Cypress (přibližně 15,3 s), následovaný Playwrightem (přibližně 28,4 s) a nejpomalejší z automatizovaných nástrojů bylo Selenium (přibližně 35,0 s). Při detailnějším zkoumání je patrné, že s přibývající náročností testů dochází u všech metod k nárůstu doby provedení. Při zaměření na automatizované metody, byl časový nárůst největší u Selenia (5 sekund rozdíl mezi prvním a třetím testem) a nejmenší u Playwrightu (3 sekundy rozdíl mezi prvním a posledním testem). To naznačuje, že Playwright je vhodnější pro složitější testy.

Z hlediska jednorázového provedení testů a času potřebného na jejich přípravu se tak manuální testování jeví jako nejméně náročný přístup. Při nárůstu počtu scénářů, zvýšení náročnosti testování nebo opakovaném testování však významně převažují časové výhody automatizovaných metod.

Směrodatná odchylka doplňuje pozorování z těchto dvou grafů. U manuálního testování je hodnota výrazně vyšší (3-4 s), než u automatizovaných metod (0,1-1,7 s). Pro metody Cypress a Selenium má směrodatná odchylka tendenci postupně růst se zvyšující se náročností testu, zatímco pro Playwright zůstává relativně stabilní.

5.2 Odhad ceny jednotlivých metod

Kapitola se zabývá odhadem nákladů na jednotlivé metody testování. Odhadované náklady vycházejí z doby implementace a u manuálního testování také z naměřených dob běhů testů. Pro výpočet nákladů byly zvoleny dvě situace. První odpovídá implementaci scénářů a běhu 1 sady testovacích scénářů, druhá implementaci scénářů a běhu 100 sad testovacích scénářů viz tabulka 11.

Odhadované hodnoty jednotlivých metod v korunách		
Metody	**Celková cena při 1 běhu sady [Kč]**	**Celková cena při 100 bězích sady [Kč]**
Manuální testování*	116,68	2015,00
Selenium	1695,83	1695,83
Cypress	1310,42	1310,42
Playwright	1541,67	1541,67

Tabulka 11 - Odhadovaná celková cena jednotlivých metod v korunách; Zdroj: vlastní

* Samotná implementace manuálních scénářů vychází na 97,5 Kč.

Z tabulky je patrné, že při jednom provedení celé sady je nejlevnější metodou manuální testování (116,68 Kč). Manuální testování má krátkou dobu implementace a tím pádem i nízké fixní náklady. Z automatizovaných nástrojů vychází nejlevněji Cypress (1310,42 Kč), následně Playwright (1541,67 Kč) a nejdražší Selenium (1695,83 Kč). Pořadí je tak v tomto scénáři nejvíce ovlivněno délkou implementace testovacích scénářů.

Při 100 bězích se pořadí změní. Manuálním testům výrazně rostou variabilní náklady a metoda se tak stává nejdražší (2015 Kč). U automatizovaných metod se naopak cena nemění, protože model uvažuje pouze fixní náklady. Nejlevnější metodou je tak Cypress (1310,42 Kč), následovaný Playwrightem (1541,67 Kč) a Seleniem (1695, 83 Kč).

Pro malé množství spuštění testovacích scénářů je z hlediska nákladů ekonomičtější manuální testování. V praxi však může testování softwaru zahrnovat desítky až tisíce testovacích scénářů. V takovém případě se z hlediska nákladů jeví jako nejvhodnější volba metoda Cypress. Pro vícekriteriální analýzu v následující kapitole tak bude uvažována varianta s celkovou cenou při 100 bězích testů.

5.3 Vícekriteriální analýza vybraných metod

Na základě výsledků praktické části byla provedena vícekriteriální analýza vybraných metod. Pro ohodnocení byla po konzultaci s odborníky z praxe zvolena kritéria ohodnotitelná z výsledků práce a byly jim stanoveny váhy. Zvolená kritéria a jejich váhy jsou shrnuty v tabulce 12, která zároveň obsahuje výsledné ohodnocení jednotlivých metod v daných kritériích.

Vícekriteriální analýza vybraných metod založená na jejich hodnocení v testovacích metodách								
Kritérium	**Náklady testu**	**Rychlost testů**	**Srozumitelnost výstupu testů**	**Snadnost údržby**	**Pokrytí testů**	**Flexibilita**	**Skalární součin**	**Pořadí**
Manuální testování	2	2	10	8	6	9	5,2	4.
Selenium	6	7	5	5	8	5	6,25	3.
Cypress	9	9	8	8	9	7	8,55	1.
Playwright	7	8	7	7	9	8	7,7	2.
Váhy	0,25	0,2	0,10	0,15	0,20	0,10		

Tabulka 12 - Vícekriteriální analýza prvků; Zdroj: vlastní

Hodnocení nákladů testů a rychlosti testů vychází přímo z výsledků kapitol 5.1 a 5.2, kde byly pro jednotlivé metody porovnány doby implementace, čas potřebný pro běh testu a náklady. Kritéria srozumitelnost výstupu testů a flexibilita byla hodnocena kvalitativně, na základě pozorování průběhu testů. Metoda manuálního testování dosáhla v těchto kritériích nejlepšího hodnocení. Během testování pracuje tester s uživatelským rozhraním aplikace a má tak možnost rovnou vidět, co se děje. Nejhůře pak v těchto kategoriích bylo hodnoceno Selenium, které jako starší framework nemá bohaté UI a výsledky jsou tak zobrazovány pouze v logách. Pro kritérium snadnost údržby byly lépe hodnoceny modernější frameworky Cypress a Playwright, jejichž skripty jsou oproti Seleniu stručnější a lépe čitelné. Metody automatizovaného testování oproti manuálnímu testování byly lépe hodnoceny i pro kritérium pokrytí testů. Díky moderním strukturám a stále se rozvíjejícím funkcím jsou dnes schopné ve větším měřítku otestovat většinu funkcionalit aplikace.

Z vícekriteriální analýzy vyšel nejlépe Cypress (skalární součin 8,55). Získal nejlepší hodnocení u kritérií s vysokou váhou – v porovnání s ostatními metodami má nejnižší náklady spojené s implementací a během testu, nejrychleji otestoval vybrané scénáře a díky modernímu prostředí pokrývá většinu potřebných funkcionalit. Na druhém místě se umístil Playwright (7,7), který v těchto kritériích s vysokou váhou lehce zaostal. Lépe byl hodnocen jen u kritéria flexibilita, kdy je díky velkému množství doplňků lépe schopný se přizpůsobit. Třetí skončilo Selenium (6,25), které bylo oproti modernějším frameworkům hodnoceno hůře ve všech kategoriích, zejména v nákladech testu a snadnosti údržby. Nejhůře ohodnoceno bylo manuální testování (5,2), které nejvíce ztratilo v kategoriích náklady testu a rychlost testů. Lépe si vedlo v kategoriích srozumitelnost výstupu testů a flexibilita, kde bylo ohodnoceno jako nejlepší. Tato kritéria však mají malou váhu.

5.4 Doporučení

Na základě provedených testů a analýzy dat lze pro praxi doporučit Cypress jako nejvhodnější framework pro projekty s velkým množstvím testovacích scénářů nebo s častějším spouštěním testů. Z hodnocených frameworků dopadl nejlépe zejména díky nejnižším odhadovaným nákladům na testování a nejkratší průměrné době běhu testů. Jako alternativu lze doporučit framework Playwright, zvláště v případech, kdy je potřeba přizpůsobit se různorodému testovacímu prostředí, například díky lepší podpoře více webových prohlížečů. Využití Selenia může mít smysl v projektech, ve kterých je tento framework již dlouhodobě používán a existuje pro něj rozsáhlá sada testovacích scénářů.

Manuální testování se podle výsledků práce jeví jako vhodná metoda pro situace, kdy je potřeba jednorázově otestovat menší část aplikace nebo kdy probíhá průzkumové (exploratorní) testování, při němž správné chování aplikace není předem známo.

5.4.1 Další studie

Zjištěná doporučení obecně odpovídají závěrům jiných prací. Ke stejnému závěru došli ve své práci i Patil a Rewatkar [58], kteří pro praxi doporučují kombinaci manuálního a automatizovaného testování. Nástroje pro automatizované testování hodnotí jako vhodnější volbu pro velký počet opakujících se testovacích běhů, zatímco manuální testování doporučují například pro průzkumové scénáře. Na vybrané automatizované nástroje se zaměřil Tymoshchuk [59], který také porovnával nástroje Selenium, Cypress a Playwright. Cypress a Playwright ohodnotil jako nejvhodnější frameworky, přičemž o něco lépe vyšel v testech rychlosti Playwright. K podobným závěrům dospěl i Moń [60], který se kromě času potřebného pro běh testů zajímal i o využití CPU a RAM v operačním systému Windows. Jeho výsledky jsou tak v souladu s hypotézou této práce, podle níž s přibývající náročností testů rostla časová náročnost jejich běhu nejpomaleji u Playwrightu. Tato práce oproti uvedeným studiím rozšiřuje výsledné porovnání o metodu manuálního testování, model nákladů a vícekriteriální analýzu, na jejichž základě byl Cypress vyhodnocen jako nejvýhodnější.

5.4.2 **Další možnosti rozvoje této práce**

Tato bakalářská práce představuje vhodný výchozí bod pro další výzkum zaměřený na komplexnější porovnání manuálního a automatizovaného testování. Navazující práce může zahrnout více testovaných aplikací a použít širší sadu testovacích scénářů, aby bylo možné ověřit zobecnění dosažených závěrů. Ekonomický pohled lze prohloubit doplněním dalších nákladových faktorů, čímž se zpřesní odhad nákladů jednotlivých metod (například licence testovacích nástrojů, odvody zaměstnavatele za zaměstnance, odpisy investic). Vícekriteriální analýzu je možné dále rozšířit na základě diskuse s odborníky z praxe o další kritéria hodnocení, což může přispět k přesnějšímu posouzení jednotlivých metod testování.

6 Závěr

Bakalářská práce se zabývala porovnáním manuálního a automatizovaného testování softwaru. Teoretická část práce představila základy testování softwaru, jednotlivé fáze testování, typy testovacích technik a úrovně testování. Dále se věnovala popisu aktuálních nástrojů pro testování softwaru. Součástí teoretické části bylo také představení metodiky vícekriteriálního rozhodování, která slouží jako nástroj pro co nejobjektivnější hodnocení jednotlivých variant.

Praktická část práce zahrnovala výběr aplikace Moodle LMS a následné vytvoření tří testovacích scénářů pro tuto platformu. Scénáře byly následně implementovány do vybraných nástrojů pro automatizované testování – Selenia, Cypressu a Playwrightu. Pro vybrané metody bylo provedeno deset platných běhů testů, během kterých nebyly odhaleny chyby a na jejichž základě byla posouzena efektivita a časová náročnost. Na základě výsledků doby běhu testů a doby implementace byl pomocí jednoduchého modelu stanoven odhad nákladů jednotlivých metod. Výsledné hodnoty byly následně použity ve vícekriteriální analýze k porovnání efektivity, kvality a pokrytí jednotlivých metod za pomocí zvolených kritérií.

Na základě výsledků byla formulována doporučení pro praxi a vhodné využití každé metody, například vhodnost Cypressu pro rychlé otestování a nízké náklady při častém spouštění testů, Playwrightu pro přizpůsobení se různorodým testovacím prostředím nebo manuální testování pro jednorázové či průzkumové (exploratorní) testování. Využití Selenia může mít smysl zejména v projektech, ve kterých je tento framework již využíván a existují pro něj testovací scénáře. Výsledky byly následně porovnány s podobnými pracemi. Výsledky této práce mohou dále sloužit jako podklad pro rozhodování při výběru vhodné testovací metody nebo konkrétního frameworku v oblasti testování webových aplikací.

7 Seznam použitých zdrojů

[1] HOMÈS, Bernard. *Fundamentals of software testing*. Revised and updated 2nd edition. London, UK: ISTE, 2024. Computer engineering series. ISBN 978-1-78630-982-2.

[2] DR. DINESH JAIN a DR. ABHAY KOTHARI. *Software Testing and Quality Assurance: Exploring testing levels, test tools, automation, and quality metrics for improved software quality (English Edition)*. India: BPB Publications, 2025. ISBN 978-93-65892-789.

[3] KAUL, Neha. *Analytic Methods of Systems and Software Testing*. Arcler Education, 2020. ISBN 9781774079157.

[4] QUADRI, S.M.K a Sheikh Umar FAROOQ. Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications* [online]. Foundation of Computer Science, 2010, 2010-9-10, **6**(9), 7-10 [cit. 2025-11-28]. ISSN 0975-8887. Dostupné z: doi:10.5120/1343-1448

[5] PARASOFT. *Develop a Strategy and Business Case for Test Automation: USING ROI CALCULATIONS TO DERIVE VALUE* [online]. Dostupné také z: https://alm.parasoft.com/hubfs/Whitepaper-Develop-Strategy-Business-Case-Test-Automation.pdf

[6] M. HANEFI CALP a UTKU KOSE. Planning Activities in Software Testing Process: A Literature Review and Suggestions for Future Research. *Journal of Science* [online]. 2018, 801-819 [cit. 2025-11-28]. Dostupné z: https://arxiv.org/pdf/1903.01222

[7] ZIMČÍKOVÁ, Jana. Testovací proces. *Itnetwork.cz* [online]. [cit. 2025-11-28]. Dostupné z: https://www.itnetwork.cz/testovani/istqb/testovaci-proces

[8] PERANZO, Pete. Why Software Testing Teams Are Critical for Success. *Imaginovation* [online]. May 7th, 2024 [cit. 2025-11-28]. Dostupné z: https://imaginovation.net/blog/importance-of-software-testing-team/

[9] SEELA, Swati. How to Write Test Cases in Software Testing (Examples). *Software Testing Help* [online]. 2021, 9. Května 2025 [cit. 2025-11-28]. Dostupné z: https://www.softwaretestinghelp.com/how-to-write-effective-test-cases-test-cases-procedures-and-definitions/

[10] What is a Test Suite & Test Case? (with Examples). *Browserstack* [online]. 2025, July 10, 2025 [cit. 2025-11-28]. Dostupné z: https://www.browserstack.com/guide/what-is-test-suite-and-test-case

[11] VIJAY. Test Plan Vs Test Strategy Vs Test Case Vs Test Scenario. *Software Testing Help* [online]. 2025, Updated May 9, 2025 [cit. 2025-11-28]. Dostupné z: https://www.softwaretestinghelp.com/difference-between-test-plan-test-strategy-test-case-test-script-test-scenario-and-test-condition/

[12] SONAL DWIVEDI. Use Case vs Test Case: Core Differences. *Browserstack* [online]. 2024, November 26, 2024 [cit. 2025-11-28]. Dostupné z: https://www.browserstack.com/guide/use-case-vs-test-case

[13] SAURABH PATIL. Difference Between Bug and Defect. *QAcraft* [online]. c2025 [cit. 2025-11-28]. Dostupné z: https://qacraft.com/difference-between-bug-and-defect

[14] SHOBHIT GUPTA a RIFA NIZAM KHAN. DIFFERENT ACTIVITIES INVOLVED IN SOFTWARE TESTING LIFE CYCLE. *International Journal Of Advanced Technology In Engineering And Science* [online]. February 2015, **3**(1), 19-22 [cit. 2025-11-28]. ISSN 2348 – 7550. Dostupné z: https://www.researchgate.net/publication/391121497_DIFFERENT_ACTIVITIES_INVOLVED_IN_SOFTWARE_TESTING_LIFE_CYCLE

[15] What Is the Software Testing Life Cycle? A Complete Guide. *Tricentis Testim* [online]. 2025, February 03, 2025 [cit. 2025-11-28]. Dostupné z: https://www.testim.io/blog/software-testing-life-cycle/

[16] Software Testing Life Cycle (STLC). *GeeksforGeeks* [online]. Last Updated : 29 Sep, 2025 [cit. 2025-11-28]. Dostupné z: https://www.geeksforgeeks.org/software-testing/software-testing-life-cycle-stlc/

[17] MARTINA STOJMANOVSKA. Software Testing Process: A Comprehensive Guide to Methods and Stages. *TestDevLab* [online]. 2024, December 13, 2024 [cit. 2025-11-28]. Dostupné z: https://www.testdevlab.com/blog/software-testing-process-methods-and-stages

[18] YOGESH SOLANKI. How to Write Test Scenarios That Ensure App Success. *TestGrid* [online]. c2025, March 21, 2025 [cit. 2025-11-28]. Dostupné z: https://testgrid.io/blog/test-scenarios/

[19] RAJKUMAR. How To Create Test Scenarios With Examples. *Software Testing Material* [online]. c2025, Updated on June 11, 2025 [cit. 2025-11-28]. Dostupné z: https://www.softwaretestingmaterial.com/test-scenarios/
[20] HADDAD, Rowan. Test Environments: Differences Between Dev, Staging, Preprod…. *AB Tasty* [online]. c2025, Feb 17, 2022 [cit. 2025-11-28]. Dostupné z: https://www.abtasty.com/blog/test-environment/
[21] Development and Test Environments: Understanding the Different Types of Environments. *Unitrends* [online]. c2025, July 23, 2021 [cit. 2025-11-28]. Dostupné z: https://www.unitrends.com/blog/development-test-environments/
[22] KULKARNI, Sourabh. The Necessity of System Integration Testing (SIT) and User Acceptance Testing (UAT) in Project Lifecycle. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT* [online]. Edtech Publishers (OPC) Private Limited, 2024, 2024-11-10, **08**(11), 1-7 [cit. 2025-11-28]. ISSN 2582-3930. Dostupné z: doi:10.55041/ijsrem7673
[23] SHEREMETA, Olga. Roles & Responsibilities in a Software testing team. *Testomat* [online]. c2025, Update Feb 09, 24 [cit. 2025-11-28]. Dostupné z: https://testomat.io/blog/roles-responsibilities-in-a-software-testing-team/
[24] Software Testing Roles and Responsibilities. *International Software Test Institute* [online]. c2025 [cit. 2025-11-28]. Dostupné z: https://www.test-institute.org/Software_Testing_Roles_And_Responsibilities.php
[25] STRAY, Viktoria, Raluca FLOREA a Lucas PARUCH. Exploring human factors of the agile software tester. *Software Quality Journal* [online]. c2025 [cit. 2025-11-28]. Dostupné z: https://d-nb.info/1242610197/34
[26] BINKS, Jonathan. Optimal Tester to Developer Ratios. *Prolifics Testing* [online]. 05 October 2023 [cit. 2025-11-28]. Dostupné z: https://www.prolifics-testing.com/news/optimal-tester-to-developer-ratios
[27] CHOWDHURY, Tasnova. Building Quality Assurance Team: Everything You Need to Know. *Technext* [online]. c2012-2025, Updated on September 27, 2023 [cit. 2025-11-28]. Dostupné z: https://technext.it/building-quality-assurance-team
[28] FLOREA, Raluca, Viktoria STRAY a Dag I.K. SJØBERG. On the roles of software testers: An exploratory study. *The Journal of Systems & Software* [online]. 2023, **204**, 1-11

[cit. 2025-11-29]. Dostupné z:
https://www.sciencedirect.com/science/article/pii/S0164121223001371
[29] TETTEH, Samuel Gbli. Software Testing Techniques and Levels in Software Development. *ESP International Journal of Advancements in Computational Technology* [online]. 2024, **2**(1), 10-19 [cit. 2025-11-28]. ISSN 2583-8628. Dostupné z: doi:10.56472/25838628/IJACT-V2I1P102
[30] DOPLNIT
[31] JAN, Syed Roohullah, Syed Tauhid Ullah SHAH, Zia Ullah JOHAR, Yasin SHAH a Fazlullah KHAN. An Innovative Approach to Investigate Various Software Testing Techniques and Strategies. *IJSRSET* [online]. 2016, **2**(2), 682-689 [cit. 2025-11-28]. ISSN 2394-4099. Dostupné z:
https://www.researchgate.net/publication/303280520_An_Innovative_Approach_to_Investigate_Various_Software_Testing_Techniques_and_Strategies
[32] SHAH ET AL., Kevin N. Redefining Software Quality Assurance: A Deep Dive into Automated Testing Strategies. *The Review of Contemporary Scientific and Academic Studies* [online]. The Review of Contemporary Scientific and Academic Studies, 2025, 2025-3-30, **5**(3), 1-11 [cit. 2025-11-28]. ISSN 2583-1380. Dostupné z: doi:10.55454/rcsas.5.03.2025.002
[33] GAVI, Kounde a Filiz SARI. Enhancing Efficiency in Electronic Component Validation: The Role of Automated Testing Systems and Signal Generation in Modern Testing Tools. *Researchgate* [online]. November 2024 [cit. 2025-11-28]. Dostupné z: https://www.researchgate.net/publication/386215490
[34] STEEGMANS, Eric, P. BEKAERT, F. DEVOS, G. DELANOTE, N. SMEETS, M. VAN DOOREN a J. BOYDENS. Black & White Testing: Bridging Black Box Testing and White Box Testing. *ResearchGate* [online]. 2004 [cit. 2025-11-28]. Dostupné z: https://www.researchgate.net/publication/251509538
[35] HUSSAIN, Taraq. A Comparative Study of Software Testing Techniques Viz. White Box Testing Black Box Testing and Grey Box Testing. *International Journal of Allied Practice, Research and Review* [online]. 2015, **2**(5), 1-8 [cit. 2025-11-28]. ISSN 2350-1294. Dostupné z: https://www.researchgate.net/publication/276028491
[36] TANLI, Mengqing, Ying ZHANG, Yulin WANG a Yan JIANG. Grey-Box Technique of Software Integration Testing Based on Message. *Journal of Physics:*

Conference Series [online]. IOP Publishing, 2021, 2021-9-1, **2025**(1), 012096 [cit. 2025-11-28]. ISSN 1742-6588. Dostupné z: doi:10.1088/1742-6596/2025/1/012096
[37] DOPLNIT
[38] TETTEH, Samuel Gbli. Software Testing Techniques and Levels in Software Development. *ESP International Journal of Advancements in Computational Technology* [online]. 2024, **2**(1), 10-19 [cit. 2025-11-28]. ISSN 2583-8628. Dostupné z: doi:10.56472/25838628/IJACT-V2I1P102
[39] NASTARAN NAZAR ZADEH. *Software Testing and User Experience* [online]. Toronto Academic Press, 2024 [cit. 2025-11-28]. ISBN 978-1-77956-199-2. Dostupné z: https://ebookcentral.proquest.com
[40] DUBEY, Sheela. Test Automation Revisited: Comparative Analysis of Tools and Frameworks for Scalable Software Testing. *International Journal for Research in Applied Science and Engineering Technology* [online]. International Journal for Research in Applied Science and Engineering Technology (IJRASET), 2025, 2025-9-30, **13**(9), 207-216 [cit. 2025-11-29]. ISSN 2321-9653. Dostupné z: doi:10.22214/ijraset.2025.73663
[41] GRAY, Courtney. The Top 10 Test Automation Tools of 2025. *Leapwork* [online]. 2025, June 24, 2025 [cit. 2025-11-29]. Dostupné z: https://www.leapwork.com/blog/top-20-test-automation-tools
[42] JOSHI, Twinkle. Breaking Down Automation Frameworks: Building Blocks for Better Testing. *European Journal of Advances in Engineering and Technology* [online]. 2024, 104-116 [cit. 2025-11-29]. Dostupné z: doi:10.5281/zenodo.15234711
[43] BRYNDZA, Martin. Master Test Automation: 7 Types of Test Automation Frameworks. *Y Soft AIVA* [online]. c2025 [cit. 2025-11-29]. Dostupné z: https://www.ysoft.com/aiva/blog/test-automation-frameworks
[44] DESHPANDE, Chinmayee. An Introduction to Selenium WebDriver. *Simplelearn* [online]. c2009-2025, Last updated on Aug 13, 2024 [cit. 2025-11-29]. Dostupné z: https://www.simplilearn.com/tutorials/selenium-tutorial/what-is-selenium-webdriver
[45] BHIMANAPATI, Viharika, Punit GOEL a Ujjawal JAIN. Leveraging Selenium and Cypress for Comprehensive Web Application Testing. *Journal of Quantum Science and Technology* [online]. Resagate Global, 2024, 2024-3-31, **1**(1), 66-79 [cit. 2025-11-29]. ISSN 3048-6351. Dostupné z: doi:10.36676/jqst.v1.i1.10

[46] ER AMAN SHRIVASTAV a KUMARESAN DURVAS JAYARAMAN. Automated Testing Frameworks: A Case Study Using Selenium and NUnit. *Nternational Journal of Research in Humanities & Soc. Sciences* [online]. 2025, **13**(1), 1-16 [cit. 2025-11-29]. Dostupné z: https://www.researchgate.net/publication/390172511

[47] Selenium Overview. *Selenium* [online]. c2025, Last modified February 6, 2024 [cit. 2025-11-29]. Dostupné z: https://www.selenium.dev/documentation/overview/

[48] Pros and Cons of Selenium as an Automation Testing tool. *GeeksforGeeks* [online]. c2025, Last Updated : 23 Jul, 2025 [cit. 2025-11-29]. Dostupné z: https://www.geeksforgeeks.org/software-testing/pros-and-cons-of-selenium-as-an-automation-testing-tool/

[49] SHOKEH, Sarah, Mohammad G. SABOBEH, Ahmad BAKER a Samer ZEIN. *A Comparative Evaluation of Cypress and Katalon for Web Application Test Automation* [online]. Palestine, 2025 [cit. 2025-11-29]. Dostupné z: https://www.researchgate.net/publication/397099356

[50] ACESKA, Emilija. Cypress Testing: What is It and Why is It Important? *TestDevLab* [online]. 2025, February 20, 2025 [cit. 2025-11-29]. Dostupné z: https://www.testdevlab.com/blog/what-is-cypress-testing

[51] JOHNSON, Elly. What is Playwright? Its Features, Advantages, and Disadvantages. *Test Automation Tools* [online]. November 20th, 2023 [cit. 2025-11-29]. Dostupné z: https://testautomationtools.dev/playwright-overview/

[52] JOSHI, Twinkle. INTEGRATING PLAYWRIGHT WITH JENKINS FOR SCALABLE AUTOMATION TESTING IN CI/CD PIPELINES. *INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING AND TECHNOLOGY* [online]. IAEME Publication, 2025, 2025-7-5, **16**(4), 12-29 [cit. 2025-11-29]. ISSN 0976-6367. Dostupné z: doi:10.34218/ijcet_16_04_002

[53] SOUKOPOVÁ, Jana. Vícekriteriální metody hodnocení [online]. Brno: Masarykova univerzita, b.r. [cit. 2025-11-29]. Dostupné z: https://is.muni.cz/el/econ/jaro2013/MKV_VZVP/um/33149329/Studijni_text_metody_vicekriterialniho_rozhodovani.pdf

[54] KUBIŠOVÁ, Andrea. *OPERAČNÍ VÝZKUM*. JIHLAVA, 2014. Disertační práce. VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

[55] *Moodle* [online]. c2025 [cit. 2025-11-29]. Dostupné z: https://moodle.com/about/

[56] HOFFMAN, Douglas. *Cost Benefits Analysis of Test Automation* [online]. Software Quality Methods, 1999.
[57] *Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost* [online]. Shanghai, China: ACM, 2006 [cit. 2025-11-29]. Dostupné z: https://scispace.com/pdf/economic-perspectives-in-test-automation-balancing-automated-472vv47gz6.pdf
[58] SWETALI RAVINDRA PATIL a PROF. ABHAY REWATKAR. A Comparative Study on Manual and Automated Software Testing Techniques at Tata Advanced Systems Limited in Nagpur. *Journal of Neonatal Surgery* [online]. 2025, **14**(15), 1573- 1580 [cit. 2025-11-29]. Dostupné z: https://www.jneonatalsurg.com/
[59] The Evolution of Test Automation: From Selenium to Playwright. A Comparison of Automation Tools: Selenium vs. Playwright vs. Cypress. *International Journal of Computer (IJC)* [online]. 2025, **54**(1), 37-45 [cit. 2025-11-29]. ISSN 2307-4523. Dostupné z: https://ijcjournal.org/index.php/InternationalJournalOfComputer/index
[60] MOŃ, Michał a Beata PAŃCZYK. A comparative analysis of web application test automation tools. *Journal of Computer Sciences Institute* [online]. Politechnika Lubelska, 2025, 2025-6-30, **35**, 159-165 [cit. 2025-11-29]. ISSN 2544-0764. Dostupné z: doi:10.35784/jcsi.7119

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam grafů

Přílohy

Součástí bakalářské práce je elektronická příloha ve formátu ZIP, která obsahuje celý projekt vytvořený v prostředí IntelliJ. Jednotlivé testovací scénáře jsou uloženy v následujících místech:

- Selenium – *AutomatedTestsMoodle/src/test/java/seleniumtest*
- Cypress – *AutomatedTestsMoodle/Cypress-test/cypress/e2e*
- Playwright – *AutomatedTestsMoodle/src/test/java/playwrighttest*