

Kontinuální testování provozovaných aplikací

Bc. Tomáš Pospíšil



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Tomáš Pospíšil
Osobní číslo:	A19530
Studijní program:	N3902 Inženýrská informatika
Studijní obor:	Informační technologie
Forma studia:	Kombinovaná
Téma práce:	Kontinuální testování provozovaných aplikací
Téma práce anglicky:	Continuous Testing of Deployed Applications

Zásady pro vypracování

1. Proved'te literární rešerši na dané téma.
2. Shromážděte požadavky na řešení s důrazem na testování webových aplikací a procesu přihlašování.
3. Navrhněte technické řešení.
4. Zdůvodněte výběr jednotlivých komponentů technického řešení.
5. Realizujte a otestujte výsledné technické řešení ve spolupráci s uživatelem.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. HUMBLE, Jez a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River: Addison-Wesley, [2011]. ISBN 978-0-321-60191-9.
2. DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk. Upper Saddle River: Addison-Wesley, c2007. The Addison-Wesley signature series. ISBN 978-0-321-33638-5.
3. CRISPIN, Lisa a Janet GREGORY. Agile testing: a practical guide for testers and agile teams. Upper Saddle River: Addison-Wesley, 2009. The Addison-Wesley signature series. ISBN 978-0-321-53446-0.
4. ARIOLA, Wayne a Cynthia DUNLOP. Continuous Testing. Scotts Valley, CA: CreateSpace, 2014. ISBN 978-1494859756.
5. HEROUT, Pavel. Testování pro programátory. České Budějovice: Kopp, 2016. ISBN 978-80-7232-481-1.
6. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.
7. COCCHIARO, Carl. Selenium framework design in data-driven testing: build data-driven test frameworks using Selenium WebDriver, AppiumDriver, Java, and TestNG. Birmingham, England: Packt Publishing, 2018. ISBN 978-1-78847-173-2. Dostupné také z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=5254598>

Vedoucí diplomové práce:

Ing. Tomáš Kadavý

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. prosince 2021**

Termín odevzdání diplomové práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan



prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 17.05.2022

Tomáš Pospíšil, v.r.

.....

podpis studenta

ABSTRAKT

V rámci diplomové práce byla vytvořena aplikace pro zajištění kontinuálního testování provozovaných služeb poskytující elektronické zdroje. Práce byla tvořena ve spolupráci s Knihovnou Univerzity Tomáše Bati ve Zlíně, která spravuje portál se seznamem elektronických zdrojů. Pro automatizaci byl vybrán nástroj Playwright. Automaty verifikují v grafickém uživatelském rozhraní proces přihlašování ve webovém prohlížeči do jednotlivých služeb. Cílem testů je ověřit dostupnost zdrojů a hlásit případné problémy správcům portálu e-zdroje. Testovací sada automatů je pravidelně spouštěna po určitých intervalech. Po skončení testů jsou výsledky poslány elektronickou poštou.

Klíčová slova: Testování, Automatické testování, Kontinuální testování, Playwright, E-zdroje

ABSTRACT

In this thesis, an application was created to ensure continuous testing of operated services providing electronic resources. The application was created in cooperation with the Library of Tomas Bata University in Zlín, which manages the portal with a list of electronic resources. The Playwright tool has been selected for automation. In the graphical user interface, tests verify the process of logging in to individual services in a web browser. The purpose of the tests is to verify the availability of resources and report any problems to the administrators of the e-resources portal. Testing methods are run regularly at certain intervals. After finishing all tests, the results are sent by e-mail.

Keywords: Testing, Automation testing, Continuous testing, Playwright, E-resources

Poděkování patří vedoucímu Ing. Tomáši Kadavému za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnoval.

Chtěl bych také poděkovat Ing. Ivanovi Masárovi z Knihovny UTB za technické konzultace a podporu při vývoji testovací aplikace.

OBSAH

ÚVOD	12
I TEORETICKÁ ČÁST	13
1 TESTOVÁNÍ.....	14
1.1 DŮVODY TESTOVÁNÍ.....	14
1.2 TYPY TESTOVÁNÍ	15
1.2.1 Funkcionální testování.....	15
1.2.2 Ne-funkcionální testování.....	16
1.3 EVOLUCE TESTOVÁNÍ.....	16
1.4 VÝVOJOVÝ PROCES	16
1.4.1 Vodopádový model a testování	18
1.4.2 Agilní vývoj a testování.....	18
1.5 KONTINUÁLNÍ INTEGRACE, NASAZENÍ A TESTOVÁNÍ.....	20
1.5.1 Kontinuální integrace	20
1.5.2 Kontinuální nasazení.....	20
1.5.3 Kontinuální testování	21
1.6 NÁSTROJE PRO CI/CD.....	22
1.6.1 Jenkins.....	22
1.6.2 Gitlab	23
1.6.3 CircleIO	23
1.6.4 Shrnutí.....	23
2 MANUÁLNÍ TESTOVÁNÍ.....	24
2.1 POSTUP MANUÁLNÍHO TESTOVÁNÍ	24
2.2 VÝHODY	24
2.3 NEVÝHODY	24
2.4 WHITE BOX	25
2.4.1 Účel testování	25
2.4.2 Výhody	25
2.4.3 Nevýhody	25
2.5 BLACK BOX.....	26
2.5.1 Postup Black box testování.....	26
2.5.2 Výhody	26
2.5.3 Nevýhody	26
2.5.4 Porovnání white box a black box testování	27
2.6 GRAY BOX	27

2.6.1	Výhody	27
2.6.2	Nevýhody	28
2.7	EXPLORATIVNÍ TESTOVÁNÍ	28
2.7.1	Průzkum vs Automatizace	28
2.7.2	Výhody	28
2.7.3	Nevýhody	29
2.8	AKCEPTAČNÍ TESTOVÁNÍ	29
2.8.1	Důvody testování	29
2.8.2	Prerekvizity akceptačního testování	29
2.8.3	Výhody	29
2.8.4	Nevýhody	30
3	AUTOMATICKÉ TESTOVÁNÍ	31
3.1	DŮVODY AUTOMATIZACE	31
3.2	ROZDÍLY MEZI MANUÁLNÍM A AUTOMATICKÉM TESTOVÁNÍ	31
3.3	ŽIVOTNÍ CYKLUS AUTOMATIZACE	32
3.3.1	Definování rozsahu automatizace	32
3.3.2	Výběr testovacího nástroje	32
3.3.3	Plánování, návrh a strategie	33
3.3.4	Příprava prostředí	33
3.3.5	Vývoj a spuštění testu	33
3.3.6	Výsledky testu	34
3.3.7	Údržba automatů	34
3.4	VÝHODY	34
3.5	NEVÝHODY	35
3.6	TESTOVACÍ PYRAMIDA	35
3.7	UNIT TESTY	36
3.7.1	Důležitost unit testů	36
3.7.2	Výhody	36
3.7.3	Nevýhody	36
3.7.4	Test Driven Development - TDD	37
3.7.5	JUnit	38
3.7.6	NUnit	38
3.8	INTEGRAČNÍ TESTOVÁNÍ	39
3.8.1	Důležitost integračních testů	39
3.8.2	Rozdíly mezi integračním a systémovým testováním	39
3.8.3	Výhody	39

3.8.4	Výzvy	40
3.8.5	Přístupy integračního testování	41
3.8.6	Nástroje	42
3.9	SYSTÉMOVÉ TESTOVÁNÍ	42
3.9.1	Důležitost testování	42
3.9.2	Typy systémového testování	43
3.9.3	Výhody	44
3.9.4	Nevýhody	44
3.9.5	Nástroje	44
II	PRAKTICKÁ ČÁST	45
4	TESTOVACÍ APLIKACE	46
4.1	ELEKTRONICKÉ ZDROJE	46
4.1.1	Přístup přes Shibboleth	47
4.1.2	WAYF	47
4.1.3	WAYFless	47
4.1.4	Proxy	47
4.1.5	VPN	48
4.2	DŮVOD TESTOVÁNÍ	48
4.3	PŘÍNOS	48
5	ŘEŠENÍ	50
5.1	POŽADAVKY NA ŘEŠENÍ	50
5.2	SELENIUM	50
5.2.1	Selenium IDE	51
5.2.2	Selenium RC	51
5.2.3	Selenium Grid	51
5.2.4	Selenium WebDriver	51
5.2.5	Výhody	51
5.2.6	Limitace	52
5.3	CYPRESS	52
5.3.1	Výhody	53
5.3.2	Limitace	53
5.4	PLAYWRIGHT	53
5.4.1	Podporující nástroje	54
5.4.2	Výhody	54
5.4.3	Limitace	55
5.5	VÝBĚR ŘEŠENÍ	55

6	POUŽITÉ TECHNOLOGIE	57
6.1	PLAYWRIGHT	57
6.2	PYTEST PLUGIN	57
6.3	PYTEST-XDIST PLUGIN	57
6.4	FLAKY PLUGIN	58
6.5	YAGMAIL PLUGIN	58
6.6	PYTEST-HTML PLUGIN	58
6.7	GIT	59
7	UŽIVATELSKÁ PŘÍRUČKA	60
7.1	KONFIGURACE PROSTŘEDÍ	60
7.1.1	Python	60
7.1.2	Knihovny jazyka Python	60
7.1.3	Konfigurační soubor config.ini	60
7.2	SPOUŠTĚNÍ TESTŮ	61
7.3	PARAMETRY SPOUŠTĚNÍ A PYTEST.INI	62
7.3.1	Parametry pro pytest	62
7.3.2	Parametr pro flaky	63
7.3.3	Parametr pro xdist	63
7.4	VÝSLEDKY TESTŮ A REPORT_SENT.TXT	63
8	PROGRAMÁTORSKÁ PŘÍRUČKA	64
8.1	NÁVRHOVÝ VZOR PAGE OBJECT MODEL	64
8.1.1	Důvod použití	65
8.2	INFRASTRUKTURA PROJEKTU	65
8.3	SLOŽKA HELPERS	66
8.3.1	Asserts.py	66
8.3.2	Data.py	66
8.3.3	Utils.py	66
8.4	SLOŽKA PAGES	67
8.4.1	LoginUtbPage.py	67
8.4.2	ProxyPage.py	67
8.4.3	ClarivatePage.py	68
8.5	SLOŽKA TESTS	69
8.5.1	TestBase.py	69
8.5.2	Automatický test	70
8.6	JAK ROZŠÍŘIT TESTOVACÍ APLIKACI O DALŠÍ TESTY	70

8.6.1	Vytvoření souboru	70
8.6.2	Vytvoření třídy s metodou setup()	71
8.6.3	Vytvoření testovacích metod	71
8.6.4	Vytvoření metody pro kontrolu přihlášení.....	72
8.7	GENERÁTOR TESTOVACÍCH PŘÍKAZŮ	74
ZÁVĚR.....		75
SEZNAM POUŽITÉ LITERATURY		76
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK		82
SEZNAM OBRÁZKŮ		83
SEZNAM PŘÍLOH		84

ÚVOD

Při vývoji softwarových aplikací se v dnešní době stále více dbá na testování produktu před vydáním aplikace k zákazníkovi. I když testování vyžaduje další finanční a časové prostředky ve vývoji, v průběhu let se ověřilo, že kontrola produktu vede k lepší jeho kvalitě a k větší spokojenosti klienta. Testování bylo nějakou formou vždy součástí vývoje aplikace. Začínalo se jednoduchým laděním programu při implementaci zdrojového kódu. Později testování bylo považováno jako individuální aktivita, která byla zařazena do vývojového cyklu aplikace. Verifikace správného chování aplikace spíše probíhala manuálně a nebyl takový tlak na automatizaci. Až později s příchodem různých podpůrných nástrojů pro automatizaci, začalo být vnímáno manuální testování jako značně neefektivní a výrazně časově náročné. Přestože k vytváření a k údržbě automatických testů je vyžadována lepší znalost a dovednost v programování, v konečném důsledku automatizace šetří čas a další prostředky. Automatické testy hrají důležitou roli v moderních přístupech při vývoji aplikace. Cílem je dodat zákazníkovi co nejrychleji novou verzi produktu v co nejlepší kvalitě. Tento způsob vývoje je označován jako kontinuální integrace a nasazení, kde automatické testy se používají pro co nejrychlejší kontrolu nové verze aplikace a získání zpětné vazby o kvalitě produktu.

Cílem této diplomové práce je vytvořit testovací aplikaci s automatickými testy, které budou kontinuálně testovat provozované služby poskytující elektronické zdroje. Aplikace bude vytvořena ve spolupráci s knihovnou Univerzity Tomáše Bati ve Zlíně, která poskytuje přístup k elektronickým zdrojům na portálu E-zdroje UTB. Automatické testy budou kontinuálně kontrolovat přihlašovací proces do jednotlivých elektronických zdrojů. Výsledkem bude zpětná vazba o úspěšnosti přihlášení a hlášení případných problémů s nedostupností služeb. To bude umožňovat správcům portálu e-zdroje rychle reagovat na případně problémy a také získají aktuální přehled o přístupnosti klíčových zdrojů.

Testovací sada automatů bude systémově testovat ve webovém prohlížeči proces přihlašování. Jedním z typů přihlášení do služby je přes technologii Shibboleth, která je založena na webovém workflow a přesměrovávání uživatele mezi stránkami. Aby bylo docíleno přesného procesu přihlašování v prohlížeči, je zapotřebí vytvořit automaty typu End-to-End. Automaty budou tedy testovat přihlášení v grafickém uživatelském rozhraní webového prohlížeče. Testovací sada s automaty bude pravidelně spouštěna a výsledky testů budou posílány elektronicky na email správce knihovny.

I. TEORETICKÁ ČÁST

1 TESTOVÁNÍ

Softwarové testování je proces vyhodnocování a ověřování, že softwarový produkt nebo aplikace funguje podle očekávání. Umožňuje předcházet chybám a problémům v aplikaci zákazníka. Různou sadou testovacích technik jsme schopni docílit správného fungování aplikace. Hlavní záměr testování je objevovat chyby v softwarovém produktu za pomoci testovacích scénářů, aby bylo možné je včas opravit. Díky verifikaci můžeme odhalit velké množství chyb, ale není možné najít všechny problémy v aplikaci. Produkt není možné otestovat za všech možných podmínek. Ovšem dokážeme zaručit, že aplikace funguje za daných podmínek, které jsme schopni nasimulovat [1].

1.1 Důvody testování

Každý zákazník používající software očekává, že aplikace bude fungovat bez chyb a podle dokumentace. Pokud zákazník v aplikaci nalézá chyby, nesprávné chování ho blokuje ve své práci a tím přichází o čas. To vede k frustraci a může to skončit nepoužíváním naší aplikace a preferováním produktu od konkurence. Některé chyby v softwaru mohou způsobit vážné škody jak na majetku, tak i na osobě [2].

Při pohledu do historie můžeme zjistit několik incidentů spojené s chybami a problémy v softwarovém produktu. Chyby způsobily nejen škody na majetku, ale mnohdy stály také i životy lidí. Testováním máme větší šanci tyto problémy najít a včas opravit. V roce 1994 26. dubna havarovalo letadlo společnosti Airbus A300 čínských aerolinií, při kterém zahynulo 264 lidí. Chyba byla jak na straně pilotů, tak ale i v palubním počítači. Letadlo se blížilo k letišti Nagoya a byl zapnut autopilot. Piloti se snažili letadlo ovládat manuálně v domněnku, že autopilot se vypne. Jak se snažili tlačit letadlo dolů k přistání, autopilot kompenzoval pohyb dolů a letadlo podle palubního počítače směřovalo směrem vzhůru. V kokpitu nebyla žádná indikace použití autopilota a čím víc piloti tlačili letadlo dolů, tím více počítač směřoval letadlo nahoru. To vedlo k pádu letadla [3]. Jednoduchým testovacím scénářem se dalo zabránit této tragédii. Pokud je zapnutý autopilot a pilot chce provést manuální vstup, je potřeba autopilota vypnout.

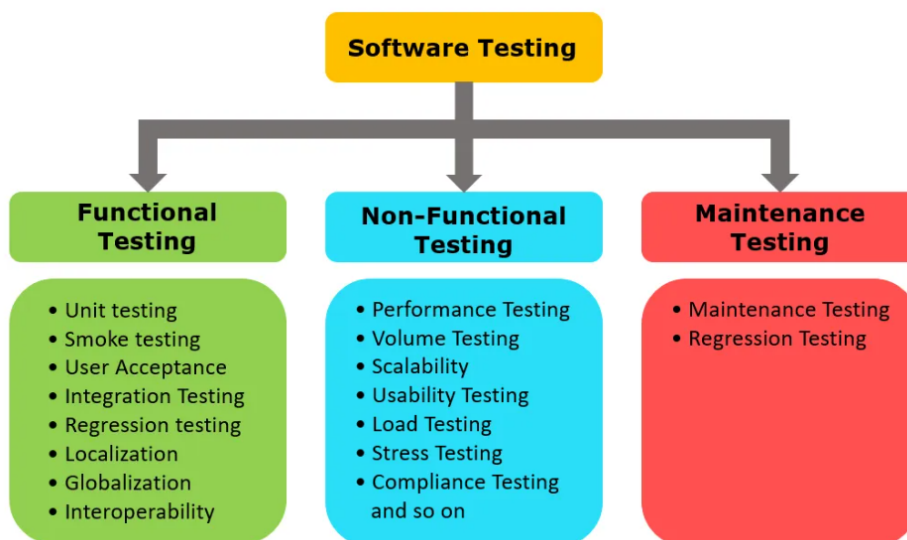
Jedna z nejznámějších chyb z vesmírného prostředí stála společnost NASA několik milionů amerických dolarů. Problém byl v nastavení vesmírné rakety, kdy softwarový inženýři se spletli v při čtení jednotek síly. V mezinárodní soustavě fyzikálních veličin je jednotka pro sílu Newton. Kdežto v anglické soustavě veličin je jednotka pro sílu Libra (lbf). To vedlo k nepatrným chybným výpočtům kurzu kosmické lodi, které se postupem času zvětšovaly [4].

Samotné testování ve vývojovém cyklu aplikace stojí čas, ale taky značnou část finančních prostředků. Pokud jsou však nastaveny správné testovací procesy a pravidla,

testování dokáže ušetřit ve vývoji spoustu času a peněz. Testování, před vydáním verze na trh, dokáže objevit mnoho chyb a problémů. Vývojové týmy mohou včas tyto chyby a problémy opravit ještě fázi vývoje softwaru. Zákazník dostane verzi aplikace, ve které by měly být vyřešeny problémy týkající se designu, bezpečnosti, škálovatelnosti a celkově problémy spojené s obecným fungováním produktu. Testování zlepšuje spolehlivost a kvalitu produktu. Systém splňující očekávání zákazníka vede k vyšší oblíbenosti a celkově lepšího použití vyvíjené aplikace [2].

1.2 Typy testování

Softwarové testování je obecně klasifikováno do třech kategorií. První kategorie jsou funkcionální testy, kde se zahrnuje testování funkcionality v aplikaci. Do druhé kategorie ne-funkcionálních testů patří testování výkonu, bezpečnosti, použití a tak dále. Jde o testování, které se zaměřuje spíše na celkové použití aplikace než na její funkce. Doplňková třetí kategorie testů je spíše pro údržbu, kde řadíme regresní testování. Regresní testování je o spuštění opět funkcionálních a ne-funkcionálních testů pro verifikaci aplikace před vydáním nové verze [5]. Grafické rozdělení testů i s příklady můžeme vidět na obrázku 1.1.



Obrázek 1.1 Rozdělení testování [6]

1.2.1 Funkcionální testování

Funkcionální testování je typ testování softwaru, který ověřuje softwarový systém podle funkčních požadavků anebo specifikací. Účelem funkčních testů je otestovat každou funkci softwarové aplikace poskytnutím vstupu a verifikace výstupu podle funkčních požadavků. Testování lze provádět ručně anebo pomocí automatických testů. Jako příklad

zde můžeme zařadit unit testování, integrační testování, systémové testování, Blackbox testování a další [7]. Jednotlivé typy testování budou uvedeny v kapitole 2.

1.2.2 Ne-funkcionální testování

Nefunkcionální testování je testování nefunkčních aspektů aplikace, jako je výkon, spolehlivost, použitelnost, bezpečnost a další. Z pohledu životního cyklu vývoje se nejdříve provádí funkcionální testování a až poté nefunkcionální testování. Funkcionální testování zajišťuje správnou funkčnost aplikace. Nefunkcionálním testováním jsme schopni vylepšit kvalitu, spolehlivost a rychlost aplikace. Zaměřuje se spíše na vylepšování uživatelského zážitku, aby například program pracoval rychle. Nefunkcionální testy je obtížné a pracné provádět manuálně, testování probíhá pomocí různých nástrojů (JMeter, Loadrunner, vPerformer). Můžeme zde zařadit testování ohledně výkonu, bezpečnosti, zátěžové testy, testy kompatibility, testování odolnosti a další [7].

1.3 Evoluce testování

Programování softwaru se v průběhu dekad vyvíjelo a v důsledku toho prošlo řadou změn i testování, které je nedílnou součástí vývoje softwaru. Vše začalo s programováním a následným laděním programu. Hledání chyb ve fázi ladění bylo považováno za formu testování. V roce 1957 testování dostalo konkrétní označení pro samostatnou aktivitu, která už byla oddělena od fáze ladění v programování. Do pozdějších 70. let testování představovalo aktivitu, která měla zaručit funkčnost softwaru podle specifických požadavků. Až poté byla aktivita rozšířena o hledání chyb v aplikaci v průběhu vývoje. V 80. letech bylo testování už považováno za proces k docílení lepší kvality aplikace a o pár let později byla fáze testování zařazena do vývojového procesu produktu [8].

1.4 Vývojový proces

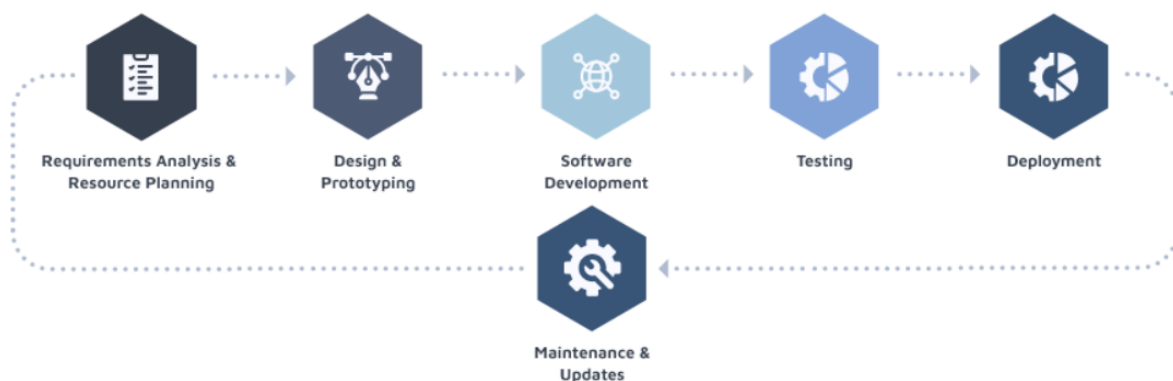
Proces vývoje softwaru je rozdělení vývoje produktu na jednotlivé fáze, které jsou sekvencně vykonávány. Grafické znázornění s jednotlivými fázemi je zobrazeno na obrázku 1.2. V první fázi vývoje je potřeba získat požadavky na vyvíjený software a provést jejich analýzu. Zahrnujeme zde i plánování, jestli dané požadavky jsme schopni splnit v požadovaný čas. Jakmile je první fáze splněna, dostaneme se do druhé fáze životního cyklu. Ta zahrnuje vytváření prototypů a designování nově vznikající funkcionality. Grafici a UX¹⁾ experti podle požadavků navrhnu chování aplikace a vytvoří adekvátní uživatelské rozhraní. Dokončením druhé fáze životní cyklus pokračuje do fáze vývoje. Je

¹⁾UX je označení pro skupinu lidí, kteří se snaží navrhnout takové uživatelské rozhraní, které bude pro uživatele jednoduché a intuitivní a poskytne příjemný uživatelský zážitek [11].

to proces, kdy vývojové týmy začnou pracovat na implementaci zadaných požadavků. Programují aplikaci pomocí zdrojového kódu tak, jak bylo navrženo v předchozích fázích [9].

Další fází je část testovací, ke které dochází po naprogramování zadaných požadavků vývojovými týmy. Týmy zajišťující kvalitu ověřují, jestli vytvořená aplikace splňuje zadané požadavky. Verifikují funkčnost a stabilitu produktu. Pomocí manuálního a automatického testování nalézají chyby a problémy, které se snaží vývojové týmy opravit ještě před dodáním softwaru k zákazníkovi. Jakmile týmy zajišťující kvalitu ověří, že vyvíjená funkcionality funguje správně podle zadaných požadavků, můžeme přejít do další části životního cyklu aplikace. Nejdůležitější částí při vývoji je nasazení nové funkcionality do produkčního prostředí zákazníka. Uživatelé mají možnost vyzkoušet si vytvořenou aplikaci v praxi a poskytnout zpětnou vazbu na její fungování. Poslední fází při vývoji je údržba produktu a případné vydání nové verze aplikace. Poslední část vývoje neznamena, že proces životního cyklu je ukončen. Uživatel může změnit své požadavky na fungování softwaru, nalézt chyby a problémy. Vývojové týmy by měly být schopni reagovat na dané změny a tím se životní cyklus opět přesouvá do první fáze [9].

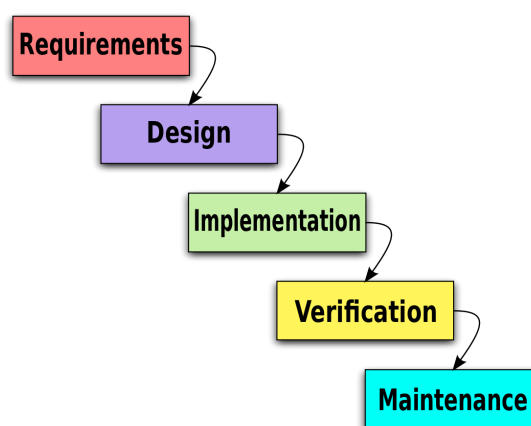
V praxi existuje mnoho konkrétních, modelů jak přistupovat k vývoji softwaru. Mezi ty nejznámější můžeme zařadit vodopádový model, inkrementální, prototypový a spirálový model. Modely se liší fázemi, mají různá pravidla pro ukončení a zahájení nové fáze. Některé kladou důraz na častější komunikaci se zákazníkem, jiné zpočátku vytvoří pouze prototyp, který v průběhu cyklů neustále vylepšují. Ovšem všechny modely mají jeden konkrétní cíl, a to vytvořit kvalitní a spolehlivý produkt.



Obrázek 1.2 Životní cyklus vývoje aplikace [9]

1.4.1 Vodopádový model a testování

Vodopádový model je jednou z nejstarších a nejtradičnějších metodologií vývoje softwaru. Vodopádový přístup se také nazývá lineární sekvenční model nebo klasický model životního cyklu. Fáze v modelu jsou stejné, jako v teoretickém modelu popsáném v kapitole 1.4. Hlavní myšlenka modelu je sekvenční vykonávání fází v procesu vývoje. Abychom mohli postoupit ve vývoji z jedné fáze do následující, všechny předchozí fáze musí být kompletně splněny [9]. Grafické znázornění modelu můžeme vidět na obrázku 1.3.



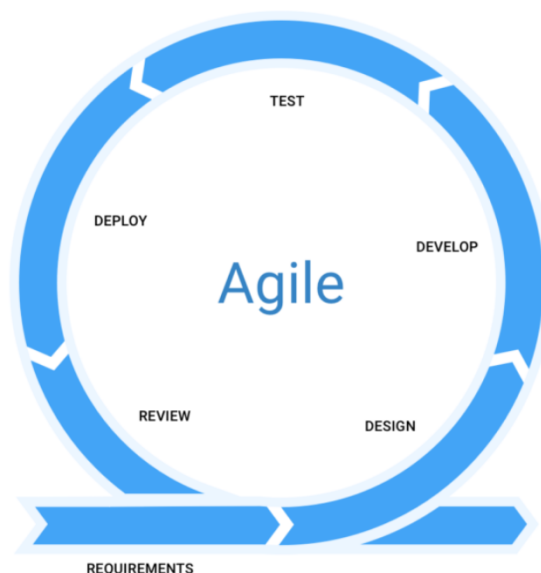
Obrázek 1.3 Vodopádový model [10]

Vývoj a testování ve vodopádovém modelu jsou oddělené aktivity. Jakmile je software z pohledu vývojáře hotový, takový software je předán testovacím týmům k verifikaci. Nevýhoda modelu spočívá v tom, že testeři nejsou aktivně zapojeni při sběru a analýze požadavků. Nemají informace o tom, jak zákazník bude software používat a chybí nadhled nad danou funkcionalitou. Testovací scénáře a plány tvoří na základě informací získaných z dokumentace a od vývojářů, kteří implementovali danou funkcionalitu. To vedlo k určitému omezení a testování nebylo komplexní [9].

1.4.2 Agilní vývoj a testování

Agilní přístup je jeden z nejpopulárnějších metodik vývoje software. Hlavní výhoda agilního vývoje je vysoká přizpůsobitelnost okolním požadavkům. Vývoj je flexibilní, dynamický a iterativní, což vede k lepší kvalitě produktu pro zákazníka. Vývoj aplikace je rozdělen na jeden nebo více sprintů. Sprint neboli iterace je označení pro časové období, které může většinou trvat od jednoho do čtyřech týdnů. Na konci každého sprintu by měla být dodána otestovaná a funkční verze aplikace [9]. Sprint pokrývá všechny fáze životního cyklu, které můžeme vidět na obrázku 1.4. Agilní vývoj podporuje i

kontinuální integraci, nasazení a testování, o kterých bude více pojednáno v kapitole 1.5.



Obrázek 1.4 Cyklus agilního vývoje [13]

Sprint začíná plánováním, ve kterém vývojové týmy vyberou z backlogu úkoly, které jsou schopni dodat v daném sprintu. Backlog je seznam úkolů, které bychom měli splnit v dané verzi aplikace. Úkoly většinou zadává produktový manažer. V backlogu jsou úkoly seřazeny podle priority a vývojové týmy vybírají do aktuálního sprintu nejprioritnější požadavky. Po vybrání požadavků do sprintu začnou designéři a UX experti pracovat na grafickém návrhu. Pokud je to možné, zpracují úkoly do vizuální podoby a předají návrhy vývojovému týmu. Vývojové týmy na základě akceptačních kritérií k danému úkolu implementují řešení pomocí zdrojového kódu. Musí brát i v potaz grafický návrh, jak by měli jednotlivé prvky požadavku vypadat v uživatelském rozhraní aplikace [9].

Testování v agilním přístupu probíhá častěji a zpravidla hned na začátku vývoje. Je to významný rozdíl oproti vodopádovému modelu, kde testéři musí čekat na kompletní řešení od vývojářů. Vývojáři a testéři pracují společně na daném úkolu a komunikace je nezbytnou součástí spolupráce. Verifikace funkcionality podle akceptačních kritérií probíhá v průběhu vývoje. Podle agilní metodologie nepíší testy pouze testéři. Vývojáři používají techniku při vývoji Test-Driven Development (TDD), což má za následek kontinuální testování implementovaného zdrojového kódu [9]. Technikou TDD se více zabývá kapitola 3.7.4.

Jakmile je funkcionality otestovaná, další fází je nasazení nové verze aplikace do produkčního prostředí. Zákazník je už schopen pracovat s novou funkcionalitou a poskytovat zpětnou vazbu. Vývojové týmy monitorují běh aplikace na produkčním prostředí

a jsou stále k dispozici, kdyby se zákazník setkal s chybou nebo jiným problémem [9].

Vydáním nové verze se dostáváme na konec sprintu. Zákazník na základě zkušenosti s naší aplikací může dostávat nové nápady s vylepšením produktu. Projektový manažeři by měli zaznamenávat jakoukoliv zpětnou vazbu od zákazníka a vytvářet nové požadavky na vývojové týmy. Tím se dostáváme opět k prvním fází agilního životního cyklu a tím je plánování [9].

1.5 Kontinuální integrace, nasazení a testování

Kontinuální integrace (CI) a kontinuální nasazení (CD) také známe pod zkratkou CI/CD. CI/CD je označení pro sadu procesů a pravidel, které jsou využívány pro spolehlivější a kvalitnější produkci softwarových aplikací. Je to také osvědčený postup, který se používá při agilním vývoji. Spojuje vývojové týmy zodpovědné za implementaci zdrojových kódů a týmy, které provádí nasazeních nových verzí aplikace na produkční prostředí zákazníka [14].

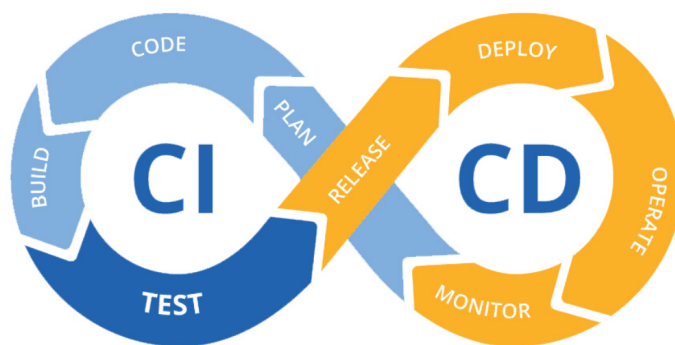
1.5.1 Kontinuální integrace

Kontinuální integrace je způsob, jakým se dá zajistit sloučení veškerých změn v projektu od jednotlivých vývojových týmu do jedné verze aplikace. Je to sada praktik a procesů, které podporují vývojáře k častějšímu dodávání malých implementací kódů do repozitáře²⁾ verzovacího systému. To vede k lepší spolupráci mezi vývojáři a ke zlepšení kvalitě kódu. V konečném důsledku i ke spolehlivější a kvalitnější aplikaci. Ve většině moderních aplikací je vývoj kódu podpořen pomocí různých platforem a nástrojů. Jako příklad můžeme uvést třeba verzovací nástroje Git nebo Team Foundation Version Control. CI takové nástroje sjednocuje a pomáhá ve verifikaci změn ve zdrojovém kódu. Pomocí automatizace a kontinuální integrace lze automaticky kompilovat kód, vytvářet balíčky a testovat kontinuálně aplikaci [14].

1.5.2 Kontinuální nasazení

Kontinuální nasazení (CD) navazuje na kontinuální integraci. Je to zautomatizovaný proces, kde jsou změny ve zdrojovém kódu automaticky zkompileovány, otestovány a připraveny pro vydání do produkce anebo na testovací prostředí. Vývojové týmy tedy mají k dispozici zkompileovanou aplikaci s jejich přidanou novou funkcionalitou, která prošla standardizovaným testovacím procesem. To vývojářům pomáhá důkladněji ověřovat aktualizace a preventivně odhalovat problémy [16]. Grafické znázornění CI/CD můžeme vidět na obrázku 1.5.

²⁾Repozitář je označení pro kontejner, kde ukládáme softwarový kód vyvíjené aplikace.



Obrázek 1.5 Kontinuální integrace a nasazení [15]

1.5.3 Kontinuální testování

Kontinuální testování je proces začlenění automatického testování do různých fází životního cyklu vývoje softwaru. Testování hraje důležitou roli v procesu CI/CD, díky kterému vývojáři dostanu ihned zpětnou vazbu na fungování jejich implementovanému požadavku [17].

Jedním z fundamentálních principů v moderním přístupu při vývoji aplikace je co nejrychleji dodat novou verzi produktu na produkční prostředí zákazníka, ale zároveň verze musí být v co nejlepší kvalitě a bez chyb. Manuální ověřování a dávání zpětné vazby vývoji v každé fázi životního cyklu je nevyhovující a neefektivní. Potřebuje spousty časových prostředků pro vykonání, tím se prodlužují fáze v životním cyklu a vydání nové verze se může opozdit [17].

Zautomatizováním manuálních testovacích procesů dokážeme zefektivnit a zrychlit ověřování nových změn v aplikaci. Poskytováním rychlé zpětné vazby v prvních fázích vývoje zlepšuje kvalitu produktu. Kontinuální testování funguje pomocí automatizovaných nástrojů, které se nahrají jako předdefinované skripty do každé fáze cyklu. Automaticky spouštěné skripty eliminují potřebu lidského zásahu při provádění automatických testů. Pokud automatický test selže, vývojové týmy jsou upozorněni na chybu. Díky tomu jsou schopni rychle zareagovat a provést nezbytné úpravy zdrojového kódu dříve, než to ovlivní další fázi vývoje. Po úspěšném projití všech automatických testů, je projekt automaticky posunut do další fáze životního cyklu [17].

Kontinuální testování poskytuje automatizované metody pro zajišťování kvality a pro poskytování zpětné vazby o funkčnosti produktu v každé fázi životního cyklu při vývoji. Vývojové týmy získávají vyhodnocením automatických testů užitečné informace, které pomáhají zlepšovat kompatibilitu a kvalitu zdrojového kódu ještě před nasazením do produkčního prostředí zákazníka [17].

Dnešní moderní vývojové architektury jsou komplexní a vícevrstvé. Kontinuální verifikace pomáhá vývojovým týmům rozdělit složité systémy na menší celky, které po-

mocí automatizovaných testovacích metod můžeme otestovat samostatně. To výrazně zlepšuje zjišťování chyb a zrychluje proces jejich oprav [17].

Pokročilé metody testování mohou simulovat různé scénáře a způsoby použití aplikace, díky kterým jsou vývojové týmy schopni sledovat chování produktu. Na základě chování ze simulací umožňují vývojářům odstranit z aplikace neefektivitu uživatelského rozhraní a vyhnout se nechtěným překvapením po nasazení produktu do produkce [17].

Ve velkých propojených systémech se může objevit chyba pouze v jednom modulu aplikace a způsobit dominový efekt. To může způsobit omezení v používání některých služeb a případně nedostupnost k aplikaci. Poskytovatelé Cloudu³⁾ se můžou setkat s chybou v systému na jednom konci, která může způsobit ochromení služby v celém regionu a vyřadit tak službu na několik hodin. To může být obzvláště ničující pro organizace, které jsou závislé na vysoké dostupnosti služeb. Kontinuální testování může odhalit tyto chyby hned na začátku na méně komplexnější úrovni. Organizace se tak může vyhnout přerušení svých služeb nebo dokonce nedostupnosti celé aplikace [17].

1.6 Nástroje pro CI/CD

Nástroje pro kontinuální integraci a nasazení mohou pomoci vývojovým týmům zautomatizovat jejich vývoj, testování a nasazení. Některé platformy se specializují na část integrační (CI), jiné nástroje jsou určeny pro kontinuální nasazování (CD) na produkční prostředí zákazníka a další jsou zaměřeny na zautomatizování testovacích procesů (CT) [20].

1.6.1 Jenkins

Jeden z kvalitních nástrojů pro automatizaci CI/CD, který je dostupný ve verzi open-source⁴⁾, je server Jenkins. Jenkins je navržen tak, aby zvládl cokoli od jednoduchého CI serveru až po kompletní proces CD. Jedná se o samostatný program založený na programovacím jazyku Java s balíčky pro operační systém Windows, MacOS a další operační systémy podobné Unixu. Se stovkami dostupných pluginů⁵⁾ Jenkins podporuje vytváření, nasazování a automatizaci při vývoji softwaru. Jenkins je jednoduchý na instalaci, má jednoduché a přívětivé uživatelské rozhraní, podporuje plánování automatizace a umožňuje upozorňovat o dokončeném běhu CI/CD [21].

³⁾Cloud je síť vzájemně propojených vzdálených serverů, které mají různé funkce, ale fungují jako jeden ekosystém. [18]

⁴⁾Open Source je software se zdrojovým kódem, který může kdokoli kontrolovat, upravovat a vylepšovat.[19]

⁵⁾Plugin je softwarový doplněk, který se instaluje do programu a rozšiřuje jeho možnosti.

1.6.2 Gitlab

Gitlab je sada nástrojů pro správu různých procesů v souvislosti s vývojem softwaru. Základním produktem je webové rozhraní pro Git repozitář s funkcemi pro sledování chyb, analýzu anebo Wiki. Gitlab umožňuje spouštět kompilace zdrojového kódu, spouštět testy a nasazovat aplikaci při každé změně v repozitáři. Všechny tyto funkce můžeme spouštět na virtuálních počítačích, v Docker kontejnerech anebo na jiných serverech. Pomáhá automatizovat a zkracovat vydávání a nasazování nových verzí aplikací k zákazníkovi [21].

1.6.3 CircleIO

Dalším známým nástroje pro CI/CD je platforma CircleIO, která umožňuje rychlý softwarový vývoj a publikaci nové verze aplikace. CircleIO podporuje automatizaci napříč všemi fázemi vývoje. Od kompilaci kódu, testování nové funkcionality až po nasazení nové verze softwaru do produkce. Stejně jako Jenkins, CircleIO integruje s různými nástroji pro správu zdrojového kódu jako je GitHub anebo Bitbucket. Automatická kompilace kódu probíhá v kontejnerech anebo na virtuálních zařízeních. Tudíž je možné kompilovat pod všemi operačními systémy. Umožňuje automatické pouštění testů, jejich paralelizaci a opět upozornění o výsledku běhu testů. Nástroj CircleIO poskytuje bezplatnou zkušební verzi aplikace na určitou časovou dobu [21].

1.6.4 Shrnutí

Mezi další populární služby pro CI/CD patří TeamCity, Buddy anebo Travis CI [21]. Existuje mnoho nástrojů, které podporují automatizaci vývojových procesů. Každý nástroj má svá specifika a způsob použití. Některé jsou open-source, jiné mají zkušební verze, ale pro další používání musíme zaplatit. To dává možnost vývojovým týmům použít takový nástroj, který nejvíce splňuje jejich požadavky.

2 MANUÁLNÍ TESTOVÁNÍ

Manuální testování je typ softwarového testování, ve kterém jsou jednotlivé testovací scénáře prováděny manuálně bez jakéhokoliv použití automatizovaného nástroje. Účelem ručního testování je identifikovat chyby, problémy a závady v softwarové aplikaci. Testováním lze dosáhnout takového stavu aplikace, aby byla bez majoritních chyb a fungovala podle specifikovaných funkčních požadavků. Manuální testování softwaru je nejprimitivnější technika ze všech typů testování, ale také pomáhá najít kritické chyby v aplikaci. Každá nová aplikace musí být ručně otestována, než bude možné její testování automatizovat. Koncepty ručního testování nevyžadují znalost žádného testovacího nástroje. I když manuální testování vyžaduje větší úsilí, tak v dnešní době CI/CD se tomu nedá vyhnout a je pořád důležité. Je možné zautomatizovat většinu funkcionality, ovšem nastanou různé testovací scénáře, které se zautomatizovat nedají. Na manuálním testováním by se měli podílet všichni, kteří jsou součástí vývoje. Manuální testování není výhradně pouze pro testery [22].

2.1 Postup manuálního testování

Ze všeho nejdříve se shromáždí veškeré dokumenty a informace potřebné k otestování implementované funkcionality. Tester zanalyzuje požadavky na testovanou funkcionalitu, aby pokryl všechny scénáře uvedené v zadání. Vytvoří testovací scénáře, podle kterých provede test. Pokud test objeví chybu, obrací se tester na vývojáře. Ti chybu zanalyzují a provedou její opravu. Po opravě chyby opět tester provede test podle testovacího scénáře. Pokud test splní očekávané výstupy, funkcionalita by měla být otestována [25] .

2.2 Výhody

Manuální testování nevyžaduje znalosti programování při použití metody Black box (viz kapitola 2.5). Používá se k testování dynamicky se měnícího grafického uživatelského rozhraní (animace). Tester pracuje se se softwarem jako skutečný uživatel, takže je schopen odhalit problémy s použitelností a uživatelským rozhraním. V případě malých projektů, manuální testování je výhodnější v poměru cena-výkon, oproti vytváření komplexnější infrastruktury k automatizaci. Manuální testování je rychlé na naučení [25].

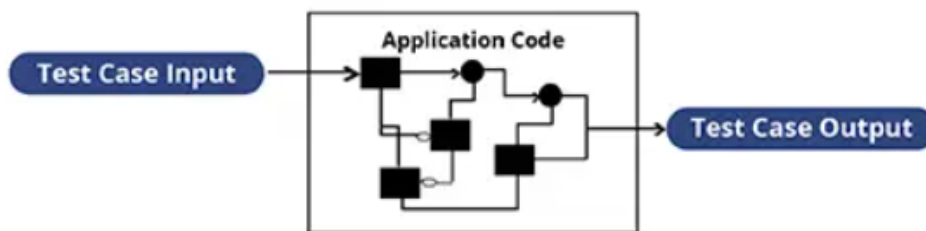
2.3 Nevýhody

Manuální testování v případě větších projektů a kontinuálního testování je časově náročné a nákladné na lidské zdroje. Testeři tvoří testovací scénáře na základě jejich

dovedností a zkušeností. To může vést k nepokrytí některých funkčních požadavků. Funkcionalita se v průběhu vývoje mění a tím pádem některé testovací scénáře se musí přizpůsobit a aktualizovat [25].

2.4 White box

White box testování je způsob testování softwaru, při kterém se testuje vnitřní infrastruktura aplikace a její zdrojový kód. Jde o funkcionální testování, kde se verifikuje funkčnost aplikace. Další názvy pro tento způsob jsou Clear box, Open box anebo Code-based testování. Hlavní myšlenkou je, že v průběhu testování je zdrojový kód aplikace viditelný pro testujícího. Testující může být v roli testera i vývojáře [23].



Obrázek 2.1 Schéma testování White box [24]

2.4.1 Účel testování

White Box testování zahrnuje testování zdrojového kódu, kdy funkce voláme se seznamem předdefinovaných vstupů proti očekávaným nebo požadovaným výstupům. Když konkrétní vstup nevede k očekávanému výstupu, je to chyba. Testují se objekty a příkazy na individuální bázi, ověřuje se funkčnost podmíněných smyček [23].

2.4.2 Výhody

Výhoda White box testování je obecně optimalizace a vylepšení zdrojového kódu. Testování je důkladnější, protože jsou obvykle pokryty všechny cesty v kódu. Provádění testů může začít už hned na začátku životního cyklu vývoje, jelikož nemusí být k dispozici grafické uživatelské prostředí [23].

2.4.3 Nevýhody

Mezi nevýhody testování se považuje složitost prováděných testů a s tím související větší náklady na čas. Testování vyžaduje znalost zdrojového kódu a pochopení implementace [23].

2.5 Black box

Black box testování je testovací technika, při které se testuje funkčnost softwarové aplikace bez znalosti zdrojového kódu, architektury nebo implementačních detailů. Primárním zdrojem testování jsou specifikace požadavků, které stanoví zákazník. Je to metoda, při které tester vybere funkci s předaným vstupem a testuje, jestli výstupní hodnota funkce odpovídá očekávaným požadavkům. Pokud funkce vrací správné hodnoty, je test úspěšný. V případě nalezení chyby je vývojový tým informován a měl by pracovat na její opravě [26].



Obrázek 2.2 Schéma testování Black box [26]

2.5.1 Postup Black box testování

Tester jako první musí shromáždit specifikované požadavky, které se vztahují k testovací funkcionalitě. Poté zanalyzuje požadavky a vytvoří testovací scénáře. Testovací scénáře jsou vytvořeny na základě vstupu a očekávaného výstupu. Scénáře by měly pokrývat jak pozitivní vstupy s platnými hodnotami, tak i nepříznivé vstupy s nevalidními hodnotami. Testování končí provedením všech testovacích scénářů a vyhodnocením jejich výsledků. Testovací scénáře, které vracely neočekávané výstupy, jsou předány vývojovým týmům pro další zkoumání a případnou opravu zdrojového kódu [26].

2.5.2 Výhody

Člověk provádějící Black box testování softwarové aplikace nemusí znát zdrojový kód ani implementační detaily o systému. Testy nejsou tak komplexní, protože testování probíhá z pohledu koncového uživatele [27].

2.5.3 Nevýhody

Vyžaduje stanovení priorit testování, ve velkých projektech je prakticky nemožné otestovat veškeré možné cesty v aplikaci. Nevýhodou s tím spojenou je určení, do jaké

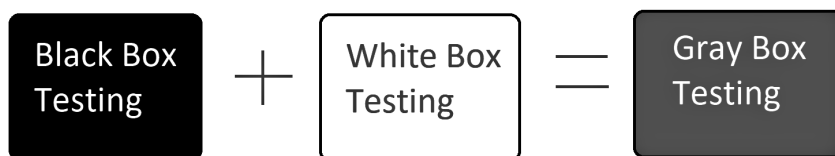
míry máme pokrytou aplikaci testy. Pokud test selže, může být časově náročné přijít na příčinu [27].

2.5.4 Porovnání white box a black box testování

Testování pomocí techniky black box usnadňuje testování komunikace mezi jednotlivými moduly v aplikaci. Oproti tomu white box testuje pouze konkrétní prvky v každém jednotlivém modulu. Black box testování poskytuje abstrakci od kódu a zaměřuje se na testování chování v aplikaci. To nám umožňuje nacházet různé cesty, kterými by se mohl vydata uživatel. U techniky white box jde o pochopení konkrétní implementace a je nutná znalost programování. Tester nemá nadhled nad danou funkcionalitu v aplikaci a je zaměřen pouze na implementační místo ve zdrojovém kódu. White box tedy testuje vnitřní strukturu aplikace, black box je zaměřen na verifikaci specifikovaných požadavků více z pohledu zákazníka. Obě techniky mají své výhody a nevýhody. Pokud software budeme testovat technikou white box i black box, jsme schopni dosáhnout lepší kvality a spolehlivosti aplikace [26].

2.6 Gray box

Gray box testování je technika testování softwarové aplikace, kdy tester má částečné znalosti o infrastruktuře a zdrojovém kódu testované aplikace. Gray box je softwarová testovací metoda, která kombinuje white box a black box testování. Účelem testování technikou gray box je vyhledat a identifikovat defekty způsobené nesprávnou strukturou kódu nebo nesprávným používáním aplikace. Ve vývoji dává Gray Box testování možnost testovat obě strany aplikace, prezentační vrstvu i kódovou část. Je primárně užitečný při integračním testování a penetračním testování [28].



Obrázek 2.3 Schéma testování Gray box [29]

2.6.1 Výhody

Testeři provádějící testování techniky gray box nemusí mít veškeré znalosti programovacího jazyka a infrastruktury aplikace jako je tomu u white boxu. Vyhovuje mít pouze částečnou znalost [29].

2.6.2 Nevýhody

Gray box testování není vhodný pro testování algoritmů. Metoda nemá kompletní přístup ke zdrojovému kódu, což může v některých případech k omezenému testování. Některé testovací scénáře je obtížné navrhnout [29].

2.7 Explorativní testování

Explorativní neboli průzkumné testování je metoda, kdy se testování aplikace netestuje podle napsaných testovacích scénářů, ale verifikace probíhá náhodně s cílem nalézt co největší počet chyb. Testování je zaměřeno spíše na přemýšlení testera, co ho zrovna napadne vyzkoušet v aplikaci. Může jít o jednoduché operace až po ty složitější, které je třeba nákladné na automatizaci, případně náročné vytvářet testovací scénáře. Průzkumné testování je široce používáno v agilním přístupu (1.4.2) vývoje softwaru. Klade důraz na osobní svobodu a odpovědnost jednotlivého testera. V rámci automatizovaného testování se nejdříve navrhne testovací scénář a později se přistoupí k automatizaci takového scénáře. Explorativní testování je simultánní proces, kde probíhá návrh testu a provedení testu současně [30].

2.7.1 Průzkum vs Automatizace

Přestože současným trendem v testování softwaru je tlak na automatizaci, explorativní testování je způsob jiného přemýšlení nad přístupem k aplikaci. Automaty fungují podle předem napsaných instrukcí, které na sebe musí navazovat. Kdežto explorativní testování je spontánní průzkum aplikace, kde se testování řídí kognitivními dovednostmi testujícího. Kroky v automatu jsou napsány tak, aby splňovaly specifikované požadavky vytvořené projektovými manažery případně zákazníkem. Průzkumným testováním můžeme nalézt jiné uživatelské cesty v aplikaci, které třeba v automatech nemusí být pokryty nebo byly nákladné na automatizaci. I automatizace má své limity, jelikož ne vše se dá zautomatizovat [30].

2.7.2 Výhody

Průzkumné testování je užitečné, když nejsou k dispozici specifikované požadavky na funkcionalitu v aplikaci. Umožňuje více proces vyšetřování, které pomáhá k nalezení více chyb než testování podle testovacích scénářů. Odhaluje chyby, které by mohly být jinými testovacími technikami ignorovány. Může tedy pokrývat různé případy a scénáře. Explorativní testování spoléhá na intuici testerů a pomáhá rozšířit znalost o produktu, což může zvýšit produktivitu. Tester může provést testování na základě svých nápadů a své kreativity [30].

2.7.3 Nevýhody

Explorativní testování není vhodné provádět delší dobu, je důležité si vždy určit časovou dobu. Testující mohou průběhu ztrácet motivaci. Testování je technika na základě zkušeností testera, záleží na jeho intuici a dovednostech [30].

2.8 Akceptační testování

Akceptační testování je typ testování prováděné koncovým uživatelem anebo zákazníkem za účelem ověření softwarové aplikace ještě před nasazením do produkčního prostředí. Testování probíhá v závěrečné fázi životního cyklu aplikace po provedení funkčního, integračního a systémového testování. Je testováno stylem Black box, kdy uživatel nezná zdrojový kód ani infrastrukturu aplikace. Koncový uživatel se soustředí na takové funkce a scénáře v aplikaci, které zákazník běžně používá [31].

2.8.1 Důvody testování

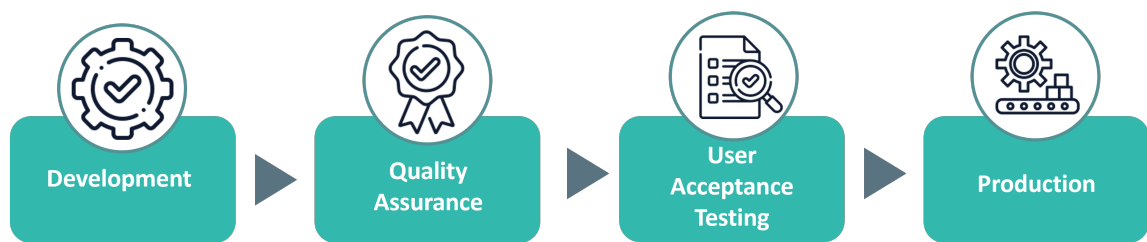
I když je akceptační testování až v závěrečné fázi vývoje po všech jiných testování, zákazník může najít chyby a problémy, které se by se měly opravit. Pokud se prováděly změny v požadavcích během vývoje, k chybám může docházet díky špatné komunikaci mezi vývojovým týmem a zadavatelem požadavků na produkt. Vývojáři vytváří funkce podle specifikovaných požadavků, ovšem může docházet k jinému pochopení požadavků než zamýšlel zadavatel. Akceptační testování je tedy nutné pro ověření požadavků koncovým uživatelem v aplikaci, aby se potvrdila jejich správná implementace v konečné fázi produktu [32].

2.8.2 Prerekvizity akceptačního testování

Předtím než se zahájí akceptační testování, musí být splněno jednotkové testování, integrační a systémové testování. Vývojáři by měli mít implementované všechny specifikované požadavky pomocí zdrojového kódu. Aplikace by měla být bez majoritních chyb a zkontrolována pomocí regresního testování. Povolené jsou pouze chyby, které se týkají kosmetických částí aplikace. A v neposlední řadě by mělo být připraveno prostředí, na kterém bude zákazník nebo uživatel provádět akceptační testování [31].

2.8.3 Výhody

Akceptační testování zvyšuje spokojenost samotných klientů při jejich testování aplikace, jelikož už si sami můžou vyzkoušet fungování produktu. Do budoucna zlepšuje definice požadavků, protože testuje požadavky podle svých potřeb. Informace shromáždě-



Obrázek 2.4 Zařazení akceptačního testování do vývoje [33]

děné díky akceptačnímu testování podporují k lepšímu pochopení požadavků cílového zákazníka. Obecně zlepšuje kvalitu produktu jako je tomu u každého testování [32].

2.8.4 Nevýhody

Aby se akceptační testování mohlo provést je zapotřebí, aby zákazník sepsal své akceptační požadavky na funkcionalitu v aplikaci. To vede k tomu, že zákazník může sepsat požadavky špatně anebo vůbec [32].

3 AUTOMATICKÉ TESTOVÁNÍ

Automatické testování je softwarová technika, která provádí testování pomocí automatických testovacích skriptů, které dokáží fungovat bez jakéhokoliv lidského zásahu. Automatický skript nebo také automat je naprogramován tak, aby prováděl jednotlivé kroky z testovacího scénáře a porovnával aktuální výsledek s očekávaným výsledkem v daném kroku. Pomocí automatizačního nástroje je možné testovací sadu složenou z automatů spouštět opakovaně. To přináší jistou výhodu při vývoji softwaru, jelikož při testování specifikovaných požadavků se můžou tvořit opakující se kroky v testovacích scénářích. Jakmile je jednou automat napsán, už nepotřebují žádný další lidský zásah a tester se může věnovat jiné činnosti [34].

Cílem automatizace je co nejvíce snížit počet manuálních testovacích scénářů. Manuální testovací scénáře, které jsou vybrány k automatizaci, by měly obsahovat často se opakující testovací kroky. Automaty by také měly testovat funkcionality, která je pro náš produkt důležitá a může být zákazníkem často používána. Celkově je přínosem zautomatizovat takové testovací scénáře, které jsou časově náročné pro ruční testování [34].

U testovacích scénářů, které pokrývají často se měnící specifikované požadavky není vhodná automatizace. Automatizace není přínosná ani u testovacích scénářů, které se provedou jen jednou. U testovacích scénářů, které by byly časově náročné na automatizaci je vždy potřeba zvážit, jestli by automatizace byla přínosná. Automatizace má své limity a u ojedinělých případů je výhodnější provést testovací scénář ručně [34].

3.1 Důvody automatizace

Automatizace testů je jedna z nejlepších cest, jak ve vývoji zvýšit rychlost testování požadované funkcionality. Automatizací lze celkově zvýšit efektivnost, spolehlivost a kvalitu vyvíjeného softwaru. Manuální testování všech možných testovacích scénářů je časově a finančně náročné. Jelikož automaty nepotřebují zásah člověka při jejich provádění, lze automaty spouštět v průběhu celého dne i noci. Automatizací můžeme pokrýt větší množství funkcionality v aplikaci. Automatickým testováním můžeme předcházet lidskými chybám v testování, pokud je automatický skript sepsán spolehlivě a kvalitně [38].

3.2 Rozdíly mezi manuálním a automatickým testováním

Automatizační testování pro provádění testovacích scénářů používají automatizační nástroje oproti manuálnímu testování, kde testovací scénáře provádí člověk. Čas potřebný na provedení testovacího scénáře je výrazně nižší u automatů než u ručního testování.

vání. U ručního testování, oproti automatům, lze využít intuici a zkušenosti testera v explorativním testování. I když je počáteční fáze automatizace výrazně náročnější na časové prostředky než u manuálního testování, z dlouhodobého hlediska se tato investice vyplatí. Manuální testování je náchylnější na lidskou chybu, jelikož testování provádí člověk. Automat je sada příkazů, která se spolehlivě provádí pokaždé stejně. Automaty jsou citlivější na změny v aplikaci, ať už jde o infrastrukturu kódu anebo grafické uživatelské rozhraní. Oproti tomu ruční testování takové změny dokáže zpracovat. Automaty lze spustit v paralelním režimu oproti více testovacím prostředím a tak snížit dobu vykonávání. U manuálního testování lze snížit dobu vykonávání tím, že se bude testování věnovat více testerů [35].

3.3 Životní cyklus automatizace

Životní cyklus automatizace je proces, při kterém dochází k vytvoření automatu a k zautomatizování testovacího scénáře. Je složen z šesti po sobě jdoucích fází, které jsou vykonávány sekvenčně. Proces je implementován zároveň s životním cyklem softwarového vývoje [36]. Grafické znázornění životního cyklu automatizace je na obrázku 3.1.

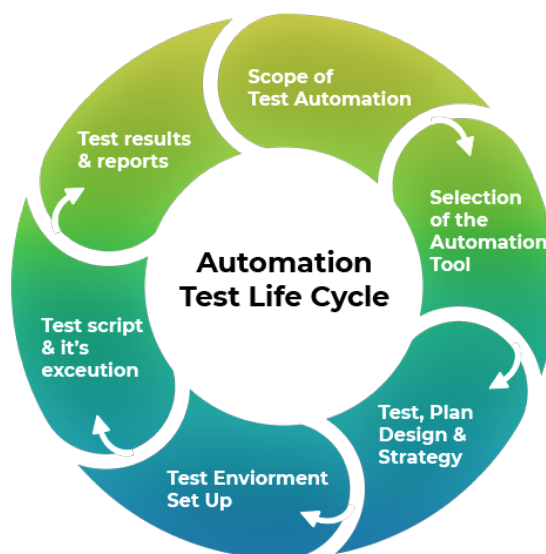
3.3.1 Definování rozsahu automatizace

V první fázi životního cyklu automatizace se rozhoduje, které části aplikace se dají zautomatizovat a které ne. Vybírají se testovací scénáře na základě jejich složitosti a časové náročnosti. Testeři diskutují o postupech, jak zautomatizovat konkrétní testovací scénáře. Zjišťují potřebný čas pro automatizaci, analyzují dostupné zdroje, které automat bude potřebovat [36].

3.3.2 Výběr testovacího nástroje

Druhá fáze životního cyklu je výběr samotného testovacího nástroje. Fáze se považuje za kritickou, jelikož automatizace testů vysoce závisí na zvoleném testovacím nástroji. Pomocí automatizačního testovacího nástroje můžeme snadno přistupovat k testovacím datům, zvládnout implementaci testu a porovnat skutečný výstup s očekávaným výsledkem. Při výběru je potřeba zvážit, pro jaký typ aplikace (webové, on-premise¹⁾) budou automaty vytvářeny. Analyzují se limitace zvoleného nástroje, jestli je nástroj open-source nebo je placený. V potaz se musí brát i zkušenosti lidí, kteří budou s testovacím nástrojem pracovat [36].

¹⁾On-premise je označení pro aplikaci, která je nainstalována lokálně na počítači.



Obrázek 3.1 Schéma životního cyklu automatizace [37]

3.3.3 Plánování, návrh a strategie

V další fázi životního cyklu automatizace se tvoří plán a postup automatizace. Probíhá plánování automatizace testovacích scénářů, určí se potřebný čas pro automatizaci a v jakém pořadí budou testovací scénáře automatizovány. Testovací tým zjistí testovací standardy a pravidla zvoleného automatizačního testovacího nástroje. Zanalyzují výhody a nevýhody testovacího nástroje a podle toho zvolí návrhový vzor pro implementaci automatu. Zvolí se strategie, jak často budou automaty poušťeny, proti jakému prostředí budou provozovány a kde budou zobrazovat výsledky [36].

3.3.4 Příprava prostředí

Fáze příprava prostředí zahrnuje přípravy ohledně prostředí, kde budou automatu poušťeny. Jde o vytvoření a konfiguraci virtuálních strojů, kontejnerů anebo cesta k webovému rozhraní. Testovací prostředí by mělo být nastaveno podobně jako bude nastavení prostředí u zákazníka, aby bylo dosaženo co největší rovnováhy mezi testovacími a produkčními daty. Probíhá konfigurace prostředí pro simulaci testů na podporovaných operačních systémech. Pokud jde o webovou aplikaci, v úvahu se bere testování ve všech podporovaných webových prohlížečích. Konfiguruje se databáze s testovacími daty, kterou budou automaty používat [36].

3.3.5 Vývoj a spuštění testu

V předposlední fázi životního cyklu automatizace probíhá samotný vývoj testu a následné jeho spuštění proti nakonfigurovanému testovacímu prostředí. Automatizace testova-

cích scénářů probíhá podle plánu, který byl vytvořen ve fázi plánování. Pomocí automatizačního nástroje probíhá přepis kroků z testovacího scénáře do příkazů pomocí zdrojového kódu. Podle navrženého testovacího vzoru se buduje infrastruktura testovacího repozitáře. Testovací kód automatu by měl být napsán tak, aby byl přehledný, srozumitelný a spolehlivý. Změny ve zdrojovém kódu automatu mohou být kontrolovány jinými testery za účelem zkvalitnění psaného automatu [36].

Jakmile automat, obsahující všechny funkční aspekty z testovacího scénáře, je hotov, test je zařazen do testovací sady. Testovací sada automatů se spustí proti nakonfigurovaných prostředí. V případě testování webové aplikace, by automaty měly být spuštěny ve všech podporovaných webových prohlížečích. Testovací sady mohou být spuštěny ručně anebo automaticky podle naplánovaného časového harmonogramu [36].

3.3.6 Výsledky testu

V poslední fázi životního cyklu je shromáždění výsledků po dokončení všech automatických testů. Týmy zanalyzují výsledky a vyhodnotí automaty, které identifikují chybu. Chyba v automatu může znamenat, že je problém v aplikaci anebo může automat spadnout díky neočekávané situaci, se kterou naprogramovaný skript nepočítal [36].

3.3.7 Údržba automatů

Údržba automatů nemá svoji vlastní fázi v životním cyklu automatizace, ale probíhá po celou dobu vývoje. Pokud je přidána nová funkcionálníta do aplikace anebo proběhla rozsáhlejší změna, je potřeba přizpůsobit stávající automaty v testovací sadě. Testovací sada se musí upravit tak, aby automaty netestovaly zastaralou funkcionálnítu, aby nedocházelo k testování duplicít. Cílem je mít co nejrychleji otestovanou funkcionálnítu, ale aby pokrytí automatů bylo co nejefektivnější [36].

3.4 Výhody

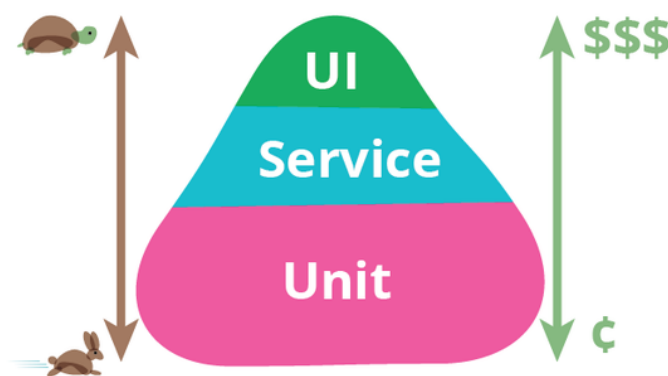
Automatické testování zabere mnohem méně času než manuální testování. Lidský zásah při vykonávání automatů je minimální a nevyžaduje tolik lidských zdrojů jako u ručního testování. Zlepšuje efektivnost využití času. Automatizační testování je spolehlivější, protože eliminuje skryté chyby opětovným prováděním testovacích případů stejným způsobem. Automaty poskytují opětovnou použitelnost testovacích scénářů při testování různých verzí stejného softwaru. Zajišťuje konzistenci v aplikaci a zlepšuje kvalitu aplikace, jelikož automaty mohou být pouštěny frekventovaně. Tester může zanalyzovat odezvu softwaru, protože se provedení stejné operace v automatu několikrát opakuje [38].

3.5 Nevýhody

Pro automatizaci testovacích scénářů volíme kvalitní automatizační nástroje, které vyžadují znalosti a zkušenosti testerů. Prvotní fáze zprovoznění automatizace je náročná na čas. Automaty mohou být nestabilní, pokud testovací prostředí vykazuje pokaždé jiné chování aplikace. Velké množství automatů v testovací sadě může být náročnější na údržbu, díky častým změnám požadavků v aplikaci. Nekvalitní automaty mohou padat na chyby, které nesouvisí s problémy v aplikaci a jsou náročnější na údržbu [38].

3.6 Testovací pyramida

Testovací pyramida automatizace představuje různé typy testů a množství, s jakou by se měli objevit v testovací sadě. Testovací pyramida má tři základní vrstvy. Nejspodnější vrstvou v pyramidě jsou Unit testy, které by měly mít v testovací sadě největší zastoupení. Jejich vytvoření a následná údržba stojí nejméně času a jejich čas běhu v testovací sadě je nejrychlejší. Více o unit testování v kapitole 3.7. Střední vrstvu v testovací pyramidě tvoří Service neboli integrační testy. Jsou to testy, které testují tok dat z jednoho modulu aplikace do druhého. Takové testy v testovací sadě jsou více časově náročné na tvorbu a správu než Unit testy a jejich čas běhu je také o něco delší. Na vrcholu pyramidy je vrstva, kterou tvoří automaty testující grafické rozhraní aplikace neboli End-to-End testy. Více o takových testech v kapitole 3.9. Jsou to testy, které jsou nejvíce časově náročné na tvorbu a následnou údržbu v testovací sadě. Čas potřebný pro otestování aplikace je nejdelší ze všech tří zmíněných typů testů. Proto by v testovací sadě UI²⁾ testy měli mít co nejmenší zastoupení. Testovací pyramida pomáhá k vytvoření optimální testovací sady, která produkt co nejrychleji otestuje a zároveň kvalita testování bude největší [39]. Grafické znázornění testovací pyramidy je na obrázku 3.2.



Obrázek 3.2 Testovací pyramida [40]

²⁾Automaty testující uživatelské rozhraní aplikace.

3.7 Unit testy

Unit testy testují jednotky nebo jednotlivé součásti systému. Cílem testování je ověřit, že jednotka softwarového kódu funguje podle očekávání. Testovací funkci je předán vstup a verifikuje se aktuální vrácená hodnota oproti očekávané. Jednotkové testy oddělují části kódu a ověřují funkčnost jednotlivých modulů, procedur, tříd a metod. Jedná se o testování technikou White box, kdy je k dispozici zdrojový kód a infrastruktura aplikace. Testy se tvoří ve vývojové fázi implementace a jejich čas na vytvoření by měl být nejkratší podle testovací pyramidy. Unit testy můžou psát jak vývojáři, tak i testéři [41].

3.7.1 Důležitost unit testů

Unit testování probíhá v testovacím cyklu jako první. Jsou na první úrovni testovací pyramidy před integračními a systémovými testy. Pokud se implementuje nová funkcionality, pomocí unit testů je vývojář schopen ihned identifikovat problém ještě v implementační fázi. Čas potřebný pro opravení chyby je v takové fázi nejkratší, jelikož se testování pohybuje na úrovni zdrojového kódu. To snižuje pravděpodobnost nalezení chyb v dalších fázích testování, kde čas potřebný na opravu chyby je delší. Unit testování pomáhá testerům a vývojářům lépe porozumět zdrojovému kódu a tím pádem umožňuje rychlejší opravy problémů. Dobré unit testy můžou sloužit i jako dokumentace projektu [41].

3.7.2 Výhody

Unit testování využívá modulový přístup, díky kterému lze testovat různé části zdrojového kódu bez čekání na dokončení jiných částí funkcionality. Správně napsané unit testy poskytují vývojářům přehled o implementované jednotce (třída, metoda, objekt, funkce), aniž by studovali zdrojový kód jednotky. Unit testy pomáhají vývojářům při migraci nebo refaktoru³⁾ kódu, jelikož poskytují očekávané hodnoty, které by měl nový kód vrátit. Unit testy snadno odhalí chybu ve zdrojovém kódu [41].

3.7.3 Nevýhody

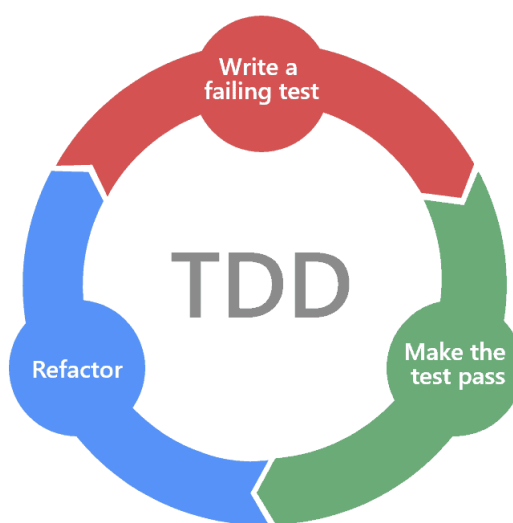
Unit testy testují pouze jednotky ve zdrojovém kódu aplikace. Unit testy nedokáží odhalit chyby na úrovni integrace s jinými moduly anebo rozsáhlé chyby na úrovni systémových testů. Unit testy je někdy obtížné napsat na rozsáhlou implementaci ve zdrojovém kódu, občas zaberou stejné množství času jako samotná implementace funkcionality. Při unit testování není možné pokrýt všechny cesty v programu. Je zapotřebí

³⁾Refaktor je označení pro činnost, kdy dochází ke zlepšení zdrojového kódu programu.

dalšího testování na vyšší úrovni [41].

3.7.4 Test Driven Development - TDD

Test Driven Development zkráceně TDD je evoluční technika ve vývoji softwaru, ve které dochází k implementaci unit testů ještě před samotným psáním zdrojového kódu. Jedná se o iterativní přístup, který kombinuje vytváření unit testů, programování a refaktor. Cílem techniky TDD je mít optimalizovaný a kvalitní kód, který je zároveň pokrytý unit testy [42].



Obrázek 3.3 Schéma cyklu TDD [43]

TDD má tři fáze vývoje. V první fázi je vytvořen unit test, aby ověřil správnost konkrétních funkcí, které jsou specifikované požadavky. Vývojáři tvoří test na základě svých předpokladů, jaký seznam vstupů bude funkce brát a seznam výstupů, které bude funkce vracet. Na začátku by měl test spadnout, jelikož zatím není implementovaná jednotka, kterou test verifikuje. Ve druhé fázi přístupu vývojář implementuje jednotku tak, aby unit test k dané jednotce procházel. Jakmile unit test prochází, nastává třetí fáze v TDD. V třetí fázi se optimalizuje jednotka tak, aby neobsahovala redundanci a celkově zvýšila svůj výkon [42]. Grafické znázornění fází v cyklu TDD je na obrázku 3.3.

Metoda TDD podporuje tvorbu optimalizovaného a kvalitního zdrojového kódu aplikace. Při tvorbě unit testů pomáhá vývojářům lépe analyzovat a porozumět požadavkům zákazníka. Testovací pokrytí aplikace v rámci TDD na úrovni unit testů je mnohem vyšší, jelikož TDD se zaměřuje už hned od začátku na vytváření testů pro každou funkcionalitu [42].

```
import org.junit.Assert;
import org.junit.Test;

public class SampleTest {

    @Test
    public void createAndSetNameTest() {

        String expected = "Y";
        String actual = "Y";

        Assert.assertEquals(expected, actual);
        System.out.println("SampleTest is successful " + actual);
    }
}
```

Obrázek 3.4 Ukázka testu v JUnit

3.7.5 JUnit

Je nejrozšířenější open-source nástroj pro unit testování v programovacím jazyce Java. Je užitečný pro psaní a opakované spouštění unit testů. Junit hraje také důležitou roli při psaní testů a zdrojového kódu technikou Test Driven Development. JUnit prosazuje myšlenku "nejříve testování a pak kódování". Poskytuje anotace k identifikaci testovacích metod. Nástroj nabízí širokou škálu assertovacích⁴⁾ metod a podporuje další nástroje pro spouštění testů. Unit testy lze organizovat do testovacích sad. S pomocí JUnit je možné psát testy rychle, efektivně a spolehlivě [44].

3.7.6 NUnit

Nunit testovací nástroj slouží pro psaní a spouštění unit testů pro všechny typy .NET programovacích jazyků. Nástroj je poskytnut jako open-source aplikace. Ve vývojových prostředích, jako je třeba Visual Studio, je nabízen NUnit jako NuGet⁵⁾ balíček. Abychom mohli spouštět testy uvnitř vývojového prostředí je zapotřebí také nainstalovat NuGet balíček NUnit Test Adapter. NUnit nabízí také plno assertovacích metod, které pomáhají vývojářům při psaní unit testů. Pomocí atributů lze označit testovací třídy anebo testovací metody. Atribut [Test Fixture] označuje testovací třídu, v rámci které jsou vykonávány jednotlivé testovací metody označeny atributem [Test]. Atributy [Setup] a [TearDown] se používají pro označení metod, které mají být provedeny před samot-

⁴⁾Metoda, která srovnává a vyhodnocuje aktuální hodnotu s očekávanou.

⁵⁾NuGet je nástroj pro správu balíčků, který je navržen pro sdílení znovupoužitelného zdrojového kódu mezi vývojáři. [45]

ným testováním a po skončení testu. NUnit nabízí implicitně několik dalších atributů a podporuje vytvoření vlastních, specifických pro účely testování [46].

3.8 Integrační testování

Hlavní cílem integračního testování je verifikace modulů nebo komponent při zapojení do systému. Jedná se o kombinaci technik testování Black, White a Gray box. Integrační testování se obvykle provádí po unit testování, kdy máme všechny jednotky vytvořené a pokryté testy. Jakmile jsou jednotky otestovány, jsou postupně integrovány jedna po druhé, dokud nejsou zintegrovány všechny moduly. Testuje se komunikace, stabilita a chování mezi moduly. Integrační testování probíhá během vývoje aplikace a některé moduly nemusí být k testování dispozici. Pokud jde o testovací pyramidu, integrační testy jsou na prostřední pozici mezi unit a systémovým testováním. Jejich čas na vytvoření a provádění testů je o něco delší než u unit testování, ale kratší než u testů kontrolující grafické rozhraní aplikace [47].

3.8.1 Důležitost integračních testů

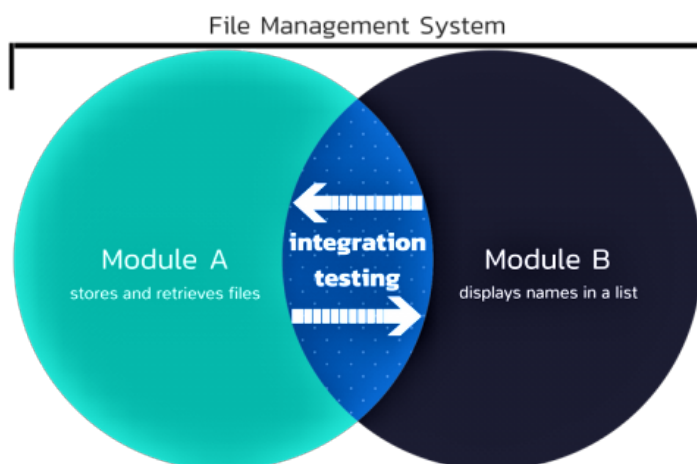
Při vyvíjení aplikace se požadavky rozdělí mezi vývojáře, kteří pracují na jednotlivých požadavcích nebo také modulech. Každý vývojář individuálně implementuje požadavky na základě svých zkušeností a dovedností. Implementace se můžou tedy lišit modul od modulu, proto se používají integrační testy ke kontrole integrace mezi jednotlivými moduly. Dalším důvodem, proč používat integrační testy, je verifikace toku dat mezi jednotlivými moduly. Na jednom modulu data můžou mít jinou strukturu než na tom dalším v celém systému aplikace. Testováním ověříme, že data vystupující z jednoho modulu jsou akceptovány dalším modulem v aplikaci. Moduly v aplikaci také komunikují s nástroji nebo API rozhraním třetích stran a integračními testy lze ověřit, že komunikace je správná [47].

3.8.2 Rozdíly mezi integračním a systémovým testováním

V systémovém testování jde o testy, které kontrolují chování systému jako celek. Nerozlišují se jednotlivé moduly zvlášť, ale všechny komponenty jsou integrovány dohromady. U integračních testů se verifikují moduly jednotlivě a zaměřují se na funkční aspekt aplikace [47].

3.8.3 Výhody

Integrační testování zajišťuje kontrolu modulů a komponent integrovaných dohromady. Testováním hledáme chyby související s komunikací a přenosem dat mezi moduly.

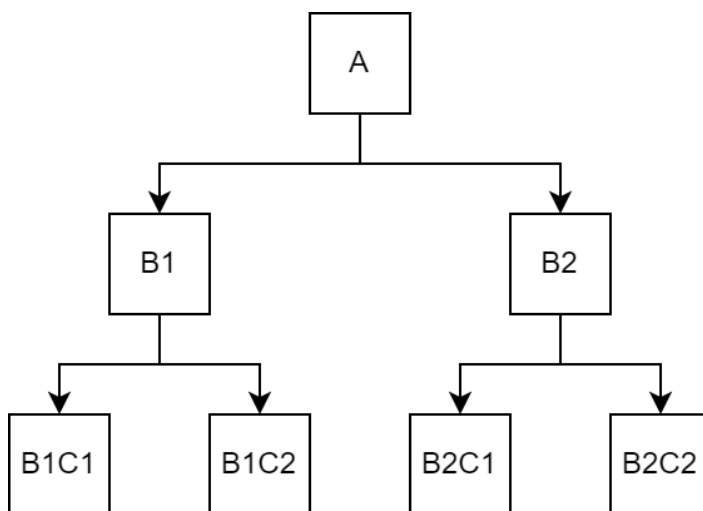


Obrázek 3.5 Integrační testování [48]

Chyba v malém celku integrovaných komponent lze snáze ladit a opravit, než pokud by chyba byla odhalena v modulu zapojeném do celého systému. Integrační testování může začít hned, jakmile je modul k dispozici. Testování nemusí čekat na dokončení všech modulů v aplikaci [47].

3.8.4 Výzvy

Integrační testování nemá nevýhody jako takové, ale má pár obtížností. Testování je komplexnější, protože do testu se zahrnují různé moduly a části, jakou jsou databáze, různé nástroje a součásti testovaného prostředí. Při integraci modulů používající rozdílné přístupy a techniky může být náročné pro sestavení integračního testu. U testování integrace modulů s nástroji třetích stran může často docházet ke změně některého z modulů, integrační testy se musí aktualizovat a přizpůsobit [47].



Obrázek 3.6 Schéma integrovaných modulů

3.8.5 Přístupy integračního testování

Existují v zásadě dva typy přístupů k integračnímu testování. První je přístup zvaný Big Bang a tím druhým je přístup Inkrementální. Inkrementální přístup se dále rozlišuje na přístupy Top-down, Bottom-up a Sandwich přístup [47].

Big Bang testování je technika integračního testování, kdy všechny jednotky nebo komponenty v systému jsou zintegrovány dohromady a jsou testovány jako jedna jednotka. Tato sada jednotek je považována při testování za entitu. Integrační testování může začít pouze tehdy, když jsou implementovány a zapojeny všechny jednotky v systému [47].

V inkrementální přístupu integračního testování jde o integraci dvou nebo více logicky souvisejících modulů, které jsou následně testovány. Postupně jsou integrovány další související moduly, dokud nejsou všechny logicky související moduly zintegrovány a otestovány. Inkrementální testování se dále dělí na Top-Down, Bottom-Up a Sandwich přístup, které se liší podle způsobů integrování a jednotlivých modulů [47].

Bottom-up přístup začíná testovat od nejnižší nebo nejvnitřnější jednotky aplikace a postupně pokračuje nahoru k modulům na vyšší úrovni. Integrace pokračuje pořád výš dokud nejsou zintegrovány všechny moduly v aplikaci. Na obrázku 3.6 je příklad zintegrovaných modulů. Typem Bottom-Up začíná integrační testování od nejnižších modulů B1C1, B1C2 a B2C1, B2C2, které jsou testování pomocí unit testů. Integrační test verifikuje komunikaci modulů B1 a B2 s nejnižšími moduly B1C1, B1C2 a B2C1, B2C2. Pokud nejsou moduly na vyšších úrovních implementovány, používají se fiktivní programy pro volání funkcionality na nižších vrstvách. Výhodou přístupu Bottom-U, že pokud existuje závažná chyba na nejnižší jednotce programu, je snazší ji odhalit a opravit [47].

Top-down přístup integračního testování začíná od nejvyššího modulu a postupně postupuje směrem k nižším modulům. Pouze nejvyšší modul je otestován pomocí unit testů. V případě schématu na obrázku 3.6 začíná integrační testování modulem A s moduly nižší vrstvy B1 a B2. Hlavní výhodou Top-down přístupu je testování kritických modulů jako první [47].

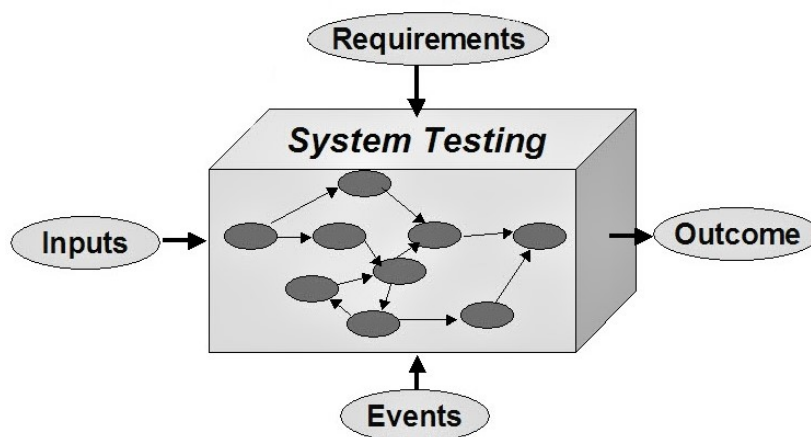
Sandwich nebo Hybrid strategie je kombinace přístupů Bottom-up a Top-down. Je to technika, ve které jsou zintegrovány moduly nejvyšší úrovně s moduly nižší úrovně a zároveň jsou nejnižší moduly integrovány s vyššími. Takto zintegrované moduly jsou brány jako jeden systém, na kterém je prováděno integrační testování [47].

3.8.6 Nástroje

Existuje několik typů nástroj pro provádění integračního testování. Prvním populárním nástrojem je VectorCAST/C++, který je používán pro unit a integrační testování. V programu je každá jednotka nebo modul testován nezávisle, aby došlo k ověření, že komponenty fungují bez jakékoliv závislosti. Poté jsou během integračního testování moduly zkombinovány a verifikuje se jejich správné chování. Program VectorCAST/C++ je určen pro vývojáře programující zdrojový kód v jazyce C++. Další nástroj pro testování je program Citrus, který je open-source software napsaný v Javě podporující automatické integrační testování [49].

3.9 Systémové testování

Systémové testování je soustava různých typů testů, které verifikují kompletně celý integrovaný softwarový systém. Testování je prováděnou technikou Black Box, kdy neznáme zdrojový kód aplikace. Cílem testování je ověřit funkčnost a chování všech specifikovaných požadavků v kompletním softwarovém systému. Jde o verifikaci koncových funkcí a scénářů, proto systémové testování se také označuje jako end-to-end testování. V testovací pyramidě (obr. 3.2) se testy nacházejí na vrcholu pyramidy. Z definice pyramidy jde o testy, které jsou nejvíce časově nákladné na jejich tvorbu a údržbu. Z hlediska času je systémové testování nejdelší. V testovacím cyklu je systémové testování prováděno po integračním a unit testováním [54].



Obrázek 3.7 Systémové testování [55]

3.9.1 Důležitost testování

Testování verifikuje funkčnost kompletního systému a zajišťuje správné chování koncových funkcí, které jsou k dispozici zákazníkovi. Systémové testování se nachází na konci cyklu testování a jde o poslední testování, které se provádí před vydáním nové verze

produktu na produkční prostředí. Testy většinou běží na testovacím prostředí, které může být nejbližší k produkčnímu prostředí u zákazníka. Tím pádem testy poskytují takovou zpětnou vazbu, která by se mohla přiblížit zpětné vazbě klienta. Automaty se použijí i po nasazení nové verze na produkci pro zajištění funkčnosti a stability nové verze aplikace. Systémové testování hraje důležitou roli, která pomáhá k vydání spolehlivé a kvalitní aplikace [54].

3.9.2 Typy systémového testování

Systémové testování zahrnuje mnoho technik, způsobů a typů, jak systémově otestovat aplikaci. Použití některého z nich závisí na produktu, organizačních procesech, časové ose a požadavcích. Mezi nejznámější typy systémového testování patří GUI testování, Smoke testování, Funkcionální, Performance, Regresní, Kompatibilní, testování použitelnosti a zátěžové testování [56].

GUI neboli testování grafického uživatelského rozhraní testuje funkcionalitu, která je viditelná pro uživatele. Testování zahrnuje kontrolu tlačítek, ikon, dialogových oken, panelů, textových polí a další komponenty, se kterými pracuje uživatel v GUI prostředí [56].

V regresním testování systémovými testy se provádí verifikace celého systému s cílem identifikovat závady, které se mohli zavést při implementaci změn a požadavků v průběhu celého vývoje. Regresní testování aplikace zajišťuje kontinuální kvalitu aplikace [56].

Smoke testování obsahuje testovací sadu testů, které mají za úkol zkontrolovat zkompilovanou verzi aplikace pro další její testování a používání. Testy ověřují pouze kritické funkce v aplikaci [56].

Performance testování nebo také testování výkonu je proces testování softwaru používaný k měření rychlosti, doby odezvy, stability, škálovatelnosti a celkově spolehlivosti aplikaci při zatížení. Hlavním účelem testování je identifikovat a odstranit místa, kde dochází k přehlcení systému při zátěži aplikace [50].

Testování použitelnosti spadá pod ne-funkcionální testování. Primárně se zaměřuje na uživatele a jejich práci s aplikací. Cílem testování je zkontrolovat, zda je aplikace použitelná a snadno pochopitelná pro koncové zákazníky. Při použití testování použitelnosti se zjišťuje, jestli je vyvinutý software jednoduchý, přímočarý a usnadňuje práci uživatelům [56].

Mezi systémové testování se zahrnují testy, které testují kompatibilitu softwaru. Testování má za cíl otestovat kompatibilitu vyvíjené aplikace s různými prohlížeči, operačními systémy, databázemi a jinými nástroji [56].

3.9.3 Výhody

Systémové testování zahrnuje verifikaci koncových funkcí, které zákazník používá. Jedná se o end-to-end testování, které obsahuje testy na kontrolu koncových uživatelských cest. Nejčastěji testy běží na testovacím prostředí, které nejvíce připomíná produkční. Test mohou být spuštěny také na produkčním prostředí u zákazníka. To může odhalit chyby a problémy, se kterými by se klient mohl setkat. Testování zajišťuje správné chování celé aplikace a pomáhá ověřit funkčnost specifikovaných požadavků od zákazníka anebo projektového manažera [56].

3.9.4 Nevýhody

Testování celého systému je časově náročné na provedení, jelikož testy pokrývají celou softwarovou aplikaci. Náklady na tvorbu a údržbu systémových testů jsou vysoké, protože pokrývají architekturu softwaru a správné chování funkčních požadavků. K vytvoření a provedení testu je zapotřebí větších znalostí a zkušeností testujícího.

3.9.5 Nástroje

Nástrojů a frameworků⁶⁾ je k dispozici celá řada pro provoz systémového testování. Nástroje pomáhají k tvorbě testovacích scénářů, implementaci testů, následně jejich spuštění a zpracování výsledků. Známým frameworkem pro automatizaci všech testů je nástroj Selenium. Má široké uplatnění ve všech typech testů a podporuje provoz testů v všech webových prohlížečích [53]. Více o tomto frameworku budu psát v kapitole 5.2.

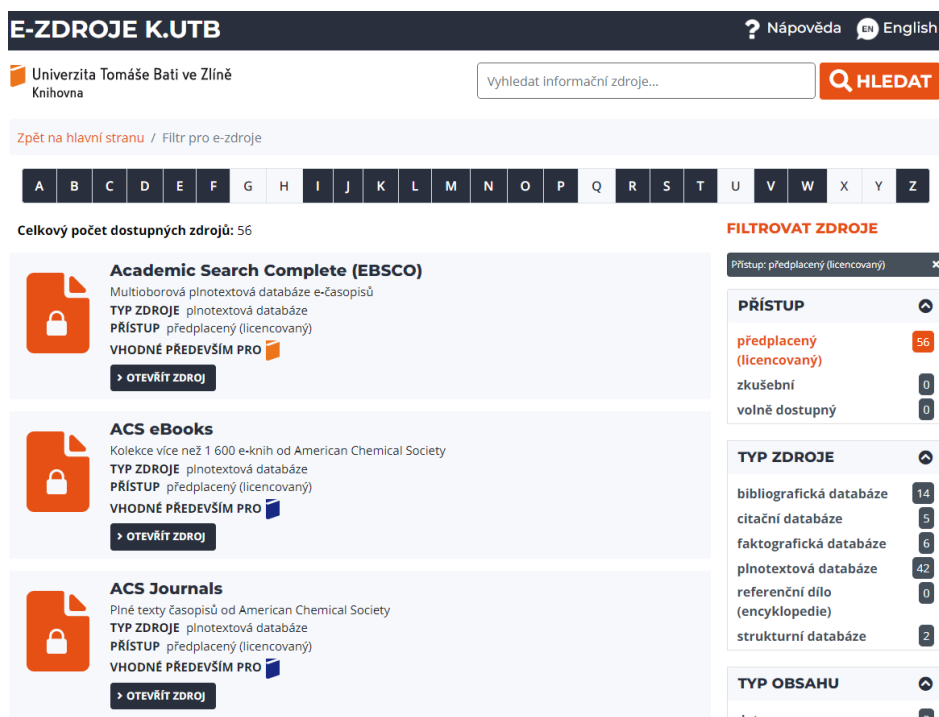
Dalším populární nástrojem pro testování je framework Robot. Framework je open-source a používá se pro automatizaci robotických procesů a automatizaci testování. Je napsán v programovacím jazyce Python a lze jej použít s dalšími vhodnými nástroji pro tvorbu automatických testů [53].

⁶⁾Framework je softwarová struktura, která pomáhá při vývoji a organizaci jiných softwarových projektů. Může obsahovat pomocné programy, knihovny API a jiné nástroje pro podporu vývoje softwarové aplikace. [52]

II. PRAKTICKÁ ČÁST

4 TESTOVACÍ APLIKACE

Testovací aparát byl vyvíjen ve spolupráci s Knihovnou UTB a jeho hlavním cílem je zaručit kontinuální testování dostupnosti služeb na internetu. Za služby jsou považovány webové aplikace, které poskytují elektronické zdroje informací. Testovací program obsahuje sadu systémových testů, které verifikují funkčnost přihlášení do jednotlivých služeb. Přihlášení probíhá přes grafické uživatelské rozhraní webového prohlížeče. Každý test kontroluje několik možností, jak se přihlásit do konkrétní služby a získat přístup do elektronického zdroje. Testovací sada bude spouštěna na linuxovém serveru v nastavených intervalech. Po dokončení aktuálního běhu testovací sady budou výsledky automatických testů shromážděny a poslány elektronickou poštou do schránky adresáta. Správce bude mít k dispozici aktuální přehled o stavu jednotlivých služeb.



Obrázek 4.1 Portál elektronických zdrojů

4.1 Elektronické zdroje

Portál E-Zdroje K.UTB nacházející se na webové stránce <https://ezdroje.k.utb.cz> nabízí seznam elektronických informačních zdrojů pro Univerzitu Tomáš Bati ve Zlíně. Ukázka portálu je na obrázku 4.1. Portál spravují techničtí pracovníci z Knihovny UTB. Je zde k dispozici několik desítek elektronických zdrojů, kde každý zdroj je zařazen do několika kategorií. Rozlišují se například podle licence, typu zdroje, typu obsahu, tematiky zdroje anebo podle jazyku. V testovací aplikaci se testy zaměřují pouze na

zdroje, které jsou předplacené s licencí. Volně dostupné zdroje a zdroje se zkušebním přístupem se na základě konzultace s pracovníky Knihovny UTB netestují. Každý elektronický zdroj má různé způsoby, jak se přihlásit a získat přístup do dané služby.

4.1.1 Přístup přes Shibboleth

Každý elektronický zdroj v databázi má přihlášení přes technologii Shibboleth. Shibboleth je jedním z nejrozšířenějších systému pro správu identit na světě. Technologii používají akademické instituce, federace identit a komerční organizace po celém světě. Umožňuje bezpečný a bezproblémový přístup k chráněným on-line zdrojům a aplikacím. S jedinou identitou se uživatel může přihlašovat do různých systémů. Projekt je open-source a jeho používání je zdarma [59].

4.1.2 WAYF

Další možností přihlášení do služby je způsobem WAYF. WAYF je zkratka pro projekt s názvem **Where Are You From**. Jde o způsob přihlášení do služby za pomoci instituce. Uživatel se naviguje na webové stránky služby, kde zvolí přihlášení pomocí instituce. Vybere ze seznamu institucí, ve které má registrovanou identitu. Po vybrání instituce dojde k přesměrování na instituci, kde uživatel zadá své přístupové iniciály a standardně se přihlásí. Pokud jsou přihlašovací údaje validní, dojde přes technologii WAYF ke přesměrování zpět do služby, ve které už je uživatel přihlášen pod svým účtem z instituce [57]. Grafické znázornění WAYF přístupu je na obrázku 4.2.

4.1.3 WAYFless

Tzv. WAYFless přihlášení do služby je provedeno navigací na webovou stránku pomocí URL odkazu, který obsahuje identifikátor poskytovatele identity (IdP) Shibboleth a tím umožňuje přeskočit nutnost volby IdP. URL odkazy umožňují přímý přístup na obsah elektronického zdroje použitím autentizace třetí strany. V našem případě je to autentizace za použití technologie Shibboleth. Po úspěšném přihlášení je uživatel přesměrován do služby, kde je rozeznána jeho identita. Přihlášení wayfless způsobem značně redukuje počet kroků, které musí uživatel udělat pro přihlášení do elektronického zdroje [58].

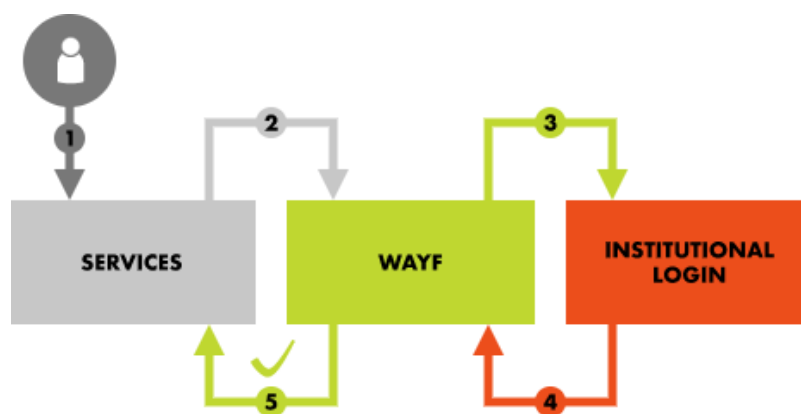
4.1.4 Proxy

Pro přístup k elektronickým zdrojům přes Knihovnu UTB může uživatel využít proxy UTB s prefixem URL `https://proxy.k.utb.cz/login?url=`. V URL proxy pak následuje odkaz na danou službu. Pomocí proxy uživatel získává vzdálený přístup k elektronickému zdroji. Na rozdíl od přihlášení přes Shibboleth, vzdálený přístup přes proxy

typicky nevyžaduje podporu ze strany provozovatele elektronického zdroje. Do proxy se lze přihlásit přes Shibboleth technologii anebo uživatel může zvolit přihlášení s údaji k LDAP. Po úspěšném přihlášení je uživatel přesměrován na službu, kde je identifikován.

4.1.5 VPN

Všechny výše zmíněné typy přihlášení přistupují k elektronickým zdrojům s libovolnou IP adresou. Další alternativou, jak se dostat ke službám je použít univerzitní IP adresu. Pokud se přihlašujeme z počítače, který není připojen do interní sítě univerzity, je potřeba se připojit do sítě vzdáleně přes VPN. Jakmile uživatel získá univerzitní IP adresu, může k elektronickým zdrojům přistupovat bez přihlášení. Uživatel může zvolit jakýkoliv způsob přístupu do služby (Shibboleth, WAYF, WAYFless, Proxy), vždy je přesměrován a ve službě identifikován. Nevýhodou přístupu přes VPN je nutnost instalovat aplikaci VPN klienta na stanici koncového uživatele.



Obrázek 4.2 Přihlášení přes WAYF [57]

4.2 Důvod testování

Vzhledem k nárůstu využívání centrálního Identity Providera (IdP) i rozšíření počtu zpřístupňovaných služeb se zviditelnily občasné problémy s dostupností elektronických zdrojů. Jednotlivé elektronické zdroje mohou být nefunkční po aktualizaci nebo změně konfigurace. O nedostupnosti zdroje se správci dozvídají až z reakce uživatelů, někdy se značným časovým odstupem.

4.3 Přínos

Vytvoření nástroje pro pravidelné automatizované testování přihlašovacího procesu do jednotlivých elektronických zdrojů bude předcházet problémům s jejich dostupností. Umožní rychle zareagovat při nedostupnosti služby, kontaktuje správce knihovny o

problému. Automaty budou výhodné i pro jednotlivé služby poskytující elektronické zdroje, jelikož budou monitorovat funkčnost přihlášení do jejich portálů. Cílem je zvýšit kvalitu zdrojů knihovny UTB a zvýšit spolehlivost při přihlašování uživatelů do služeb.

5 ŘEŠENÍ

Cílem je vytvořit nástroj pro pravidelné automatizované testování přihlašovacího procesu. Přihlašování přes Shibboleth je založeno na webovém workflow a přesměrování uživatele ze služby poskytující elektronické zdroje na IdP a zpět. Pokud chceme tento workflow proces automatizovat a zároveň zajistit, že se bude chovat stejně pro uživatele i testovací automat, nejvhodnější volbou je spustit workflow ve skutečném prohlížeči a přehrát předem zaznamenané činnosti uživatele. Tento typ testování je nazýván End-to-End. V End-to-End testování jde o zajištění funkčnosti všech komponent přihlašovacího řešení z pohledu uživatele. Verifikuje řešení od zdroje dat o uživateli (Active Directory) přes Identity Providera po jednotlivé poskytovatele elektronických zdrojů. Testovací aplikace bude provozována na linuxovém serveru, kde bude testovací sada pouštěna v pravidelných intervalech. Shromažďování a zobrazení výsledků v html formátu bude také provádět testovací aplikace za použití python pluginů. Nástroje pro kontinuální integraci a nasazení (CI/CD) nebyly použity, jelikož by to vytvářelo větší komplexitu řešení než ulehčení práce s testovací aplikací pro pracovníky knihovny UTB. Automatické spouštění testovací sady bude pravidelně naplánováno každou hodinu pomocí aplikace `cron` na linuxovém stroji.

5.1 Požadavky na řešení

Testovací aplikace bude určena pro kontinuální testování provozovaných služeb ve webovém rozhraní. Je požadavkem zajistit spouštění testů v podporovaných webových prohlížečích. Automaty budou používat end-to-end testování, tudíž provedení přihlášení bude pomocí grafického uživatelského rozhraní. Je potřeba vybrat takový nástroj k automatizaci, který bude open-source, zdarma a bude mít zákaznickou podporu. V případě nově přidávaných služeb poskytující elektronické zdroje, bude mít možnost napsat testovací skript i zaškolený pracovník knihovny UTB. Požadavkem pro vybrání technologie k psaní automatů je tedy uživatelsky přívětivý framework, díky kterému půjde snadno psát nové automaty. Řešení bude mít možnost spouštět testovací sadu pravidelně a výsledky automatů posílat na email. Nástroj by měl podporovat spouštění testů na operačním systému Linux.

5.2 Selenium

Selenium je jedním z nejpoužívanějších open-source nástrojů pro automatické testování webového uživatelského rozhraní. Selenium neumí testovat desktopové aplikace, ale je určen testování webu. Nástroj podporuje automatizaci napříč různými webovými prohlížeči (Google Chrome, Opera, Firefox, Internet Explorer, Safari). Umožňuje spuštění ve všech operačních systémech (Windows, Mac, Unix/Linux). Pro implementaci

automatických testů framework podporuje několik programovacích jazyků (C#, Java, Perl, Php, Python, Ruby). Selenium se používá také pro psaní funkcionálních testů a umožňuje integraci dalších automatických nástrojů k dosažení kontinuálního testování (Maven, Jenkins nebo Docker). Lze také použít pro správu a spouštění automatů nástroje jako je NUnit anebo JUnit. Selenium se skládá z několika komponent, které lze použít pro automatizaci [60].

5.2.1 Selenium IDE

Selenium IDE slouží primárně pro nahrávání a spouštění automatických skriptů ve webovém prohlížeči. Rozšíření pro prohlížeč Firefox nebo Chrome snadno vytváří testy pomocí funkcí pro nahrávání a spouštění automatických skriptů [60].

5.2.2 Selenium RC

Selenium RC nástroj umožňuje psaní automatických testů ve skriptovacím jazyce, které testují webové rozhraní aplikace. Skládá se ze serverové a klientské knihovny. Testovací skript je vložen do webového prohlížeče, ve kterém skript provádí různé interakce [60].

5.2.3 Selenium Grid

Selenium Grid podporuje souběžné provádění automatických testů na různých prohlížečích, počítačích a operačních systémech současně. Nástroj usnadňuje testování kompatibility mezi prohlížeči. Zkracuje také celkový čas běhu automatů [60].

5.2.4 Selenium WebDriver

Selenium WebDriver je vylepšená verze komponenty Selenium RC. Je to nejdůležitější součást sady Selenium. S pomocí WebDriver lze testovací skripty vyvíjet pomocí libovolných podporovaných programovacích jazyků a umožňuje spouštět tyto skripty ve všech moderních prohlížečích. Testují grafické rozhraní aplikace [60].

5.2.5 Výhody

Selenium umožňuje testovat webové aplikace ve všech podporovaných prohlížečích a na všech operačních systémech. Nástroj je zdarma a volně přístupný jako open-source. Implementace automatických jazyků lze pomocí několika programovacích jazyků. Automaty se dají spouštět paralelně což zkracuje celkovou dobu vyhodnocování testů a zvyšuje efektivnost. Selenium se dá integrovat s JUnit, NUnit, Maven a dalšími nástroji, které slouží pro rychlejší a snadnější testování pomocí automatických skriptů [60].

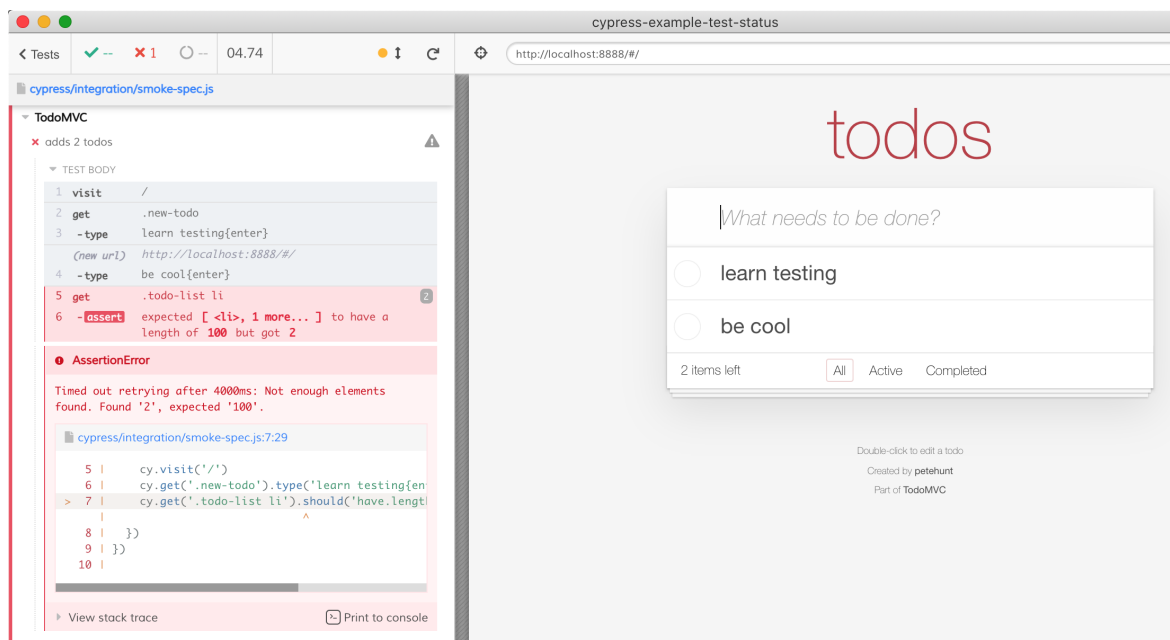
5.2.6 Limitace

Nástroj slouží pouze pro testování webových aplikací. Pro efektivní vytváření automatických skriptů vyžaduje Selenium kvalitní znalosti a dovednosti v oblasti programování. Nelze provádět automatické testování webových služeb SOAP nebo REST. Konfigurace a příprava testovacího prostředí pro běh Selenium zabere delší čas [60].

5.3 Cypress

Cypress je moderní, zdarma a open-source nástroj, který slouží pro automatizaci systémových testů, integračních i unit testů. Automaty se programují pomocí programovacího jazyka Javascript a především slouží pro testování front-end¹⁾ aplikace. Je kompatibilní s různými webovými prohlížeči a také operačními systémy (Windows, Mac, Unix/Linux). Umožňuje kontrolovat a spravovat síťový provoz. Cypress lze také integrovat s několika nástroji pro kontinuální integraci a nasazení [61].

Kromě běžného psaní automatických skriptů Cypress poskytuje vizuální rozhraní, ve které zobrazuje aktuálně spuštěné testy a provedené příkazy ve skriptu. Umožňuje testovat vysoce interaktivní a responzivní aplikace. S nástrojem Cypress můžeme přistupovat a manipulovat v aplikaci přes HTML DOM webové stránky, vyplňovat textové pole, odesílat formuláře a testovat přítomnost grafických prvků na stránce [61].



Obrázek 5.1 Cypress [62]

¹⁾Část webové aplikace, se kterou uživatel komunikuje a interaguje. Jde o grafické rozhraní webové aplikace.

5.3.1 Výhody

Používání nástroje Cypress poskytuje mnoho výhod a vylepšení. Jedna z hlavních výhod oproti nástroji Selenium je zabudované automatické čekání na zobrazení elementu na stránce pro pozdější jeho interakci. Automaty se stávají stabilnějšími a spolehlivějšími. Další implicitní funkcí v Cypress je poskytování video záznamu spuštěného testu, který slouží pro pozdější ladění a správu automatického skriptu. Poskytování obrázků při spadnutí testu je samozřejmostí. K dispozici je také vizuální rozhraní Cypressu, kde můžeme testy ladit, spravovat a spouštět. Testy lze integrovat s několika nástroji pro kontinuální testování, testy lze spouštět v různých webových prohlížečích a na různých operačních systémech [63].

5.3.2 Limitace

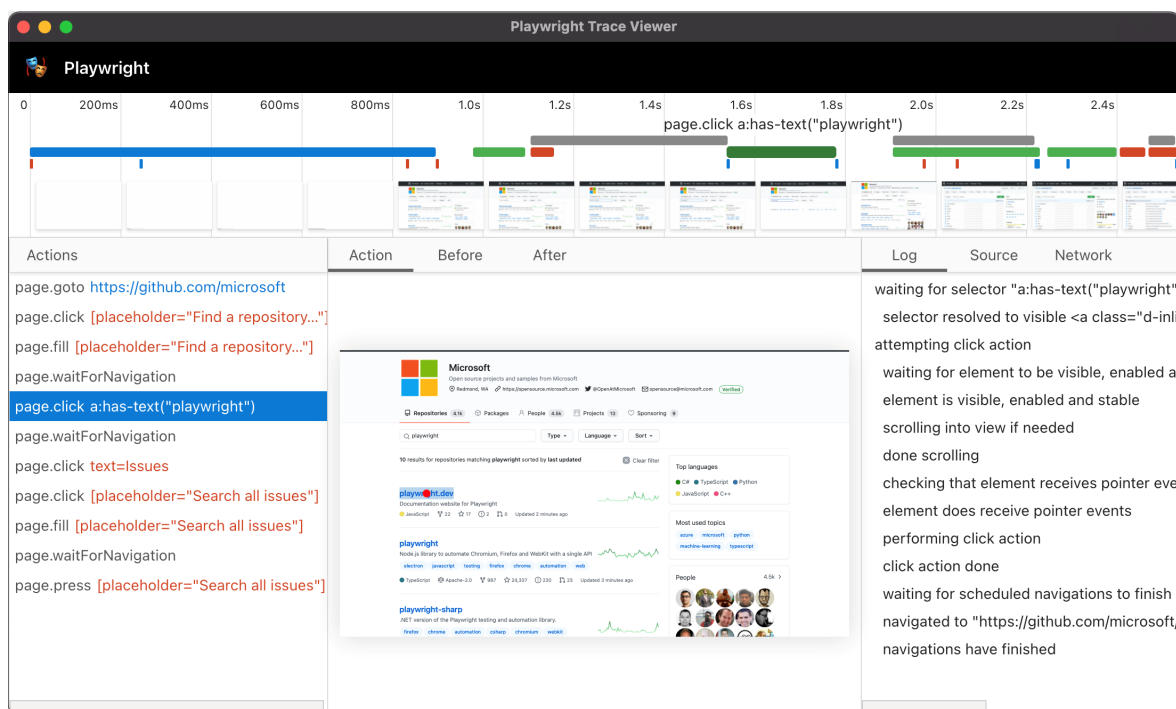
Cypress je založen pouze na JavaScriptu, tudíž je potřeba nutná znalost tohoto programovacího jazyku. Nástroj neumožňuje používat více záložek v jednom testovacím webovém prohlížeči a s tím související používání více webových oken v jednom testovacím automatu. Cypress nemá možnost automaticky generovat příkazy automatického skriptu a také nepodporuje testování mobilních aplikací [63].

5.4 Playwright

Nástroj Playwright od společnosti Microsoft je open-source nástroj pro psaní a spouštění automatických testů ve webovém uživatelském rozhraní a v nativních mobilních emulátorech. Jeho používání je zdarma a má dobrou uživatelskou podporu při potížích. Je založen na Node.js a za vytvořením stojí stejní vývojáři, kteří vyvinuli framework Puppeteer. Umožňuje spouštění automatických testů ve většině moderních webových prohlížečů a na všech operačních systémech. Implementaci automatických skriptů v nástroji Playwright lze pomocí několika programovacích jazyků (TypeScript, JavaScript, Python, .NET, Java). Nástroj implicitně poskytuje několik funkcí, díky kterým jsou automatické skripty stabilnější a spolehlivější. Test automaticky čeká, než dojde k zobrazení elementu na stránce pro následující akci s elementem. Má také bohatou sadu introspekčních událostí, které vedou k větší stabilitě testů. V době dynamických webů Playwright nabízí plno assertovacích metod, které se automaticky opakují, dokud nejsou splněny dané podmínky. Assertovací metoda porovnává a kontroluje očekávaný výsledek s aktuálním. V Playwright je implicitně zabudované opakování této kontroly při nesplnění podmínek, jelikož může docházet k nestabilitám mezi automatem a webovým prohlížečem [64].

5.4.1 Podporující nástroje

Playwright umožňuje generovat testovací kód v libovolném programovacím jazyce na základě akcí v nástroji Playwright Codegen. Uživatel ve webovém rozhraní klikne na element a Codegen vygeneruje tuto akci ve zdrojovém kódu zvoleného programovacího jazyka, který lze použít v automatickém skriptu. Playwright inspector poskytuje uživatelské rozhraní, ve kterém uživatel může hledat elementy na stránce, generovat selektory k elementům, procházet jednotlivé kroky po provedení testu a další funkce. Playwright Trace Viewer zachycuje všechny informace o provedení testu, jakmile dojde k jeho selhání. Zobrazuje všechny akce provedené v automatické skriptu, zobrazuje dynamické snímky HTML DOMu, zachycuje síťovou komunikaci a mnoho dalších funkcí, které pomáhají při prozkoumávání spadlého automatu [65].



Obrázek 5.2 Ukázka nástroje Trace Viewer [66]

5.4.2 Výhody

Playwright je nástroj, který poskytuje prostředí pro psaní stabilních a spolehlivých automatických testů, které lze spouštět ve většině moderních webových prohlížečů. Je multiplatformní a zdrojové kódy automatických skriptů lze psát v několika jazycích. Hlavní výhodou je, že poskytuje implicitně automatické čekání na elementy na webové stránce před provedením akce a assertovací metody několikrát opakují své vyhodnocování na dynamické webové stránce. To umožňuje lepší stabilitu a větší spolehlivost

automatů. Playwright vytváří jednotlivé kontexty prohlížeče pro každý test, to umožňuje izolaci a nezávislost testů. Playwright také nabízí další funkce, které zrychlují a zlepšují automatizaci. Pomocí Codegen uživatel dokáže rychle a snadno vytvořit nový kód a s pomocí nástroje Trace Viewer uživatel dokáže rychle zjistit problém u neprocházejícího testu [67].

5.4.3 Limitace

Playwright nepodporuje pouštění automatických skriptů ve webovém prohlížeči Internet Explorer 11 a starší verze prohlížeče Microsoft Edge. Další limitací je, že nelze spustit automaty na reálných mobilních zařízeních. Možností, jak obejít tuhle limitaci je použití mobilního emulátoru v prohlížeči [67].

5.5 Výběr řešení

Při výběru řešení pro automatické testování procesu přihlašování do služeb poskytující elektronické zdroje knihovny UTB, bylo bráno v potaz několik faktorů. Nástroj pro automatizaci měl být zdarma, jeho zdrojový kód měl být open-source a měl by umožnit vytvářet stabilní a spolehlivé automatické testy kontrolující grafické rozhraní webové aplikace. Implementace automatických skriptů by měla být co nejjednodušší, aby nové automaty mohl vytvářet i zaškolený pracovník knihovny. Testovací nástroj by měl být ověřený v praxi a měl by mít dobrou dokumentaci. Testy by měly být provozovány ve většině moderních webových prohlížečích a nástroj musí být spustitelný na linuxovém operačním systému.

Vybíral jsem z několika nástrojů, které jsou open-source, jejich používání je zdarma a snadno dostupné. Existuje mnoho frameworků pro automatické testování grafické webového rozhraní. Zanalyzoval jsem tři frameworky Selenium, Cypress a Playwright, které jsou osvědčenými nástroji pro automatizaci testovacích skriptů.

Selenium je velmi používaný nástroj pro automatické testování grafického rozhraní ve webovém prohlížeči. Umožňuje psát automaty v různých programovacích jazycích. Bohužel neposkytuje generátor automatického skriptu a je tedy nutná dobrá znalost programovacího jazyka. Jelikož Selenium nemá implicitně zabudované automatické čekání na přítomnost elementu na stránce, vyžaduje při psaní automatů explicitně vytvářet dodatečná čekání před provedení akce na elementu. Ne vždy se taková čekání snadno vytváří a také může dojít k přehlédnutí. To způsobuje nestabilitu a nepřesnost automatických skriptů. Selenium je velmi často používaný frameworkem pro psaní automatů, ale v případě vytváření testovací aplikace pro kontrolu elektronických zdrojů přes své limitace není vhodný nástroj.

Další nástroje pro automatické testování jsem vybral k analýze Cypress a Play-

wright. Oba dva jsou moderní nástroje, které mají automatické čekání na přítomnost elementu na stránce předtím, než se provede akce na elementu. Nástroje umožňují sekvenční i paralelní pouštění automatických testů v různých webových prohlížečích. Playwright je vyvíjen velkou společností Microsoft poskytující kvalitní dokumentaci a širokou uživatelskou podporu. I když je Cypress vyvíjen menší společností, také poskytuje detailní dokumentaci a má rozsáhlou komunitu lidí řešící různé problémy spojené s automatizací v Cypressu. Automaty v Cypressu jsou psané pouze pomocí programovacího jazyka Javascript, kdežto u Playwright si uživatel může vybrat z několika jazyků (TypeScript, JavaScript, Python, .NET, Java). Cypress i Playwright zachycují různé informace, které se můžou použít pro analýzu spadlého automatu. Jsou to především záznamy obrazovky, videozáznamy, dynamické zobrazení HTML DOMu a další informace spojené s během spadlého automatu.

Jelikož testovací aplikace se bude v budoucnu rozšiřovat o další nově přidané služby poskytující elektronické zdroje, důležitým faktorem pro zvolení automatizačního nástroje byla jednoduchost psaní nových automatických skriptů. Playwright poskytuje nástroj, díky kterému může zaškolený uživatel snadno generovat zdrojový kód automatu. Takový zdrojový kód dále vloží do testovací aplikace a automat pustit. Cypress žádný generátor kódu nemá. Po konzultaci s pracovníky technického oddělení knihovny UTB, kteří budou spravovat a vytvářet nové automaty v testovací aplikaci, byl zvolen nástroj Playwright pro kontinuální testování provozovaných služeb.

6 POUŽITÉ TECHNOLOGIE

Pro vytvoření testovací aplikace, spouštění automatických testů a následné zpracování výsledků jsem použil několik technologií. Technologie by měly být kompatibilní se všemi moderními webovými prohlížeči, zdarma a umožňuje jejich používání na linuxovém stroji. Jde zejména o automatizační framework Playwright a další funkce importovány přes plugin¹⁾ balíčky.

6.1 Playwright

Hlavní nástroj pro psaní, správu a spouštění automatických testů jsem zvolil open-source aplikaci Playwright. Důvody, proč byl zvolen tento framework, jsou sepsány v kapitole 5.5. Automatické skripty jsou psány programovacím jazykem Python. Python je interpretovaný a objektově orientovaný jazyk, který je multiplatformní a jednoduchý na naučení. Zdrojový kód v pythonu je přehledný a snadno čitelný. Tyhle všechny faktory ovlivnily výběr programovacího jazyka, ve kterém se budou psát automatické skripty v nástroji Playwright. V budoucnu se počítá s rozšiřováním automatických skriptů v testovací aplikaci, jelikož bude zapotřebí testovat nově přidávané služby poskytující elektronické zdroje. V programovacím jazyce Python by měli pracovníci knihovny UTB snad vytvořit a dále spravovat takové automatické skripty.

6.2 Pytest plugin

Pytest je open-source framework, který umožňuje snadno vytvářet automatické testy podporující komplexní funkční testování aplikací a knihoven. Automaticky detekuje testy v repozitáři testovací aplikace. Pomocí anotací můžeme testovací metody označit kategorií, případně celou testovací metodu přeskočit. Pomocí pytestu dokážeme spustit specifický test nebo testy patřící do stejné kategorie [68].

6.3 Pytest-XDist plugin

Plugin Pytest-XDist rozšiřuje pytest o další možnosti spouštění testů. Nejpoužívanější technika je distribuce testů do více procesů, což poskytuje rychlejší provádění testů. Uživatel může předem určit, kolik procesů má být k dispozici při spouštění testů anebo je zde možnost automatického přiřazení procesů podle výkonu počítače.

¹⁾Označení pro balík funkcí, které nepracují samostatně, ale různým způsobem rozšiřují funkcionality aplikace

6.4 Flaky plugin

Flaky plugin je nástroj pro automatické spouštění flaky automatických testů. Flaky je označení pro nestabilní test, který vrací náhodně pozitivní a negativní výsledky, aniž by se zdrojový kód aplikace anebo testu změnil. Díky nástroji Flaky můžeme anotovat takové testovací metody, které jsou nestabilní. Při neúspěchu testu se takový test použít několikrát, dokud nevrátí úspěšný výsledek [69].

6.5 Yagmail plugin

Yagmail plugin je rozšíření, díky kterému lze snadno posílat elektronickou poštu. Je možnost využít klienta pro Gmail službu od společnosti Google anebo klienta SMTP. Pro testovací aplikaci byl vytvořen účet na Gmailu a emaily se posílají přes klienta Gmail v pluginu YagMail. Pro Yagmail posílá textové emaily, emaily s HTML obsahem a také emaily s libovolnou přílohou.

Summary

173 tests ran in 2638.26 seconds.

(Un)check the boxes to filter the results.

☒ 146 passed, ☒ 96 skipped, ☒ 27 failed, ☐ 0 errors, ☐ 0 expected failures, ☐ 0 unexpected passes

Results

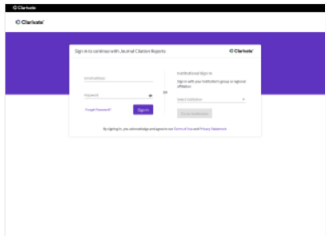
Show all details / Hide all details

Result	Test	Duration	Links
Failed (show details)	Tests/Services/academic_search_complete_ebsco_test.py::AcademicSearchComplete::test_proxy_ldap	0.82	
Failed (show details)	Tests/Services/academic_search_complete_ebsco_test.py::AcademicSearchComplete::test_proxy_shibboleth	0.78	
Failed (show details)	Tests/Services/academic_search_complete_ebsco_test.py::AcademicSearchComplete::test_wayf	0.84	
Failed (show details)	Tests/Services/aip_publishing_journal_of_chemical_physics_test.py::JournalOfChemicalPhysics::test_wayfless	30.88	
Failed (show details)	Tests/Services/aip_publishing_physics_of_fluids_test.py::PhysicsOfFluids::test_wayfless	30.90	
Failed (show details)	Tests/Services/aip_publishing_the_journal_rheology_test.py::TheJournalRheology::test_wayfless	30.87	
Failed (show details)	Tests/Services/bookport_test.py::Bookport::test_wayf	22.09	
Failed (hide details)	Tests/Services/journal_citation_reports_test.py::JournalCitationReports::test_proxy_ldap	32.37	


```

done, _ = await asyncio.wait(
    {
        self._connection._transport.on_error_future,
        callback.future,
    },
    return_when=asyncio.FIRST_COMPLETED,
)
if not callback.future.done():
    callback.future.cancel()
result = next(iter(done)).result()
> playwright._impl._api_types.TimeoutError: Timeout 30000ms exceeded.
===== logs =====
E   waiting for selector "//*[@id="InstLogoTa-0"]"
=====
C:\Users\pospe\AppData\Local\Programs\Python\Python39\lib\site-packages\playwright\_impl\_connection.py:63:
TimeoutError

```



Obrázek 6.1 Ukázka výstupu HTML pluginu

6.6 Pytest-Html plugin

S pomocí pluginu pytest-html testovací aplikace dokáže shromáždit výsledky po běhu automatických skriptů a vygenerovat HTML dokument s výsledky testů. HTML stránka umí zobrazit a filtrovat počet úspěšných, neúspěšných a přeskočených testů. U každého

spadlého automatu zobrazí chybu a záznam obrazovky. Ukázka takové HTML souboru je na obrázku 6.1.

6.7 Git

Git je verzovací nástroj, který slouží pro ukládání a správu zdrojového kódu aplikace. Umožňuje sledovat změny v kódu programu. Díky nástroji Git může vývojář verzovat svoji aplikaci, vracet zpět nežádoucí funkcionalitu a případně spolupracovat s dalšími vývojáři v rámci jedné aplikace.

7 UŽIVATELSKÁ PŘÍRUČKA

Uživatelská příručka slouží jako návod pro uživatele, ve kterém je popsána konfigurace prostředí a jak testy spustit.

7.1 Konfigurace prostředí

Před samotným spuštěním automatických testů musí uživatel nakonfigurovat prostředí, ve kterém testy budou běžet. Pro správné fungování testovací aplikace uživatel musí splnit všechny kroky, které jsou popsány v této kapitole.

7.1.1 Python

Základem před spuštěním automatických testů je mít nainstalován skriptovací jazyk Python. Na oficiálních stránkách <https://www.python.org/downloads> uživatel stáhne instalační soubor a nainstaluje ho na zařízení, ve kterém budou testy spuštěny. Po nainstalování skriptovacího jazyka si uživatel může ověřit verzi přes příkaz `python --version`. Testovací aplikace byla vyvíjena na verzi Pythonu 3.9.6 a je kompatibilní od této verze výše.

7.1.2 Knihovny jazyka Python

V projektu testovací aplikace uživatel najde soubor s názvem `requirements.txt`. Soubor obsahuje seznam knihoven s jejich verzemi, které jsou nezbytně nutné pro správné chování testovací aplikace. Knihovny se mohou nainstalovat jednotlivě standardním příkazem v Pythonu `pip install [název_pluginu]`. Další možností je nainstalovat knihovny najednou přes příkaz `pip install -r requirements.txt`, ale příkaz musí být provolán na stejné cestě jako se nachází soubor se seznamem knihoven.

7.1.3 Konfigurační soubor `config.ini`

Testovací projekt obsahuje konfigurační soubor, pomocí kterého uživatel může nastavovat různé parametry pro automatické testy. Na prvních řádcích konfiguračního souboru jsou autentizační parametry, pod kterými se budou automatické testy přihlašovat do služeb poskytující elektronické zdroje. Dalším parametrem je URL adresa, kterou automaty budou používat při přihlašování do služeb přes Proxy UTB. Předposledním parametrem v konfiguračním souboru lze přepínat mezi normálním režimem a režimem VPN. S režimem VPN se automatické testy spoléhají na připojení do služby z univerzitní IP adresy. Tudiž bez potřeby zadávání přihlašovacích údajů a samotného přihlášení do služby. Poslední konfigurační nastavení je o povolení/zakázání notifikací

```
1  [USER]
2    name = t1_pospisil
3    password = ***
4
5  [PROXY]
6    proxy_url = https://proxy.k.utb.cz/login?url=
7
8  [VPN]
9    isEnabled = False
10
11 [NOTIFICATION]
12   isEnabled = True
13   email = pospecnik95@gmail.com
14
```

Obrázek 7.1 Ukázka konfiguračního souboru config.ini

elektronickou poštou s výsledky automatických testů. Při povolení notifikací uživatel vyplní emailovou adresu, na kterou se výsledky automatů budou posílat.

7.2 Spouštění testů

Spouštění automatických testů je za pomoci pluginu `pytest`. Uživatel se naviguje do složky projektu s testovací aplikací. Než uživatel spustí automatické skripty, je potřeba vyplnit parametry v konfiguračním souboru `config.ini`, který je popsán výše.

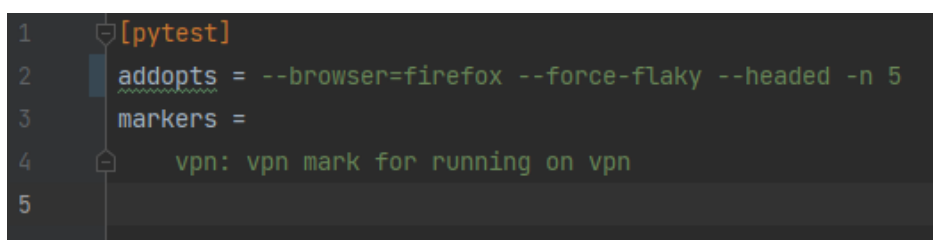
Je několik možností, jak testy spouštět. Hromadné spuštění testů začne po zavolání obecného příkazu `python -m pytest` v kořenovém adresáři testovací aplikace. Spustí se všechny testy, který mají předponu anebo koncovku `_test` v názvu třídy automatického testu. Pokud uživatel chce spustit konkrétní automatický skript, obohatí příkaz `python -m pytest` názvem daného automatu a cestou k němu. Příkaz s názvem konkrétního automatu je na obrázku 7.2. Automaty jsou rozdělené ve složkách a více o infrastruktuře projektu testovací aplikace je popsáno v kapitole 8.2.

```
python -m pytest Tests/Services/academic_search_complete_ebsco_test.py
```

Obrázek 7.2 Příkaz pro spuštění konkrétního testu

7.3 Parametry spouštění a pytest.ini

Testovací aplikace umožňuje spouštět automatické testy v různých režimech a s různým nastavením. Tyto režimy a nastavení se provádějí přidáním parametrů do příkazového řádku při spuštění aplikace. Příkaz, který je na obrázku 7.4, je obohacen o několik argumentů pro nastavení a spuštění testů. Jelikož pro uživatele je náročné vyplňovat tyto argumenty při každém spuštění, plugin pytest umožňuje tyto argumenty uložit do souboru `pytest.ini`. Tyto argumenty budou načteny ze souboru a použity při zavolání obecného příkazu `python -m pytest`. Argumenty se vyplňují ručně k parametru `addopts` (viz. obrázek 7.3). Dalším parametrem v souboru `pytest.ini` testovací aplikace je `markers`, kterým jsou označeny automaty běžící v režimu VPN.



```
1 [pytest]
2 addopts = --browser=firefox --force-flaky --headed -n 5
3 markers =
4     vpn: vpn mark for running on vpn
5
```

Obrázek 7.3 Ukázka souboru `pytest.ini`

7.3.1 Parametry pro pytest

Samotný plugin pytest nabízí mnoho parametrů, které lze použít v příkazové řádce při spouštění testů. Prvním důležitým parametrem je `--headed`, který umožňuje pustit testy v grafickém okně. Bez uvedení tohoto parametru testy běží v headless režimu, ve kterém jsou spuštěny na pozadí bez zobrazení grafického rozhraní webového prohlížeče. Dalším parametrem je `--browser`, ve kterém uživatel specifikuje webový prohlížeč pro automatické testy. Hodnoty jsou `chromium`, `firefox` anebo `webkit`. Existují další příkazové parametry, díky kterým uživatel může ovlivnit běh testů a jejich výstup. Například testy se můžou spustit ve zpomaleném režimu, pro každý test se může zaznamenávat video jeho provádění, případně záznam obrazovky při neúspěchu a další možnosti.

```
python -m pytest --headed --browser=firefox --force-flaky -n 5
```

Obrázek 7.4 Příkaz s různými parametry

7.3.2 Parametr pro flaky

Testovací aplikace obsahuje plugin flaky, díky kterému lze označit nestabilní automaty. Takto označené automaty se v případě selhání pustí několikrát. K označení automatů se používají dekorátory v Pythonu. V případě pluginu flaky vypadá dekorátor takhle `@flaky(max_runs=3)`. V tomto případě se automat spustí opakovaně maximálně třikrát, pokud bude automat nestabilní. V testovací aplikaci nejsou dekorátory označeny žádné automaty, ale ověřená metoda je spustit testy s globálně nastaveným atributem `--force-flaky`. Takto jsou všechny testy považovány za nestabilní a v případě selhání se test pustí znova. K nestabilitě může dojít například zpomalením internetové sítě anebo jiných okolností, které dopředu neznáme.

7.3.3 Parametr pro xdist

Díky pluginu xdist je uživatel schopen pustit testovací sadu paralelně. Tím se ušetří čas na provedení všech testů a k dosažení rychlejší zpětné vazby od služeb poskytující elektronické zdroje. Pro paralelní spuštění uživatel zadá do příkazové řádky atribut `-n` a hodnotu `auto` anebo konkrétní počet procesů. V případě `-n auto` režimu plugin vytvoří počet procesů podle výkonu počítače, na kterém jsou testy provozovány. Procesy jsou oddělené a v jednom procesu může běžet právě jeden automat. Po skončení testování se výsledky automatů z každého procesu shromáždí a uloží jako jeden celek.

7.4 Výsledky testů a `report_sent.txt`

Po každém běhu testovací sady jsou výsledky testů shromážděny a vyexportovány do HTML souboru. V projektu jsou HTML soubory s výsledky uloženy ve složce `htmlReports`. Jakmile automatický test narazí na chybu při svém běhu, vyfotí aktuální zobrazenou stránku na obrazovce a uloží v podobě obrázku. Záznamy obrazovky neúspěšných automatů jsou uloženy v testovacím projektu ve složce `screenshots`. Tyto záznamy obrazovky se nacházejí i v HTML reportu s výsledky automatů. Ukázka HTML souboru je na obrázku 6.1.

Testovací aplikace obsahuje soubor `report_sent.txt`, ve kterém se ukládá seznam spadlých automatů. Upozorňující email s výsledky automatických skriptů je poslán, jakmile je do souboru se seznamem spadlých automatů uložen nový záznam o neúspěšném automatu. Pokud v aktuálním testovacím běhu spadnou automaty, které jsou už obsažené v souboru `report_sent.txt`, email s výsledky se nepošle. Uživatel v tomto případě bude upozorňován pouze na nově neúspěšné automaty. Tento mechanismus předchází zahlcení emailové schránky adresáta. Uživatel může soubor ručně upravovat.

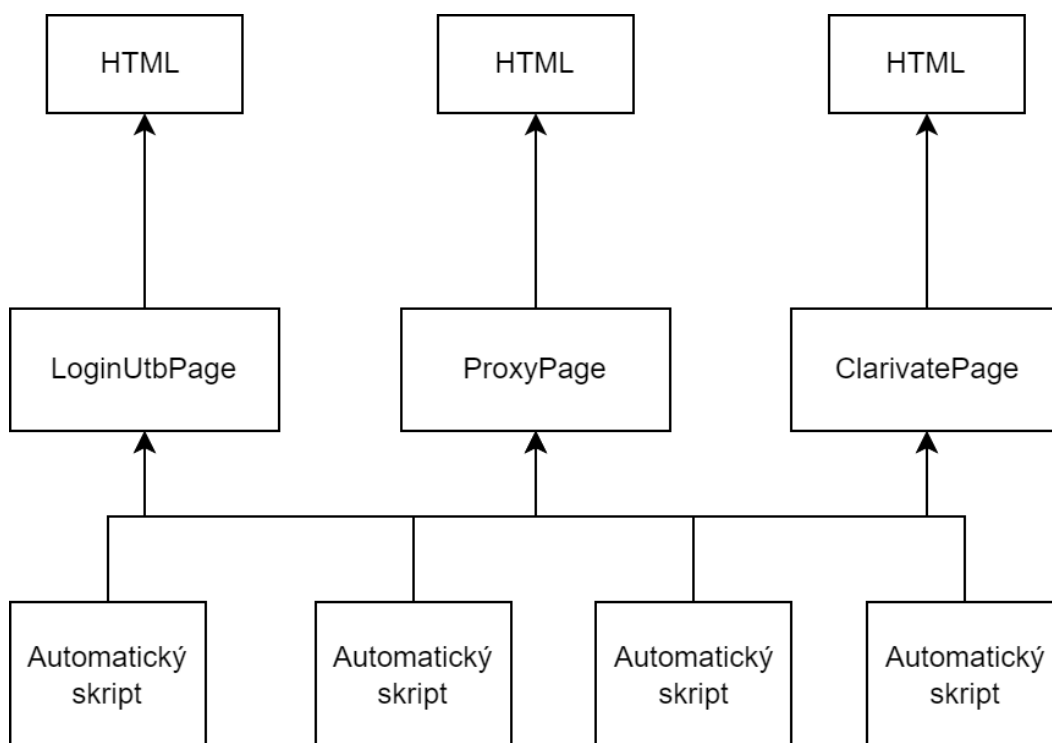
8 PROGRAMÁTORSKÁ PŘÍRUČKA

V programátorské příručce rozeberu infrastrukturu projektu. Jaké složky a soubory testovací aplikace obsahuje a popíšu z jakých částí se implementačně skládá automatický skript. Uvedu zde i návod, jak rozšířit testovací aplikaci o další automaty, které budou testovat nově přidáné služby.

8.1 Návrhový vzor Page Object Model

V testovací aplikaci je použit návrhový vzor Page Object Model (POM). Je jedním z populárních vzorů, které se používají při automatizaci testů kontrolující grafické rozhraní softwaru. Primárním cílem použití tohoto vzoru je vyhnout se duplicitě zdrojového kódu a zlepšit znovupoužitelnost napsaného kódu. To zvyšuje přehlednost, údržbu automatických testů a snadnější psaní nových automatů.

Automatické testy se navigují na různé webové stránky v prohlížeči. Pomocí vhodných lokátorů jsou identifikovány elementy na HTML stránce, se kterými se provádějí různé akce. S využitím návrhového vzoru POM tyto elementy v rámci jedné HTML stránky zabalí do jediného objektu. Elementy v objektu jsou zapouzdřeny a interakce jsou prováděny přes metody objektu. Jakmile se automatický test naviguje na webovou stránku, vytvoří se nová instance objektu reprezentující danou stránku. Přes tuto instanci automat provádí interakce se HTML stránkou [70].



Obrázek 8.1 Schéma použití návrhového vzoru POM

8.1.1 Důvod použití

Při automatizaci testů grafického rozhraní se automat naviguje na různé webové stránky a používá pro ovládání stránky různé grafické prvky (tlačítka, textová pole, přepínače a další). Pro identifikování a následnou akci s elementy se používají lokátory. Neustálé psaní lokátorů elementů v každém automatu zvyšuje náročnost na správu takového automatu, dochází k duplicitním použití lokátorů a celkově zvyšuje náklady na údržbu. Při změně grafického uživatelského rozhraní aplikace by musely být lokátory přepsány na všech místech a to by opět zvýšilo časovou náročnost na opravu automatů. Tyto důvody vedly k použití návrhového vzoru Page Object Model, kdy elementy s lokátory jsou zapouzdřeny v objektu na jednom místě a pomocí funkcí dochází k interakci s elementy. Elementy v objektu lze volat v různých testovacích automatech, tím se zvyšuje znovupoužitelnost zdrojového kódu a nedochází k psaní duplicitních lokátorů pro elementy. Jelikož jsou objekty s lokátory odděleny od automatů, automatické testy se stávají přehlednějšími a je snadnější čtení kódu [70].

8.2 Infrastruktura projektu

V kořenovém adresáři testovacího projektu se nacházejí soubory:

- `requirements.txt` (7.1.2)
- `pytest.ini` (7.3)
- `config.ini` (7.1.3)
- `report_sent.txt` (7.4)
- `conftest.py`

`Conftest.py` je skriptovací soubor, ve kterém jsou definované metody provádějící se před anebo po spuštění testovací sady. Před spuštěním testů se ve funkci `read_report_file()` načítá seznam automatů ze souboru `report_sent.txt`. Po skončení automatů se porovnává seznam načtených automatů se seznamem neúspěšných automatů z aktuálního běhu. Proveďte se množinová operace rozdíl mezi těmito seznamy. Pokud v seznamu právě neúspěšných automatů přebývají záznamy, provede se emailová notifikace právě s těmito přebývajících automaty. V emailu se nachází příloha typu ZIP, která je vytvořena funkcí `create_zip_file_with_report()`. ZIP soubor obsahuje HTML s výsledky a záznamy obrazovky spadlých automatů.

```
def check_inner_text(self, selector, text):
    self.test.assertTrue(self.utils.is_exist(selector, 20000),
                          f"Element with {text} is not exist.")
    it = 0
    while it < 5 and text not in self.page.inner_text(selector).rstrip():
        self.page.wait_for_timeout(2000)
        it += 1
    self.test.assertIn(text, self.page.inner_text(selector))
```

Obrázek 8.2 Metoda pro kontrolu textu

8.3 Složka Helpers

Jak již název napovídá, složka **Helpers** obsahuje skriptovací soubory, které pomáhají při práci s daty, obsahuje pomocné assertovací funkce a další.

8.3.1 Asserts.py

Třída **Asserts** obsahuje funkce, které slouží pro efektivnější kontrolování datových prvků ve webovém prohlížeči. Funkce v třídě:

- `is_exists(self, selector, message, timeout=20000)` - slouží pro kontrolu, jestli element existuje na stránce
- `check_inner_text(self, selector, text)` - verifikuje, jestli element obsahuje text, který je předán v parametru
- `check_inner_html(self, selector, html)` - kontroluje, jestli element obsahuje HTML kód, který je předán v parametru

8.3.2 Data.py

Třída **Data** zpracovává a ukládá informace, které byly načteny z konfiguračního souboru `config.ini`. Informace se týkají přihlašovacích údajů pro uživatele, nastavení notifikací, proxy a režimu VPN. Dalším atributem ve třídě **Data** jsou informace o elektronických zdrojích, které jsou dostupné na URL adrese `https://www-new.k.utb.cz/eresources-list.php`. Třída obsahuje funkce, které jsou potřebné pro správnou manipulaci a uložení takových dat.

8.3.3 Utils.py

Třída **Utils** obsahuje pouze pomocné funkce, které jsou často volány a zlepšují tím přehlednost zdrojového kódu.

8.4 Složka Pages

Složka obsahuje tři třídy, které představují konkrétní stránky ve webovém prohlížeči. Každá třída obsahuje selektory k vyhledávání elementů na stránce. Třídy také obsahují funkce, které s těmito elementy pracují. Jedná se o použití návrhového vzoru Page Object Model (8.1), ve kterém je webová stránka v prohlížeči reprezentována třídou s atributy ve zdrojovém kódu.

8.4.1 LoginUtbPage.py

Třída `LoginUtbPage` představuje webovou stránku pro přihlášení do služby Shibboleth. Jednotlivé atributy jsou vytvořeny na základě selektorů k elementům na webové stránce. Atributy jsou:

- `login_username_input(self)` - atribut, který slouží pro lokalizaci textového pole pro zadání přihlašovacího uživatelského jména
- `login_password_input(self)` - atribut, který slouží pro lokalizaci textového pole pro zadání přihlašovacího uživatelského hesla
- `login_button(self)` - atribut, který lokalizuje tlačítko pro přihlášení
- `gdpr_confirm_button(self)` - atribut, který lokalizuje tlačítko pro souhlas se zpracováním osobních údajů

Funkce třídy `LoginUtbPage`:

- `login_and_accept_gdpr(self)` - funkce, která za použití atributů uživatelského jména a hesla vyplní textová pole pro přihlášení a potvrdí akci kliknutím na tlačítko `Přihlásit se`

8.4.2 ProxyPage.py

Třída slouží pro reprezentaci webové stránky, která je používána pro přihlašování uživatele přes PROXY UTB. Atributy jsou:

- `shibboleth_button(self)` - atribut reprezentující tlačítko pro přihlášení přes Shibboleth
- `ldap_button(self)` - atribut reprezentující tlačítko pro přihlášení přes LDAP
- `login_ldap_button(self)` - atribut reprezentující tlačítko pro potvrzení přihlášení

- `login_ldap_username_input(self)` - atribut reprezentující textové pole pro zadání uživatelského jména
- `login_ldap_password_input(self)` - atribut reprezentující textové pole pro zadání uživatelského hesla

Funkce třídy `ProxyPage`:

- `navigate(self, link_native_home)` - funkce, která se naviguje na webovou stránku přes URL adresu. URL adresa je vytvořena spojením PROXY URL s URL dané služby
- `login_shibboleth(self)` - funkce provádějící přihlášení přes Shibboleth
- `login_ldap(self)` - funkce provádějící přihlášení přes LDAP

8.4.3 `ClarivatePage.py`

Existuje pár služeb s elektronickými zdroji, do kterých se přihlašuje přes poskytovatele Clarivate. Ve třídě `ClarivatePage` jsou elementy a funkce, které s takovou webovou stránku pracují. Atributy jsou:

- `select_institution_combobox(self)` - element, pomocí kterého se vybírá typ přihlášení
- `czech_academic_value(self)` - element představující hodnotu v kombinovaném poli
- `go_to_institution_button(self)` - atribut reprezentující tlačítko pro přihlášení
- `bata_input(self)` - atribut reprezentující textové pole

Funkce třídy `ClarivatePage`:

- `login_via_clarivate(self)` - metoda, která vybere typ přihlášení, vyplní hodnoty a klikne na tlačítko pro přihlášení
- `assert_login(self)` - metoda, která kontroluje úspěšné přihlášení do služby

8.5 Složka Tests

Složka `Tests` obsahuje:

- Složku `Services` - obsahuje automatické testy, které testují dostupnost přihlášení do služby poskytující elektronické zdroje
- Test `CheckListOfServices()` - test, který kontroluje seznam aktuálních elektronických zdrojů získaných z knihovny UTB se seznamem služeb, které jsou pokryté automatickými testy. Jakmile dojde ke změně seznamu služeb Knihovny UTB, automat zahlásí chybu.
- Bázovou třídu `TestBase(unittest.TestCase)` - bázová třída slouží umístění společných funkcí a elementů, které jsou využívány v jednotlivých automatech testující přihlášení.

8.5.1 TestBase.py

Bázová třída, ze které dědí každý automatický test. Obsahuje funkce a atributy, které jsou využívány všemi automaty. Funkce jsou:

- `setup(self, page, source_id)` - jde o funkci, která se provolá před spuštěním automatického testu. Jejím účelem je načtení dat a příprava prostředí pro automat.
- `get_e_source(self, source_id)` - metoda, která na základě identifikátoru elektronického zdroje načte informace o daném zdroji
- `wayfless(self)` - funkce, která obsahuje společnou funkcionalitu pro testování přihlášení způsobem WAYFless
- `proxy_shibboleth(self)` - funkce, která obsahuje společnou funkcionalitu pro testování přihlášení přes PROXY způsobem Shibboleth
- `proxy_ldap(self)` - funkce, která obsahuje společnou funkcionalitu pro testování přihlášení přes PROXY způsobem LDAP
- `wayf(self)` - funkce, která obsahuje společnou funkcionalitu pro testování přihlášení způsobem WAYF
- `vpn(self)` - funkce, která obsahuje společnou funkcionalitu pro testování přihlášení přes VPN.

8.5.2 Automatický test

V rámci jednoho automatického testu se testuje jedna služba, která poskytuje elektronické zdroje. Každý elektronický zdroj obsahuje jedinečný identifikátor (například `source_id=UTB00015`), pod kterým je uveden na portále Knihovny UTB. Identifikátor je ručně přidán každému automatu, aby došlo k rozpoznání elektronického zdroje. Automat je složen z funkcí:

- `setup(self, page: Page)` - jde o funkci, která se provolá před spuštěním automatického testu. Metoda v sobě obsahuje volání `setup(self, page, source_id)` z bazové třídy. V automatu je předán pouze parametr `source_id`.
- `test_wayfless(self)` - testovací metoda, která volá funkci `wayfless(self)` z bazové třídy a verifikuje úspěšné přihlášení do služby způsobem WAYFless
- `test_proxy_shibboleth(self)` - testovací metoda, která volá funkci `proxy_shibboleth(self)` z bazové třídy a verifikuje úspěšné přihlášení do služby přes PROXY s možností Shibboleth
- `test_proxy_ldap(self)` - testovací metoda, která volá funkci `proxy_ldap(self)` z bazové třídy a verifikuje úspěšné přihlášení do služby přes PROXY s možností LDAP
- `test_wayf(self)` - testovací metoda, která volá funkci `wayf(self)` z bazové třídy a verifikuje úspěšné přihlášení do služby způsobem WAYF
- `test_vpn(self)` - testovací metoda, která volá funkci `wayf(self)` z bazové třídy a verifikuje úspěšné přihlášení do služby přes VPN
- `assert_access(self)` - assertovací funkce, která obsahuje kontrolu přístupu na službu po úspěšném přihlášení

8.6 Jak rozšířit testovací aplikaci o další testy

Testovací aplikace je navržena tak, aby vytvoření nového automatického testu dokázal i zaškolený pracovník knihovny UTB. V případě přidání nové služby poskytující elektronické zdroje je uživatel schopen vytvořit snadno nový automatický skript pro tuto službu. Automat bude přidán do testovací sady a spouštěn stejným způsobem jako dosavadní automaty.

8.6.1 Vytvoření souboru

Prvním krokem k vytvoření nového automatu je vytvořit nový skriptovací soubor s koncovkou `.py` ve složce `Test/Services`. Soubor je vhodné pojmenovat stejným názvem

```
class SpringerLink(TestBase):  
  
    @pytest.fixture(autouse=True)  
    def setup(self, page: Page):  
        super(SpringerLink, self).setup(page, "UTB00001")
```

Obrázek 8.3 Třída s metodou setup()

jako se jmenuje služba, která bude automaticky testována. Pro identifikaci automatu pluginem Pytest je nutné, aby název souboru začínal anebo končil slovem `test`. Například název souboru pro automat testující službu Springer Link je `springer_link_test.py`.

8.6.2 Vytvoření třídy s metodou setup()

Do nově vytvořeného skriptovacího souboru uživatel definuje třídu, která bude mít název testované služby. Třída bude dědit z bazové třídy `TestBase` (8.5). Uživatel začne implementací první metody s názvem `setup`, která provádí nastavení automatu. Uživatel pomocí dekorátoru `@pytest.fixture(autouse=True)` označí tuto metodu, aby se volala jako první před spuštěním automatu. Metoda `setup` obsahuje pouze volání stejnojmenné metody z bazové třídy, ve které je pomocí parametru předáno `source_id`. Hodnota `source_id` obsahuje identifikátor, kterým je označena každá služba na portále e-zdroje knihovny UTB.

8.6.3 Vytvoření testovacích metod

Automat testuje různé typy přihlášení do služby, která poskytuje elektronické zdroje. Ne všechny služby podporují všechny typy přihlášení (WAYF, WAYFless, Proxy). Uživatel v automatu definuje takové testovací metody, které jsou validní pro danou službu. Případně může využít dekorátor `@pytest.mark.skip(reason="not working in WAYF")` pro přeskočení testovací metody v testovací sadě.

U služby, která podporuje všechny typy přihlášení, uživatel definuje všechny testovací metody (8.5.2). Testovací metody `test_wayfless`, `test_proxy_shibboleth` a `test_proxy_ldap` obsahují pouze volání stejnojmenných metod z bazové třídy a kontrolující metodu úspěšného přihlášení.

```
def test_wayfless(self):  
    super(SpringerLink, self).wayfless()  
    self.assert_access()
```

Obrázek 8.4 Testovací metoda pro typ přihlášení WAYFless

```
def test_proxy_shibboleth(self):  
    super(SpringerLink, self).proxy_shibboleth()  
    self.assert_access()
```

Obrázek 8.5 Testovací metoda pro typ přihlášení PROXY Shibboleth

```
def test_proxy_ldap(self):  
    super(SpringerLink, self).proxy_ldap()  
    self.assert_access()
```

Obrázek 8.6 Testovací metoda pro typ přihlášení PROXY LDAP

Testovací metoda `test_vpn` kontrolující přihlášení do služby přes VPN obsahuje opět volání базové metody, URL navigaci na webovou stránku služby a volání metody pro kontrolu přihlášení. Testovací metoda je označena dekorátorem `@pytest.mark.vpn`, který zařazuje test do kategorie VPN. Uživatel je potom schopný spustit automaty pouze s touthle kategorií.

```
@pytest.mark.vpn  
def test_vpn(self):  
    super(SpringerLink, self).vpn()  
    self.page.goto(self.e_source.link_native_home)  
    self.assert_access()
```

Obrázek 8.7 Testovací metoda pro přihlášení přes VPN

Nejsložitější pro uživatele je definice testovací metody `test_wayf`, protože průběh takového testu je velmi variabilní. Základ testovací metody je stejný a to volání stejnojmenné metody z базové třídy. Po navigaci automatu na webovou stránku musí automat provést přihlášení do služby. Bohužel obsah HTML stránky je jiný pro každou službu podporující WAYF a nelze definovat takové obecné lokátory pro elementy, které by platily ve všech službách. Uživatel tedy musí napsat lokátory a testovací kroky do automatu, aby docílil automatického přihlášení přes WAYF. Testovací kroky s lokátory může uživatel vytvořit sám anebo použít nástroj od Playwright pro generování příkazů (8.7). Ukázka testovací metody WAYF pro službu Springer Link je na obrázku 8.8.

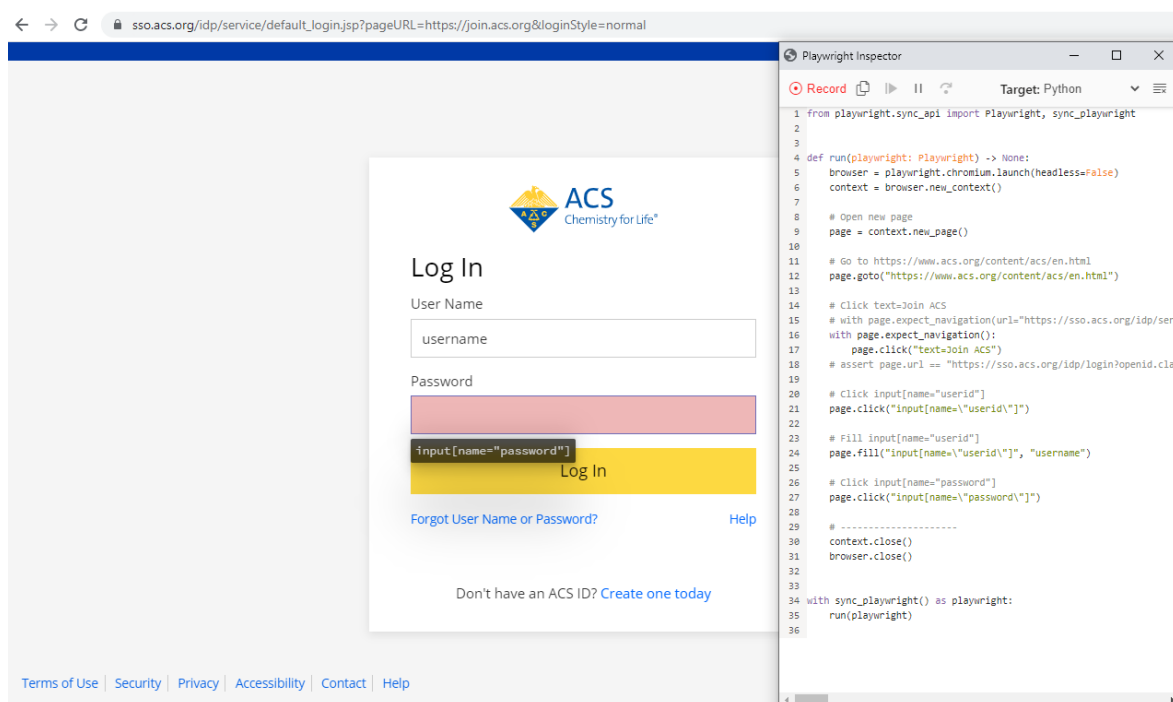
8.6.4 Vytvoření metody pro kontrolu přihlášení

Posledním krokem pro napsání automatického skriptu je definování metody, která bude provádět kontrolu přihlášení do služby. V testovací aplikaci je název této metody `assert_access` obsažen v každém automatu za účelem lepší čitelnosti zdrojového


```
def test_wayf(self):
    super(SpringerLink, self).wayf()
    self.page.goto(self.e_source.link_native_home)
    self.page.click("text=» Sign up / Log in")
    self.page.click("text=Access via your institution")
    self.page.fill("input[name=\"search\"]", "bata")
    self.page.press("input[name=\"search\"]", "Enter")
    with self.page.expect_navigation():
        self.page.click("text=Tomas Bata University in Zlín")
    login_page = LoginPage(self.page, self.data)
    login_page.login_and_accept_gdpr()
    self.assert_access()
```

Obrázek 8.8 Testovací metoda pro typ přihlášení WAYF do služby Springer Link

kódu. V těle metody je volána assertovací funkce, která kontroluje požadovaný výsledek s aktuálním. Většinou se jedná o kontrolu konkrétního textu, který je zobrazen po přihlášení. Příklad pro službu Springer Link je na obrázku 8.10.



Obrázek 8.9 Ukázka použití code generátoru

```
def assert_access(self):  
    self.asserts.check_inner_text("/*[@id=\"diagnostic-info\"]",  
        "the Library of Tomas Bata University in Zlin")
```

Obrázek 8.10 Kontrolující metoda pro službu Springer Link

8.7 Generátor testovacích příkazů

Playwright nabízí nástroj, pomocí kterého uživatel může generovat testovací příkazy do zdrojového kódu. V případě testovací aplikace uživatel musí generovat příkazy do programovacího jazyka Python. Uživatel v příkazové řádce zavolá příkaz spolu s webovou stránku, kterou má webový prohlížeč načíst. Příkaz je ukázán na obrázku 8.11. Nástroj otevře okno Playwright Inspector, ve kterém je vytvořen základní automatický skript. Spolu s tímto oknem je otevřen i webový prohlížeč, ve kterém je načtena webová stránka zadána v příkazu. Uživatel svými akcemi s webovou stránkou generuje zdrojový kód, který je postupně přidáván do automatu v oknu Playwright Inspector. Například pokud klikne uživatel na tlačítko nebo do textového pole, vygeneruje se kód v Pythonu `page.click("input[name=\"userid\"]")`. Takový kód je potom zkopírován do testovací aplikace do automatického skriptu. Pokud je spuštěn nástroj Playwright Inspector, uživatel při najetí myši na element také zjistí jeho lokátor, pod kterým lze daný element hledat na webové stránce. Ukázka použití nástroje Playwright Inspector a také zjištění lokátoru pro textové pole se nachází na obrázku 8.9.

```
npx playwright codegen https://www.acs.org/content/acs/en.html
```

Obrázek 8.11 Příkaz pro otevření code generátoru

ZÁVĚR

Ve spolupráci s knihovnou UTB byla vytvořena testovací aplikace se sadou automatických testů, které kontinuálně testují proces přihlašování do služeb poskytující elektronické zdroje. Seznam elektronických zdrojů je uveden na portále e-zdroje, který spravuje knihovna Univerzity Tomáše Bati ve Zlíně. Testy pro kontrolu přihlašování do služeb používají grafické uživatelské rozhraní ve webovém prohlížeči. Verifikují různé typy přihlášení, mezi které patří typ WAYF, WAYFless, PROXY a přihlášení pod univerzitní IP adresou přes VPN. Díky kontinuálnímu testování budou mít správci portálu e-zdroje přehled o dostupnosti jednotlivých zdrojů a v případě problémů budou moci rychle reagovat.

Automaty byly navrženy tak, aby jejich správa a případně rozšíření bylo co nejjednodušší. Byl vybrán open-source nástroj Playwright, který poskytuje podporu při automatizaci testů. Disponuje různými nástroji a pluginy, díky kterým je vytváření, spouštění a údržba automatů snadná. Testovací aplikace umožňuje pomocí konfiguračních souborů spustit automaty v různých režimech a v různých webových prohlížečích. Aplikace je také spustitelná ve všech operačních systémech.

Testovací sada automatů bude pravidelně spouštěna po určitých intervalech. Aplikace poběží na linuxovém stroji a plánování bude zajištěno linuxovým nástrojem cron. Po skončení běhu testovací sady budou výsledky automatů shromážděny a poslány elektronickou poštou. Email bude obsahovat seznam aktuálně neúspěšných automatů a bude připojena příloha v podobě ZIP souboru. ZIP soubor obsahuje výsledky automatů ve formě HTML spolu se záznamem obrazovky u neúspěšných testů.

SEZNAM POUŽITÉ LITERATURY

- [1] Software Testing Tutorial. *Javatpoint* [online]. 2022 [cit. 2022-04-10]. Dostupné z: <https://www.javatpoint.com/software-testing-tutorial>
- [2] Why software testing is important. *IBM* [online]. 2022 [cit. 2022-04-10]. Dostupné z: <https://www.ibm.com/topics/software-testing>
- [3] HAMILTON, Thomas. What is the need of Testing. *Guru99* [online]. 2022 [cit. 2022-04-10]. Dostupné z: <https://www.guru99.com/software-testing-introduction-importance.html>
- [4] SAWYER, Kathy. Mystery of Orbiter Crash Solved. *WashingtonPost* [online]. 1999 [cit. 2022-04-10]. Dostupné z: <https://www.washingtonpost.com/wp-srv/national/longterm/space/stories/orbiter100199.htm>
- [5] MEDEWAR, Sameeksha. Types of Software Testing. *HackrIo* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://hackr.io/blog/types-of-software-testing>
- [6] The types of testings. *SoftwareTestingMaterial* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://www.softwaretestingmaterial.com/non-functional-testing/>
- [7] HAMILTON, Thomas. What is Functional Testing. *Guru99* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://www.guru99.com/functional-testing.html>
- [8] GUPTA, Abhinav. Software Testing Evolution & Methodologies. *Webomates* [online]. 2021 [cit. 2022-04-12]. Dostupné z: <https://www.webomates.com/blog/software-testing/evolution-of-software-testing/>
- [9] PATEL, Bhaval. Software Development Process. *SpaceOTechnologies* [online]. 2022 [cit. 2022-04-13]. Dostupné z: <https://www.spaceotechnologies.com/blog/software-development-process/>
- [10] SMITH, Paul. Waterfall Model. *Wikipedia* [online]. 2022 [cit. 2022-05-13]. Dostupné z: [https://en.wikipedia.org/wiki/File:Waterfall_model_\(1\).svg](https://en.wikipedia.org/wiki/File:Waterfall_model_(1).svg)
- [11] UI a UX design: Co to je a jak je dělat správně. *MyTimi* [online]. 2021 [cit. 2022-04-15]. Dostupné z: <https://www.mytimi.cz/jak-na-ui-a-ux-design/>
- [12] KRÜGER, Nico. Agile Testing Methodology — 5 Examples for the Agile Tester. *Perforce* [online]. 2018 [cit. 2022-04-15]. Dostupné z: <https://www.perforce.com/blog/alm/what-agile-testing-5-examples>

- [13] PETERHERIA, Natalia. Why Agile SDLC Model Is The Best for Your Startup. *ProductTribe* [online]. 2018 [cit. 2022-05-13]. Dostupné z: <https://producttribe.com/project-management/agile-sdlc-guide>
- [14] SACOLICK, Isaac. What is CI/CD? Continuous integration and continuous delivery explained. *InfoWorld* [online]. 2022 [cit. 2022-04-15]. Dostupné z: <https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html>
- [15] ACARER, Cansin Cagan. CI/CD – Continuous Integration, Continuous Delivery, and Continuous Deployment. *Cansin* [online]. 2020 [cit. 2022-04-15]. Dostupné z: <https://cacarer.com/ci-cd-continuous-integration-continuous-delivery-and-continuous-deployment>
- [16] What is Continuous Delivery. *Amazon* [online]. 2022 [cit. 2022-04-15]. Dostupné z: <https://aws.amazon.com/devops/continuous-delivery/>
- [17] MINICK, Eric. Continuous Testing. *IBM* [online]. 2022 [cit. 2022-04-15]. Dostupné z: <https://www.ibm.com/cloud/learn/continuous-testing>
- [18] What is Cloud. *Microsoft* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://azure.microsoft.com/cs-cz/overview/what-is-the-cloud/>
- [19] What is open source software. *Opensource* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://opensource.com/resources/what-open-source>
- [20] What are some common CI/CD tools. *RedHat* [online]. 2018 [cit. 2022-04-16]. Dostupné z: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>
- [21] Best 14 CI/CD Tools You Must Know | Updated for 2022. *Katalon* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://katalon.com/resources-center/blog/ci-cd-tools>
- [22] HAMILTON, Thomas. Manual Testing Tutorial: What is, Concepts, Types & Tool. *Guru99* [online]. 2022 [cit. 2022-04-16]. Dostupné z: <https://www.guru99.com/manual-testing.html>
- [23] HAMILTON, Thomas. What is WHITE Box Testing. *Guru99* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.guru99.com/white-box-testing.html>
- [24] MEZQUITA, Ty. White Box Testing. *Cyberhhot* [online]. 2020 [cit. 2022-04-17]. Dostupné z: <https://cyberhoot.com/cybrary/white-box-testing/>

- [25] How to perform Manual Testing. *Javatpoint* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.javatpoint.com/manual-testing>
- [26] Black box testing. *Javatpoint* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.javatpoint.com/black-box-testing>
- [27] Black Box Testing Pros and Cons. *Imperva* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.imperva.com/learn/application-security/black-box-testing/>
- [28] HAMILTON, Thomas. What is Grey Box Testing. *Guru99* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.guru99.com/grey-box-testing.html>
- [29] Gray Box Testing. *Geeksforgeeks* [online]. 2019 [cit. 2022-04-17]. Dostupné z: <https://www.geeksforgeeks.org/gray-box-testing-software-testing/>
- [30] HAMILTON, Thomas. What is Exploratory Testing. *Guru99* [online]. 2022 [cit. 2022-04-17]. Dostupné z: <https://www.guru99.com/exploratory-testing.html>
- [31] HAMILTON, Thomas. What is User Acceptance Testing. *Guru99* [online]. 2022 [cit. 2022-04-18]. Dostupné z: <https://www.guru99.com/user-acceptance-testing.html>
- [32] Acceptance testing. *Javatpoint* [online]. 2022 [cit. 2022-04-18]. Dostupné z: <https://www.javatpoint.com/acceptance-testing>
- [33] GUPTA, Ayushi. What is User Acceptance Testing. *Dtlabs* [online]. 2021 [cit. 2022-05-13]. Dostupné z: <https://www.dtlabs.com/blog/what-is-user-acceptance-testing-828267>
- [34] HAMILTON, Thomas. Automation Testing. *Guru99* [online]. 2022 [cit. 2022-04-18]. Dostupné z: <https://www.guru99.com/automation-testing.html>
- [35] HAMILTON, Thomas. Automation Testing Vs. Manual Testing. *Guru99* [online]. 2022 [cit. 2022-04-20]. Dostupné z: <https://www.guru99.com/difference-automated-vs-manual-testing.html>
- [36] PEDAMKAR, Priya. Automation Testing Life Cycle. *Educba* [online]. 2022 [cit. 2022-04-18]. Dostupné z: <https://www.educba.com/automation-testing-life-cycle/>
- [37] Automation Testing. *BeyondKey* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://www.beyondkey.com/software-testing-and-qa-services>

- [38] Automation Testing. *Javatpoint* [online]. 2022 [cit. 2022-04-18]. Dostupné z: <https://www.javatpoint.com/automation-testing>
- [39] KINSBRUNER, Eran. The Testing Pyramid. *Perfecto* [online]. 2021 [cit. 2022-04-19]. Dostupné z: <https://www.perfecto.io/blog/testing-pyramid>
- [40] FOWLER, Martin. Test Pyramid. *MartinFowler* [online]. 2012 [cit. 2022-05-13]. Dostupné z: <https://martinfowler.com/bliki/TestPyramid.html>
- [41] Unit Testing. *Javatpoint* [online]. 2022 [cit. 2022-04-20]. Dostupné z: <https://www.javatpoint.com/unit-testing>
- [42] UNADKAT, Jash. What is Test Driven Development. *BrowserStack* [online]. 2021 [cit. 2022-04-20]. Dostupné z: <https://www.browserstack.com/guide/what-is-test-driven-development>
- [43] Why Test-Driven Development. *Marsner* [online]. 2021 [cit. 2022-05-13]. Dostupné z: <https://marsner.com/blog/why-test-driven-development-tdd/>
- [44] JUnit overview. *TutorialsPoint* [online]. 2022 [cit. 2022-04-21]. Dostupné z: https://www.tutorialspoint.com/junit/junit_overview.htm
- [45] What is NuGet. *MicrosoftDocs* [online]. 2022 [cit. 2022-04-21]. Dostupné z: <https://docs.microsoft.com/en-us/nuget/what-is-nuget>
- [46] NUnit. *DotnetPattern* [online]. 2022 [cit. 2022-04-21]. Dostupné z: <http://dotnetpattern.com/nunit-introduction>
- [47] What is integration testing. *SoftwareTestingHelp* [online]. 2022 [cit. 2022-04-22]. Dostupné z: <https://www.softwaretestinghelp.com/what-is-integration-testing/>
- [48] A Closer Look at Integration Testing. *QualityLogic* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://www.qualitylogic.com/2018/01/11/closer-look-integration-testing/>
- [49] Top 10 Integration Testing Tools To Write Integration Tests. *SoftwareTestingHelp* [online]. 2022 [cit. 2022-04-23]. Dostupné z: <https://www.softwaretestinghelp.com/integration-testing-tools/>
- [50] HAMILTON, Thomas. Performance Testing Tutorial. *Guru99* [online]. 2022 [cit. 2022-04-23]. Dostupné z: <https://www.guru99.com/performance-testing.html>
- [51] HAMILTON, Thomas. What is System Testing. *Guru99* [online]. 2022 [cit. 2022-04-23]. Dostupné z: <https://www.guru99.com/system-testing.html>

-
- [52] What is a Framework. *CodeInstitute* [online]. 2022 [cit. 2022-04-24]. Dostupné z: <https://codeinstitute.net/global/blog/what-is-a-framework/>
- [53] RAJORA, Harish. 13 Best Test Automation Frameworks. *LambdaTest* [online]. 2021 [cit. 2022-04-24]. Dostupné z: <https://www.lambdatest.com/blog/best-test-automation-frameworks-2021/>
- [54] System Testing. *Javatpoint* [online]. 2022 [cit. 2022-04-24]. Dostupné z: <https://www.javatpoint.com/system-testing>
- [55] GUPTA, Yogindernath. System Testing. *SoftwareTestingGenius* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://www.softwaretestinggenius.com/all-about-system-testing-an-important-part-of-our-software-testing-effort/>
- [56] What Is System Testing. *SoftwareTestingHelp* [online]. 2022 [cit. 2022-04-24]. Dostupné z: <https://www.softwaretestinghelp.com/system-testing/>
- [57] WAYF. *WAYF* [online]. 2022 [cit. 2022-04-24]. Dostupné z: <https://wayf.dk/en/how-use-wayf>
- [58] WAYFless Access to Resources. *TheChineseUniversity* [online]. 2020 [cit. 2022-04-24]. Dostupné z: <https://library.cuhk.edu.cn/article/164>
- [59] What is Shibboleth. *Shibboleth* [online]. 2022 [cit. 2022-04-24]. Dostupné z: <https://www.shibboleth.net/about-us/the-shibboleth-project/>
- [60] Selenium. *Javatpoint* [online]. 2022 [cit. 2022-04-30]. Dostupné z: <https://www.javatpoint.com/selenium-tutorial>
- [61] Cypress. *Cypress* [online]. 2022 [cit. 2022-05-01]. Dostupné z: <https://docs.cypress.io/guides/overview/why-cypress>
- [62] Cypress. *Cypress* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests>
- [63] UNADKAT, Jash. Cypress vs Selenium: Key Differences. *BrowserStack* [online]. 2022 [cit. 2022-05-01]. Dostupné z: <https://www.browserstack.com/guide/cypress-vs-selenium>
- [64] TIWARI, Garima. Playwright vs Selenium: A Comparison. *BrowserStack* [online]. 2021 [cit. 2022-05-01]. Dostupné z: <https://www.browserstack.com/guide/playwright-vs-selenium>

-
- [65] Powerful Tooling. *Playwright* [online]. 2022 [cit. 2022-05-01]. Dostupné z: <https://playwright.dev/>
- [66] Trace Viewer. *Playwright* [online]. 2022 [cit. 2022-05-13]. Dostupné z: <https://playwright.dev/docs/trace-viewer>
- [67] Why playwright. *Playwright* [online]. 2022 [cit. 2022-05-01]. Dostupné z: <https://playwright.bootcss.com/docs/why-playwright>
- [68] Pytest doc. *PytestDocs* [online]. 2022 [cit. 2022-05-02]. Dostupné z: <https://docs.pytest.org/en/stable/>
- [69] Flaky. *PypiOrg* [online]. 2022 [cit. 2022-05-02]. Dostupné z: <https://pypi.org/project/flaky/>
- [70] Page Object Model (POM). *LambdaTest* [online]. 2022 [cit. 2022-05-07]. Dostupné z: <https://www.lambdatest.com/blog/page-object-model-in-selenium-python/>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CD	Continuous delivery
CI	Continuous integration
CI/CD	Continuous integration and Continuous delivery
CT	Continuous testing
GUI	Graphical User Interface
IdP	Identity providera
lbf	Pound-force jednotka
POM	Page Object Model
UI	User interface
UTB	Univerzita Tomáše Bati ve Zlíně
UX	User Experience
WAYF	Where Are You From

SEZNAM OBRÁZKŮ

Obr. 1.1.	Rozdělení testování [6]	15
Obr. 1.2.	Životní cyklus vývoje aplikace [9]	17
Obr. 1.3.	Vodopádový model [10]	18
Obr. 1.4.	Cyklus agilního vývoje [13]	19
Obr. 1.5.	Kontinuální integrace a nasazení [15]	21
Obr. 2.1.	Schéma testování White box [24]	25
Obr. 2.2.	Schéma testování Black box [26]	26
Obr. 2.3.	Schéma testování Gray box [29]	27
Obr. 2.4.	Zařazení akceptačního testování do vývoje [33]	30
Obr. 3.1.	Schéma životního cyklu automatizace [37]	33
Obr. 3.2.	Testovací pyramida [40]	35
Obr. 3.3.	Schéma cyklu TDD [43]	37
Obr. 3.4.	Ukázka testu v JUnit	38
Obr. 3.5.	Integrační testování [48]	40
Obr. 3.6.	Schéma integrovaných modulů	40
Obr. 3.7.	Systémové testování [55]	42
Obr. 4.1.	Portál elektronických zdrojů	46
Obr. 4.2.	Přihlášení přes WAYF [57]	48
Obr. 5.1.	Cypress [62]	52
Obr. 5.2.	Ukázka nástroje Trace Viewer [66]	54
Obr. 6.1.	Ukázka výstupu HTML pluginu	58
Obr. 7.1.	Ukázka konfiguračního souboru config.ini	61
Obr. 7.2.	Příkaz pro spuštění konkrétního testu	61
Obr. 7.3.	Ukázka souboru pytest.ini	62
Obr. 7.4.	Příkaz s různými parametry	62
Obr. 8.1.	Schéma použití návrhového vzoru POM	64
Obr. 8.2.	Metoda pro kontrolu textu	66
Obr. 8.3.	Třída s metodou setup()	71
Obr. 8.4.	Testovací metoda pro typ přihlášení WAYFless	71
Obr. 8.5.	Testovací metoda pro typ přihlášení PROXY Shibboleth	72
Obr. 8.6.	Testovací metoda pro typ přihlášení PROXY LDAP	72
Obr. 8.7.	Testovací metoda pro přihlášení přes VPN	72
Obr. 8.8.	Testovací metoda pro typ přihlášení WAYF do služby Springer Link ...	73
Obr. 8.9.	Ukázka použití code generátoru	73
Obr. 8.10.	Kontrolující metoda pro službu Springer Link	74
Obr. 8.11.	Příkaz pro otevření code generátoru	74

SEZNAM PŘÍLOH

P I. CD/DVD

PŘÍLOHA P I. CD/DVD

Zde je stručný popis obsahu, který je dodržen v přiloženém CD/DVD.

doc/

- **source/** - zdrojový kód Latex a obrázky diplomové práce
- **diplomovaPrace.pdf** - text diplomové práce ve formátu PDF

src/

- zdrojové kódy testovací aplikace

readme.txt

- instrukce pro instalaci a spuštění automatických testů