Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Tvorba automatizovaných testů

Bc. Jakub Vévoda

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jakub Vévoda

Systémové inženýrství a informatika
Informatika

Název práce

Tvorba automatizovaných testů

Název anglicky

Creating automated tests

Cíle práce

Hlavním cílem diplomové práce je zhotovit automatizované testy pro desktopovou aplikaci vytvářenou zvolenou firmou. Dílčími cíli je provést porovnání manuálního testování a testování automatizovaného.

Metodika

Na základě studia odborných a vědeckých literárních zdrojů, prakticky získaných informací a znalostí budou navrženy metody testování, vytvořeny vlastní automatizované testy. K testování budou použity vybrané technologie – SikuliX, Ranorex, TestArchitect, Robot Framework, AutoIt. Na základě provedeného manuálního a automatizovaného testování budou porovnány zvolené varianty a provedno jejich vyhodnocení.

Doporučený rozsah práce

60 stran

Klíčová slova

software, test, automatizované, desktop, aplikace

Doporučené zdroje informací

BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

LEWIS, W E. – VEERAPILLAI, G. *Software testing and continuous quality improvement.* Boca Raton: Auerbach Publications, 2005. ISBN 0849325242.

SPILLNER, Andreas. Software testing foundations: a study guide for the certified tester exam : foundation level. 2014. ISBN 978-1-937538-42-2

STEPHENS, Matt a Doug ROSENBERG. Testování softwaru řízené návrhem. Brno: Computer Press, 2011. ISBN 978-80-251-3607-2.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

doc. Ing. Edita Šilerová, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 29. 7. 2020

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 21. 10. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 31. 03. 2021

Čestné prohlášení

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

V Praze dne 31.03.2021 _____

Poděkování

Rád bych touto cestou poděkoval vedoucí mé diplomové práce doc. Ing. Editě Šilerové, Ph.D. za odborné vedení práce a za možnost konzultací, kde mi byly poskytnuté cenné rady, které jsem využil pro svou diplomovou práci.

Dále bych chtěl poděkovat zadavatelům práce za možnost zpracovávat diplomovou práci v jejich firmě a dalším lidem, kteří mi během vypracovávání práce poskytovali užitečné rady, které jsem využil pro svou praktickou část diplomové práce

Tvorba automatizovaných testů

Abstrakt

Diplomová práce se zabývá tvorbou automatizovaných testů pro desktopovou aplikaci vyvíjenou zvolenou firmou. Automatizace desktopové aplikace je prováděna v programu SikuliX. Testy jsou navrženy a tvořeny pomocí již existujících testovacích scénářů. Práce pojednává o metodách testování, procesu testování a vysvětluje důležité pojmy a principy vztahující se k dané problematice. Dále se práce zabývá výhodami a nevýhodami automatizovaného testování oproti testování manuálnímu a porovnává rozdíly mezi testováním automatickým a manuálním.

Teoretická část práce se zabývá definicemi, metodami a principy dané problematiky.

Praktická část má více částí. První část se zabývá tvorbou automatizovaných testů. Druhá část porovnává vytvořené automatizované testy s testy manuálními.

Klíčová slova: testování, automatizované, manuální, aplikace, software, desktop, SikuliX, scénáře, výhody, nevýhody.

Creation of automated tests

Abstract

The diploma thesis deals with the creation of automated tests for a desktop application developed by a selected company. Desktop application automation is performed in the SikuliX program. Tests are designed and created using existing test scenarios. The thesis deals with testing methods, testing process and explains important concepts and principles related to the issue. Furthermore, the work deals with the benefits

and the disadvantages of automated testing over manual testing and compares the differences between automatic and manual testing.

The theoretical part of the thesis deals with definitions, methods and principles of the issue.

The practical part has several parts. The first part deals with the creation of automated tests. The second part compares the created automated tests with manual tests.

Keywords: testing, automated, manual, applications, software, desktop, SikuliX, scenarios, advantages, disadvantages.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V současnosti je vyvíjena spousta nových aplikací a informační technologie se používají takřka všude, například ve volnočasových aktivitách, ale i v odborných odvětvích. Je potřeba si uvědomit, že vytvořit webovou nebo desktopovou aplikaci není vůbec jednoduché. Vývoj stojí spoustu práce a úsilí. Aby vývoj aplikace nepřišel nazmar, je potřeba mít kvalitní zpětnou vazbu, kterou získáváme od zákazníků neboli uživatelů aplikace. Dříve než se aplikace vypustí na trh, je potřeba aplikaci pořádně otestovat.

Diplomová práce se věnuje testování software. V roli testera aplikací autor strávil téměř 3 roky. Poslední rok začal vyvíjet automatizované testy pro desktopové aplikace, aby ulehčil práci a čas ostatním testerům. Tyto testy fungují na principu rozpoznávání obrazu.

Diplomová práce je zpracována za použití odborné literatury, internetových zdrojů a dokumentací k příslušným použitým technologiím.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem práce je zhotovit automatizované testy pro desktopovou aplikaci, která je právě vyvíjena ve zvolené firmě. Následně bude v práci srozumitelně vysvětlen rozdíl mezi automatizovaným a manuálním testováním.

Dílčími cíli diplomové práce je porovnat rozdíly mezi automatizovaným testováním a manuálním testováním. Předpokládaný výsledek je, že automatizované testy ušetří čas i práci testerům.

2.2 Metodika

Teoretická část diplomové práce se zabývá teoretickými východisky pro automatizované a manuální testování.

První část diplomové práce je zaměřena na odbornou literaturu vztahující se k problematice automatizovaného testování. Budou vysvětleny základní pojmy vztahující se k problematice testování a popsány principy testování. Dále budou představeny principy manuálního testování a technologie určené k automatizaci testování. Bude popsáno, jaké testy se používají.

V praktické části bude provedena analýza požadavků na automatizované testy a následně na vývoj automatizovaných testů. Analýza se bude zabývat zjištěním požadavků, které mají automatizované testy splňovat. Při vývoji testů bude dbáno na dodržení všech požadavků zadaných vedením firmy. Po zprovoznění automatizovaných testů budou výsledky srovnány s požadovanými hodnotami a následně bude zhodnocen přínos automatizovaných testů porovnáním automatizovaných testů s testy manuálními.

3 Teoretická východiska

V této kapitole se diplomová práce bude zabývat vysvětlením všech důležitých pojmů. Tyto pojmy bude nutné znát pro orientaci v problematice testování i v problematice tvorby automatizovaných testů.

V první kapitole bude vysvětleno, co si pod slovem testování a automatizované testování lze představit. Tzn. hlavní důvody, cíle testování a role, které se při testování využívají. Následně budou představeny chyby, které se při testování mohou objevit. Poté bude představeno automatizované testování, tzn. technologie, které se využívají při testování, rozdělení testů, typy testů, a dále program SikuliX a jeho použití.

3.1 Testování

Testování softwaru je disciplína, která je velmi různorodá. Podle standardu ISTQB, což je certifikační rada, která funguje na mezinárodní úrovni, je testování závislé na kontextu. Software pro elektronické bankovnictví vyvíjený agilním způsobem, či mobilní aplikaci, budeme testovat podstatně jinak než řídicí software pro dopravní letadla. [1]

Dále je potřeba zmínit, že téměř každý IT projekt s sebou nese nutnost testování. Provádí se diskuze mezi sponzorem projektu, businessovým vlastníkem, dodavatelem a manažerem testování o tom, do jaké hloubky bude software testován. [1]

Testování má za úkol najít určité informace o produktu technikou zkoumání. Nalezené informace je potřeba poskytnout všem stakeholderům neboli všem zainteresovaným stranám. [2]

Testovací proces je součástí dalších procesů. Hlavními procesy jsou ověřování a plánování kvality. Úkoly testování a testovacího týmu jsou často velmi rozsáhlé. Testování často přímo zasahuje do celého vývoje softwaru a také často nahrazuje celý proces zjišťování kvality. [2]

3.1.1 **Proces testování**

Proces testování spadá do životního cyklu softwaru. Životní cyklus softwaru viz. Obrázek č. 1 se skládá z pěti následujících etap [3]:

První etapa – **Analýza a specifikace požadavků**

- Tato etapa usiluje o přesné určení toho, co zákazník požaduje.

Druhá etapa – **Architektonický a podrobný návrh**

- Tato etapa rozdělí problém na podproblémy a dojde k vytvoření specifikace funkcionalit.

Třetí etapa – **Implementace**

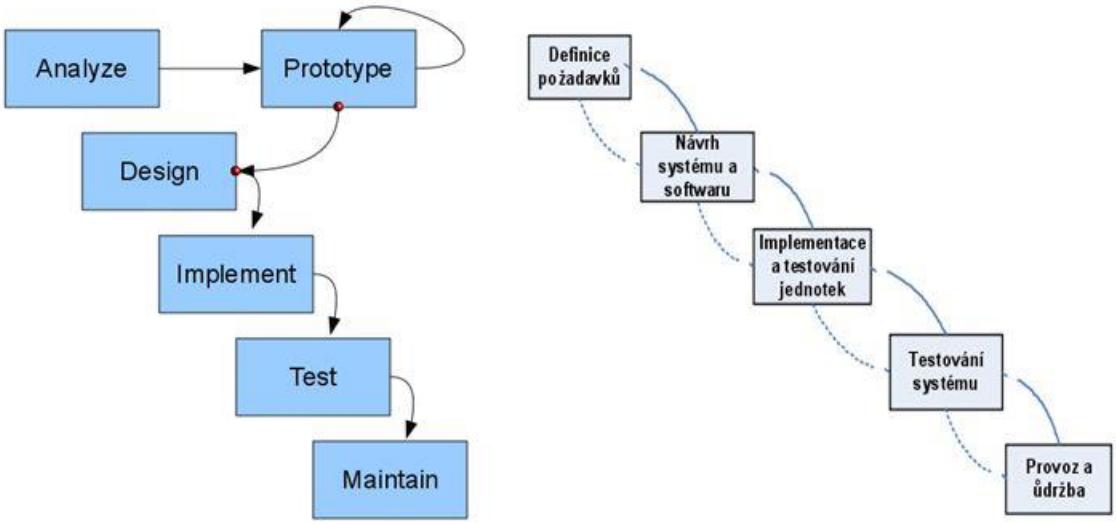
- Tato etapa obsahuje programování, realizaci a dokumentace.

Čtvrtá etapa – **Integrace a testování**

- Integrace spojuje jednotlivé moduly dohromady a při testování se často vrací k předchozím etapám kvůli opravám nalezených chyb.

Pátá etapa – **Provoz a údržba**

- V této etapě se řeší problémy vzniklé při provozu a opravují se nalezené chyby.



Obrázek č. 1 - Životní cykly SW [47]

Dále se software může rozšiřovat o nové funkce. Každá nová funkce prochází výše zmíněným cyklem. [3]

Diplomová práce se zabývá především testováním, tudíž se vychází ze čtvrté etapy životního cyklu softwaru. Testovací proces začíná u stanovení cílů testování. Je potřeba přesně definovat čeho má testování dosáhnout a jaký má být výsledek. Dále se určuje, do jaké hloubky bude software testován a co všechno bude testováno. Poté se volí testy, shromažďují se data a připravují se nástroje pro testování. Také se kontroluje, zda jsou všechny požadavky na software ve formě, která umožňuje jednoznačně zkontrolovat jejich splnění. [4]

Testování probíhá procesem zkoumání softwaru. Software se zkoumá na několika různých úrovních a reportují se všechny nalezené chyby. Celý tento proces se několikrát opakuje. Dále se pak opakuje při každé nové verzi softwaru. [4]

3.1.2 **Proč testovat**

Je třeba zmínit, proč přesně je testování potřeba. Testování má několik následujících důvodů [5]:

1) Nalezení defektů, které se objevily během vývojové fáze softwaru.

2) Pohled vnějšího okolí na firmu dodávající software. Pokud firma dodává chybný software, neudrží se dlouho na trhu.

3) Pokud je chyba nalezena příliš pozdě, náklady na její odstranění jsou mnohem vyšší. Pokud je chyba nalezena již při vývoji (implementaci) u vývojářů, chyba se opraví rychle za minimální náklady. Často se stává, že se chyba nalezne až po zavedení softwaru u zákazníka. Pro firmu je to finančně náročné, protože může nastat situace, kdy bude nutné software na pár dní odstavit.

4) Pokud se vynaloží adekvátní úsilí na testování před dodáním softwaru zákazníkovi, je mnohem menší pravděpodobnost, že se software porouchá při provozu.

3.1.3 **Cíle testů**

Jedním z hlavních cílů testování je ujistit se, že se software bude chovat podle požadavků zákazníka. Požadavky jsou definované v dokumentaci, která by měla vycházet ze zadání od zákazníka. [6]

Cílem je samozřejmě i nalezení chyb a jejich následná oprava. Je třeba brát v potaz i to, že žádné chyby nemusí být nalezeny, protože programátoři odvedli skvělou práci a dodali program bez chyb. Testování, které nenalezne žádnou chybu je stejně úspěšné jako testování, při kterém je nalezena spousta chyb. [6]

3.1.4 **Role**

Další důležitou částí testování jsou role, které do procesu testování zasahují. Většina lidí si myslí, že do testování zasahují pouze osoby z testovacího týmu. Není tomu tak, protože do testování zasahuje mnohem více osob.

V následujících podkapitolách budou všechny role zasahující do procesu testování vysvětleny. Podrobně bude vysvětlena role testera, protože ta je pro testování nejdůležitější.

3.1.4.1 Tester

Tester je základ testovacího týmu. Práce testera se odvíjí od velikosti projektu, na kterém zrovna pracuje. Závisí také na vedení firmy a zadání projektu. Náplň práce testera se tedy velmi liší. Nejčastěji se jedná o manuální testování, reporting nalezených chyb a retestování oprav od programátorů. Tester také musí kvalitně vyhodnotit, zda se jedná o chybu či nikoli. Tuhle informaci nejčastěji získá z testovacích scénářů, které tvoří test analytik. [8][16][21]

Tester musí mít specifické vlastnosti, které se mohou lišit v závislosti na oblasti testování. Základní vlastnosti, které tester musí mít jsou následující. Pečlivost, protože software, který je výstupem projektu by měl být kvalitní. Další důležitou vlastností je

analytické myšlení, které tester využije při vyhodnocování chyb. Nutnou vlastností testera je také dochvilnost a odpovědnost. [8][16][21]

Znalosti testera se také liší podle toho, na jakém softwaru tester pracuje. Pokud tester pracuje přímo s kódem, jedná se především o znalost programovacích jazyků. Tester musí dobře znát software, který testuje a případně další software, se kterým testovaný software spolupracuje. [8][16][21]

3.1.4.2 Test analytik

Test analytik je role, díky které vznikají testovací scénáře, podle kterých tester testuje. Hlavní činností test analytika je analýza funkčností aplikace. Vytváří důležité dokumenty, které jsou využívány především testovacím týmem. Snaží se hledat mezery v analýze, které se snaží opravit. Shání testovací data, vytváří automatické testy. [16][39]

3.1.4.3 Test manager

Test manager je role, která má na starosti celou oblast testování. Řídí testery a komunikuje se všemi ostatními články testovacího týmu. Přebírá zodpovědnost za veškerá rozhodnutí, co se testování týče. Úkolem test managera je dohlížení na vývoj softwaru a dodržování standardů týkajících se vývoje softwaru. Měl by umět dobře vést lidi, být pečlivý a velmi dobře komunikovat s ostatními členy týmu. [42][16]

3.1.4.4 Analytik

Analytik zajišťuje požadavky od zákazníka. Jedná se o velmi důležitou pozici v týmu. Po získání a analyzování všech požadavků od zákazníka začíná analytik s tvorbou specifikace, kde definuje všechny požadované funkce a chování aplikace. Tato specifikace musí být co nejpřesnější, protože se jí řídí vývojáři a testeři. Dále musí analytik zpracovávat změnové požadavky a hlásit rizika, která by se při vývoji mohla vyskytnout. Kvalitně odvedená práce analytika je základ úspěšného řešení projektu. [39][16]

3.1.4.5 Vývojář

Role vývojáře zahrnuje především samotný vývoj aplikace. Musí správně implementovat všechny požadavky, které obdržel od týmu analytiků. Je potřeba zmínit, že se jedná o roli, která provádí první testy vyvíjeného softwaru. Tyto testy se nazývají UNIT testy a budou vysvětleny v kapitole „Typy testů“. [8] [16]

3.1.4.6 Zákazník

Dalo by se říci, že zákazník je také velice důležitou rolí. Právě díky této roli se vývoj softwaru odstartoval. Pracuje se na základě jeho požadavků. Zákazník bude spokojen tehdy, když dostane to, co požaduje. Software se tedy musí důkladně otestovat. Zákazník určitě nebude spokojený s chybovým programem. Komunikace mezi zákazníkem a vývojářskou firmou je klíčová, jelikož prvotním cílem je vždy uspokojit zákazníka splněním jeho požadavků. Zde se ale může vyskytnout velmi častá chyba. Tato chyba spočívá ve znalostech zákazníka. Pokud zákazník nemá potřebné softwarové znalosti, jeho požadavky mohou být často neuskutečnitelné. [8] [16]

3.2 Základní pojmy

Je potřeba doplnit ještě pár základních pojmů z oblasti testování softwaru. I když jsou tyto pojmy celkem známé, mnohdy se stává, že jsou nesprávně chápány. Je tedy potřeba přesně definovat jejich význam. [7]

3.2.1 Verifikace

Verifikace je ověření pravdivosti. U testování se jedná především o ověření, zda jednotlivé fáze vývoje softwaru splňují požadavky definované v předchozí fázi. [7]

Jedná se o aktivitu, která je spíše technického charakteru. Vychází ze znalosti požadavků kladených na software a ze softwarové specifikace. Verifikace zajišťuje detaily, které ale nejsou pro zákazníka tak důležité. [7]

3.2.2 Validace

Validace na rozdíl od verifikace neřeší detaily, ale zkoumá, jestli software dělá to, co zákazník požaduje. Lze říci, že validace je proces vyhodnocování, zda software na konci vývoje dělá to, co bylo požadováno. [7]

3.2.3 Ladění

Ladění je postup, který má za úkol vyhledávat a snižovat množství chyb v softwaru tak, aby fungoval podle předpokladů. Ladění bývá obtížné, pokud je software velmi provázaný. K ladění existuje spousta nástrojů. Nejznámějším nástrojem pro ladění je debugger. [7]

Debugger je softwarový nástroj, používaný pro hledání chyb při vývoji. Častokrát zobrazuje zdrojový kód, a tím je lépe vidět, kde se chyba vyskytla. [7]

3.2.4 Selhání testu

Pod selháním testu je možné si představit, že software se neukončí správně. Například se výstupy budou lišit od očekávaných výsledků a tím dojde k selhání. [7]

3.2.5 Očekávaný výsledek

Očekávaný výsledek je takový výsledek, který vznikne pouze když testovaný software pracuje podle očekávaní. [7]

3.2.6 Prefixové a postfixové hodnoty

Prefixové hodnoty jsou vstupy, které potřebujeme k uvedení softwaru do stavu, ve kterém bude moci přijmout hodnoty testovacího případu. [7]

Postfixové hodnoty jsou vstupy, které jsou potřeba předat softwaru po hodnotách testovacího případu. [7]

3.2.7 **Testovací případ**

Testovací případ je složen z několika hodnot. Těmito hodnotami jsou hodnoty testovacího případu, očekávané výsledky, prefixové a postfixové hodnoty. Tyto hodnoty jsou potřebné pro správné dokončení běhu a vyhodnocení softwaru. [7]

3.2.8 **Testovací sada**

Testovací sada obsahuje testovací případy. Dá se říci, že testovací sada je množinou testovacích případů. [7]

3.2.9 **Testovací scénář**

Testovací scénář se liší od testovacího případu především komplexností. Testovací případy jsou z pravidla jednokrokové. Testovací scénáře většinou obsahují více kroků. [7]

Scénáře většinou neobsahují detaily, často se vyskytují v podobě diagramů testovacího prostředí. Pomáhají testerovi při řešení složitého problému. [7]

3.2.10 **Překladač**

Překladač neboli kompilátor je softwarový nástroj, který se využívá pro vývoj softwaru. Slouží pro překlad algoritmů zapsaných ve vyšším programovacím jazyce (např. Python) do nižšího jazyku (strojového kódu). Z širšího pohledu se dá říci, že je kompilátor program, který provádí překlad ze vstupního jazyka do jazyka výstupního. [13]

3.2.11 **Synchronizační primitiva**

Synchronizační primitiva jsou prostředky, které umožňují paralelně běžícím aplikacím přistupovat ke sdíleným prostředkům současně. [14]

3.2.12 **Logování**

Logování je činnost, při které vznikají záznamy (logy) za účelem pozdější analýzy. Logování si během automatizovaného testování můžeme představit například tak, že průběh testu je zaznamenáván do souboru. V souboru najdeme veškeré kroky testu a jejich průběh. [17]

3.3 Testovací standardy

V této kapitole budou představeny standardy, které se využívají pro tvorbu testů a testování samotné. Nejprve je třeba zmínit, že v dnešní době je všechno tvořeno podle nějakých standardů. Tyto standardy vydávají organizace IEEE a ISO. Standardy prošly určitým vývojem v čase a aktuální verze k 18.01.2021 je následující [18]:

- ISO / IEC / IEEE 29119-1: 2013, Část 1: Koncepty a definice

  Tato norma je základem pro ostatní části normy. Usnadňuje použití ostatních částí a definuje slova ve slovníku, na kterém je norma postavena. Dále poskytuje příklady použití v praxi. [18]

- ISO / IEC / IEEE 29119-2: 2013, Část 2: Zkušební procesy

  Druhá část definuje normy, je určena pro organizace a vysvětluje, jak software testovat. Obsahuje popisy testovacích procesů a rozlišuje testovací procesy pro 3 úrovně. První úroveň je organizační, druhá úroveň se týká správy testů a třetí se týká dynamických testovacích procesů. [18]

- ISO / IEC / IEEE 29119-3: 2013, Část 3: Zkušební dokumentace

  Část třetí se zabývá především dokumentací testů. Obsahuje šablony a příklady dokumentace. Jedná se o dokumentaci procesu organizačního testu, procesu řízení testů a procesu dynamického testu. [18]

  Dokumentace organizačního testu obsahuje testovací politiku a strategii organizačních zkoušek. [18]

  Dokumentace procesu řízení obsahuje testovací plán včetně testovací strategie. Dále obsahuje stav testu a dokončení testu. [18]

  Dokumentace procesu dynamického testu obsahuje: specifikace návrhu testu, specifikace testovacího případu, specifikace zkušebního případu, požadavky na testovací údaje, testovací zprávu o připravenosti dat, požadavky na testovací prostředí, zprávu o připravenosti testovacího

prostředí, aktuální výsledky, výsledky testu, protokol o provedení testu, test incidentu. [18]

- ISO / IEC / IEEE 29119-4: 2015, Část 4: Zkušební techniky

  Čtvrtá část popisuje standardní definice technik navrhování testů softwaru (testovací metody a návrhy testovacích případů). Dále je jejím obsahem odpovídající opatření pokrytí, která lze během návrhu a implementace testů uplatnit. [18]

  Techniky návrhu testů založené na specifikacích jsou založeny na funkční specifikaci testovaného systému. Označujeme je jako techniky testování černé skříňky, které bude vysvětleno v pozdější kapitole. Tato skupina obsahuje tyto návrhy: rozdělení ekvivalence, metoda klasifikačního stromu, analýza hraničních hodnot, testování syntaxe, techniky návrhu kombinatorických zkoušek, testování rozhodovací tabulky, grafy příčin a následků, státní přechodové testování, testování scénářů a náhodné testování. [18]

  Techniky návrhu zkoušek na základě struktury jsou založeny na sktruktuře testovaného systému. V této skupině jsou tyto techniky: pobočkové testování, rozhodovací testování, testování stavu větve, kombinované testování podmínek větví, testování pokrytí rozhodnutím o změně podmínek MC/DC a testování toku dat. [18]

  Poslední jsou **techniky navrhování testů založené na zkušenostech.** Tyto techniky se spoléhají na zkušenosti testera a jedná se o takzvané průzkumné testování. Do této skupiny patří technika hádání chyb. [18]

- ISO / IEC / IEEE 29119-5: 2016, Část 5: Testování na základě klíčových slov

  Poslední část normy obsahuje testování pomocí klíčových slov. Tato technika je přístupem ke specifikaci softwarových testů, které jsou většinou

automatizované. Norma je určena pro ty, kteří chtějí testovat na základě klíčových slov nebo vytvořit specifikaci na základě klíčových slov. [18]

3.4 Chyby neboli bugy

Tato kapitola se zabývá pojmem softwarová chyba (bug). Dále se bude zabývat vysvětlením pojmů chyba, defekt a selhání softwaru. Všechny tyto pojmy označují nesprávné chování systému, přesto se od sebe liší. Aby bylo možné popsat nesprávné chování softwaru, je nejprve potřeba znát chování správné. Chybu lze chápat jako rozdíl současného stavu od stavu požadovaného. Přesné definice softwarové chyby podle Rona Pattona (2002) jsou:

- Software nedělá něco, co by podle specifikace produktu dělat měl.
- Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
- Software dělá něco, o čem se produktová specifikace nezmiňuje.
- Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
- Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný.

Díky uvedeným pravidlům lze určit, zda se jedná o chybu či o očekávané chování softwaru. Z definice lze říci, že softwarová chyba zahrnuje velmi široký rozsah chování. Chování chyb, které jsou známy [10]:

- Vyprodukování špatného výstupu.
- Chování v rozporu s uživatelskými požadavky.

- Chování v rozporu se standardy.
- Havárie programu.
- Zacyklení programu.
- Poškození programu.
- Přepsání programu.
- Smazání dat a souborů.
- Smazání programů nebo části operačního systému.
- Vyčerpání prostředků systému.
- Ignorování událostí.
- Selhání komunikace.
- Nevyhovující rychlost nebo ovládání programu.
- Nedostatečná nebo matoucí komunikace programu s uživatelem.
- Neadekvátní signalizace chyb nebo stavu programu.
- Nemožnost spuštění programu.

Tento výčet chyb by mohl být ještě delší. Pro lepší přehlednosti jsou vybrány pouze nejdůležitější chyby.

3.4.1 **Zdroje chyb**

Nejčastějším zdrojem chyb bývá dokumentace k aplikaci. V dokumentaci se často vyskytují chyby, například funkce nejsou úplné nebo přímo chybí. Některé funkce jsou definované úplně špatně. Důležité je dokumentaci kvalitně vypracovat, aby se zbytečně nevytvářely chyby. [8][15][9]

Druhým nejčastějším zdrojem chyb je návrh, který vzniká u vývojářů aplikace. Příkladem vzniku chyb může být uspěchaný návrh. Některé věci mohou v návrhu zcela chybět nebo dojde k nějakým nečekaným změnám, které pak vývojáři již nezahrnou do návrhu. Návrh by měl vycházet ze specifikace a vývojář v něm popisuje, jak bude jeho práce provedena. [8][15][9]

Dalším významným důvodem je chyba v kódu. Každý dělá chyby i dokonalý, programátor chybu udělá. Chyby v kódu jsou způsobeny i kvůli předchozím zdrojům chyb. Pokud programátor vychází z nekvalitní dokumentace, nečekaně se změní zadání produktu, nebo je funkčnost nedostatečně popsána a programátor ji kvůli špatnému popisu nepochopí, zcela jistě se dopustí chyby. Dalšími důvody může být stres kvůli nedostatku času nebo horší komunikace ve vývojářském týmu. [8][15][9]

Posledním zdrojem chyb jsou například chyby, které ve skutečnosti chyby nejsou. Může se jednat o špatně sepsané testovací scénáře, takže tester označí za chybu něco, co chybou nebylo. Tento poslední zdroj chyb nebývá tak důležitý, a proto se mu většinou neklade velká váha a neřeší se. Skupinu těchto chyb tvoří pouze malé procento. [11]

Lze říci, že chyby v programech jsou hlavně důsledkem lidského faktoru. Vznikají tím, že programátor něco přehlédne, nebo dojde k nepochopení uvnitř vývojového týmu během specifikace kódování a tvorby dokumentace. [11]

Složitější chyby vznikají, když na jednom programu pracuje více lidí. V tomto případě mohou mezi jednotlivými částmi programu vznikat interakce, které nejsou požadované. Tyto nežádoucí interakce je velmi složité dohledat. Právě z toho důvodu se zpracovávají velmi podrobné dokumentace jednotlivých částí programu. [11]

Následující kategorie chyb je vztažena k vláknům. Pokud je proces zpracovávaný na více než jednom vlákně a vlákna nejsou správně synchronizována, dochází k chybě. [9]

Vlákno (thread) je spuštěná instance počítačového programu. Jednotlivá vlákna jsou schopna v rámci procesu vykonávat různé činnosti. Například: zpracování dat, vykreslování výstupu na výstupní zařízení (obrazovku), síťovou komunikaci atd. [37]

3.4.2 **Druhy chyb**

Chyby se dělí podle zdroje. Dále se dělí na sémantickou chybu, syntaktickou chybu a vznik neočekávané události. V následujících podkapitolách bude toto rozdělení vysvětleno. [9]

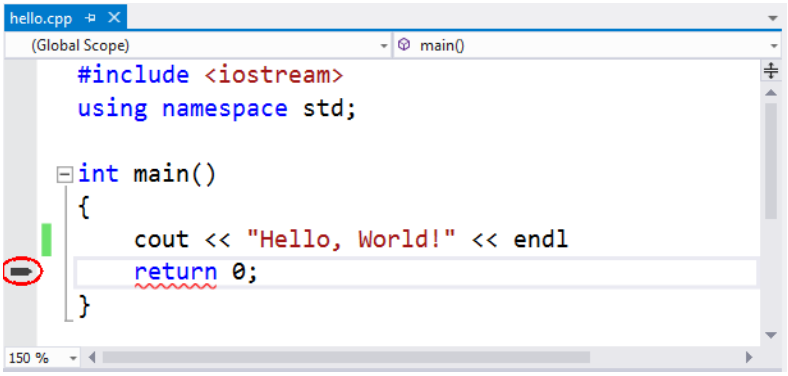
3.4.2.1 Sémantická chyba

Sémantická chyba nastane tehdy, když se program bez jakéhokoliv problému přeloží, ale nedělá to, co se očekává. Program například může skončit v nekonečném cyklu (zacyklí se), spadne (násilné ukončení ze strany operačního systému například kvůli porušení přidělených práv) nebo neudělá naprosto nic (nevrátí žádný výsledek). Nevrácení výsledku je nejhorší možná varianta, která může nastat. Pokud má program něco vracet a nevrátí nic, obtížně se dohledává, kde vznikla chyba. [9]

U složitých programů, jako je třeba operační systém může nastat to, že program sám pozná, že se dostal do chybné situace a není schopen pokračovat ve své činnosti. V takovéto situaci se stává, že se program sám ukončí a podá o chybě specifické hlášení (modrá smrt u Windows). [9]

3.4.2.2 Syntaktická chyba

Syntaktická chyba, je taková chyba, kdy se program ani nepřeloží. Tuto chybu odhalí překladač během syntaktické analýzy. Většina moderních vývojářských prostředí po tomto selhání označí či podá zpětnou vazbu o tom, kde se syntaktická chyba vyskytuje viz. Obrázek č. 2. [10]



Obrázek č. 2 - Syntaktická chyba [44]

3.4.2.3 Neočekávaná událost

Neočekávaná událost vzniká již po přeložení programu a jeho běhu. Pod neočekávanou událostí si lze představit situaci, se kterou program nepočítá a neumí na ni správně zareagovat. Jednou z těchto situací může být například zápis na disk. Program zapisuje nějaká data na disk a po určitém čase se disk zaplní, tudíž na něj nelze zapsat žádná další data. Program se o zápis ale stejně pokouší. Pokud programátor tuto situaci neošetřil nějakou chybovou hláškou, nastane neočekávaná situace a program pravděpodobně zamrzne nebo spadne. [10]

Častokrát může tuto chybu způsobit i obyčejný překlep. Chyby, při kterých došlo k záměně znaků se obvykle obtížně hledají. [10]

Dalším významným zdrojem chyb je chybné nebo nedostatečné použití synchronizačních primitiv při přístupu ke sdíleným zdrojům. Chyba tohoto typu se v programu může udržet velmi dlouho a projeví se až při přesném sledu událostí. Chyba zpravidla nastává, když se vlákna na procesoru seřadí do specifického pořadí. Toto pořadí způsobí, že dva programy žádají o přístup ke stejnému zdroji. Pokud se tak stane, většinou dojde k vzájemnému zablokování (deadlock) a ani jeden program nemůže pokračovat dál ve své činnosti. [10]

3.4.3 **Typy chyb**

Existuje několik typů chyb, které programátoři dělají. Jedná se například o logickou chybu, překlep, opomenutí kontroly nebo použití nebezpečných funkcí. Opomenutí kontroly, logická chyba a překlep se dají zařadit do stejné skupiny chyb. [10]

3.4.3.1 Opomenutí kontroly, logická chyba, překlep

Chyby tohoto typu jsou velmi nebezpečné, pokud program aktuálně zpracovává nedůvěryhodné vstupy. Naopak pokud program zpracovává důvěryhodné vstupy (data jsou připravována interně), problém není tak závažný. Jedná-li se o vstupy zvenčí, kdokoliv může cokoliv podvrhnout. V současné době je tato skutečnost velmi velkým problémem, jelikož spousta programů přijímá právě nedůvěryhodné vstupy. Zdrojem nedůvěryhodných vstupů je převážně internet a program následně zpracovává vše, co z internetu obdrží. [10]

3.4.3.2 Použití nebezpečných funkcí

Tato chyba vzniká tak, že programátor nedodržuje doporučení, které použitá funkce vyžaduje. Například funkce strcpy(). Tuto funkci používá jazyk C. Dochází zde k systematické chybě, protože to, co funkce dostane v parametru, kopíruje do paměti a nemá žádné omezení na délku řetězce. Chyba vzniká tehdy, kdy je řetězec příliš dlouhý a může tak dojít k přemazání neznámého obsahu paměti. [10]

3.4.4 **Softwarové chyby a jejich efekt**

Efekt softwarových chyb je dán podle závažnosti chyby. Nastat může i dominový efekt, který může mít těžké následky pro uživatele, nebo se efekt chyby nemusí projevit vůbec. Nějaké chyby mají na funkci programu minimální vliv, a proto ještě nejsou objeveny. Chyby mohou být ale i vážnější a mohou vést ke ztrátě dat nebo zamrznutí programu. Softwarové chyby jsou velmi nebezpečné a mohou vést například

k neoprávněnému přístupu k datům. Podle NIST bylo zjištěno, že právě softwarové chyby stojí ekonomiku USA přibližně 60 miliard dolarů ročně. [9][38]

3.4.5 **Hlášení, priorita chyb a retesty**

Když je chyba nalezena testerem, tester zadá zprávu o chybě do softwaru, který se používá pro jejich správu. Díky tomuto softwaru se chyba přiřadí ke správnému týmu, který chybu opraví. Programátor se také dozví prioritu oprav chyb zapsaných v softwaru na správu chyb. [12]

3.4.5.1 Priority chyb z hlediska podniku

Z hlediska podniku se dělí chyby podle priority jejich řešení následovně [12]:

- Minor

  Chyba typu Minor se vyznačuje velmi nízkou prioritou opravy. Často se jedná o lehkou grafickou úpravu, pravopis atd. Chybu tohoto typu může programátor odložit na později a mezitím se může věnovat chybám s větší prioritou. Také se může stát, že chyba Minor nebude vůbec opravena.

- Normal

  Chyba typu Normal nemá tak vysokou prioritu řešení, ale programátor by chybu opravit měl. Nemusí ji ale opravovat ihned.

- Critical

  Chyba typu Critical má nejvyšší prioritu řešení a měla by být opravena v co nejkratším čase. Chyby typu Crtitical většinou ohrožují běh programu nebo obsahují nějakou bezpečnostní nedokonalost.

- Blocker

Chyba typu Blocker je velmi specifická. Vlastnosti této chyby brání testerovi otestovat nějakou důležitou funkcionalitu. Priorita odladění této chyby závisí na

domluvě testera s programátorem a test managerem. Může se stát, že chyba typu Blocker přesáhne svou prioritou chybu typu Critical.

3.4.5.2 Retestování

Všechny nahlášené chyby se vrací od programátora zpět k testerovi. Toto vracení má několik důvodů [43]:

- Chyba byla programátorem opravena a tester musí opravu prověřit. Oprava se prověří opětovným otestováním chyby. Pokud byla chyba opravena správně, tester akceptuje opravu. Pokud se chyba stále vyskytuje, tester opravu nepřijímá a vrací chybu zpět programátorovi. Do systému na správu chyb zapíše, jak se oprava projevila a změní chybu ze stavu čeká na přetestování na stav reopened (znovu otevřeno). [43]

- Chyba programátorovi nelze navodit. To znamená, že programátor nemůže chybu reprodukovat u sebe a požaduje po testerovi dodání přesného postupu vzniku chyby. Může se také stát, že se chyba projevuje pouze na instalaci u testera a u programátora se chyba nevyskytne. Zde se tedy musí vývojářský tým zaměřit na hardwarovou výbavu počítače testera a porovnat rozdíly. Z těchto rozdílů by pak mohlo vyplynout, proč chyba nastává. [43]

- Nalezená chyba podle programátora neexistuje (není validní) a tester se mýlí. [43]

- Nalezenou chybu nelze opravit. Tyto chyby vznikají špatnou dokumentací nebo špatným zadáním od zákazníka, jak již bylo zmíněno v předchozích kapitolách. Tato chyba se pak musí opravit změnou dokumentace a zadání. Jinak je chyba brána jako funkčnost systému. [43]

3.4.5.3 Důvody neopravení nalezené chyby

Existuje několik důvodů, proč se nalezená chyba neopraví.
Důvody jsou následující [43]:

- Pro opravu chyby není dostatek času – zde se jedná především o minoritní chyby, které nijak zvlášť neovlivňují aplikaci.

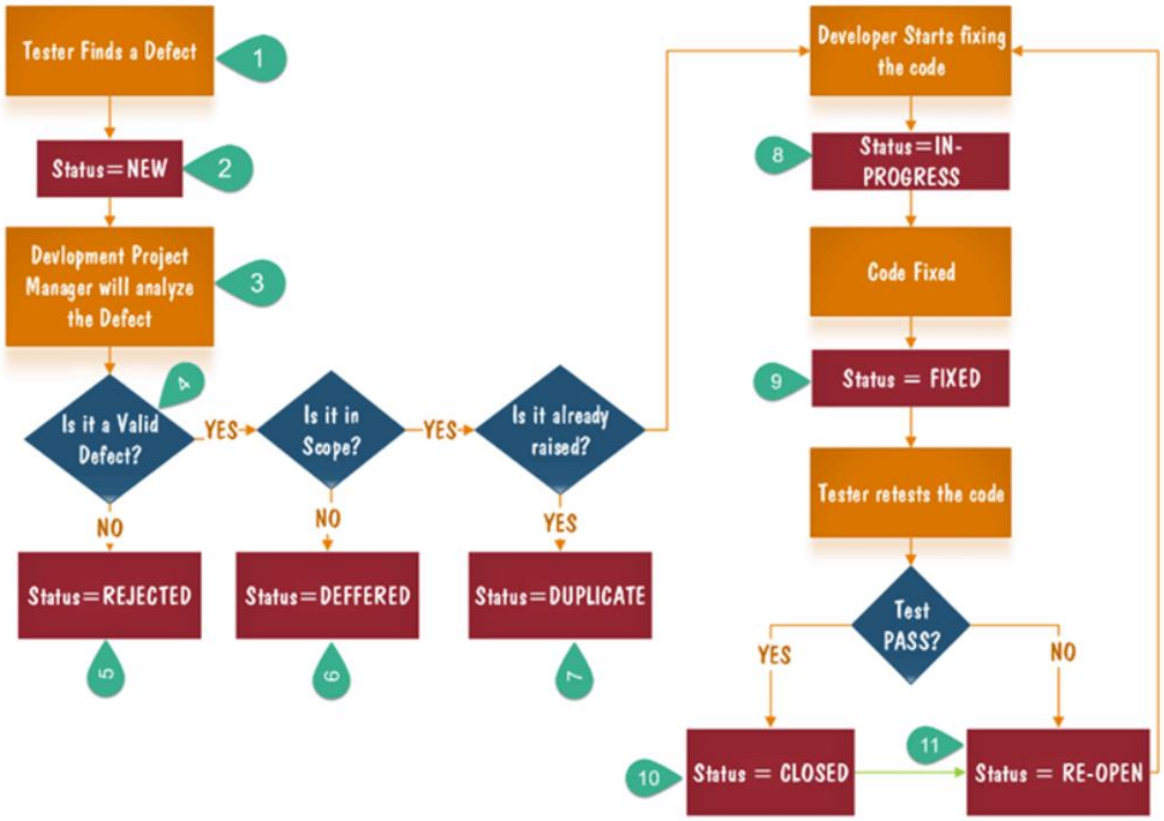
- Tester nedokáže dodat programátorovi dostatek informací pro opravu nalezené chyby.

- Opravení nalezené chyby je příliš nebezpečné pro fungování programu. Mnohdy se při vývoji stane, že je lepší nechat aplikaci ve funkčním stavu, protože oprava chyby může být příliš riskantní a zásadně tak může ovlivnit chod programu.

- Chyby, které nestojí za opravení jsou takové chyby, jejichž funkčnost se příliš nepoužívá. Tato oprava se odloží a řeší se raději důležitější úkol.

3.4.6 **Životní cyklus chyby**

Pod životním cyklem chyby si lze představit specifickou sadu stavů, kterými chyba prochází během celé své existence. Účelem životního cyklu chyby je snadná koordinace a sledování aktuálního stavu chyby, který se mění podle toho, u koho se právě chyba nachází. Na následujícím obrázku je ukázán jeden typ životního cyklu chyby. [15]



Obrázek č. 3 - Životní cyklus chyby [15]

Na Obrázek č. 3 je znázorněn životní cyklus chyby. Jednotlivé položky životního cyklu jsou označeny zelenými čísly, která budou vysvětleny níže [15]:

1. Tester najde chybu.

2. Chybě se přidělí status „New (nová)“.

3. Chyba přejde k analýze manažerem.

4. Manažer rozhoduje, jestli je chyba platná.

5. Pokud chyba není platná, manažer změní status chyby na „Zamítnuto“.

6. Pokud je chyba platná, manažer musí zkontrolovat, jestli chyba spadá do rozsahu vydání. Chyba může být například v nové funkci, která ale není

součástí aktuálního vydání. Tato chyba se může odložit a změní se její status na „Odloženo“.

7. Dále se ověřuje, jestli chybu nenalezl již jiný tester a nezaznamenal ji. Pokud nově nalezená chyba již existuje, změní se stav chyby na „Duplicitní“.

8. Pokud chyba není duplicitní, spadá do rozsahu vydání a je validní, pokračuje chyba k vývojáři, který chybu opravuje. Status chyby se změní na „Probíhající“.

9. Po opravě chyby se změní stav ze stavu „Probíhající“ na stav „Opraveno“ a chyba se vrací zpět k testerovi.

10. Tester přetestuje opravenou chybu. Pokud je chyba opravena správně, tester změní stav chyby na uzavřena. Tato chyba se bere jako vyřešená, ale stále zůstává v systému uložena, kdyby se náhodou opakovala v nějakém příštím vydání. Poté by se změnil stav ze stavu uzavřena na stav znovu otevřena.

11. Pokud opravená chyba neuspěje u přetestování testerem, tester změní status chyby na „Znovu otevřena (reopened)“, zapíše do systému nové poznatky a chyba se přidělí zpět k programátorovi, který bude chybu opravovat.

3.5 Rozdělení testů

Testování se dá rozdělit na funkční a nefunkční testy, statické a dynamické testy, testování černé a bílé skříňky. Tyto testy budou vysvětleny v následujících podkapitolách. [19]

3.5.1 Funkční testy

Funkční testování softwaru pracuje s požadavky a se specifikacemi. Software je tedy testován vůči těmto požadavkům. Funkce se testují pomocí vkládání vstupů a kontrolování očekávaných výstupů. Tento typ testů není zaměřený na fungování softwaru, ale zaměřuje se na správnost výsledků. Těmito testy se simuluje skutečné využití systému, avšak

nepřináší nám žádné předpoklady k správnosti systému. Aby bylo dosaženo nejvyšší efektivity funkčního testování, podmínky testu musí být vytvořeny zákazníkem a uživateli systému. Pokud by tomu bylo jinak, mělo by to velký vliv na kvalitu systému. [19]

3.5.2 **Nefunkční testy**

Nefunkční testování je testování, které se zaměřuje například na bezpečnost, spolehlivost, použitelnost, efektivitu aplikace atd. To znamená, že pomocí nefunkčního testování se testují nefunkční aspekty aplikace. Nezkoumá se tedy funkčnost, ale spíše chování. Například testování stability systému, kde se zkoumá, jak se systém bude chovat, pokud bude přihlášeno více lidí současně. Nefunkční testování by se mělo dělat na všech úrovních testování a jeho realizace by měla proběhnout co nejdříve, protože nalezená chyba pomocí nefunkčních testů může být často velmi nebezpečná a velmi složitá na opravu. [20]

Nefunkční testování může vyžadovat speciální znalosti a zkušenosti. Mohou sem patřit například znalosti o zranitelnosti systému. [20]

3.5.3 **Statické testování**

Do statického testování patří techniky založené na manuálním zkoumání. Například se jedná o hodnocení kódu nebo jiných pracovních produktů pomocí speciálních nástrojů. Statické testování se využívá už v počátečních fázích vývoje softwaru a většinou ještě ani není vytvořený funkční prototyp aplikace. Je možné ho využít i před začátkem psaní kódu, a to na analýzu požadavků a kódu. Není nutné, aby byla aplikace v chodu. Statické testování je důležité hlavně pro bezpečnostní systémy. [21]

Využívá se například při agilní fázi vývoje softwaru, kde se využívá systém na řízení verzí. Výhodou statického testování je to, že dokáže odhalit chyby už na začátku vývoje. To znamená, že oprava bude méně finančně náročná, protože jak již bylo zmíněno, čím dříve je chyba odhalena, tím je oprava méně finančně náročná. [21]

3.5.4 **Dynamické testování**

Dynamické testování lze provádět pouze na spuštěném softwaru. Proto je potřeba minimálně spustitelný prototyp. To znamená, že dynamické testování se spouští mnohem

později než testování statické. Dynamické testování má za úkol najít chyby, které způsobují selhání softwaru. Dalo by se tedy říci, že dynamické testování je přesný opak testování statického. Nalezené chyby jsou dražší na opravu, ale je díky tomu získáván komplexnější pohled na vyvíjený software. [21]

3.5.5 **Testování černé skříňky**

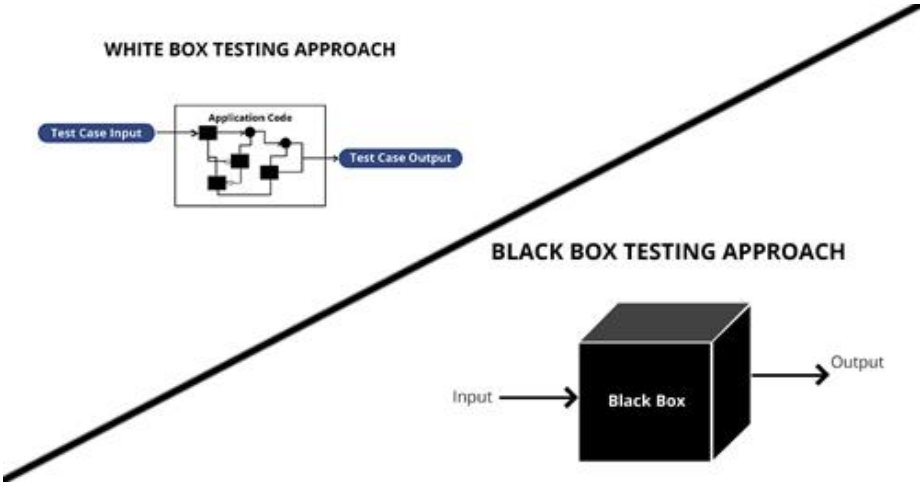
Při testování černé skříňky se zaměřujeme na vstupy a výstupy programu bez znalosti, jak je naimplementován. Produkt je černou skříňkou, do které se nelze podívat. Vidíme jen jak vypadá a jak se chová navenek. Smyslem je analyzovat chování softwaru vzhledem k očekávaným vlastnostem tak, jak ho vidí uživatel. Testování černé skříňky je tedy funkční testování bez znalosti vnitřní struktury, datové struktury a programové struktury. Tester nemá přístup ke kódu ani k dokumentaci. [22] [23]

V této variantě testování si tester může vytvářet testovací scénáře sám. Lze zde využít i automatizovaného testování v kombinaci s manuálním. Tester posuzuje pouze očekávaný výstup s reálným výstupem. Pokud se výstupy liší, tester označí chybu, ale neřeší, jestli je program napsaný efektivně. Pokud se výstupy shodují, neznamená to, že je aplikace napsaná správně.[22] [23]

3.5.6 **Testování bílé skříňky**

Tester má přístup ke zdrojovému kódu a testuje produkt na jeho základě. Vidí nejen co se děje na povrchu skříňky, ale i vnitřní reakce systému. Tím poněkud ztrácí pohled uživatele, ale může lépe odhadnout, kde hledat chyby a kde ne. Testování probíhá pomocí automatizovaných testů nebo manuálně. Často dochází i ke kombinaci těchto metod. Testování bílé skříňky může vyžadovat speciální znalosti, mezi které patří například programování nebo znalost ukládání dat. Testování bílé skříňky se často využívá na začátku softwarového vývoje, aby vývojář s testerem mohli spolupracovat na hledání chyb. Bílá skříňka často odhalí nějaký nežádoucí kód (mrtvý kód) nebo bezpečnostní hrozbu. To, že bílá skříňka odhalí nežádoucí kód, je její největší výhoda. Pokud se v aplikaci vyskytuje nežádoucí kód, aplikace může stále pracovat správně. Tento nežádoucí kód však může být velkou bezpečnostní hrozbou, protože díky němu může docházet ke krádeži dat nebo k nějakému jinému citlivému úniku. Mrtvý kód lze chápat tak, že určitá část kódu se

pro běh aplikace vůbec nevyužívá. Například se může jednat o nějakou starou funkci, která byla nahrazená jinou funkcí, ale nebyla smazána. [23][24]



Obrázek č. 4 - Testování černé a bílé skříňky [45]

3.6 Typy testů

3.6.1 Testování programátorem

Testování programátorem neboli Assembly test, je první test, který se při vývoji nějakého softwaru provádí. Tento test provádí sám vývojář. Často se používá metoda čtyř očí. To znamená, že kód testuje jiný vývojář než ten, který ho psal. Touto metodou vzniká mnohem větší šance, že nějaká chyba bude nalezena. Tento test se naneštěstí nevyužívá tak často. Chyby, které by mohly být nalezeny hned, jsou nalezeny až po delší době. Tím vznikají zbytečné náklady na jejich odstranění. [25]

3.6.2 Jednotkové testy (UNIT testy)

Jednotkové testování, je první testování prováděné po dokončení vývoje. Software je ve stavu po dokončení, ale ještě se neví, jestli software funguje na úrovni strojového kódu. UNIT testování testuje jednotky. Jedná se o testování na úrovni programového kódu a zjišťuje se, jestli jsou jednotlivé části softwaru správně naprogramovány. Nejlépe je rozdělit části softwaru a testovat každou samostatně. Tak se prokáže správnost jednotky bez návaznosti na zbytek programu. Díky tomuto postupu lze odhalit chyby již v počátku a jejich odstranění je o to levnější. Při jednotkovém testování se využívají simulátory, které pomáhají testerovi s testy. Tester jednotkových testů je samotný vývojář. Pro zavedení

jednotkových testů v pozdější fázi vývoje musí být častokrát proveden u velmi složitých programů refactoring (úprava kódu). To většinou není možné u menších aplikacích financovat. U velkých a složitých aplikací se refactoring provádí s odporem. Proto je vhodné UNIT testy zavádět co nejdříve. [25][26]

3.6.3 **Integrační testy**

Po provedeném testování vývojářem, přichází na řadu testy, které provádí testovací tým. Testy, které testovací tým provádí jako první, jsou právě testy integrační. Ty spočívají v tom, že se kontroluje, jak spolu všechny části aplikace komunikují. Zkoumá se také komunikace s hardwarem nebo operačním systémem. Integrační testování lze provádět manuálně i automatizovaně. Často se však toto testování vynechává, jelikož chybná komunikace v aplikaci se odhalí i při dalších testech. Jak již bylo zmíněno, oprava chyby, která se najde později, se vždy prodraží. To znamená, že vynechání jakéhokoliv testu je velmi nebezpečné, nicméně u integračních testů je toto riziko menší než u testů ostatních. [25][26]

3.6.4 **SIT testy – Systémové testování**

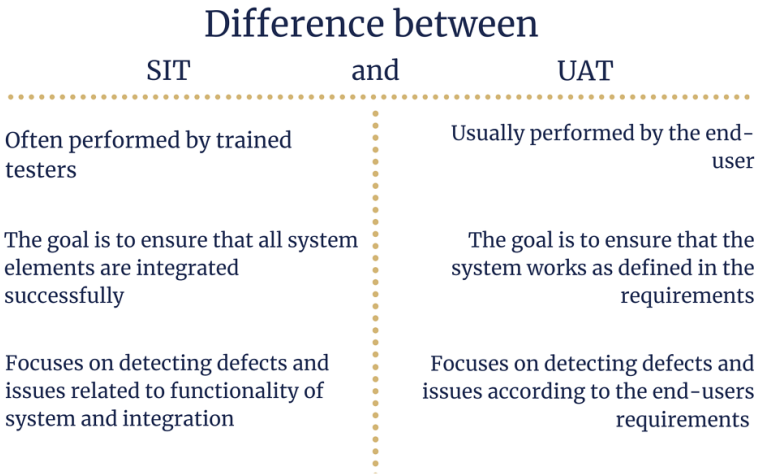
Systémové testování spojené s integračním testováním se nazývá zkratkou SIT – System Integration Tests. Jedná se tedy o testy, které nastávají po ověření integrace. Při průběhu systémových testů se ověřuje celý systém dohromady. Systém tedy už není rozdělen na dílčí části, jak tomu bylo u předchozích testů, ale testuje se jako celek. Systémové testování se provádí až v pozdější fázi vývoje, právě protože je nutné, aby systém fungoval jako celek. Tyto testy kontrolují aplikaci z pohledu zákazníka. Testuje se podle připravených scénářů a snaží se navodit situace, které reálně mohou nastat. Systémové testy se několikrát opakují. Nalezené chyby se opraví a v dalším kole testování se retestují. Při systémovém testování se provádí jak testování funkční, tak testování nefunkční. Systémové testy jsou tedy posledními testy, které se provádí na straně dodavatele sofwaru. Systémové testování slouží jako kontrola softwaru před předáním zákazníkovi. Musí se také zmínit, že systémové testování je časově velmi náročné a často může docházet k chybám v komunikaci s ostatními prvky, které se systémem mají spolupracovat. Tyto problémy nelze předvídat, mohou se pouze vytyčit kritická místa, na která by se měla při implementaci dávat větší pozor. Na realizaci těchto testů by se mělo

myslet už v přípravě celého projektu, právě aby se mohly testy co nejvíce přizpůsobit očekávanému chování softwaru. [25][26]

3.6.5 **Akceptační testování – UAT**

UAT neboli User Acceptance Test jsou testy na straně zákazníka. Jestliže všechny předchozí fáze testování proběhly bez větších komplikací a nedostatků, může se výsledný software předat zákazníkovi. Zákazník si akceptační testy většinou testuje sám se svým týmem testerů podle připravených scénářů. Scénáře pro akceptační testy připravuje zákazník společně s dodavatelem. Samotné testy už však probíhají u zákazníka na jeho testovacím prostředí. Pokud je nalezena nějaká nesrovnalost mezi specifikacemi a reálným stavem, musí se nesrovnalost vyřešit a vše se vrací zpět k dodavateli. Chyby se opraví a ihned nasadí zpět k zákazníkovi. Je nutno si také domluvit, jakým způsobem si zákazník s dodavatelem budou předávat nalezené chyby. [25][26]

U akceptačního testování si dodavatelé softwaru musí dávat velký pozor. Zde může dojít k velké nespokojenosti zákazníka a tím pak přijít o možnou budoucí spolupráci. To, že zákazník s jeho testovacím týmem nalezne nějakou chybu nevadí. Pokud jich ale nalezne mnoho, může nastat velký problém. Dále se musí dbát na rychlost oprav, protože chyby, které zákazník nalezne, musí být opraveny co nejrychleji, aby byl zákazník spokojený. [25][26]



Obrázek č. 5 - SIT a UAT [46]

3.6.6 **Další testy**

Existuje ještě spousta dalších testů, které se využívají při testování softwaru. Každá firma si může vymýšlet vlastní formy testů. Záleží pouze na zkušenostech týmu testerů a jejich kreativitě. V této kapitole budou jen ve zkratce zmíněny testy, které se při testování ještě využívají. [25][26]

Jedním z testů, které se v praxi často využívají jsou **Crash testy (zátěžové testy) neboli performance testy**. Úkolem těchto testů je vystavovat software takové zátěži, která hraničí se selháním softwaru. Díky těmto testům se zjišťují minimální požadavky a doporučené požadavky na systém. Dále se také zjišťuje, jestli má cenu vylepšovat stabilitu nebo ne. Pracuje se zde s připravenými daty, která by měla odpovídat předpokládanému dennímu zatížení softwaru. Toto zatížení se pak systematicky zvedá s počtem uživatelů, velikostí databáze atd. Také je vhodné zkoumat funkce aplikace při využití pomalejšího internetového připojení. [25][26]

Penetrační testy jsou další testy, které se v praxi často využívají. Jedná se o testy bezpečnosti především webových aplikací. Při těchto testech se simulují různé útoky na zabezpečení aplikace. Testují se útoky zevnitř i zvenku. Cílem těchto testů je zjistit, jestli je software bezpečný či nikoliv. Většinou se penetrační testy nechávají provádět u nějaké jiné firmy, nebo alespoň jinými vývojáři. [25][26]

Testy splnění a nesplnění jsou využívány velmi často. Tyto testy se často používají v kombinaci s testem černé skříňky. Při testech splněním se využívá množina dat, kterou musí aplikace pokaždé akceptovat. Při testech selháním do aplikace vkládáme pouze nestandardní data. Snahou testů nesplnění je způsobit neočekávané ukončení aplikace. [25][26]

Progresní a regresní testy jsou testy zaměřené na nové funkce. Progresní testy se používají při kontrole nových funkcí nebo vlastností aplikace. K provedení těchto testů je nutná přesná dokumentace nových funkcí a vlastností. [25][26]

Regresní testy se používají pro opětovnou kontrolu nových funkcí a nových vlastností aplikace. Smyslem regresních testů je kontrola, že nově implementované funkce nezměnily funkčnost starších funkcí. Kontrolují se především ty funkce, které jsou ovlivněny novými funkcemi. [25][26]

Smoke testy se využívají tehdy, kdy je vývoj dokončen a aplikaci lze spustit. Jedná se o rychlé ověření, zda je aplikace schopná projít dalším testováním. Proběhne tedy pouze jednoduchá kontrola, zda je vše tak, jak má být. [25][26]

3.7 Automatizované testování

Součástí každého testování jsou opakující se procesy. To znamená, že při vývoji softwaru se testování několikrát opakuje. Přidání nových funkcí může ovlivnit staré funkce. Proto je potřeba přetestovat vše, co by mohlo být ovlivněno. Každá oprava chyby nese riziko zavedení chyby nové. Toto riziko není malé, a proto se dělají retesty. Tomuto opakovanému testování se říká testování regresní, které bylo zmíněno výše. Opakované manuální testování je často velmi časově náročné. Místo toho, aby testovací tým testoval nové funkce, musí stále dokola testovat funkce staré. Právě kvůli tomu se vytváří automatizované testování. [8][27]

Automatizované testování se tedy využívá především na stále stejné opakující se akce. Pokud je jasné, že nějaký úkon probíhá vícekrát, je vhodné provést jeho automatizaci. Výhodou automatizovaného testování je tedy šetření času i peněz. Není totiž nutné, aby tester opakoval testování starých funkcí stále dokola, ale může se věnovat novým funkcím a na ty staré spustí automatický test, který vyhodnotí, jestli jsou staré funkce chybové či ne. Zde se tedy jedná o velké usnadnění a testování se díky tomu může velmi zrychlit. Musí se ale brát v potaz to, že vývoj automatizovaných testů také něco stojí a někdo tím stráví čas, který musí být zaplacen. Je tedy na vedení projektu, které musí zvážit, jestli se vývoj automatizovaných testů vyplatí nebo ne. Výhodou i nevýhodou automatizovaného testování je eliminace lidského faktoru. Výhodné je to v tom směru, že se automatizovaný test nedopustí chyb z nepozornosti a únavy. Avšak při testech je důležité zohlednit také lidský faktor. Člověk lépe vnímá, zda software není moc složitý,

vymýšlí různé kreativní postupy k hledání chyb a vidí v softwaru souvislosti. Je třeba tedy zvážit, zda je vhodné automatizaci testů provést či nikoliv. [8][27]

3.8 Nástroje pro tvorbu automatizovaných testů a jejich využití

Nejdříve je třeba zmínit, že existuje spousta nástrojů používaných pro tvorbu automatizovaných testů. Každý z nástrojů je vhodný používat pro něco jiného. Nástroje se dělí podle vhodnosti použití. Existují nástroje vhodné pro Unit testy, nástroje vhodné pro integrační testy a API testy (rozhraní pro programování aplikací) a poslední testovací nástroje jsou vhodné pro testování GUI (grafické uživatelské rozhraní). Z každé kategorie budou představeny dva testovací nástroje. [29]

3.8.1 Nástroje pro UNIT testy

U těchto nástrojů záleží především v jakém jazyce je software vyvíjen. Každý programovací jazyk má několik možností. Pro jazyk Python máme například nástroj PyUnit. Pro JavaScript máme nástroje Jasmine a Jest. Pro programovací jazyk Java máme nástroje TestNG a jUnit atd. [29]

3.8.1.1 PyUnit

Nástroj PyUnit je součástí standardní knihovny Pythonu od verze 2.1. Umožňuje snadno psát testy, hromadně je spouštět, a to v textovém nebo grafickém režimu. PyUnit je založen na jUnit od Javy. Využívá jeho osvědčenou testovací architekturu. Dále se PyUnit pyšní počtem stažení, které přesahuje pět tisíc. [28]

3.8.1.2 jUnit

JUnit je nástroj, který pomáhá s jednotkovými testy v programovacím jazyce Java. Funguje na všech operačních systémech a je pravděpodobně nejúspěšnější z rodiny xUnit frameworků. Aktuální verze jUnit je verze 5. Všechny třídy jsou nyní umístěny do jednoho balíčku a tato verze využívá Java8. [30]

3.8.2 **Nástroje pro Integrační testy a testy API**

Testovacích nástrojů pro integrační a API testy je velké množství. U testovacích nástrojů již tolik nezáleží, pro jaký programovací jazyk jsou psány, ale spíše záleží na podpoře protokolů nebo na formátu dotazů a odpovědí. [29]

3.8.2.1 Postman

Jedná se o nástroj, který je v současné době velmi populární. Využívá se pro testování API a je možné si ho nainstalovat jako rozšíření do prohlížeče nebo jako samostatnou aplikaci do počítače. Postman funguje na všech operačních systémech a je využíván jak testery, tak programátory. Postman je bezplatný testovací software, což je velká výhoda. Největší výhodou nástroje Postman je ale jeho jednoduché posílání žádostí a přijímání odpovědí. Má na výběr velké množství metod a u jednotlivých metod může testovat hlavičky opovědí, čas odpovědí atd. Lze vykonávat testy splnění a nesplnění. Nevýhodou nástorje je monitorování testovacích případů, které však lze vyřešit placenou verzí. [31]

3.8.2.2 Insomnia

Insomnia je výkonný klient, který obstarává správu souborů cookies s proměnnými pro různá prostředí. Obsahuje i generátor kódu a zvládá autentifikaci pro operační systémy Linux, Windows a Mac. Insomnia je bezplatný nástroj a má velmi příjemné pracovní prostředí. Je vhodná i pro testery začátečníky, kteří mají menší zkušenost s programováním. Insomnia má velmi propracovanou dokumentaci, která pomůže skoro se vším. [32]

3.8.3 **Testovací nástroje pro GUI**

Testovacích nástrojů pro grafické uživatelské rozhraní existuje hodně. Některé se specializují na počítačové hry, některé jsou vhodné pro mobilní zařízení, některé fungují bez scriptů, některé jsou vhodné pouze pro webové aplikace, zkrátka je z čeho vybírat. [29]

3.8.3.1 Selenium

Selenium se zaměřuje především na webové aplikace. Dalo by se říci, že většina testerů, kteří někdy zkoušeli automatizaci testů, má zkušenost se Selenium. Selenium je

populární hlavně díky tomu, že je bezplatné a open source (volně přístupný zdrojový kód). Je využíváno i známými společnostmi jako je Google a Facebook. Nevýhodou nástroje Selenium je složité prvotní nastavení. Po prvním nastavení Selenium poskytuje velmi efektivní práci, jednoduché generování skriptů a ověřování funkčností. [33]

3.8.3.2 Katalon

Katalon je unikátní tím, že ho může využívat jakýkoliv tým na jakékoliv úrovni. Drží ocenění zákazníků za automatizaci testování z roku 2020. Je tedy zařazen mezi špičkové programy pro automatizaci. Katalon zajišťuje snadné generování automatizovaných testů pro všechny platformy operačních systémů, bez ohledu na složitost aplikace. Jedná se o výkonný záznamový nástroj pro ukládání všech prvků uživatelského rozhraní. Katalon se dá jednoduše integrovat do již zaběhnutého systému, a tak transformovat automatizované testování na testování průběžné. [34]

Obrázek č. 6 - Katalon[34]

3.9 SikuliX

SikuliX je nástroj pro tvorbu automatizovaných testů. Tento nástroj je vhodný pro testování GUI. Tomuto nástroji je přezdíváno „boží oko“, protože pracuje na principu

analýzy obrazu. To znamená, že prakticky vše, co je vidět na obrazovce uživatele, lze nějakým způsobem automatizovat. Vývoj SikuliX započal v roce 2009 jako výzkumný projekt na MIT. SkuliX funguje na všech operačních systémech a je třeba zmínit že využívá OpenCV k analýze obrazu. [36]

OpenCV je open source knihovna počítačového vidění a učení. Tato knihovna byla vytvořena s cílem poskytnout základy všem programům zabývajících se počítačovým viděním a učením. Dalším cílem bylo zrychlit vnímání strojů komerčních produktů. Knihovna má více než 2500 optimalizovaných algoritmů, které obsahují klasické i moderní algoritmy počítačového vidění a strojového učení. Tyto algoritmy lze použít například pro detekci obličeje, rozpoznávání tváří, identifikaci objektů, klasifikaci lidských akcí ve videích, sledování pohybů na kamerách a sledování pohyblivých objektů. Dále dokáže extrahovat trojrozměrné modely objektů a vytvářet spousty trojrozměrných objektů. OpenCV je velmi populární nástroj se spoustou funkcí, má více než 47 tisíc uživatelů a něco přes 18 milionů stažení. Tato knihovna je hojně využívána například ve společnostech, výzkumných projektech a také vládními orgány. OpenCV využívá například Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota a mnoho dalších. Využívá se například pro spojování obrazů z ulic, detekce narušení sledovacího videa v Izraeli, monitorování důlního vybavení v Číně, pomáhá robotům při navigaci a vyzvedávání objektů atd. [35]

SikuliX tedy využívá OpenCV, dále podporuje programovací a skriptovací jazyky jako jsou například Python (Jython), RobotFramework, Ruby (JRuby) a JavaScript. Ačkoli SikuliX v současné době není podporován na mobilních zařízeních, lze jej použít pomocí emulátorů. [36]

Kromě lokalizace obrázků na obrazovce může SikuliX pracovat s klávesnicí a myší a používat tak prvky grafického uživatelského rozhraní. Lze využívat i v prostředí s více monitory. SikuliX je vhodné i pro vzdálené systémy. SikluliX také zvládá rozpoznávání textu a rozpoznávání textu v obrázcích. [36]

SikuliX je vhodné využít pro automatizaci často opakujících se akcí, testování GUI a webových stránek. Pomocí SikuliX se dají hrát hry a spravovat IT sítě. [36]

4 Vlastní práce

Diplomová práce se věnuje tvorbě automatizovaných testů pro desktopovou aplikaci, kterou vyvíjí zvolená firma. K tvorbě automatizovaných testů bude využit nástroj SikuliX, který nejlépe splňuje požadavky pro automatizované testy firmy. V následujících kapitolách se diplomová práce bude věnovat tvorbě automatizovaných testů. Testy budou tvořeny na základě požadavků získaných od zadavatele. To znamená, že se diplomová práce bude nejprve věnovat konzultaci požadavků. Požadavky poté budou zpracovány do verze, která bude odsouhlasena zadavatelem. Po dokončení vývoje automatizovaných testů proběhne analýza těchto testů a porovnání s testy manuálními.

4.1 Získání požadavků

Diplomová práce začíná získáním požadavků pro automatizované testy. Tyto požadavky byly získány pomocí konzultace s firmou. Konzultace proběhla se všemi členy, kteří zasahují do vývoje softwaru a do testování vyvíjené aplikace. Jelikož automatizované testy budou určeny pro testery, kteří je budou spouštět, kontrolovat a dále vyhodnocovat nalezené chyby, byli testeři hlavním prvkem konzultace.

Konzultace proběhla hromadně se všemi účastníky. Konzultace se tedy zúčastnili testeři věnující se testování aplikace, pro kterou jsou automatizované testy určeny. Přizváni byli i testeři z jiného projektu, jelikož i ti mohou mít dobrý nápad a požadavek k automatizaci. Konzultace se také zúčastnili i testeři z ostatních projektů. (Testeři z ostatních projektů se tedy také zúčastnili konzultace.) Dále se zúčastnil manažer testovacího týmu, vývojáři aplikace, manažer vývojového týmu, test analytik a manažer projektu.

Poté začala diskuse na téma: Požadavky na automatizované testy. Z této diskuse vyšly spousty požadavků, které ale musí nejdříve schválit vedení firmy za účasti test manažera a manažera projektu. Získané požadavky byly schváleny a budou uvedeny v následující kapitole diplomové práce. Zadavatel očekává předložení návrhu řešení těchto

požadavků. Po předložení návrhu, který se schválí, započne vývoj automatizovaných testů. Návrh řešení bude uveden v následujících kapitolách.

4.1.1 Získané požadavky na automatizované testy

Požadavky, které byly na základě konzultace získány musí být obsaženy v automatizovaných testech. Nejdříve bude vypracován návrh řešení těchto požadavků a po jeho schválení dojde k samotnému vývoji automatizovaných testů.

Požadavky na automatizované testy:

- Automatizovat manuální testy GUI podle stávajících scénářů.
- Vytvořit souhrnný test.
- Logování testů s viditelným výsledkem.
- Odlišení jednotlivých částí testů.
- Rozlišovat testy podle dne, času a názvu.
- Notifikace s výsledkem testu po jeho ukončení.
- Vytvořit zálohu databáze pro automatizované testy.

4.1.1.1 Upřesnění požadavků

V této kapitole budou získané požadavky na automatizované testy upřesněny a vysvětleny.

První požadavek je jasný a není ho potřeba upřesňovat. Podle stávajících testovacích scénářů budou navrženy automatizované testy.

Druhý požadavek je požadavek na souhrnný test. Bylo určeno, že jeden test bude obsahovat všechny automatizované testy.

Třetí požadavek je požadavek na logování. To znamená, že výsledek testu by měl být ukládán do samostatného testu, kde bude zobrazen průběh testu s výsledkem.

Další dva požadavky se dají sloučit. V logovacím souboru budou jednotlivé testy viditelně odděleny a budou rozděleny podle data, času a názvu testu.

Notifikace o ukončení testu. Jako notifikaci po ukončení testu si zadavatel představuje „vyskakovací okno“ s výsledkem testu.

Posledním úkolem je vytvoření zálohy databáze pro automatizované testy, aby si tuto zálohu mohli všichni testeři nahrát a spustit automatizované testy u sebe.

4.1.2 **Návrh řešení získaných požadavků**

Tato kapitola se zaměří na možná řešení získaných požadavků od zadavatelů a uživatelů automatizovaných testů.

První požadavek je zcela jasný. Automatizované testy budou navrženy podle testovacích scénářů.

Druhý požadavek se týká souhrnného testu. Tento požadavek je řešen následujícím způsobem. Vytvoří se jednoduchý skript, který bude mít možnost zadání od uživatele. V tomto skriptu budou možnosti spustit kompletní testovací sadu nebo vybrat konkrétní test, který se spustí. Řešení je takto zvolené kvůli budoucím úpravám. Pokud bude potřeba upravit nějaký test, nebude se muset procházet jeden velký kód se všemi testy, ale vybere se konkrétní test, kterého se úprava týká. Tento test se pak upraví. Tím je ovlivněno i to, že pokud by se upravoval jeden velký kód, mohly by se do něj zanést chyby. A určitě je lepší, že vypadne jeden test, než že se pokazí testy všechny.

Další tři požadavky mají jedno prosté řešení. Vzhledem k tomu, že budou testy rozděleny a nebude se jednat o jeden velký kód, odlišení testů je vyřešené. Logování testů bude umístěno do jednoho souboru, jelikož zadavatel požaduje mít všechny výsledky pohromadě. Průběh testu se tedy bude zapisovat do souboru. Pokud test proběhne v pořádku, vypíše se vedle názvu testu „OK“. Pokud některá část testu selže, vypíše se vedle testu „NOK (not OK)“. Tím bude zřetelné, jestli test proběhl v pořádku, nebo se vyskytla chyba. Jednotlivé testy budou rozděleny podle kroků, které jsou v testovacích scénářích tak, aby při kontrole logu bylo vidět, kde přesně v testu byla chyba. To velmi usnadní dohledání chyby. Následně si tester díky řešení požadavku číslo 2 může spustit pouze test, ve kterém se chyba vyskytla, a tím může získat vizuální upřesnění. Dále budou všechny testy začínat s datem a časem spuštění testu, jak si zadavatel žádal.

Dalším požadavkem byla notifikace po ukončení testu. Tento požadavek bude řešen vyskakovacím oknem s viditelným výsledkem testu. Pokud test dopadne dobře, výsledek bude zobrazen jako „OK“. Pokud test selže v nějakém kroku, výsledek testu bude „NOK“. Tento požadavek je vylepšen o notifikaci pomocí e-mailu. Po skončení testu bude odeslána notifikace e-mailem na zvolenou adresu. E-mailovou adresu zadá tester při spouštění testu. Dále se notifikace odešle na speciální e-mail, vytvořený přímo pro správu automatizovaných testů, kam budou mít přístup všechny zainteresované osoby.

Záloha databáze a automatizované testy budou nahrány na firemní uložiště, kam budou mít přístup všechny zainteresované osoby. Budou zde umístěny pro jistotu dvě verze, aby nedošlo k nějakému poškození testů. Jedna verze bude s názvem záloha, do které se bude přistupovat jen v případě nouze. Druhá verze bude volně přístupná všem zainteresovaným osobám.

Návrh řešení požadavků byl prokonzultován se zadavateli. Znovu proběhla schůzka, jak tomu bylo při získání požadavků. Na tomto zasedání bylo představeno uvedené řešení požadavků. Dále byl představen program SikuliX, ve kterém se automatizované testy vytvářely. Poté byl zadavateli předán přibližný harmonogram práce. Řešení naplnilo očekáváné požadavky, tudíž prošlo schválením. Po schválení požadavků započala tvorba automatizovaných testů.

4.2 Představení aplikace a důvod automatizace

Před začátkem tvorby automatizovaných testů je třeba představit aplikaci, pro kterou budou testy vyvíjeny. Aplikace obstarává zabezpečování oblastí a sledování osob. Je určena pro velíny ostrahy, kde uživatelé sledují pohyb osob ve střežené oblasti. Uživatelé mohou nastavovat různé zóny, do kterých osoby mohou či nemohou vstupovat.

Zvolená firma podepsala kontrakt na dlouhodobý vývoj této aplikace pro více společností, právě proto se firma rozhodla automatizovat testovací scénáře.

4.2.1 Výběr testovacího nástroje

Pro tvorbu byl zvolen testovací nástroj SikuliX. Nástroj SikuliX byl zvolen, protože jako jediný nástroj z testovacích nástrojů pro GUI dokáže pracovat přímo s grafickým rozhraním aplikace. Dále byl vybrán kvůli zkušenostem autora s programováním v jazyce Python, na kterém je SikuliX založeno. Dalším důvodem, proč byl SikuliX vybrán je, že jeho pomocí lze přesně napodobit pohyby uživatele v aplikaci, což je právě u aplikace zvolené firmy velkou výhodou.

4.3 Tvorba automatizovaných testů

V této kapitole bude popsána tvorba automatizovaných testů. Kapitola je zaměřená hlavně na zpracování získaných požadavků a dále na problémy, které při tvorbě testů byly řešeny.

4.3.1 Studium testovacích scénářů

Před zahájením vývoje bylo potřeba se podrobně seznámit s testovacími scénáři, protože na jejich základě byly automatizované testy vyvíjeny, což bylo i požadavkem od zadavatele. Testovací scénáře určené k automatizaci jsou následující:

- **Přihlašování a hesla**: tento testovací scénář se zabývá přihlašováním uživatele do aplikace, zkoušení prolamování hesla, vkládání dlouhých znakových řetězců, kopírováním hesel a speciálními znaky v hesle uživatele.

- **Testy uživatelského rozhraní:** tyto testy spočívají v kompletním otestovaní aplikace. Test je zaměřen na přívětivost uživatelského rozhraní a zda všechny tlačítka v aplikaci dělají to, co uživatel očekává. Testovací scénář je časově velmi náročný.

- **Testy přístupů portály**: testy přístupů jsou také časově velmi náročné. Jedná se o přístupy do různých zón, které uživatel nastaví. Pomocí přístupových tokenů, nebo simulátoru přístupů se simulují všechny různé kombinace vstupů a kontroluje se, zda je vstup oprávněný, nebo ne.

- **Testy přístupů výtahy:** tento scénář je podobný jako testovací scénář testy přístupů portály. Rozdíl je zde v tom, že přístupový token se nevyužívá na zóny, ale na jednotlivá patra, do kterých výtah jede.

- **Atributy subjektů:** tento testovací scénář se věnuje subjektům a jejich atributům. Některé atributy jsou již předdefinované a spoustu dalších atributů může uživatel vytvořit sám. Testovací scénář popisuje, jak by se měly atributy správně vytvářet a propisovat do zpráv například při průchodu některou zónou.

- **Tokeny:** testovaní tokenů je jednodušším testem. Zkoumá se přidávání tokenů do systému. To lze provádět importem, nebo ručně pomocí čtečky tokenů. Token se pak následně přiřazuje k subjektu. Ověření funkčnosti tokenu probíhá v předchozích testech.

- **Virtuální zóny:** test virtuálních zón hodně zasahuje do testu přístupů, jelikož existují zóny fyzické, které jsou určeny rozmístěním čteček tokenů po střežené oblasti. Virtuální zóny do toho zasahují tak, že uživatel si může na fyzické zóně vytvořit zónu virtuální. Poté uživatel nastaví přístupy subjektům

do virtuálních zón. Vstupy do zón se následně propisují do zpráv a tím uživatel vidí, kde se jaký subjekt pohybuje.

- **Kniha návštěv:** tento testovací scénář se věnuje návštěvám ve střežené oblasti. Návštěva se vytvoří v systému a přidělí se jí nějaký token. Pomocí tokenu se pak návštěva může pohybovat ve fyzických a virtuálních zónách. Návštěva se vytváří ručně, nebo pomocí skenu občanského průkazu. Existuje také opakovaná návštěva, používá se například pro externí firmu, která se ve střeženém objektu věnuje uklízení.

- **Plánovač a svátky**: testy plánovače a svátků slouží pro zapínání a vypínání čteček tokenů. Uživatel nastaví datum, kdy budou všechny čtečky zamčené a žádný subjekt nebude mít přístup do hlídané oblasti. Podobně fungují i svátky, které se importují do plánovače. Uživatel pak nastaví, jak se mají čtečky v daný svátek chovat.

- **Historie:** historie událostí se ukládá kvůli neoprávněným vnikům po dobu jakou uživatel nastaví. Tento test je jednoduchý. Má-li se testovat v reálném prostředí stává se velmi dlouho trvající.

- **Události:** posledním scénářem, který je určený k automatizaci je testovací scénář událostí. Tento scénář sleduje v záložce události všechny vygenerované události, které se staly během běhu aplikace. Zapisují se zde průchody portály, změny módů portálů, otevírání dveří, poplachy a další volitelné události.

Všechny zmíněné scénáře jsou určené pro automatizaci.

4.3.2 Vývoj automatizovaných testů

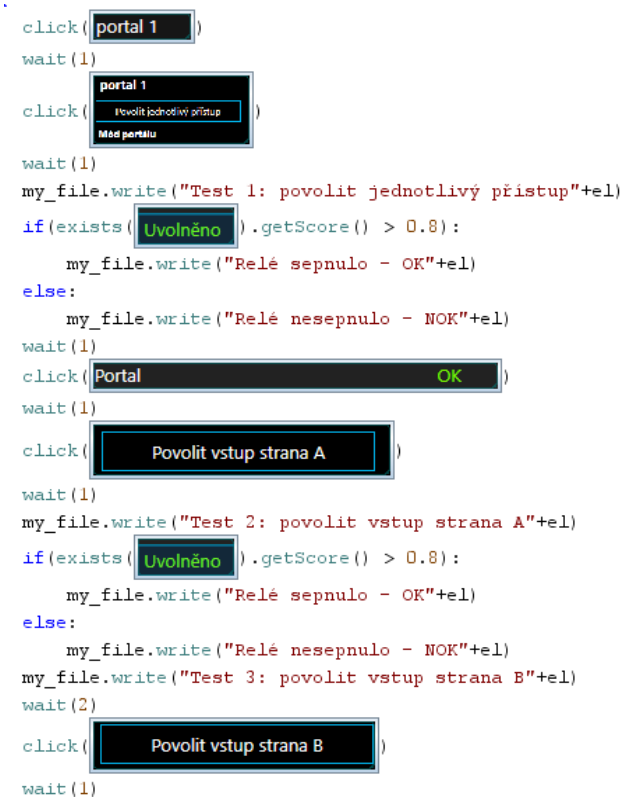
Jak bylo zmíněno v požadavcích na automatizované testy, zadavatel chtěl vytvořit souhrnný test, který by otestoval výše zmíněné scénáře najednou. Bylo zvoleno jiné řešení, které bylo zadavatelem schváleno. Nejprve je tedy potřeba vytvořit všechny

automatizované testy zvlášť a poté se spojí pod jeden, ze kterého se budou ostatní testy volat.

4.3.2.1 Automatizace scénářů

V této kapitole jsou představeny důležité části automatizovaných scénářů.

Testy uživatelského rozhraní. Při této automatizaci bylo potřeba projít všechny záložky aplikace. Proto je rozsah tohoto testu nejrozsáhlejší a nejsložitější. Na následujícím Obrázek č. 7 je znázorněna část práce s portály. Při této práci uživatel musí v záložce portálů povolit vzdálený přístup jedné osobě. Odpovědí na tuto akci je sepnuté relé a otevřené dveře. Automatizovaně to vypadá následovně:



Obrázek č. 7 – GUI

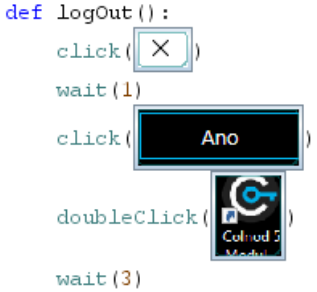
Automatizace průchodu portálem byla kritická tím, že automatizací nelze sledovat otevírání a zavírání dveří. Řešení je tedy takové, že se sleduje status portálu, kde je pomocí příkazu getScore() sledováno, jestli reálný status portálu odpovídá požadovanému. Pokud je zachycena shoda více jak 80 %, otevření dveří bylo úspěšné a výsledek je zapsán do logovacího souboru. Pokud se tak nestane, negativní výsledek je opět zapsán do

logovacího souboru a pokračuje se další testem. Příkaz wait(), který je vidět také na předchozím obrázku znamená čekací dobu, která je zde kvůli výkyvům v aplikaci. Občas se změna statusu u portálu propíše hned a občas to chvilku trvá, proto je všude nastavená čekací doba 1 vteřinu. Během této doby se status propíše. Pokud by tomu tak nebylo, zapíše se negativní výsledek do logu.

Testy událostí. Na následujícím obrázku je znázorněna část práce s událostmi. Je zde proklik do záložky událostí, následné vyfiltrování událostí podle času. Poté je potřeba vyfiltrovat všechny změny, tudíž do kolonky typ program vloží znaky „zm“. Zobrazí se všechny události o změně a pokud program nalezne požadované změny s přesností nad 80 % zapíše do logu správný výsledek.

Obrázek č. 8 – Události

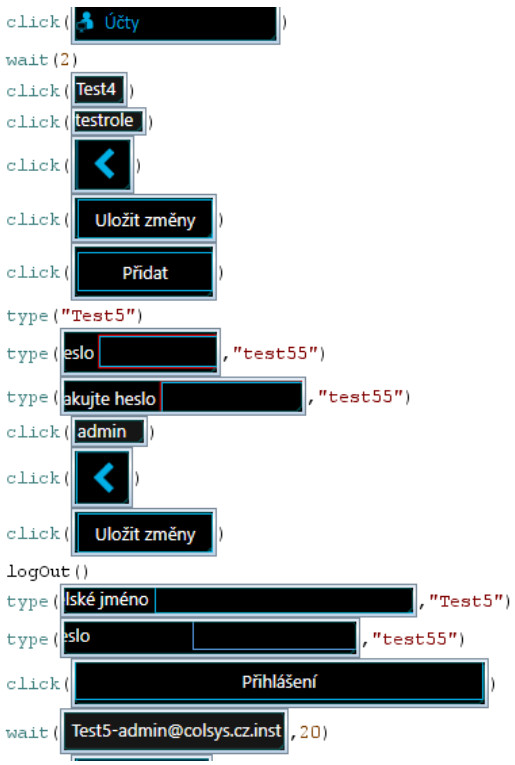
Přihlašování a hesla. Tento test často pracuje s přihlašováním a odhlašováním, proto je zde vytvořena funkce logOut() pro odhlašování, aby se tato část nemusela stále opakovat. Tato funkce je opravdu jednoduchá, jak je vidět na následujícím Obrázek č. 9.



Obrázek č. 9 – Odhlášení

Funkce pro odhlášení se pak několikrát v průběhu testu volá. Na konci této funkce je nastavena čekací doba 3 vteřiny protože, po odhlášení z aplikace nějakou dubu trvá, než se zobrazí nové přihlašovací okno.

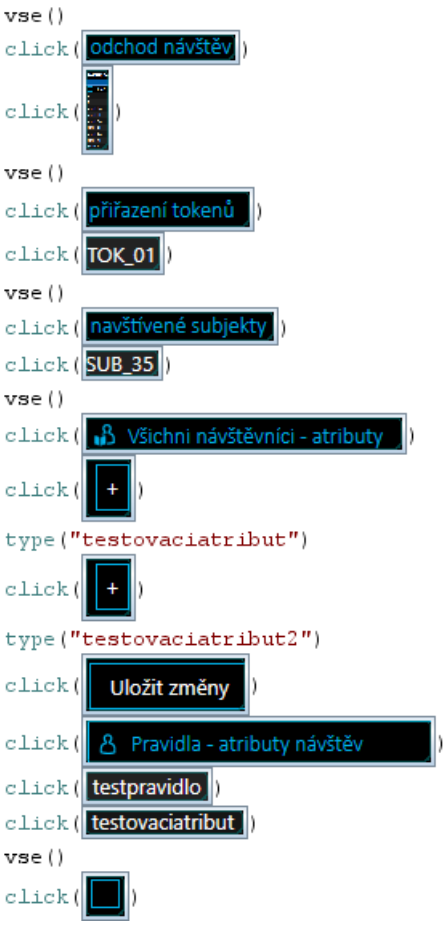
Na dalším obrázku je znázorněn postup vytváření nových uživatelských účtů a následné testování, zda tato funkcionalita funguje správně.



Obrázek č. 10 - Tvorba nového uživatele

V první části programu je vytvořen nový uživatel s názvem Test5. Po úspěšném vytvoření nového uživatele se zavolá funkce logOut(), která provede odhlášení z aplikace. Následně je do aplikace přihlášen nově vytvořený uživatel.

Návštěvy. Další zajímavou částí automatizace je automatizace návštěv. Při práci s návštěvami musí uživatel nastavovat spoustu atributů. Díky automatizaci je tato práce nahrazena jednoduchou funkcí vse(), která pomocí volané klávesové zkratky ctrl+a vybírá všechny atributy najednou a rovnou je přiřazuje ke zvolené návštěvě.



Obrázek č. 11 – Návštěvy

Na Obrázek č. 11- Návštěvy je zobrazení použití funkce vse(), která vybírá všechny atributy a přiřazuje je. Dále je na obrázku vidět tvorba atributů návštěvy a zvolení tokenu a navštívené osoby.

4.3.2.2 Plnění požadavků na automatizované testy

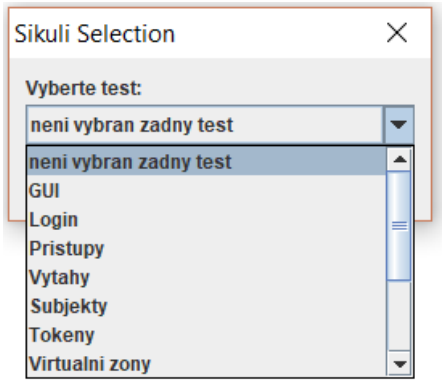
V této kapitole je znázorněno splnění zadání od zadavatelů automatizace.

Jedním z nejdůležitějších požadavků je souhrnný test, který je řešen rozdělením všech testů po jednotlivých scénářích. Tester si při spuštění programu all.py vybere, který test chce pustit. Toto je řešeno pomocí načítání vstupu od uživatele a následné volání konkrétního testu.

```
testy = ("neni vybran zadny test", "GUI", "Login", "Pristupy", "Vytahy",
test = select("Vyberte test: ", options = testy)
if (test == testy[0]):
    popup("Test nevybran")
    exit(1)
elif (test == testy[1]):
    os.system("GUI.py")
elif (test == testy[2]):
    os.system("login.py")
elif (test == testy[3]):
    os.system("pristup.py")
elif (test == testy[4]):
    os.system("elevator.py")
```

Obrázek č. 12 - Volba testu

Na obrázku Volba testu je znázorněn kód obstarávající výběr konkrétního testu uživatelem a na dalším obrázku je vyobrazeno okno s výběrem testů.



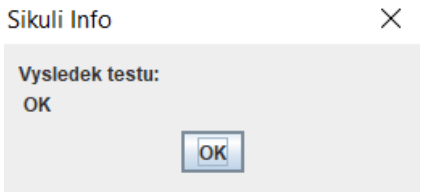
Obrázek č. 13 - Výběr testu z nabídky

Dalším důležitým požadavkem bylo zaznamenávání výsledků do speciálního souboru a oddělení jednotlivých testů pomocí názvu a času s datem. Postup testů je ukládán do souboru log.txt a je zobrazen na následujícím Obrázek č. 14.

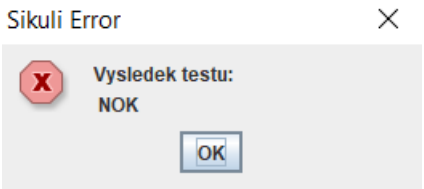
```
****************************************
2021-03-13 14:37:38.561000

Test 1: povolit jednotlivý přístup
Relé sepnulo - OK
Test 2: povolit vstup strana A
Relé sepnulo - OK
Test 3: povolit vstup strana B
Relé sepnulo - OK
Test 4: Změna módu na zamčeno
Mód změněn - OK
Test 5: Změna módu na odemčeno
Mód změněn - OK
Test 6: Změna módu na Token+Pin
Mód změněn - OK
Test 7: Změna módu na Token
Mód změněn - OK
Test 8: Změna módu na Token+Token
Mód změněn - OK
Test 9: Změna módu na Token+Vzdálené potvrzení
Mód změněn - OK
Test 10: Změna módu na První uvolnění
Mód změněn - OK
Test 11: Změna módu na Libovolná karta
Mód změněn - OK
Test 11: Změna módu na Odemknutí kartou
Mód změněn - OK
Test 12: Události
Události o změně módu nenalezeny - OK
```

Obrázek č. 14 – Logování

Notifikace po dokončení testu jsou řešeny vyskakovacím oknem s výsledkem testu. Pokud test proběhne úspěšně, objeví se okno s výsledkem OK. Pokud test selže, objeví se okno s výsledkem NOK (not OK).

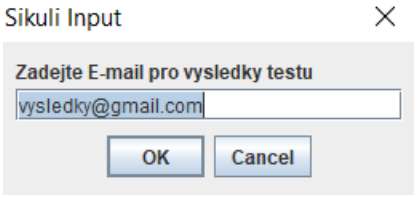


Obrázek č. 15 - Výsledek testu OK



Obrázek č. 16 - Výsledek testu NOK

Další notifikace pro testera je řešena pomocí e-mailu, lze vidět na Obrázek č. 17 a Obrázek č. 18. Tester zadá svůj e-mail při spouštění testovacího scénáře a až test skončí, tester dostane notifikaci na vložený e-mail. Notifikace přijde ještě na další e-mail, na který mají přístup všechny zainteresované osoby.



Obrázek č. 17 - Okno pro e-mail

```
import smtplib

email = input("Zadejte E-mail pro výsledky testu:","vysledky@gmail.com ")
li = ["vysledky@gmail.com", email ]

for dest in li:
    s = smtplib.SMTP('smtp.gmail.com', 587)
    s.starttls()
    s.login("automatizacevev@gmail.com", '[redacted]')
    message = "Test dokončen"
    s.sendmail("automatizacevev@gmail.com", dest, message)
    s.quit()
```

Obrázek č. 18 - E-mail

4.4 Nasazení automatizovaných testů

Po dokončení vývoje automatizovaných testů bylo potřeba nasadit automatizované testy do provozu. Prvních pár dní se testy ještě ladily, aby vše fungovalo na testovacích strojích. Nejčastěji docházelo k pomalejší odpovědi aplikace na nějakou změnu, která během testování v aplikaci vznikla. Tento problém se vyřešil úpravou čekací funkce wait(). Čekací doby se zkracovaly a prodlužovaly podle potřeby, aby test běžel tak, jak má a byl co nejvíce efektivní. Po dokončení ladění testů se začalo reálně testovat.

Reálné testování probíhalo na vytvořené záloze databáze. Tato záloha je určena pro testy a simuluje reálné vytížení aplikace, kdy se ve střežené oblasti pohybuje okolo 500 osob. Kompletní testovací sada úspěšně otestovala vybrané scénáře.

Po otestování scénářů muselo proběhnout zaškolení testerů do práce s automatizovanými testy. I toto školení se musí počítat do nákladů na vývoj automatizovaných testů. Nasazení automatizovaných testů do provozu proběhlo úspěšně. Nasazení i s laděním testů trvalo přesně 6 pracovních dní.

4.5 Manuální testování

Tato kapitola je věnována manuálnímu testování testovacích scénářů, které byly automatizovány.

Aby bylo možné porovnat automatizované testy s manuálními, je potřeba všechny manuální testy, které podlehly automatizaci otestovat ručně. Při manuálním testování bylo postupováno podle testovacích scénářů, které byly předlohou při tvorbě automatizovaných testů. Manuální testy provedl autor diplomové práce a další čtyři testeři. Je potřeba zmínit, že dva testeři měli s těmito scénáři již velké zkušenosti a další dva testeři měli s těmito scénáři zkušenosti minimální, neboť pracují na jiném projektu. Protože jsou některé testy velmi komplexní a složité muselo být měření testování rozděleno na více dní. Denní pracovní doba je stanovena na standardních 8 hodin. Dále je potřeba uvést, že manuální testování scénářů proběhlo ještě před začátkem vývoje automatizovaných testů. Začátek testů byl 11. 05. 2020. V následující tabulce jsou uvedeny výsledky manuálního testování jednotlivých scénářů.

Tabulka 1 - Manuální testy

	Časová náročnost [hod:min:sec]				
Scénář	**Vévoda Jakub**	**Tester č.2**	**Tester č.3**	**Tester č.4**	**Tester č.5**
Přihlašování a Hesla	2:36:24	1:56:45	1:58:23	2:15:34	2:10:13
GUI	17:11:42	16:20:34	16:17:49	16:58:21	17:02:19
Přístupy – portály	15:27:54	15:04:18	15:12:34	15:21:14	15:23:43
Přístupy – výtahy	9:08:41	8:56:14	8:52:32	9:02:04	8:55:36
Atributy subjektů	5:12:29	4:53:18	4:51:23	4:59:02	4:57:53
Tokeny	1:26:26	1:12:06	1:34:54	1:15:27	1:17:26
Virtuální zóny	7:54:34	7:28:32	7:37:27	7:26:18	7:29:43
Kniha návštěv	8:28:15	8:16:37	8:10:57	8:24:19	8:35:37
Plánovač a svátky	7:29:17	7:10:47	7:18:31	7:21:19	7:24:42
Historie	16:24:51	16:10:11	16:14:26	16:21:29	16:24:34
Události	3:28:36	3:13:27	3:09:46	3:16:02	3:21:54
Celková časová náročnost	94:49:09	90:42:49	91:18:42	92:41:09	93:03:40
Průměr časové náročnosti	92:31:06				

Z tabulky manuálního testování vyplývá, že kompletní otestování aplikace trvá velmi dlouho, což je způsobeno velkou komplexností testované aplikace. Další nevýhodou v tomto případě je, že testy se nedají provádět současně.

4.6 Automatizované testování

Aby bylo možné hodnotit automatizované testování, všechny testovací scénáře musely být pečlivě nastudovány. Kompletní sada automatizovaných testů v reálném provozu byla otestována autorem diplomové práce a dalšími čtyřmi testery. Testovací podmínky byly stanoveny pro všechny stejně Byla použita záloha databáze na testování a všichni měli k dispozici testovací stanici se stejnými komponenty. Test proběhl 20. 09. 2020 a jeho výsledky jsou znázorněné v následující tabulce.

Tabulka 2 - Výsledky automatizace

Datum testu:	**20. 09. 2020**			
Id	**Záloha testovací DB**	**Testovací stanice**	**Dokončení testu**	**Čas [min:s:ms]**
Jakub Vévoda	ANO	Dell Precision T3640 MT	OK	16:34:16
Tester č.2	ANO	Dell Precision T3640 MT	OK	15:58:59
Tester č.3	ANO	Dell Precision T3640 MT	OK	15:47:34
Tester č.4	ANO	Dell Precision T3640 MT	OK	16:25:37
Tester č.5	ANO	Dell Precision T3640 MT	OK	16:18:49
			Průměrný čas:	**16:13:03**

Z předchozí tabulky vyplývá průměrný testovací čas v délce 16 min a 13 vteřin. Je potřeba říct, že sloupec „dokončení testu“ neznamená, že se při testování nenašly žádné chyby. Tento sloupec vyjadřuje pouze to, že automatizované testy během testu neskončily s nějakou chybou v kódu automatizovaných testů. Váha testu, při kterém se chyba najde je stejná, jako když se chyba nenajde žádná. Toto tvrzení je zmíněno již v teoretické části diplomové práce.

4.7 Automatizované testy vs. manuální testy

V této podkapitole jsou uvedeny rozdíly, klady a zápory mezi automatizovanými a manuálními testy, které vyplynuly během tvorby automatizovaných testů. Dále tato podkapitola obsahuje finanční náročnost manuálního a automatizovaného testování.

4.7.1 Výhody a nevýhody automatizovaného testování

Výhodou automatizovaného testování je samozřejmě úspora času. Automatizované testy jsou oproti manuálním testům velmi rychlé. Počítač dokáže až několikrát rychleji otestovat aplikaci než člověk. Další výhodou automatizovaného testování je nenáročnost na znalosti testera. Automatizované testy může obsluhovat osoba, která je pouze proškolená k obsluze testů a která nemusí mít znalosti testování. Výhodou automatizovaného testování je i neúčast testera u testů. Tester nemusí být přítomen po celou dobu testování a může se tak věnovat jiným testům nebo řešení nalezených chyb.

Největší nevýhodou automatizovaného testování je úprava testů. Pokud vznikne nějaká velká změna v aplikaci, bude se muset test také upravit. Zde je to díky rozdělení testů podle jednotlivých scénářů trochu lehčí, nicméně to zásah do testů úplně nevyloučí. Další nevýhodou automatizace pomocí Sikulix je, že pokud se změní vizuální stránka aplikace, testy se budou muset upravit také. SikuliX totiž pracuje na bázi analýzy obrazu, jak je uvedeno v teoretické části diplomové práce. Úprava ale není tak složitá, nahradí se pouze obrázek za nový. Další nevýhodou může být právě nepřítomnost testera, jelikož tester při testování dokáže být kreativní a vymýšlet různé způsoby a kombinace testování, tím může odhalit něco, co automat nedokáže.

4.7.2 Finanční náročnost

V této kapitole bude znázorněna finanční náročnost na manuální testování a na vývoj automatizace testů.

4.7.2.1 Manuální testování

V této podkapitole je znázorněna finanční náročnost manuálního testování.

Nejprve je potřeba zmínit, že manuální testování započalo tvorbou testovacích scénářů. Bohužel již zpětně nelze spočítat, kolik tvorba testovacích scénářů stála. Naštěstí tento náklad může být zanedbaný, jelikož testovací scénáře posloužily k manuálnímu testování, i k vývoji automatizovaných testů. Z tabulky č.1 je zřejmé, že celkové manuální otestování všech scénářů, které se automatizovaly zabere průměrně 92,5 hodin. Denní pracovní doba je standardních 8 hodin. Z toho plyne, že manuální testy zaberou cca 11,5 pracovních dnů. Tester je placen 150Kč/h v hrubé mzdě. Náklady zaměstnavatele na manuální testování je 18 465 Kč. Vše uvedené je znázorněno v tabulce č.3.

Tabulka 3 - Náklady na manuální testování

Pracovní den [hod]	8
Počet pracovních dnů měsíčně	21,74
Hrubá mzda [Kč/h]	150
Hrubá mzda testera měsíčně [Kč]	26 088
Sociální a zdravotní pojištění [%]	33,8
Celkový náklad zaměstnavatele na měsíc [Kč]	34 906
Délka manuálního testování [hod:min:sec]	92:31:06
Délka manuálního testování [pracovní dny]	11,5
Celkový náklad na den [Kč]	1 606
Celkový náklad na manuální testování [Kč]	18 465

4.7.2.2 Automatizované testování

V této podkapitole je znázorněna finanční náročnost vývoje automatizovaného testování.

Při vývoji automatizovaných testů byl přesně zapisován čas práce pro pozdější výpočet nákladů na vývoj. V tabulce č.4 je uveden náklad na vývoj automatizovaných testů. Vývoj automatizovaných scénářů trval 193 hodin. Zaškolení a ladění automatizovaných testů trvalo 6 pracovních dní, jak bylo zmíněno výše v diplomové práci. Celková délka vývoje automatizovaných testů byla 30,125 pracovní dní. Celkový náklad na vývoj automatizovaných testů je 54 818 Kč. Je potřeba také zmínit, že každá úprava

automatizovaných testů bude dražší, než úprava testovacího scénáře testů manuálních i přesto, že jsou testy navrženy tak, aby úpravy byly co nejjednodušší.

Tabulka 4 - Náklady na vývoj automatizovaného testování

Pracovní den [hod]	8
Počet pracovních dnů měsíčně	21,74
Hrubá mzda [Kč/h]	170
Hrubá mzda testera měsíčně [Kč]	29 566
Sociální a zdravotní pojištění [%]	33,8
Celkový náklad zaměstnavatele na měsíc [Kč]	39 560
Délka vývoje automatizovaných testů [hod]	241
Délka vývoje automatizovaných testů [pracovní dny]	30,125
Celkový náklad na den [Kč]	1 820
Celkový náklad na vývoj automatizovaného testování [Kč]	54 818

4.7.3 **Finanční návratnost**

Tato kapitola je věnována finanční návratnosti automatizovaných testů. Z tabulky č.3 a z tabulky č.4 je již známý celkový náklad. Celkový náklad na jedno manuální testování všech scénářů, které podlehly automatizaci je 18 456 Kč. Celkový náklad na vývoj automatizovaného testování včetně ladění a školení testerů je 54 818 Kč. Manuální testy doposud probíhaly průměrně jednou za dva měsíce. Z toho vyplývá, že investice do automatizovaných testů se vrátí přibližně za půl roku. Tato informace je velmi příznivá, jelikož firma počítá minimálně se třemi roky dalšího vývoje. Jestliže vývoj aplikace bude opravdu trvat minimálně další 3 roky, investice do automatizovaných testů se vrátí až šestkrát.

5 Výsledky a diskuse

V této kapitole diplomové práce budou uvedeny výsledky z praktické části práce, dále budou výsledky vysvětleny a srovnány mezi sebou.

5.1 Výsledky vývoje automatizovaných testů

Vývoj automatizovaných testů trval 193 hodin, což je 24,125 dní. Lze říci, že toto číslo je velmi dobré vzhledem ke složitosti manuálních testů. Dále byly úspěšně splněny všechny požadavky od zadavatele. Velkou výhodou je rozdělení testů do samostatných skriptů. To zaručuje přehlednost a lehčí zanášení úprav do testů. Pokud tedy dojde k nějaké změně v testované aplikaci a tato změna ovlivní pouze nějakou funkcionalitu, a ne celý systém, stačí upravit ten test, který tuto funkcionalitu obsahuje.

Další výhodnou funkcí automatizovaných testů je logování, které rozlišuje jednotlivé testy, což zaručuje přehlednost a lehkou orientaci v chybách. Jednoduché je poté i dohledávání chyb v logu. Pomocí klávesové zkratky ctrl+f lze jednoduše vyfiltrovat všechny neúspěšné (NOK) záznamy. Tím se testerovi velmi usnadní dohledávání chyb. Výhodou tohoto řešení a rozdělení testů do samostatných skriptů je také to, že tester si po nalezené chybě může zopakovat pouze chybový test. Zde také velmi pomáhá testování na základě analýzy obrazu, jelikož tester může sledovat průběh automatizovaného testu a potom vizuálně vidět, jak chyba vypadá.

Nevýhodou a zároveň výhodou je to, že testování na základě analýzy obrazu pro sebe zabere celou testovací stanici. Na tomto počítači již nelze dělat nic jiného. Výhodou je již zmíněné sledování průběhu testu a vizuální kontrola chyb.

5.2 Porovnání manuálních a automatizovaných testů

Porovnání manuálních a automatizovaných testů je velmi složité. Manuální testování má své výhody oproti automatizovaným testům a naopak. Díky výsledkům z praktické části diplomové práce toto porovnání lze provést. Porovnání manuálních testů by mělo být provedeno z vícero hledisek. Hlediska, která jsou pro firmy podstatná jsou finance a kvalita. Z těchto hledisek tedy bude provedeno porovnání.

5.2.1 **Finanční hledisko**

Na první pohled se může zdát, že se manuální testování vyplatí mnohem více než testování automatizované. Zde však hraje velkou roli doba vývoje testované aplikace a také jak firma svou aplikaci testuje. Velkou roli také hraje typ aplikace a typ testů. Zde se jedná o desktopovou aplikaci a o testování s grafickým rozhraním aplikace. Všechny testy všech funkcionalit zmíněných v praktické části diplomové práce probíhají včetně grafického rozhraní. Na první pohled se opravdu zdá, že vývoj automatizovaných testů není nejvýhodnější. Díky technice analýzy obrazu bylo v tomto případě dosaženo velmi dobrých výsledků.

Celkové náklady na vývoj automatizovaných testů jsou 54 818 Kč. To je oproti jednomu manuálnímu testování skoro trojnásobná částka. Zde hraje velkou roli doba vývoje testované aplikace. V případě zvolené firmy se jedná minimálně o 3 roky vývoje. To znamená, že se bude minimálně 3 roky testovat. Jak je z praktické části diplomové práce známo, jedno manuální testování zabere celkem 11,5 pracovních dní. To je cca 92 hodin práce. Z těchto výsledků vyplývá, že jedno manuální testování stojí zvolenou firmu 18 465 Kč. Jedno automatizované testování trvá cca 16 minut. To není ani 150 Kč. Z dlouhodobého hlediska se tedy automatizované testování určitě vyplatí. Z toho vyplývá, že pokud mají firmy jistotu dlouhodobého vývoje a je zde jistota opakování testů, automatizace testů by měla být zařazena do testovacích procesů. Pokud se jedná pouze o malou aplikaci, která není náročná na vývoj a testování, automatizace není potřeba zavádět.

5.2.2 **Kvalitní hledisko**

Kvalita je pro spoustu firem na prvním místě. Zde hodně záleží na kvalitě testera. Pokud je tester kvalitní, zkušený, zapálený pro testování a dokáže prosadit svoje tvrzení, může vyvíjený software otestovat mnohem lépe než automatizované testy. Testování takového testera může být i trochu delší než normálně, ale to firmám dbajícím na kvalitu mnohdy nevadí. V případě zvolené firmy je toto hledisko pojato ještě trochu jinak. Firma má kvalitní testery, a přesto zvolila automatizaci testování. Je potřeba říci, že toto automatizované testování nemá testery nahradit. Automatizace je zde zavedena z důvodu

uvolnění kapacit testerů na podstatné problémy. Tester spustí automatizovaný test a dále se věnuje věcem, které mají větší prioritu. Firma tak získá ještě kvalitněji otestovaný software, jelikož automatizované testy drží svou kvalitu na stejné úrovni a k tomu tester může pracovat na testování nových funkcionalit, popřípadě se věnovat retestům oprav softwaru. Ve zvolené firmě je tedy zavedena kombinace automatizovaných testů a testů manuálních.

Z výsledků práce můžeme říci, že automatizace urychlí testování a dále může zlepšit jeho kvalitu. Vždy tomu tak nemusí být. Pokud je tester kvalitní, může z hlediska kvality odvést lepší práci než automatizované testy. Pokud tomu tak není, automatizace odvede lepší práci než tester. Úplné vynechání testera z testování připraví firmu například o lidský faktor, který v testování hraje také velkou roli. Člověk dokáže vymýšlet různé kombinace, které nejsou v automatizaci zavedené a tím může odhalit více chyb než automatizovaný test (jak již bylo výše zdůrazněno). Z tohoto důvodu se úplné vynechání testera nevyplácí. Nejlepším řešením je tedy kombinace manuálního testování a automatizovaného testování. Vše má svoje výhody a nevýhody, a právě kombinací testů je získáno nejvíce. To znamená, že by se měly automatizovat opakující se úkony a manuálně testovat nové funkce a retestovat opravené chyby. Dále pokud automatizace nalezne chybu v nějaké části aplikace, tester tuto část aplikace může manuálně otestovat a tím získá lepší pohled na věc. Vše výše uvedené se vyplatí v případě dlouhodobého vývoje.

5.3 Shrnutí automatizace a zhodnocení výsledků

V této části práce je uvedena aktuální situace ve zvolené firmě k datu 20. 03. 2021. Automatizované testy jsou ve firmě nasazeny od 20. 09. 2020. Nyní je to tedy půl roku od nasazení. Bylo vypočítáno, že za půl roku by se měla investice do automatizovaných testů vrátit. Jelikož firma nyní vydává spoustu drobných i větších úprav testované aplikace, rozhodla se automatizované testy spouštět při každé vydané změně. Investice do automatizovaných testů se podle zadavatelů automatizace vrátila již několikrát a firma je s testy velmi spokojena. Testeři využívají automatizované testy k běžnému testování a v průběhu testování se věnují retestování oprav. Lze tedy konstatovat, že automatizace firmě prospěla a byla úspěšně zavedena do testovacích procesů firmy. Testerům se

zjednodušila práce a uvolnily se kapacity na důležité úkony, které je potřeba vykonávat při testování opravených chyb a testování nových funkcích dodaných do aplikace.

6 Závěr

V teoretické části diplomové práce byly vysvětleny všechny důležité a potřebné pojmy zabývající se problematikou manuálního testování a automatizovaného testování. Byly vysvětleny základní pojmy testování, uvedena problematika chyb a testovacího procesu. Dále byly uvedeny role, které jsou při testování důležité a co obstarávají. Také byly vysvětleny druhy testů a kdy je vhodné jejich použití. V závěrečných kapitolách diplomové práce byly představeny technologie automatizovaného testování a technologie, ve které byla zpracována praktická část diplomové práce SikuliX.

Praktická část započala získáním požadavků na automatizované testy od zvolené firmy. Tyto požadavky byly získány na jednání se všemi zainteresovanými osobami. Následně byly tyto požadavky zpracovány a zadavateli bylo nastíněno řešení získaných požadavků. Po schválení návrhu řešení požadavků proběhl vývoj automatizovaných testů na základě testovacích scénářů a požadavků od zadavatele. Po dokončení vývoje automatizovaných testů proběhlo nasazení těchto testů do reálného provozu. Prvních pár dní probíhalo ladění testů na testovací stanice testerů a následně proběhlo zaškolení testerů. Automatizované testy jsou nyní ve firmě využívány již půl roku a velmi zjednodušily práci testerů, kteří teď mají čas se věnovat důležitějším úkonům.

Po úspěšném nasazení proběhlo porovnání manuálních a automatizovaných testů z hlediska kvality a financí. Na závěr práce byla popsána aktuální situace ve firmě a využívání automatizace v kombinaci s manuálními testy.

Seznam použitých zkratek

GUI – Grafické uživatelské rozhraní.
MIT – Massachusettský technologický institut.
IT – Informační technologie.
Atd – A tak dále.
Tzn – To znamená.
OK – Dobře.
NOK – Špatně.
IBM – International Business Machines Corporation.
API – Rozhraní pro programování aplikací.
UNIT – Jednotkové testování.
UAT – Uživatelské akceptační testování.
SIT – Systémové testování.
FAT – Funkční testování.
NIST – Národní institut standardů a technologie.
USA – Spojené státy americké.
SW – Software.
MC/DC – Modified condition/decision coverage – Upravené podmínky / pokrytí rozhodnutí.
ISO – Mezinárodní organizace pro normalizaci.
IEC – Mezinárodní elektrotechnická komise.
IEEE – Institut pro elektrotechnické a elektronické inženýrství.
ISTQB – Rada pro mezinárodní testování softwaru.

7 Seznam použitých zdrojů

[1] BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

[2] GRAHAM, Dorothy, Rex BLACK a Erik VAN VEENENDAAL. Foundations of software testing: ISTQB certification. Fourth edition. Andover, Hampshire: Cengage, [2020]. ISBN 9781473764798.

[3] Životní cyklus softwaru [Kalábovi]. Kalábovi [Kalábovi] [online]. Dostupné z: https://kalabovi.org/pitel:isz:zivotni_cyklus_softwaru

[4] Test Governance Framework for contracted IS development [online]. Copyright ©h [cit. 23.11.2020]. Dostupné z: https://nb.vse.cz/~qdolm05/files/Test_Governance_Framework_for%20Contracted_ISD.pdf

[5] Capturing Architectural Requirements [online]. 2005 [cit. 2018-03-08]. Dostupné z: https://www.ibm.com/developerworks/rational/library/4706.html#N100A7

[6] 15. ČERMÁK, Miroslav. Testování SW [online]. 2009 [cit. 2019-03-14]. Dostupné z: http://www.cleverandsmart.cz/testovani-sw/

[7] Ammann, P. a Offutt, J. Introduction to Software Testing. 1. vyd. New York: Cambridge University Press, 2008. 346 s. ISBN 978-0-521-88038-1.

[8] PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002. Programování. ISBN isbn80-722-6636-5.

[9] Software Errors Cost U.S. Economy $59.5 Billion Annually: NIST Assesses Technical Needs of Industry to Improve Software-Testing. Wayback Machine [online]. Dostupné z: https://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm

[10] KEN, Robinson. Ariane 5 Flight 501 Failure—A Case Study of Errors [online]. http://www.cse.unsw.edu.au/~se4921/PDF/ariane5-article.pdf

[11] DUSTIN, Elfriede, Tilo LINZ a H. SCHAEFER. Effective software testing: 50 specific ways to improve your testing. 4th edition. Boston: Addison-Wesley, 2002. ISBN 978-0-201-79429-8.

[12] YouTrack: The Issue Tracking and Project Management Tool for Software

Teams. JetBrains: Developer Tools for Professionals and Teams [online]. Copyright ©
2000 [cit. 08.03.2019]. Dostupné z: https://www.jetbrains.com/youtrack/

[13] AHO, Alfred V. Compilers, Principles, Techniques and Tools. Canada, 1994. ISBN 0-201-10194-7.

[14] What is Thread Synchronization? - Definition from Techopedia. Techopedia: Educating IT Professionals To Make Smarter Decisions [online]. Copyright © 2021 Techopedia Inc. [cit. 05.01.2021]. Dostupné z: https://www.techopedia.com/definition/24349/thread-synchronization

[15] Defect/Bug Life Cycle in Software Testing. Meet Guru99 - Free Training Tutorials & Video for IT Courses [online]. Copyright © Copyright [cit. 05.01.2021]. Dostupné z: https://www.guru99.com/defect-life-cycle.html

[16] HOŘANOVÁ, Andrea. Testování softwaru se zaměřením na automatizované testy. Praha, 2016. Diplomová práce. Česká zemědělská univerzita v Praze.

[17] Logování [online]. Dostupné z: https://www.it-slovnik.cz/pojem/logovani

[18] ISO / IEC 29119 - ISO/IEC 29119 ISO / IEC 29119 - https://cs.qaz.wiki/wiki/ISO/IEC_29119 [online]. Dostupné z: https://cs.qaz.wiki/wiki/ISO/IEC_29119

[19] Functional Testing - SOFTWARE TESTING Fundamentals. Software Testing Fundamentals (STF) ! - SOFTWARE TESTING Fundamentals [online]. Dostupné z: https://softwaretestingfundamentals.com/functional-testing/

[20] What is Non Functional Testing? Types with Example. Meet Guru99 - Free Training Tutorials & Video for IT Courses [online]. Copyright © Copyright [cit. 19.01.2021]. Dostupné z: https://www.guru99.com/non-functional-testing.html

[21] ASH, Lydia. The Web testing companion: the insider's guide to efficient and effective tests. Indianapolis, Ind.: Wiley Pub., c2003. ISBN 978-047-1430-216.

[22] Black box test - CleverAndSmart Management Consulting. CleverAndSmart Management Consulting [online]. Copyright © 2008 [cit. 20.01.2021]. Dostupné z: https://www.cleverandsmart.cz/black-box-test/

[23] Poeta.cz - moderní literární server [online]. Copyright © [cit. 20.01.2021]. Dostupné z: http://www.poeta.cz/Zaklady_testovani.pdf

[24] White box test - CleverAndSmart Management Consulting. CleverAndSmart Management Consulting [online]. Copyright © 2008 [cit. 20.01.2021]. Dostupné z: https://www.cleverandsmart.cz/white-box-test/

[25] Druhy, Typy A Kategorie Testů | Testování softwaru. Testování softwaru [online]. Dostupné z: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/

[26] CEM, Kaner; JAMES, Bach.; BRET, Pettichord. Lessons learned in software testing : A Context-driven approach. New York : John Wiley & Sons, 2001. ISBN 0-471- 08112-4.

[27] Brown, C. T.; Gheorghe, G.; Huggins, J.: An Introduction To Testing Web Applications With Twill And Selenium. O´Reilly. California. 2007. ISBN 0596527802, 9780596527808.

[28] PyUnit - the standard unit testing framework for Python. PyUnit - the standard unit testing framework for Python [online]. Dostupné z: http://pyunit.sourceforge.net/

[29] Nejlepší testovací nástroje - Radek Kitner. Radek Kitner - konzultant, lektor testování softwaru [online]. Dostupné z: https://kitner.cz/testovani_softwaru/testovaci-nastroje/

[30] JUnit 5. [online]. Copyright © 2021 The JUnit Team [cit. 22.01.2021]. Dostupné z: https://junit.org/junit5/

[31] Postman | The Collaboration Platform for API Development. Postman | The Collaboration Platform for API Development [online]. Copyright © 2021 Postman, Inc. All rights reserved. [cit. 22.01.2021]. Dostupné z: https://www.postman.com/

[32] Insomnia | The API Design Platform and REST Client. Insomnia | The API Design Platform and REST Client [online]. Copyright © [cit. 22.01.2021]. Dostupné z: https://insomnia.rest/

[33] SeleniumHQ Browser Automation. SeleniumHQ Browser Automation [online]. Dostupné z: https://www.selenium.dev/

[34] Katalon Solution. Katalon Solution [online]. Copyright © 2020 Katalon LLC. All rights reserved. [cit. 22.01.2021]. Dostupné z: https://www.katalon.com/

[35] About - OpenCV. Home - OpenCV [online]. Dostupné z: https://opencv.org/about/

[36] RaiMan's SikuliX. RaiMan's SikuliX [online]. Copyright © 2017 Raimund Hocke [cit. 23.01.2021]. Dostupné z: http://sikulix.com/

[37] O ústavu. Fakulta mechatroniky, informatiky a mezioborových studií [online]. Dostupné z: https://www.fm.tul.cz/ustavy/ustav-novych-technologii-a-aplikovane-informatiky/o-ustavu

[38] Faculty of Informatics Masaryk University. Faculty of Informatics Masaryk University [online]. Dostupné z: https://www.fi.muni.cz/usr/jkucera/pv109/2005/xobsivac.html

[39] Encyklopedie profesí: IT analytik. Práce.cz [online]. Praha, 2012 [cit. 2016-03-31]. Dostupné z: http://www.prace.cz/poradna/encyklopedie-profesi/i/it-analytik/

[40] Práce IT architekt. Profesia.cz [online]. Praha, 2016 [cit. 2016-03-31]. Dostupné z: http://www.profesia.cz/prace/it-architekt/

[41] Projektový manažer. Jobs.cz [online]. Praha, 2016 [cit. 2016-03-31]. Dostupné z: http://www.jobs.cz/poradna/profese/p/projektovy-manazer/

[42] Test Manager. SmartRecruiters Job Search [online]. USA, 2016 [cit. 2016-03-31]. Dostupné z: https://www.smartrecruiters.com/PrincipalEngineeringSro/72228329-test-manager

[43] Fáze a úrovně provádění testů. Testování softwaru [online]. 2016 [cit. 2019-03-14]. Dostupné z: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu/

[44] 2.9 Bugs and Debugging. [online]. Dostupné z: https://icarus.cs.weber.edu/~dab/cs1410/textbook/2.Core/debugging.html

[45] What Is the Difference between White Box Testing and Black Box Testing? - Invensis Technologies. Software Development, IT, BPO, Call Center, F&A Outsourcing, Ecommerce Support – Invensis [online]. Copyright © All Rights Reserved [cit. 27.03.2021]. Dostupné z: https://www.invensis.net/blog/difference-between-white-box-testing-black-box-testing/

[46] System Integration Testing (SIT) and User Acceptance Testing (UAT). News, Events & Blog from TM Group [online]. Copyright © All rights reserved [cit. 27.03.2021]. Dostupné z: https://news-events-blog.tm-group.com/what-is-system-integration-testing-sit-and-user-acceptance-testing-uat

[47] Životní cyklus SW [online]. Dostupné z: http://statnice.dqd.cz/mgr-szz:ap-ap:1-obr