Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Tvorba testovacího scénáře vybrané aplikace

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Viktoriia Petrova

Systémové inženýrství a informatika
Informatika

Název práce

Tvorba testovacího scénáře vybrané aplikace

Název anglicky

Creating a test scenario of the selected application

Cíle práce

Cílem bakalářské práce je seznámit čtenáře s problematikou testování softwaru a popsat teoretická východiska pro tuto problematiku.

Cílem teoretické části práce je nastudování a vymezení základních pojmů a postupů, které se při testování software používají. Dílčími cíli jsou celková charakteristika metodik testování, popis základních typů testů a jednotlivých rolí, které se vyskytují v průběhu celého procesu testování.
Teoretická část se dále zabývá jednotlivými modely životního cyklu vývoje softwaru, definicí chyb, které vznikají při vývoji a popisem manuálního a automatizovaného odvětví testování.

Cílem praktické části bakalářské práce je vytvoření testovacího scénáře zvolené aplikace pomocí dvou různých nástrojů a následná analýza a porovnání obou výsledků a metod testování.

Metodika

Metodika řešení problematiky bakalářské práce je založena na studiu a analýze odborných literárních zdrojů (včetně zkušenosti z firmy, která se zabývá testováním), které budou použity jako podklady pro teoretickou část práce.

Na základě tohoto teoretického základu bude vytvořena praktická část práce, která bude zaměřena na tvorbu testovacího scénáře pomocí dvou metod testů: manuální a automatické. Výsledky testování budou shrnuty a zhodnoceny z hlediska časového a finančního.

Doporučený rozsah práce

30 – 50 stran

Klíčová slova

vývoj software, testování software, definice chyb, manuální testovaní, automatizované testování, testovací scénáře

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.

KANER, Cem, Jack L. FALK a Hung Quoc NGUYEN. Testing computer software. 2nd ed. New York: John Wiley, 1999. ISBN 04-713-5846-0.

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.

Předběžný termín obhajoby

2019/20 LS – PEF

Vedoucí práce

Ing. Dana Vynikarová, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 24. 2. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 2. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 04. 03. 2020

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Tvorba testovacího scénáře vybrané aplikace" jsem vypracovala samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autorka uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušila autorská práva třetích osob.

V Praze dne 23. 03. 2020 _____

Poděkování

Ráda bych touto cestou poděkovala paní Ing. Daně Vynikarové, Ph.D. za poskytnuté rady a vstřícnost při tvorbě mé bakalářské práce.

Tvorba testovacího scénáře vybrané aplikace

Abstrakt

Tato bakalářská práce je zaměřena na problematiku na testovaní softwaru, kde čtenář se seznámí se základními pojmy z oblastí testingu, získá představu o procesu a metodách testování softwaru a dostane dostatečný teoretický základ. První kapitola popisuje definici testování společně s jeho cílem. Dále byla popsána struktura testovacího týmu, zmíněny metodiky pro vývoj a testování softwaru. Na to navazuje kapitola zabývající se softwarovými chybami a samotnou klasifikací testovacích metod. V posledních dvou kapitolách literární rešerše práce měla za úkol popsat nástroje určené k testování webových aplikací. Další důležitou kapitolou je základní rozdělení druhů testů, které se provádějí během testování.

Hlavním cílem praktické části je podrobně popsat proces vytvoření jednotlivých kroků testů, které mají kompletně pokrýt testovanou funkčnost. V první kapitole práce je představena samotná testovaná aplikace a popsán důvod, který vede k testování. Další dvě kapitoly se zaměřují na vytvoření testovacího scénáře zvolené aplikace s používáním několika nástrojů pro manuální a automatické testování. Dále probíhá exekuce vytvořených testovacích scénářů a zatím následuje shrnutí výsledků. Poslední kapitoly praktické části se věnují porovnání obou výsledků a metod testování.

Klíčová slova: vývoj software, testovací scénář, testování software, definice chyb, manuální testování, automatizované testování, testovací scénáře

Creating a test scenario of the selected application

Abstract

This bachelor thesis deals with software testing, where the reader gets acquainted with the basic concepts of testing, gets an idea of the process and methods of software testing and gets a sufficient theoretical basis. The first chapter describes definitions of testing together with its goal. The next chapter describes the structure of the testing teams, it also characterizes the methodologies for software development and testing. This is followed by a chapter that deals with software error and classification of test methods. In the last two chapters of the literary research the task was to describe the tools intended for testing web applications and to introduce the reader to the basic di vision of the type of tests that are performed during testing.

In the practical part, the main goal is to describe in detail the process of creating individual test steps, which should completely cover the tested functionality. The tested application is introduced in the first chapter of the thesis, here also is described the reason that leads to testing. The next two chapters are focused on creating a test scenario for the selected application using several tools for manual and automatic testing. Execution of created test scenarios is followed by summary of results. The last chapters of the practical part are devoted to comparing both results and testing methods.

Keywords: software development, test scenario, software testing, defects, manual testing, automated testing, test script

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Dnes vidíme rychlý rozvoj informačních technologií a jejich integraci do všech oblastí činnosti.

Informační technologie se v současnosti staly nedílnou součástí v lékařské, průmyslové, finanční, vzdělávací a mnoha dalších oblastech. Obecně se předpokládá, že do vytváření funkčního softwaru patří hlavně jeho vývoj. Avšak důležité jsou i kroky následující k úspěšnému dodání produktu, který má sloužit lidem. S rostoucí složitostí jednotlivých softwarových řešení napříč všemi odvětvími rostou také nároky na kvalitu. Každý uživatel chce pracovat se systémem, který vykonává své funkce správně a umožňuje dosáhnout uživateli očekávaného výsledku co nejrychleji a nejjednodušší. Softwarové produkty nízké kvality samozřejmě nevyvolávají důvěru mezi uživateli, a proto nakonec zaniknou a na jejich místo se na trhu objeví velké množství lepších softwarových řešení. Pravě k tomu, aby se program na trhu prosadil, se provádí testování. Testování softwarových produktů umožňuje nejen splnit očekávání koncového uživatele, ale také zabránit jak vlastníka a distributora, tak i tvůrce softwaru proti významným ztrátám.

Kompetentní testování je klíčem k úspěchu velkých softwarových systémů i malých softwarových produktů a je stejně důležité jako fáze analýzy a samotného vývoje aplikací. Proto není dobrým nápadem testování podceňovat. Je důležité nejen zkontrolovat, zda produkt vyhovuje požadavkům, ale také ujistit se, že je vhodný pro koncové uživatele a že je schopen vykonávat deklarované funkce. Je nutné zařídit testování jako proces, který zahrnuje přímé provedení testovacích scénářů, plánování práce, přidělení pracovních zdrojů, vytvoření testovacího modelu, sledování procesu testování spolu s analýzou výstupních kritérií, jakož i aktivitu po dokončení testování.

K úspěšnému vytvoření procesu testování se používá celá řada nástrojů. Kromě toho lze automatizovat nejnáročnější a často se opakující testovací scénáře, které zpravidla mohou výrazně zkrátit čas potřebný k ověření dostatečně velkých a složitých částí funkčnosti systému. Tomuto způsobu testování spolu s manuálním je věnována praktická část této práce, ve které jsou názorně předvedeny jejich možnosti a detailně vysvětlen samotný proces vytvoření a exekuce testovacích scénářů.

Teoretická část se zaměřuje na objasnění termínu testování software, kde čtenář se seznámí se základními pojmy z oblastí testingu, získá představu o procesu a metodách testování softwaru a dostane nezbytný teoretický základ, který umožní práce v oboru, i když zatím čtenář neměl žádné praktické zkušenosti.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem teoretické části práce je nastudování a vymezení základních pojmů a postupů, které se při testování software používají. Dílčími cíli jsou celková charakteristika metodik testování, popis základních typů testů a jednotlivých rolí, které se vyskytují v průběhu celého procesu testování.

Teoretická část se dále zabývá jednotlivými modely životního cyklu vývoje softwaru, definicí chyb, které vznikají při vývoji, a popisem manuálního a automatizovaného odvětví testování.

Cílem praktické části bakalářské práce je vytvoření testovacího scénáře zvolené aplikace pomocí dvou různých nástrojů a následná analýza a porovnání obou výsledků a metod testování.

2.2 Metodika

Metodika řešení problematiky bakalářské práce je založena na studiu a analýze odborných literárních zdrojů (včetně zkušeností z firmy, která se zabývá testováním), které budou použity jako podklady pro teoretickou část práce.

Na základě tohoto teoretického základu bude vytvořena praktická část práce, která bude zaměřena na tvorbu testovacího scénáře pomocí dvou metod testů: manuální a automatické. Výsledky testování budou shrnuty a zhodnoceny z hlediska časového a finančního.

3 Teoretická východiska

Teoretická část práce se zabývá shrnutím problematiky testování. V této problematice na začátku části byla popsána definice testování společně s jeho cílem. Dále v této části byla popsána struktura a role v testovacím týmu, zmíněny metodiky pro vývoj a testování softwaru. Na to navazuje celkem rozsáhlá kapitola zabývající se softwarovou chybou a samotnou klasifikací testovacich metod. Dále práce má za úkol popsat nástroje určené k testování webových aplikací a další důležitou částí je základní rozdělení druhu testů, které se provádějí během testování.

3.1 Co je testování

Testování softwaru je komplexní proces, který zahrnuje celou řadu typů činností, zcela analogicky procesu vývoje softwaru. Bohužel názor, že testování je jednoduchá a jednotvárná práce, která vyžaduje pouze spuštění aplikace a následný mechanický průchod testovacím scénářem krok po kroku, je stále poměrně rozšířený. Avšak ve skutečnosti, proces testování je daleko složitější a rozsáhlejší. Zpravidla sem patří sběr a analýza požadavků na aplikaci, vytvoření testovacích scénářů, příprava vyhovujících testovacích dat a nástrojů, které tým potřebuje k testování. Samotný proces testování se často provádí ve více iteracích, na konci, kterých proběhne závěrečné vyhodnocení výsledků a reportování provedeného testování. [1]

Tedy ze širšího pohledu testování softwaru je řízený proces se specifickými kroky, který musí vždy být zaměřen. Zahrnuje v sobě nejen přímé provedení testovacích scénářů, ale také plánování práce, analýzu požadavků, přidělení pracovních zdrojů, vytvoření testovacího modelu, sledování procesu testování, analýzu výstupních kritérií, jakož i aktivitu po dokončení testování. [1]

V užším smyslu pod testováním můžeme chápat kontrolu souladu mezi skutečným a očekávaným chováním programu, která je provedena na určité sadě testů vybraných určitým způsobem. [2]

3.2 Cíle testování

Mezi hlavní cíle testování softwaru patří především zajištění toho, aby software fungoval v souladu s produktovou specifikací a dosáhl požadovaného stupně spokojenosti zákazníků. Lidé se často mylně domnívají, že testování je proces, který potvrzuje správnost programu a prokazuje, že v programu nejsou žádné chyby. Přestože intuitivně tušíme,

že hledání chyb a jejich následný reporting a debugging je důležitý krok, ve skutečnosti to není podstatou testovacího procesu, a to hlavně protože není možné zaručit absenci chyb. V nejlepším případě se při testování může ukázat, že defekty jsou přítomny. Pokud se program chová správně pro solidní sadu testů, není důvod říkat, že v něm nejsou žádné chyby. [3] Testování zkoumá softwarový produkt a pomáhá získat informace o jeho kvalitě. Avšak je důležité mít neustále na paměti, že testování není synonymem pro zajišťování kvality (Qualty Assuuance). Smyslem testování není prokazování ničeho konkrétního, nýbrž snížení rizika.[4]

Dosažení cílů testování softwaru je možné podle následujících zásad:

- Povinná přítomnost procesu spouštění softwaru za účelem zjištění chyb (funkční aplikace)
- Softwarová aplikace by během testování měla zůstat nezměněna
- Testování by se mělo provádět na externí straně tedy lidmi mimo společnost, která produkt vyvinula
- Testovací případy a všechny výsledky by měly být zaznamenány do dokumentace

3.3 Role při testování

Lidská složka hraje při testování dominantní roli. Obvykle, na větších projektech, na které nestačí jeden tester, je podle potřeb projektu a daných omezení sestaven různý dle počtu členů tým, ve kterém každý má specifickou roli. Úroveň zúčastněných odborníků určuje, jak dobře a plně bude software zkoumán. Je žádoucí, aby odborníci na testování měli nezbytné znalosti v oblasti obecného zajišťování kvality výroby a aby nepatřili k vývojovému týmu (například správci databází, techničtí specialisté, koncové uživatele atd.). [5]

Minimální počet specialistů poskytujících testování softwaru ve struktuře středních a větších projektů zpravidla zahrnuje v sobě manažera testování (Test manager), analytika (Test analyst či Test engineer) a testera. Test manažer a vedoucí jsou přímo zodpovědní za organizaci celkových testovacích procesů a kontrolu zaměstnanců, zatímco analytik a tester jsou přímo zodpovědní za praktické testování softwaru.

3.3.1 Manažer testování

Manažer testování je zodpovědný na projektu za veškeré aktivity související s testováním a je se svým týmem v podstatě mezičlánek mezi analýzou a vývojem. Často spolupracuje s vedoucím testování (Test lead) či schvaluje jeho činnost – například při

vytvoření plánu testování, je třeba poznamenat, že u menších projektů bývá tato role často sloučená. Podle ISTQB (International Software Testing Qualifications Board) pod test manažerem se rozumí osoba zodpovědná za projektové řízení testovacích aktivit a zdrojů a za vyhodnocení předmětu testování, což obnáší jeho management, řízení, správu, plánování a regulaci. [6] Jeho obvyklá činnost zahrnuje volbu testovacích postupů a cílů, stanovení potřebných zdrojů pro testování a stavu testovacích scénářů, určení kritérií kvality produktu, hodnocení naplnění kritérií, tvorbu a správu průběžných a výsledných dokumentů, koordinaci samotného testování a spolupráci s vývojovým a QA odděleními. Test manažer věnuje hodně času řízení v oblasti lidských zdrojů – rozhoduje o sestavení testovacího týmu, stanoví role odborníků zapojených do testování, hodnotí jednotlivé členy a nalézá cesty ke zlepšování svého týmu. [4, 7]

3.3.2 Analytik testování

Test analytik se zabývá především tvorbou test analýzy a na základě výstupů od analytiků se snaží pochopit aplikaci, zvážit rizika a stanovit prioritu pro provádění jednotlivých testovacích scénářů tedy sestaví harmonogram provádění testů. Povinnosti analytika obvykle zahrnují tvorbu všech potřebných dokumentů pro testovací tým, vytváření a často exekuce automatických a manuálních testovacích případů, vyhodnocování výsledků testů, lokalizaci chyb, statistickou analýzu defektů softwaru a stanovení použitých funkčních testovacích metod. Tato role bývá často spojena s rolí testera.

Analytik odpovídá za podporu všech fází životního cyklu testování. [5]

3.3.3 Tester

Tester je především zodpovědný za provádění testů podle předem připravených testovacích scénářů a za následnou dokumentaci získaných výsledků, případně zjištěných závad. Poté, co vývojový tým dokončí software a odstraní zjištěné defekty, jeho úkolem je přetestování provedených testů a správa stavu nahlášených chyb. Běžná je taký příprava a údržba testovacích dat a generace zprav o testovacím prostředí – jejich funkčnost, dostupnost a použitelnost.

Projekty využívající automatizaci disponují specialisty automatizovaného testování. Tito členové testovacího týmu typicky spolupracují s analytiky testování – dle strategie a požadavků volí vyhovující nástroj a automatizují testovací případy či celé scénáře. Mezi povinnosti specialisty automatizovaného testování také patří statická a dynamická softwarová

analýza, implementace sestavování a spouštění jednotlivých testů, dokumentace výsledků testů a kontrola řádného proběhnutí testů, analýza chybových stavů a návrat z nich. [4]

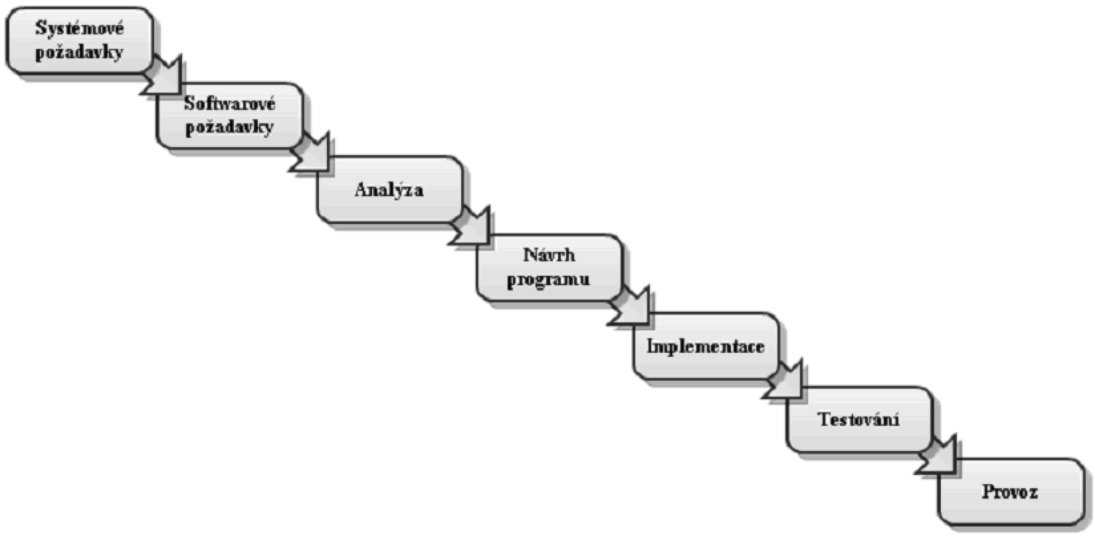
Je třeba poznamenat, že role manažera, analytika a testera se vzájemně nevylučují. Některé povinnosti vedoucího mohou být přeneseny na další členy týmu a naopak.

3.4 Modely životního cyklu vývoje softwaru

Jako modely životního cyklu softwaru dále budou zvažovány kaskádový model (vodopád), model ve tvaru V, spirálový model a flexibilní model.

3.4.1 Kaskádový model

V modelu vodopádu funkční rozklad testování lze provádět třemi způsoby: „shora dolů“, „zdola nahoru“ a „velkého třesku“. [8] Při prvním rozkladu začíná volání procedur od nejvyšší úrovně (hlavní procedury a funkce) a poté jsou kontrolovány podřízené procedury. V každém přechodném bodě jsou funkce nižší úrovně nahrazeny jedinečnými útržky – jednorázovým kódem. Rozklad „zdola nahoru“ je obrácená sekvence, ve které kontrola začíná konečnými bloky a funkcemi a končí hlavními bloky. S přístupem „velkého třesku“ jsou všechny bloky, procedury a funkce testovány současně. Problémem této metodiky je především to, že tato metodika je v podstatě nastavena na bezproblémový průchod a je nepružná, neboť neumožňuje některé fáze opakovat. Z pohledu testování je nepříjemné, že se předpokládá testování až kompletně vyvinuté aplikace, a proto v moderních projektech není kaskádový model prakticky použitelný. [7, 9]



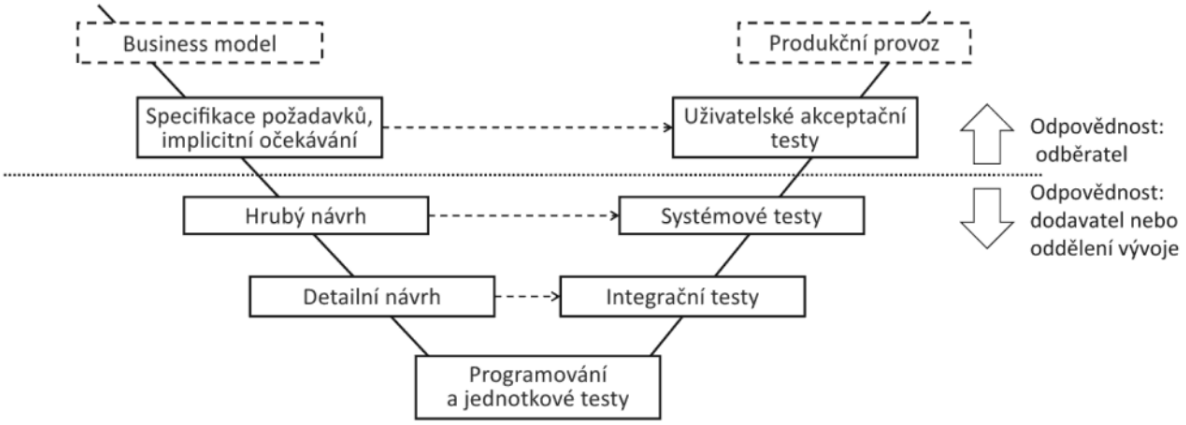
Obrázek 1: Kaskádový model [16]

3.4.2 V-Model

V-Model je logickým vývojem vodopádu. [10] Je třeba poznamenat, že obecně jak vodopádový, tak i V-Model životního cyklu softwaru má stejnou sadu fází, ale zásadním rozdílem je způsob, jakým jsou tyto informace použity v procesu implementace projektu. Hlavním rysem je, že vývoj a testování jsou stejně důležité procesy.

Model V používá dobře rozvinutou strukturovanou metodu, ve které je každá fáze implementována po podrobné dokumentaci předchozí fáze. Vývoj testovacích aktivit a testovacích scénářů se provádí na úplném začátku projektu, ještě před přímým programováním aplikace a tím šetří velké množství času na uskutečnění celého projektu. Účelem V-modelu je zvýšit působnost a efektivitu vývoje softwaru a zajistit vztah mezi testovacími a vývojovými postupy, jak je znázorněno na obrázku č. 2.

V-model je nejtradičnější model používaný pro testování softwaru.



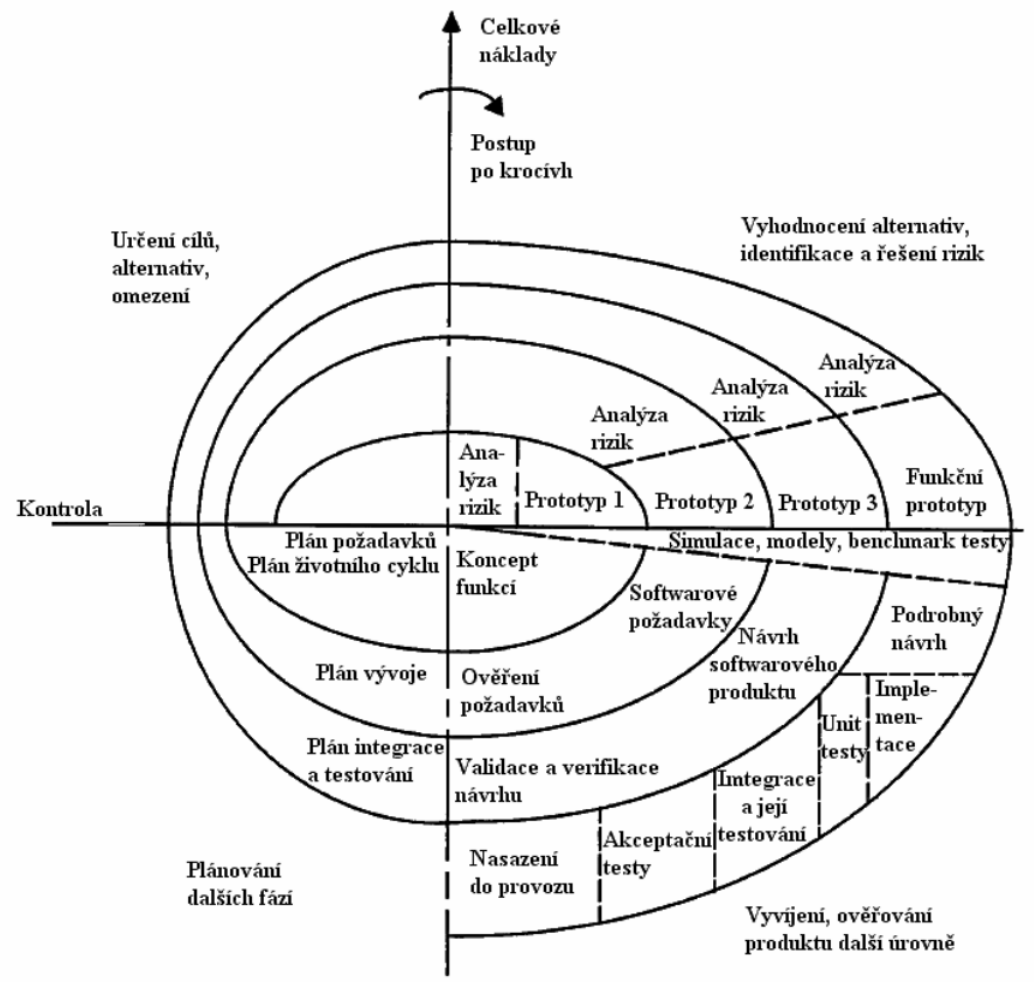
Obrázek 2: V-Model [14]

3.4.3 Spirálový model

Spirálový model je specifickým případem iteračního inkrementálního modelu, ve kterém je zvláštní pozornost věnována řízení rizik, která ovlivňují zejména organizaci procesu vývoje projektu a kontrolních bodů. [8, 11] Největší rozdíl oproti základnímu modelu spočívá v tom, že přírůstky jsou stanoveny hlavně s ohledem na míru rizika a nikoli na základě požadavků klienta. Tento model je ideální pro větší projekty.

Spirála je položena na souřadnou rovinu x–y, přičemž horní levý kvadrant souvisí s definováním cílů, pravý horní slouží pro analýzu rizik, dolní pravý pro vývoj (a testování) a dolní levý pro plánování další iterace. Tyto čtyři fáze – stanovení cílů, analýza rizik, vývoj

a testování, jakož i plánování další iterace – se opakují evolučním (iteračním) způsobem. V každé fázi vývoje se spirála rozšiřuje.



Obrázek 3: Spirálový model [16]

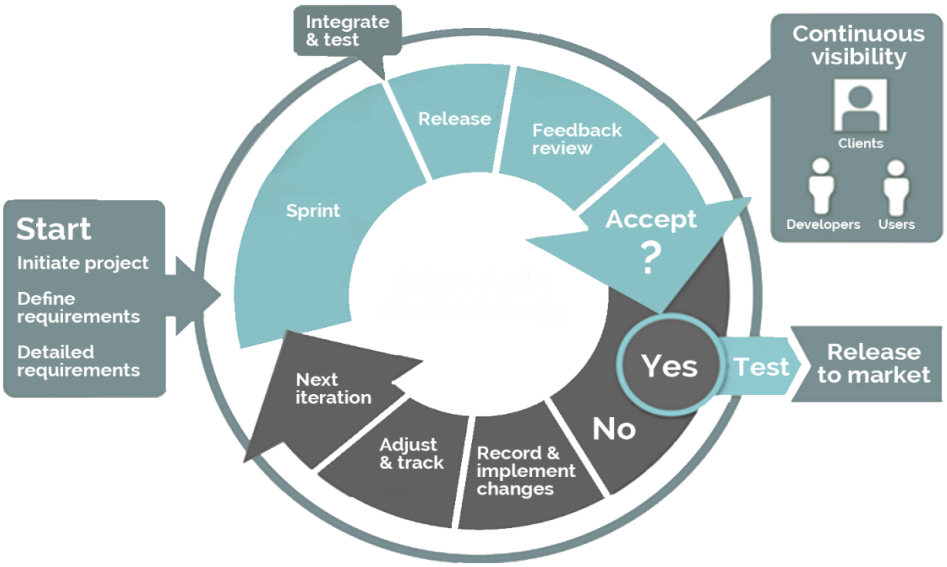
3.4.4 Agilní metod

Flexibilní model (agilní model) je skupina metod původně určených pro vyvíjení softwaru a je založen na tzv. manifestu agilního programování, který byl napsán v únoru 2001 sedmnácti poradci Agile Alliance. Tento dokument byl přeložen do 42 jazyků a radikálně změnil svět vývoje a testování softwaru. Hlavní vlastnosti všech flexibilních modelů životního cyklu softwaru jsou:

- Focus zaměření na zákazníka
- Postupný vývoj zdola nahoru
- Flexibilita, pokud jde o požadavky
- Fázová dodávka hotových funkčních komponentů

- Funkční software je důležitější než vyčerpávající dokumentace

Přístupy založené na flexibilním modelu jsou logickým vývojem a pokračováním všeho, co bylo vytvořeno a otestováno po desetiletí u dříve uvažovaných vodopádových, spirálových a V-modelů. [12] Obrázek č. 4 ukazuje diagram modelu flexibilního životního cyklu softwaru.



Obrázek 4: Agilní metod [20]

3.5 Chyba

3.5.1 Definice chyby

Se složitostí jednotlivých softwarových řešení se také zvyšuje i potřeba jejich spolehlivosti, bezpečnosti a udržovatelnosti. Software jako produkt je výsledkem spolupráce různých specialistů, kteří neúmyslně, ale budou zavlékat defekty, způsobující chyby a selhání do procesu softwarového vývoje. Přestože intuitivně tušíme, co je chápáno pod termínem „defekt“, pro bližší porozumění problematiky, je důležitě vymezit a nastudovat některé základní pojmy a principy související s ní.

Z hlediska specifikace a požadavků Ron Patton definuje chybu za předpokladu, že je splněna alespoň jedna z níže uvedených podmínek:

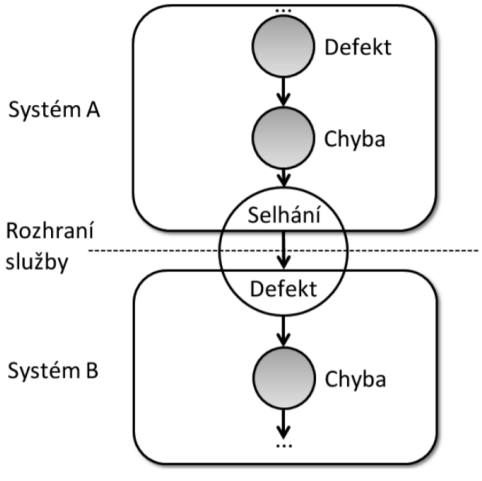
- Software vykonává něco, co není specifikováno
- Software vykonává něco, co by podle specifikace vykonávat neměl
- Software nevykonává něco, co by podle specifikace měl

- Software nevykonává něco, o čem se produktová dokumentace nezmiňuje, ale měla by zmiňovat
- Software je těžko pochopit, obtížně se používá, je pomalý nebo – podle názoru testera softwaru – nebude koncovým uživatelem považován za správný.

Tedy v širším smyslu lze mluvit o defektu jako o chybě v programu nebo v systému, díky níž program způsobuje neočekávané chování a v důsledku poskytovaná systémem služba neimplementuje jeho funkci. [1]

Z této definice plynou tři další velmi blízké, avšak vzájemně nezaměnitelné ponětí: defekt, chyba a selhání.

Termín defekt se obvykle používá k označení vad, které se projevují ve fázi programového provozu a jsou nejčastěji následkem pochybení člověka, programátora či analytika anebo kohokoli jiného, kdo se ve vývoji zúčastni. Defekt většinou pak způsobí chybu ve stavu interního komponentu systému, která následně může vést k selhání. O selhání mluvíme, pokud služba, která poskytována systémem, se odchýlí od specifikovaného chování systému. Tento problém nastane ve chvíli, kdy se chyba stane součástí externího stavu systému a bude zřejmá pro uživatele jako událost, při které není komponenta schopna vykonat požadovanou funkcionalitu v rámci definovaných podmínek. [4]

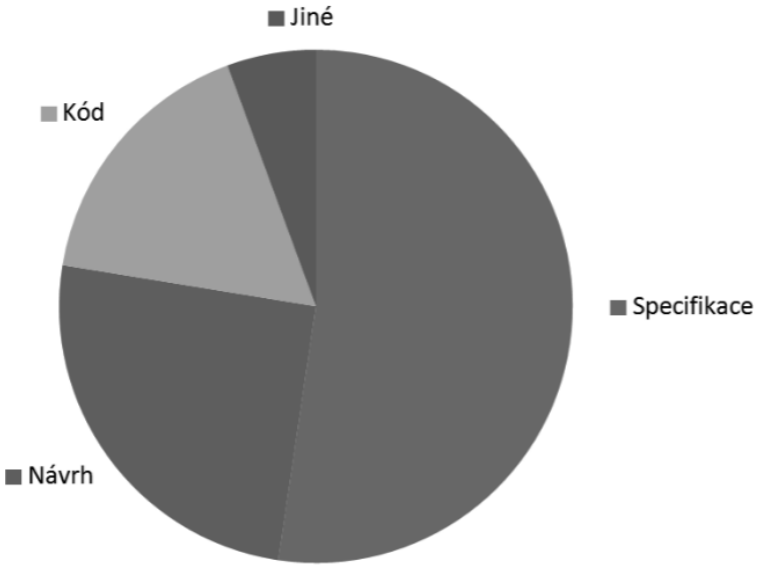


Obrázek 5: Šíření chyby [14]

3.5.2 Důvod vzniku chyby

Existuje několik důvodů, proč se defekty vyskytují. Často se lze setkat s tvrzením, že nejčastější příčinou je specifikace či návrh, kde zpracovatel nedostatečně promyslí nějaký proces nebo vůbec na něj zapomene.

Jak je to vidět na grafu uvedeném níže chyba, která vznikla kvůli specifikaci je prvním a hlavním důvodem vzniku defektů, dalšími zdroji jsou návrh systému nebo funkcionality, kde vývojář to udělá způsobem, který nemůže v praxi fungovat, samotná implementace a chybný programový kód anebo veškeré ostatní důvody, do kterých obvykle spadají omyly testerů apod. [1]



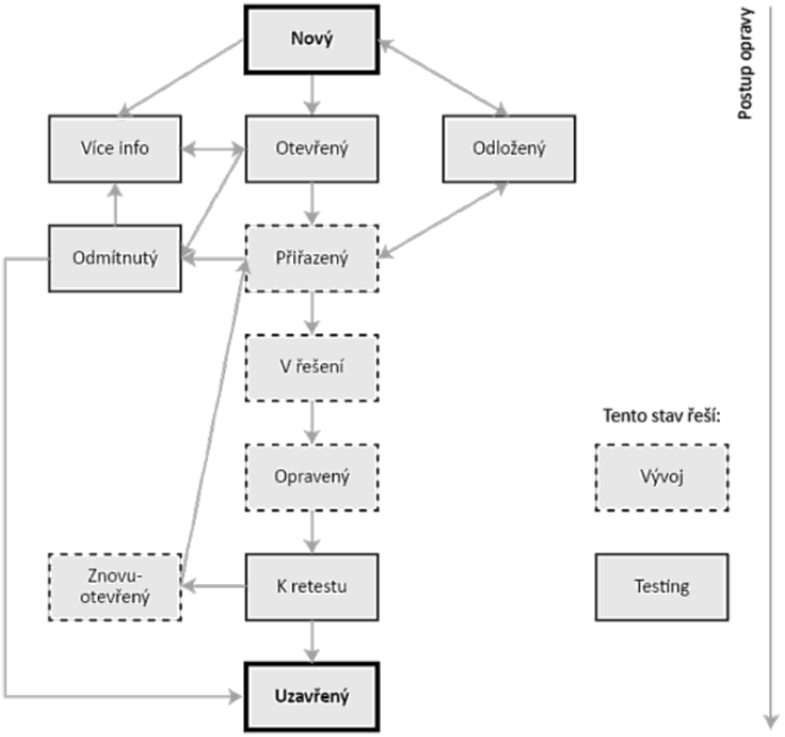
Obrázek 6: Rozložení příčin defektů [1]

Paul Jorge sen v knize „Software Testing: A Craftsman's Approach“ uvádí mezi hlavními příčiny výskytu chyb nekvalitní interakci mezi vývojáři v rámci projektu, komplexní logickou konstrukci vyvíjeného softwaru, opakované úpravy požadavků během projektu, nesprávnou dokumentaci zdrojového kódu a omezené časové rámce, chyby vývojářů softwaru, nekvalitní nástroje pro vývoj softwaru, nedostatek kontroly zdrojového kódu, komplexní softwarové architektury nebo nedostatečné financování velkého projektu. [13]

3.5.3 Životní cyklus chyby

Obecně platí, že jedním s hlavních výstupů, které vytváří každý testovací tým, jsou záznamy o chybách. Z pohledu řízení testování je nutno, aby každý defekt report prošel

určitým životním cyklem, od nahlášení testerem, přes opravu defektu vývojáři, až po uzavření defektu, přičemž počet fází, kterými prochází chyba, se může v jednotlivých aplikacích lišit. Obrázek č. 7 ilustruje příklad životního cyklu defektu a znázorňuje jednotlivé stavy a přechody mezi nimi.



Obrázek 7: Životní cyklus chyby [14]

ISTQB neuvádí žádná doporučení, jak by měl životní cyklus defektu vypadat tedy neexistuje žádná standardizovaná podoba.

Chyba v programu začíná „žít“ v okamžiku návrhu, implementace či jiného výstupu, který vývojový tým vytvoří. Takový defekt považujeme jako neaktivní do té doby, než dojde k jeho aktivaci provedením příslušného kódu vývojářem nebo testerem. Chyba může být nalezena řadou způsobů, avšak platí, že každá chyba by měla být zaevidována. Záznam o chybě je přesný a trvalý záznam o selhání softwaru, který musí co nejpřesněji popsat chování aplikace a způsob, jakým bylo tohoto chování dosaženo.

Každá chyba by měla být označena atributy, které pomáhají přesněji kategorizovat defekt, odhalit místa jeho vzniku a usnadnit následné přetestování po opravě. Klíčové atributy chyby podle standartu ISTQB jsou následující:

- Datum a čas nalezení
- Očekávaný a skutečný výsledek

- Název – měl by poskytovat přesní shrnutí celého záznamu a dostatek informací k pochopení podstaty chyby. Nesmí být krátký, ani příliš dlouhý a vágní
- Popis – poskytuje informace, které nejsou zřejmé ze samotného názvu. Zasahuje jak shrnutí chyby, očekávané a skutečné výsledky tak i příslušní logy, screenshoty obrazovky, videa apod.
- Stav defektu – popisuje práce, které byly provedeny. Nové chyby jsou ve stavu „Aktivní“ a zůstanou v něm, dokud nebude nalezeno řešení. Následně přejde do stavu „Vyřešená“ a čeká na přetestování. Poté, pokud v softwaru nejsou detekovány žádné závady, je stav zkontrolován a upraven na „Uzavřená“. Pokud chyba přetrvává i po opravě vývojářem, tester změní stav na „Aktivní“ a postup pro její odstranění je duplikován.
- Číslo chyby
- Oblast předmětu testování a stav testovacího prostředí
- Kroky, vedoucí k reprodukci – musí být maximálně stručné a pochopitelné
- Přiřazení – osoba, která bude zodpovědná za správu chyby (opravení či její předání někomu jinému)
- Severita – závažnost popisuje míru negativního dopadu na systém, jeho službu nebo zákazníka a je zpravidla označená jako číslo od 1 do 4 anebo pomocí abecedního označení, kde A (fatal) nejvyšší závažnost a znamená fatální, kritickou chybu, která zastavuje celý systém. B (Critical) je kritická závažnost, při které defekt způsobuje nepoužitelnost nebo brání plnému využívání požadované funkcionality systému. C (Major) chyba nemá žádný významný vliv na použitelnost systému, avšak může ovlivnit přijetí produktu zákazníkem. D (Minor) jsou kosmetické chyby, drobné nedostatky.
- Priorita opravy definuje pořadí, v jakém bychom měli odstranit závadu
- Výsledek – pokud chyba není detekována, pak je stav Odmítnuto. Duplicitní: existují dva rozdílné záznamy o stejné chybě s opakujícím se obsahem. Odloženo: vada nemá prioritu nebo bude vyřešená v budoucích releasech. Nelze reprodukovat: nelze nasimulovat chování aplikace, při kterém došlo k chybě [7, 14]

3.6 Klasifikace a metody testování

3.6.1 Statické a dynamické testování

Statické testování je typ testování, při kterém se nespustí zdrojový kód a nespustí se aplikace. Při tomto typu testování se zdrojový kód a dokumenty pro softwarový projekt kontrolují ručně nebo automaticky**.** Statické testování začíná v raných fázích vývoje a někdy se také nazývá ověřovacím testováním. Metody použité během statického testování zahrnují:

- inspekci (formální způsob kontroly)
- studijní návod
- technickou kontrolu (soulad zdrojového kódu se specifikacemi; obvykle se tady zvažují plány, strategie a testovací scénáře)
- neformální kontrolu (zdrojový kód je neformálně zkontrolován a jsou poskytovány neformální komentáře)

Během dynamického testování je spuštěn zdrojový kód aplikace. Jsou zavedeny různé konfigurace počátečních hodnot, výsledek je zkontrolován a výstup je porovnán s očekávanými.

Při tomto přístupu se kontroluje funkční chování aplikace, monitoruje se systémová paměť, doba odezvy procesoru a výkon systému. Dynamické testování se používá, když je potřeba vyhodnotit hotový software. Dynamické testování lze klasifikovat podle následujících kritérií:

- úroveň přístupu ke zdrojovému kódu (testování pomocí metod „bílá skříňka“, „černá skříňka“ a „šedá skříňka“)
- úroveň testování (unit, integrace, systémové, akceptace)
- rozsah softwaru (funkční, zatížení, testování, zabezpečení atd.) [3]

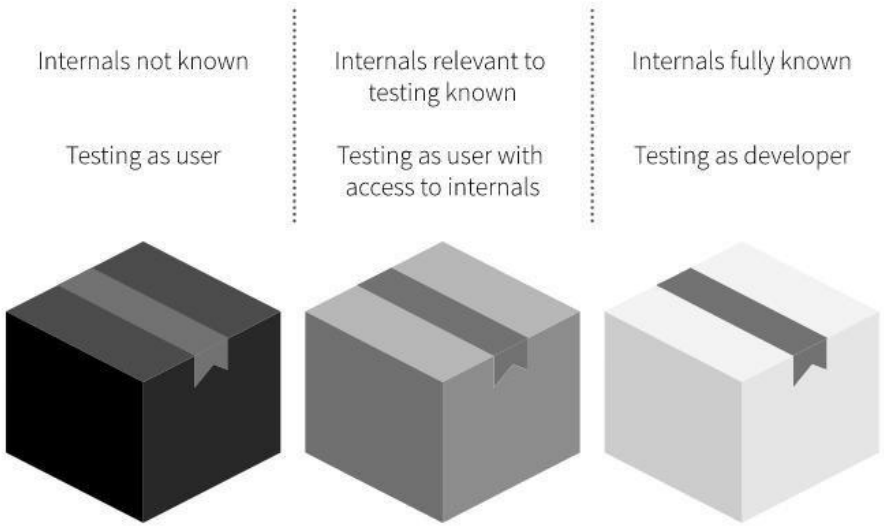
3.6.2 Funkční a nefunkční testy

Funkční testování je typ testování zaměřený na ověření správného fungování funkčnosti aplikace (správná implementace funkčních požadavků). [10] Funkční testování je často spojeno s testováním pomocí metody black box, ale je také možné ověřit správnost implementace funkce pomocí metody white box.

Nefunkční testování je typ testování zaměřený na kontrolu nefunkčních funkcí aplikace (správná implementace nefunkčních požadavků, jako použitelnost, kompatibilita, výkon, zabezpečení atd.).

3.6.3 Černá, bílá a šeda skřínka

Při těchto typech testování se používají parametry přístupu ke kódové a softwarové architektuře. [15]



Obrázek 8: Testování metodou Black Box, Grey Box a White Box [21]

Při testování pomocí metody „black box“ tester má přístup k softwaru pouze prostřednictvím stejných rozhraní jako zákazník a uživatel. Testování černé skříňky se zpravidla provádí pomocí specifikací nebo jiných dokumentů, které popisují požadavky na systém. Při tomto typu testování se snaží zajistit pokrytí požadavků a vstupních dat. Je nejvhodnější pro rychlé testování skriptů a používá se k detekci následujících chyb:

- nesprávné nebo chybějící funkce
- chyby rozhraní
- chyby v datových strukturách
- chyby výkonu
- chyby inicializace a ukončení

Na této úrovni testeři nezkoumají vnitřní fungování komponent softwarového produktu, přesto je implicitně kontrolují. Testovací skupina zkoumá vstupní a výstupní data softwarového produktu: například tester, který nezná vnitřní strukturu webu, testuje webové stránky pomocí prohlížeče, ověřuje funkčnost zařízení pro poskytování vstupních dat (kliknutí, stisknutí kláves) a kontroluje, zda se očekávané výsledky shodují se skutečným chováním.

Mezi výhody techniky černé skříňky patří účinnost, nestrannost, nenápadnost (není vyžadován přístup ke zdrojovému kódu), snadná implementace, nevyžaduje znalosti a předchozí zkušenosti s programováním.

Nevýhody techniky černé skříňky zahrnují nízkou lokalizaci, neefektivní autorské testování, přítomnost slepých míst, redundanci.

Testování „white box“ je testování, které přesahuje rámec uživatelského rozhraní a během kterého jsou prověřovány všechny podrobnosti o aplikaci. Tato metoda získala svůj název kvůli tomu, že aplikace pro testera vypadá jako bílá (průhledná) krabička, uvnitř které je vše jasně viditelné. Testování bílé skříňky je opakem testování černé skříňky. Při testování pomocí této metody má tester přístup k testovanému zdrojovému kódu a může s ním propojit testovací kód. Tato situace je typická pro testování jednotek, ve kterém se testují pouze jednotlivé komponenty systému. Při testování metodou bílé skříňky lze použít znalosti o vnitřní struktuře testovaného softwaru, včetně toho jak se probíhá ověření a zpracování platných, hraničních a nesprávných údajů. Kromě toho tento typ testování umožňuje vyhodnotit úroveň pokrytí kódu testy.

Cílem této metody není identifikovat chyby syntaxe, protože obvykle kompilátor detekuje vady tohoto druhu. Metody bílé schránky jsou zaměřeny na lokalizaci logických chyb, které jsou obtížněji odhalit, najít a opravit. Například tester zkontroluje zdrojový kód pro implementaci konkrétního pole na webové stránce, určí všechna platná a neplatná vstupní data a zkontroluje výstup na základě očekávaných výsledků, což se také určuje prozkoumáním implementačního kódu.

Výhody techniky bílé skříňky zahrnují zvýšenou účinnost, schopnost studovat úplnou cestu zdrojového kódu, včasnou identifikaci vady a identifikaci skrytých nedostatků ve zdrojovém kódu.

Hlavní nevýhodou techniky bílé skříňky je nutnost specializovaných nástrojů pro analýzu zdrojového kódu.

Účelem testování šedé skříňky je najít vady, které vzniknou v důsledku nesprávné struktury zdrojového kódu nebo nesprávného použití aplikací. Šedé testování je také známé jako testování, kde tester má přístup ke zdrojovému kódu, ale při přímém provádění testů přístup ke kódu obvykle není vyžadován. Testování se provádí stejným způsobem jako v metodě „black box“, avšak při vytvoření testovacích scénářů se využívají znalosti o vnitřní

struktuře programu. Testování v šedé skříni je vhodné pro webové aplikace, které mají distribuovaný systém, a obecně pro funkční testování.

Mezi výhody techniky šedé schránky patří kombinovaný zisk z použití přístupů dvou hlavních metod, inteligentní tvorba testovacích skriptů a nestrannost.

Nevýhody techniky zahrnují částečné pokrytí zdrojového kódu a celkovou složitost identifikace chyb.

3.6.4 Manuální a automatizované testy

Manuálnímu testování lze rozumět jako interakci profesionálního testera a softwaru za účelem hledání chyb. Tedy během manuálního testování je možné získat zpětnou vazbu, což neumožňuje automatické ověření. Jinými slovy, při přímé interakci s aplikací tester může porovnat očekávaný výsledek se skutečným a nechat doporučení. [15] Současně odborníci provádějí kontrolní testy a generují zprávy bez pomoci nástrojů pro automatizaci testování softwaru. Jedná se o klasickou metodu, která zaručí nalezení chyb softwaru.

Manuální testování je proces hledání závad v programu, kdy tester kontroluje výkon všech součástí programu, jako by byl uživatelem. Pro přesnost testu tester často používá předem připravený plán testu, který nastiňuje nejdůležitější aspekty programu. Systematický přístup k testování zahrnuje několik fází:

1. Volba metodiky testování, získání potřebného vybavení (počítače, software), přijetí lidí na pozici testerů;
2. Příprava testů s popisem implementace a očekávaného výsledku;
3. Předání testovací sady testerům, kteří manuálně provádějí testy a zaznamenávají výsledky;
4. Přenos výsledků testů vývojářům v podrobné zprávě s popisem všech identifikovaných problémů pro diskusi a opravu vad.

Ruční testování může trvat hodně času, ale z krátkodobého hlediska ušetří mnohem více peněz. Jeho cena závisí pouze na testeru, a ne na automatizačních nástrojích. Avšak manuální testování lze použít pouze u programů, které mají omezený počet případů použití. Při vývoji složitých softwarových systémů jsou možnosti ručního testování velmi omezené, protože při provádění změn v kódu je nutné organizovat opakované provádění testů. Nicméně s ručním testováním mohou být detekovány extrémně sofistikované chyby, což je velmi obtížné pomocí automatizovaného testování.

Automatické testování používá specializovaný software k provádění testů a ověřování správnosti výsledků, což zjednodušuje testování a zkracuje jeho trvání. V automatizovaném testování se používá předem připravený skript (skript), který je spuštěn automaticky za účelem porovnání skutečného výsledku softwarového provozu s očekávaným. [15] Tato metoda testování pomáhá optimalizovat časové náklady, ale má vyšší pravděpodobnost promeškat chybu. Hlavní výhodou automatizovaného testování je schopnost opakovat testy bez zásahu člověka.

Tradičním a nejoblíbenějším způsobem mezi vývojáři je organizovat automatizaci testů na úrovni kódu. Automatizované testování na úrovni kódu je často kritizováno za nemožnost otestovat uživatelské rozhraní programu. Příznivci TDD však ukázali, že při správném použití řady vzorů MVC (Model-View-Controller) je možné organizovat softwarovou simulaci uživatelských akcí bez použití grafického uživatelského rozhraní (GUI). Tento přístup nám umožňuje organizovat testování obsluhy akcí uživatelů a ponechat pouze část související s přímým zobrazením dat, na které se testy nevztahují.

Druhým způsobem automatizace testování je simulace akcí uživatelů pomocí speciálních nástrojů (testování GUI). Tento typ testování se týká testování černé skříňky.

Existují čtyři generace nástrojů a technik pro organizování testování GUI.

1. Nástroje pro snímání/přehrávání (capture/playback tools) zaznamenávají akce během ručního testování, což výrazně zvyšuje produktivitu a eliminuje opakování jednotvárných akcí. Hlavní nevýhodou nástrojů této generace je to, že jakákoli změna umístění vizuálních prvků programu vede k nutnosti opakovaného záznamu ručních testů.
2. Skript (skriptování) - forma automatizace testování pomocí specializovaných skriptovacích jazyků.
3. Testování založené na datech – metodika automatizace testování založená na použití parametrů provádění testu ve skriptech. Tento přístup umožňuje organizovat provádění skriptů s různými sadami vstupních parametrů a zvýšit flexibilitu testování.
4. Při použití testování založeného na klíčových slovech je vytvořen specializovaný slovník klíčových slov, který popisuje systémové události (například „Přihlašovací uživatel“). Každé klíčové slovo je spojeno s nezbytnými parametry (například: „UserID“, „Heslo“) a očekávanými výsledky. Pro každé klíčové slovo musí být uveden popis. Tento přístup

umožňuje psát funkční testy téměř přirozeným jazykem, aniž by byly nutné programovací dovednosti testerů. [4, 7]

Automatizované testy jsou zpravidla regrese (od lat. Regrese – zpětný pohyb), tj. jsou zaměřeny na detekci chyb v již testovaných částech zdrojového kódu při provádění změn.

Jedním z hlavních problémů automatizovaného testování je jeho složitost: navzdory tomu, že automatizace umožňuje odstranit část rutinních operací a zrychlit provádění testů, lze na aktualizaci testů vynakládat velké prostředky. Tyto investice jsou však ve většině případů odůvodněné, protože ruční testování vyžaduje mnohem více zdrojů.

3.7 Nástroje pro testování

Správný výběr vhodného nástroje je důležitým krokem, na kterém závisí úspěšnost celého procesu testování. V dnešní době existuje vcelku široký rozsah nástrojů sloužících k testování jak z řad velkých komerčních řešení, tak i volně dostupných produktů. Software, který tester využívá přímo pro testování, lze rozdělit na základě použití do několika kategorií:

- Bugtracking – softwarový nástroj, který se používá pro evidenci a správu nalezených chyb
- Test Management – typ nástrojů zaměřených přímo na testování. Jedná se o systémy pro správu testovacích případů, které slouží k definování, vytváření, uchovávání a provádění testovacích případů, a to buď přímo s použitím nějakého nástroje automatizovaného testování, nebo jen v podobě manuálních testů, které vyžadují, aby tester ručně provedl všechny uložené kroky. Nástroje test managementu se v mnoha směrech podobají již zmíněným bugtrackingovým systémům a jsou velice často s nimi spojeny, čím umožňují sledovat důležité metriky, jako je počet spuštěných testů a procento jejich úspěšnosti, průchodnost. Díky tomu se zvyšuje podrobnost dostupné informací, které mají k dispozici vedoucí testovacího procesu.
- Nástroje pro automatizované testování – jsou systémy, které umožňují automatizovat část nebo celý proces testování, používají se u větších projektů pro regresní testy a testování souladu aplikace se specifikací. [9]

Tato kapitola se především věnuje nástrojům, které jsou určeny k automatizaci a správě testovacích případů.

3.7.1 HP Application Lifecycle Management

Software Application Lifecycle Management, který je momentálně vyvíjený nadnárodní společností HP (Hewlett-Packard) poskytuje centralizovanou platformu pro správu a automatizaci činností potřebných pro základní životní cyklus aplikací a pomáhá organizacím řídit kompletní správu aplikací od plánování projektu, shromažďování požadavků až po testování a nasazení, což je jinak časově náročná úloha. ALM je software, který sdružuje v sobě sadu vestavěných modulů jako Dashboard, Management, Requirements, Testing a Defect. Všechny tyto nástroje tvoří integrální celek. Jsou mezi sebou provázané tak, že informace z jednoho se propisují do všech ostatních dotčených nástrojů. To je extrémně výhodné pro přehled a reportování průběhů testování, jeho chybovost apod.

Modul Management se především používán pro plánování a sledování projektu. Pomáhá vytvářet a spravovat releasy a cykly, což je první krok před vytvořením jakékoli pracovní položky, jako jsou požadavky, testy nebo defekty. V modulu průběh procesu správy aplikací lze sledovat v reálném čase zjišťováním, kolik chyb bylo vyřešeno a kolik jich stále zůstává otevřených. Výsledky pak mohou být analyzovány na úrovni releasu nebo cyklu a zkontrolovány, jak dobře odpovídají cílům vydání.

Requirements podrobně popisují veškeré potřeby aplikace a požadavky, které zákazník na vyvíjenou aplikaci má. Modul se používá jako základ pro vytvoření plánu testování. Těmito požadavky by se měli zabývat testy, které analytik vytvoří během fáze plánování

Dalšími hodně využívanými a z hlediska testování nejpodstatnějšími moduly jsou Testing a Defects. První slouží k vytváření a správě testovacích případů, podporuje editování, mazání a uložení. Podmodul Tets Lab umožňuje provádět manuální a automatizované testy a kontrolovat výsledky testů. Modul Defects v ALMu je systém sledování chyb integrovaný se všemi již zmiňovanými moduly. To znamená, že je možné zaregistrovat a sledovat chybu jak v požadavku (modul Requirements), tak i v samotném testovacím procesu (Testing). [17]

3.7.2 Visual Studio

Visual Studio, je integrované vývojové prostředí (IDE) od společnosti Microsoft. Může být použito pro vývoj konzolových aplikací, počítačových programů, webů, webových aplikací, webových služeb a mobilních aplikací. Aktuálně je na trhu Visual Studio 2019.

Program umožňuje zároveň sledování chyb a správu testovacích případů. Testování jednotek je k dispozici ve všech edicích Visual Studio. Další testovací nástroje, jaké jsou testování živých jednotek, IntelliTest a testy s kódovaným uživatelským rozhraním, jsou k dispozici pouze ve verzi Visual Studio Enterprise.

3.7.3 Selenium

Selenium je open source nástroj, který se používá k automatizaci testů prováděných na webových prohlížečích. Je vyvinut v programovacím jazyku Java a je možné ho používat na různých platformách. Dnes Selenium existuje ve čtyřech verzích: Selenium RC, Selenium IDE, Selenium Grid a Selenium WebDriver.

Selenium RC (Remote Control) je prvním produktem z nástrojů rodiny Selenium, který umožňují vytváření automatických testů ve velké škále programovacích jazyků. Skládá se ze dvou komponent:

- Selenium RC Server – komunikuje pomocí jednoduchých požadavků HTTP GET / POST.
- Selenium Core – knihovna JavaScriptových funkcí, které interpretují a spouští Selenese příkazy uvnitř internetového prohlížeče.

Selenium IDE (Integrated Development Environment) je produkt japonského vývojáře Shinya Kasatani. Jedná se o rozšíření pro webové prohlížeči Chrome a Firefox a obecně je nejúčinnějším způsobem vývoje testovacích případů. Test je možné vytvořit jeho nahráním, nástroj zaznamenává akce uživatelů v prohlížeči pomocí existujících příkazů Selenium s parametry definovanými v kontextu daného prvku. Hlavní výhodou nástroje Selenium IDE je přístup, který umožňuje vytváření automatizovaných testů i uživatelům, kteří nemají žádné zkušenosti s programováním.

Selenium Grid vyvinul Patrick Lightbody a byl používán v kombinaci s RC pro exekuci testů na vzdálených serverech. Ve skutečnosti lze pomocí Gridu provádět více testovacích skriptů současně na více počítačích.

Selenium WebDriver je často vnímán jako upgrade na RC, jeho největší výhodou je možnost využití objektově orientovaného API pro tvorbu automatizovaných testů. Je mnohem rychlejší hlavně proto, že není nutné používat Selenium Server ke spouštění testů, WebDriver volá každý prohlížeč sám o sobě. Podporuje různé programovací a skriptovací jazyky jako je Java, C #, Python nebo Ruby. Navíc existují knihovny poskytující podporu pro další jazyky, např. Perl nebo PHP. WebDriver API umožňuje pracovat s HTML a JavaScript aplikacemi v internetových prohlížečích. I když je Firefox nativním prohlížečem, není závislý jen na něm, podporuje např. Internet Explorer, Safari, Google Chrome a mnoho dalších.

3.7.4 JUnit

Junit je framework určený pro jednotkové testování pro programovací jazyk Java. Tento nástroj patří do rodiny xUnit frameworků pro různé programovací jazyky a je pravděpodobně

nejpoužívanějším rámcem pro psaní a spouštění testů. JUnit nevyžaduje server pro testování webové aplikace, což urychluje proces testování.

Framework JUnit také umožňuje rychlé a snadné generování testovacích případů a testovacích dat. Balíček org.Junit se skládá z mnoha rozhraní a tříd pro testování JUnit, jako jsou Test, Assert, After, Before, atd. [18]

3.7.5 NUnit

NUnit je open-source framework pro vývoj jednotkových testů na platformě .NET, podporovaný Visual Studio a Selenium WebDriver. Jedná se o portaci frameworku JUnit určeného pro Javu. NUnit je nejpoužívanější framework pro testování jednotek nad všemi jazyky platformy Microsoft .NET především z toho důvodu, že prezentuje výsledky testů v čitelném formátu a umožňuje testerovi ladit automatické testy.

Základními atributy tohoto frameworku jsou:

- TestFixture – atribut, označující třídu, která obsahuje testy a případně metody spojené s testováním, jaké jsou například SetUp a TearDown.
- SetUp – metodu sloužící k přípravě běhového prostředí, spuštěna před každým testem v aktuální sadě.
- Test – označuje metodu, která pokrývá určitý testovací případ tedy konkrétní jednotkový test
- TearDown – pro metody, které jsou automatické volány po skončení jednotlivých testů
- Values – se používá k určení sady hodnot, které mají být poskytnuty pro jednotlivý parametr parametrizované testovací metody. Protože NUnit kombinuje údaje poskytované pro každý parametr do sady testovacích případů, musí být poskytnuty údaje pro všechny parametry, pokud jsou poskytovány pouze pro některý z nich. [19]

Při výběru konkrétního nástroje pro tvorbu automatizovaných testů musíme předem zanalyzovat testovanou aplikaci (velikost aplikace, použitá technologie, stupeň dokončenosti, předpokládaný další vývoj, životnost aplikace apod.) a na základě zjištěných informací vybrat konkrétní nástroj pro tvorbu automatizovaných testů. V praktické části práce budou použity nástroje NUnit, Selenium WebDriver a jeho implementaci v jazyce C#. Jedná se o jednu z nejpoužívanějších kombinací pro nástroj Selenium. K programování, evidenci testů a jejich spuštění bude použité Visual Studio 2019.

V případě volby manuálního testování by mělo být testovací prostředí ještě rozšířeno o nástroje pro management testování. Příkladem takového řešení pro manuální testování je nastroj Application Lifecycle Management.

3.8 Typy testů

3.8.1 UNIT testy

Jednotkové testování se také nazývá UNIT testování. Zaměřuje na nejmenší testovatelné části aplikace a zaručuje, že blok zdrojového kódu splňuje požadavky na něj. Testování jednotky obvykle provádějí vývojáři a programátory pod vedením test manažera nebo projektového manažera. Hlavním účelem je otestovat každý blok samostatně a individuálně. Testování jednotky obvykle přichází na řadu nejdříve, a to ve fázi realizace projektu a týká se metod „white box“. [16]

3.8.2 Assembly testy

Assembly testy jsou o jeden krok vyšší než jednotkové testování. [13] Pomocí těchto testů se prokáže, zda moduly mohou správně a stabilně interagovat, jak je definováno ve funkčních specifikacích poskytnutých zákazníkem. Testy assembly jsou typem černé skříňky a nazývají se také jako testy integrace.

3.8.3 Integrační testy

Integrační testování ověřuje interakci mezi jednotlivými komponentami uvnitř vyvinuté aplikace. [13] Integraci však lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. Graf funkční dekompozice je základem pro integrační testování, který ukazuje strukturální vztah aplikačních prvků mezi sebou. Je třeba poznamenat, že před integračním testováním musí být všechny moduly testovány samostatně.

3.8.4 Smoke testy

Typ testování softwaru, který zahrnuje krátký testovací cyklus, zaměřený především na ověření fungování kritických funkcí. [9] Výsledky těchto testů se používají ke stanovení stupně stability verze softwaru za účelem dalšího vývoje. Pojem „smoke test“ v softwaru pochází z podobného typu testování hardwaru, při kterém zařízení prochází testem, pokud při

prvním zapnutí nedochází k požáru (smoke). Tyto testy jsou již v působnosti testerů. Je ale možné, že je provádí pracovník odpovědný za správu testovacích prostředí.

3.8.5 Sanity testy

Sanitární testování je druh testování výkonu softwaru, při kterém jsou jednotlivé funkce a postupy kontrolovány z hlediska souladu s požadavky uvedených ve specifikaci. Používá se k určení funkčnosti určité části aplikace po změnách provedených v ní nebo v prostředí. Obvykle se provádí ručně. Může obsahovat vlastnosti funkčních a regresních testů, testů uživatelského rozhraní, atd. Podle metodiky jsou sanity testy podobné smoke testům, ale první jsou zaměřeny na „hloubku“ a druhé na „šířku“, a pokrývají co nejvíce procedur a funkcí.

3.8.6 Perfomance testy

Testy výkonnosti (Perfomance tests) se používají ke stanovení rychlosti, nákladů na RAM a stability softwaru při zatížení. Testování výkonnosti může zahrnovat kvantitativní testy prováděné ve specializované laboratoři nebo konající se v produkčním prostředí za omezených podmínek. Mezi typické testovací parametry patří rychlost zpracování, rychlost přenosu dat, šířka pásma sítě a spolehlivost při pracovním zatížení. Zákazník může například měřit dobu odezvy programu, když dojde k milionům požadavků za sekundu.

3.8.7 Stress testy

Jsou určeny pro kontrolu výkonnosti aplikace a systému pod stresem jako celku a také pro vyhodnocení schopnosti systému k regeneraci, tj. schopnosti vrátit se do normálu po ukončení stresu. Stresem v této souvislosti může být zvýšení intenzity operací na velmi vysoké hodnoty nebo abnormální změna konfigurace serveru. Jedním z úkolů při stresovém testování může být také posouzení zhoršení výkonu, takže cíle stresového testování se mohou překrývat s cíli výkonového testování. Protože zátěžové testování úzce souvisí s povahou původu testované aplikace, metody zátěžového testování také závisí na aplikaci. Při tomto typu zkoušení se vyhodnocuje spolehlivost a stabilita aplikace, pokud jsou překročeny přípustné meze normálního zatížení.

3.8.8 Systémové testy

Během systémových testů je vyvinutá aplikace kontrolována jako celek. [13] Hlavním cílem testování systému je ověřit funkční i nefunkční požadavky na aplikaci. Současně jsou

detekovány závady, jako jsou nesprávné použití systémových prostředků, neočekávané kombinace dat na úrovni uživatele, nekompatibilita s prostředím, neočekávané scénáře použití, chybějící nebo nesprávná funkčnost, nepříjemnosti použití atd.

3.8.9 Regresní testy

Jedná se o typ testování zaměřeného na kontrolu změn provedených v aplikaci nebo prostředí (oprava vady, slučování kódu, migrace do jiného operačního systému, databáze, webového serveru nebo aplikačního serveru), aby se potvrdilo, že dříve existující funkčnost funguje jako předtím. [13]. Pokud se například jedna z tříd změní, je nutné spustit všechny testovací případy pro všechny interagující třídy, aby se zajistilo, že žádné změny neovlivní interakci s jinými objekty. Testery tedy během regresních testů spouští všechny skripty, aby se ujistily, že nic nebylo ovlivněno.

4 Vlastní práce

Praktická část této práce se zabývá převedením získaných v teoretické části informací k vytvoření vlastního testovacího scénáře zvolené webové aplikace, jeho exekuci manuálním a automatickým způsobem testování a analýzou získaných výsledků.

V první kapitole práce je představena samotná testovaná aplikace a popsán důvod, který vede k testování. Další kapitolou je vytváření manuálního testu, ve kterém čtenář se seznámí s testovacím nástrojem vybraným k otestování. Dále se uskuteční exekuce testu a následné zhodnocení výsledků.

Začátkem následující kapitoly je tvorba testu, založeného na použití metody automatického testování, popis použitých nástrojů. Poté proběhne samotné testování a jeho vyhodnocení.

Poslední kapitoly praktické části se věnují porovnáním obou výsledků a metod testování.

4.1 Testovaná aplikace

Předmětem testování je Univerzitní Informační Systém (zkráceně UIS). Jedná se o složitější softwarový systém, který slouží k evidenci veškeré studijní agendy na univerzitě. Mezi používané funkce patří zobrazení rozvrhu, kontrola osobních údajů studenta, přehled známek a studijního plánu, zápis povinně volitelných a volitelných předmětů, přihlašování na zkoušky a jiné.

V této práci bude otestováno především GUI (grafické uživatelské rozhraní) a jednotlivé komponenty systému, které není ale třeba testovat do hloubky, stačí jejich základní funkčnost. Hlavním cílem je testování UX , funkčnosti webové aplikace, dostupné informací a nalezení případných chyb. Jejich následné odstranění nebo návrh možného řešení.

Oblast, která byla použita v této práci, se vztahuje na základní funkčnosti systému a to zejména na přihlášení do aplikace, kontrolu osobních údajů studenta, zobrazení rozvrhu a studijního plánu.

První testovací scénář je zaměřen na otestování vkládání nového e-mailu do osobních údajů studenta a ověření, zda se údaje propsaly do osobních informací. Druhý testovací scénář je zaměřen na otestování funkce vyhledávání a zobrazení studijního plánu. Třetí testovací scénář je zaměřen na zobrazení rozvrhu dle vybraných parametrů

Pro odhad celkového stavu a kvality komponent budou použity automatické a manuální testy.

4.2 Manuální test

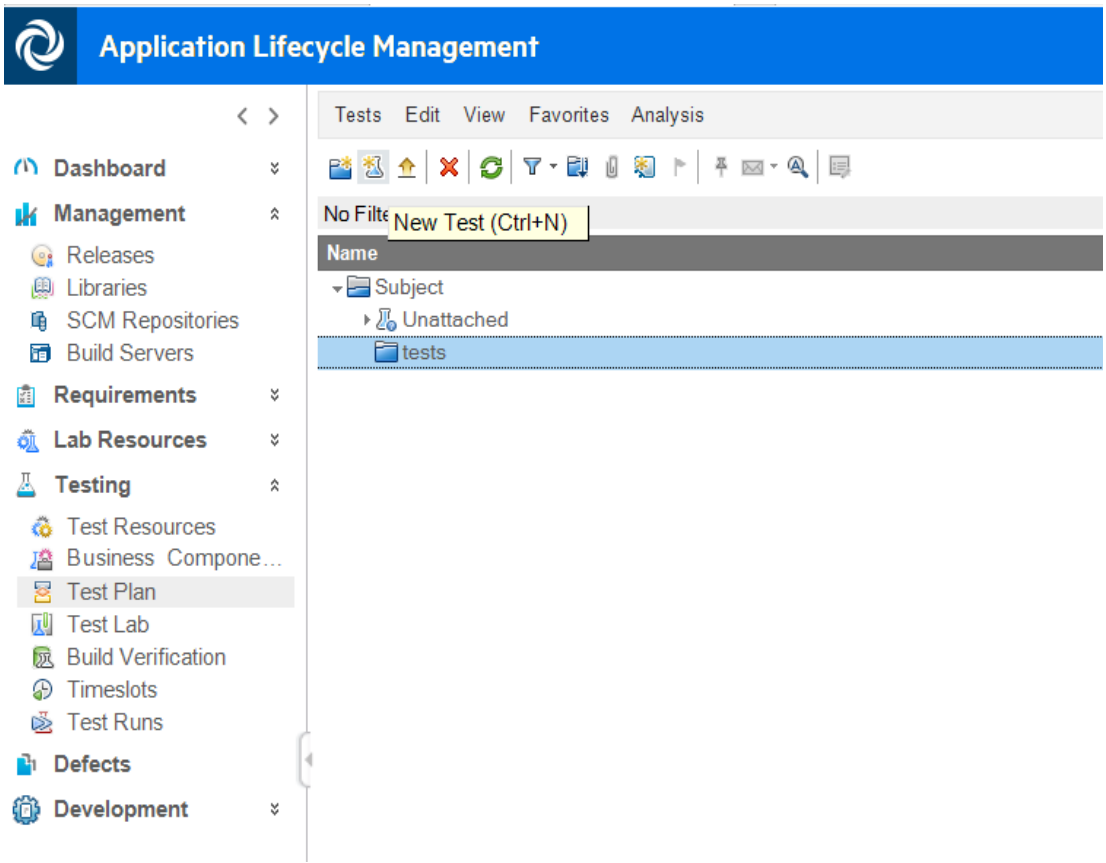
Pro tvorbu a exekuci manuálního testu byla použita specializovaná sada nástrojů Application Lifecycle Management, se kterými, autor již pracoval během své praxe na pozici tester softwarových aplikací. Jak již bylo v teoretické části uvedeno, ALM má v sobě sadu vestavěných modulů jako jsou Management, Requirements, Testing a Defects.

Pro potřeby bakalářské práce byl použit pouze modul Testing.

4.2.1 Tvorba testovacího scénáře

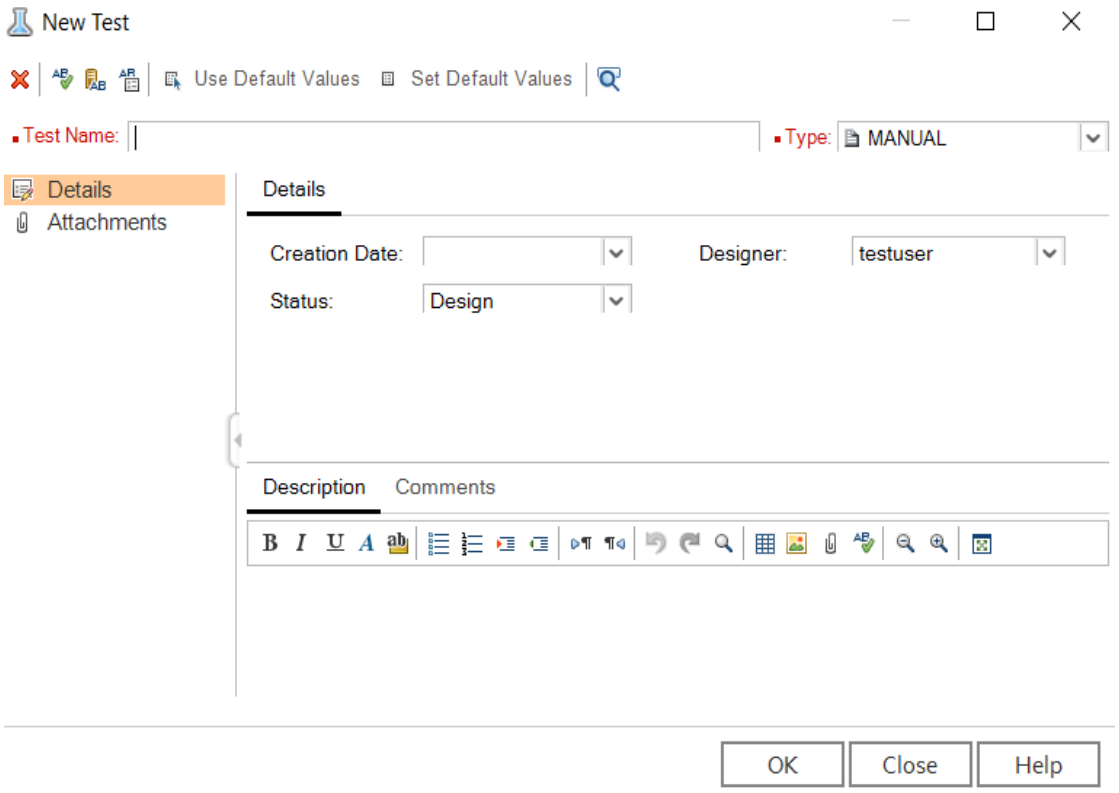
Pomoci již zmíněného modulu Testing, v rámci sekci Testovacího plánu testovací scénáře lze v ALMu vytvořit dvěma způsoby. Kromě návrhu vlastního řešení (testu) v modulu softwaru je možné také naimportovat do svého ALM projektu data testovacího plánu z aplikace Microsoft Word nebo Microsoft Excel.

Vytváření nového testu se provádí pomocí tlačítka „New Test“



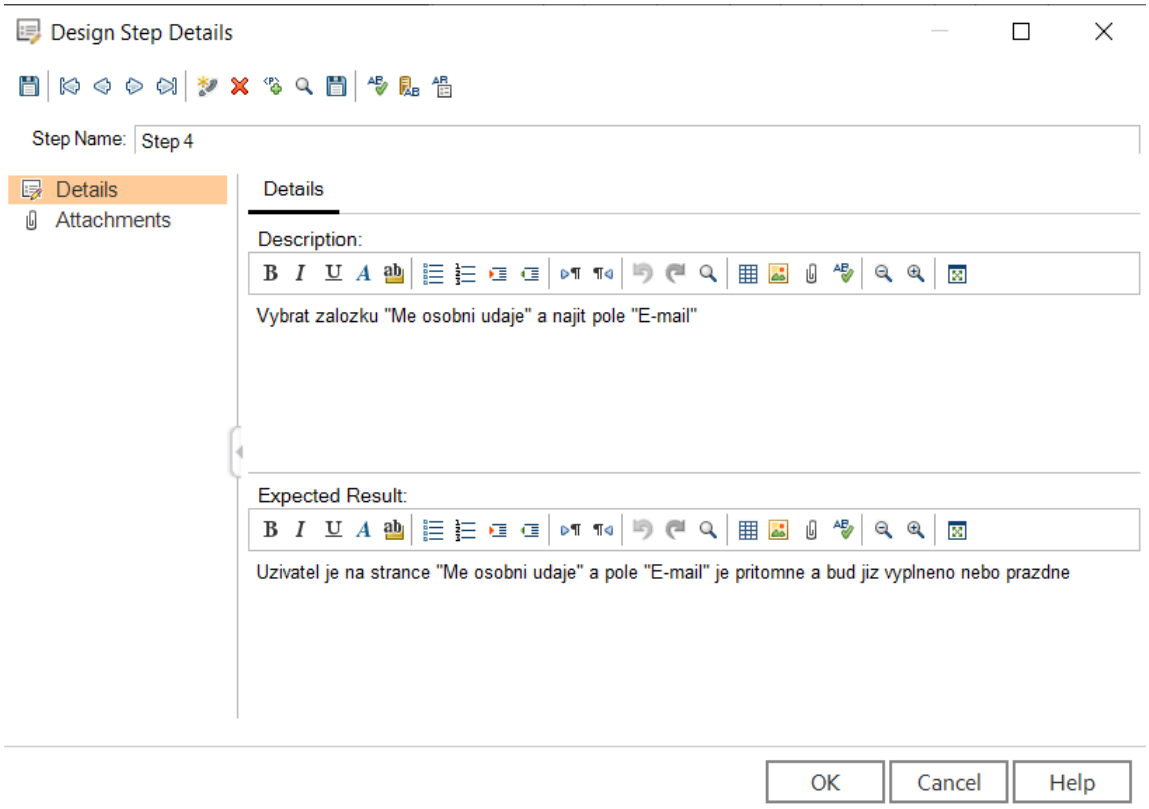
Obrázek 9: ALM – Test plan

Po stisknutí ikonky se otevře dialogové okno, které se zeptá na název a atributy nového testu. Bez vyplnění povinných položek „Test Name“ a „Type“ program neumožní testovací scénář založit. Obecně se předpokládá, že než začneme provádět jakýkoliv test, musíme pečlivě zvážit, co ten test dělá, a pojmenovat ho tak, aby z názvu vyplývalo to, co daný test pokrývá. Dalším důležitým polem je „Description“, kde by měly být změněny přístupové role testeru a všechny předpoklady pro korektní exekuci testovacího scénáře. Tady by se měl vyskytnout detailnější přehled, o čem bude průběh testu.



Obrázek 10: Vytváření nového testovacího scénáře

Po vytvoření testovacího scénáře se začíná specifikace testu a design jednotlivých kroků. Kroky se vytváří pomocí tlačítka „Next step“ v sekci „Design Steps“, která je znázorněna na obrázku č. 11.



Obrázek 11: Vytváření nového tetovacího kroku

Do pole „Description“ se zadává co nejvýstižněji popis akce, kterou má tester provést, pole „Expected“ popisuje normální chování aplikace podle webových standardů, dokumentace produktu, specifikace zákazníka či již provedené analýzy produktu.

Tabulka 1: Jednotlivé kroky v rámci testovacího scénáře add_new_email

Step Name	Description	Expected
Step 1	Otevřít webovou aplikaci UNIVERZITNI INFORMACNI SYSTEM (https://is.czu.cz /?lang=cz /)	Otevřela se webová aplikace
Step 2	Přihlásit se loginem a heslem	Uživatel se přihlásí a je na stránce "Osobní administrativa"
Step 3	Kliknout na odkaz "Uživatelská nastavení"	Otevře se stránka "Uživatelská nastavení"

Step 4	Vybrat záložku "Mé osobní údaje" a najít pole "E-mail"	Uživatel je na stránce "Mé osobní údaje". Pole "E-mail" je přítomné a buď je již vyplněno, nebo je zatím prázdné
Step 5	Stisknout tlačítko "Přidat další"	Objeví se pole pro přidání nového e-mailu
Step 6	Vložit do pole nový e-mail a stisknout tlačítko "Uložit změny"	Dojde k uložení e-mailu a zobrazí se informace o úspěšném průběhu operace
Step 7	Vrátit se zpátky na stránku "Osobní administrativa"	Uživatel je na stránce "Osobní administrativa"
Step 8	Kliknout na odkaz "Kontrola osobních údajů"	Otevře se stránka "Kontrola osobních údajů"
Step 9	Ověřit, zda je e-mail v systému a zkontrolovat, aby se údaje propsaly do osobních informací	e-mail byl úspěšně uložen a je přítomen na obrazovce
Step 10	Odhlásit se pomocí ikony „Odhlášení“	Uživatel je odhlášen

Tabulka 2: Jednotlivé kroky v rámci testovacího scénáře rozvrh

Step Name	Description	Expected
Step 1	Otevřít webovou aplikaci UNIVERZITNI INFORMACNI SYSTEM (https://is.czu.cz /?lang=cz /)	Otevřela se webová aplikace
Step 2	Přihlásit se loginem a heslem	Uživatel se přihlásí a je na stránce "Osobní administrativa"

Step 3	Zadat do vyhledávače text "stud" a zkontrolovat, zda je v nabídce položka "Studijní plány" obsahující požadovanou kombinaci písmen	Položka je přítomná
Step 4	Kliknout na položku "Studijní plány"	Načte se stránka "Prohlídka studijních programů" - výběr fakulty a aplikace zobrazí přehled jednotlivých fakult
Step 5	Zvolit "Provozně ekonomická fakulta"	Načte se stránka "Prohlídka studijních programů" - výběr počátečního období a aplikace zobrazí přehled počátečního období studia zvolené fakulty
Step 6	Pomocí ikony "Prohlížet" zobrazit studijní období ZS 2019/2020	Otevře se stránka "Prohlídka studijních programů" - výběr typu studia. Na stránce je vidět již zvolenou informaci o fakultě a studijním období
Step 7	Zvolením ikony ve sloupci "Prohlížet" vybrat bakalářský typ studia	Uživatel je přesunut na stránku "Prohlídka studijních programů" - výběr programů nebo specializace. Aplikace zobrazuje přehled jednotlivých studijních programů, oborů a specializací
Step 8	Zvolit program "B – SI Systémové inženýrství a informatika" jazyk výuky – čeština	Otevře se stránka "Prohlídka studijních programů" - výběr studijního oboru. Na stránce je vidět již vybrané informace a aplikace zobrazuje

		přehled jednotlivých studijních
Step 9	Zobrazit studijní plán prezenční formy oboru zvolením ikony ve sloupci "Prohlížet".	Aplikace zobrazuje základní informace o vybraném oboru a studijní plán tedy přehled všech předmětů pro vybrané studium. Studijní plán je rozdělen do studijních období, tj. semestrů.
Step 10	Vrátit se zpět na stránku "Osobní administrativa" a vyhledat ve vyhledávači textem "odhlášení"	Otevře se stránka "Odhlášení", kde kliknutím na tlačítko „Odhlásit se “ proběhne odhlášení z Univerzitního informačního systému.

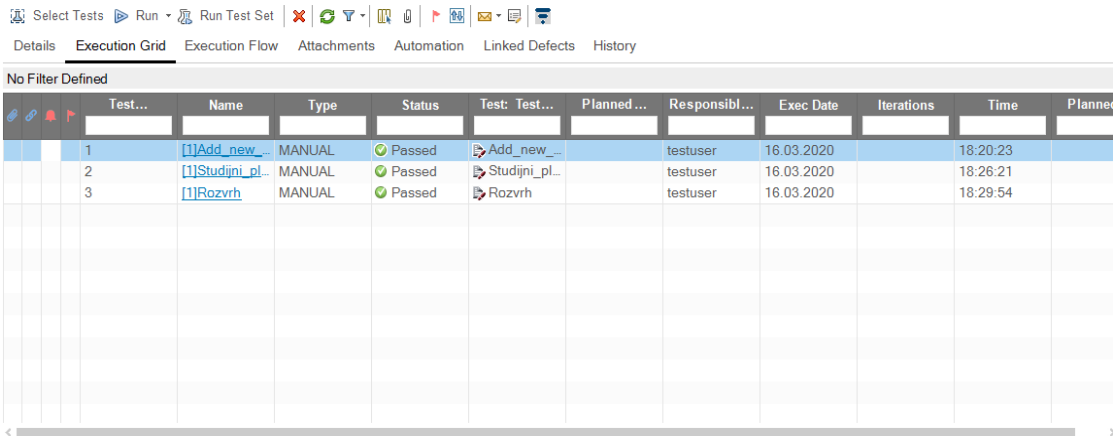
Tabulka 3: Jednotlivé kroky v rámci testovacího scénáře studijni_plan

Step Name	Description	Expected
Step 1	Otevřít webovou aplikaci UNIVERZITNI INFORMACNI SYSTEM (https://is.czu.cz /?lang=cz /)	Otevřela se webová aplikace
Step 2	Přihlásit se loginem a heslem	Uživatel se přihlásí a je na stránce "Osobní administrativa"
Step 3	Kliknout na odkaz "Rozvrhy"	Otevře se stránka "Zobrazení rozvrhu" - výběr kritérií, kde je možné prohlížet jednotlivé akce vybraných rozvrhů podle různých kritérií. V první části lze zobrazit osobní rozvrh studenta zvolený při zápise. Další výstupy lze získat volbou jednoho nebo více

		omezení v rozbalovacích seznamech a stiskem tlačítka "Zobrazit" ve formuláři níže.
Step 4	V rozbalovacích seznamech zvolit si katedru informačního inženýrství (KII PEF); typ studia: Bakalářský; program: N – SI Systémové inženýrství a informatika; ročník: 3. a stisknout tlačítko "Zobrazit"	Otevře se stránka "Rozvrh předmětů a tabulka", která zobrazuje HTML náhled na zvolený rozvrh.
Step 5	Odhlásit se pomocí ikony „Odhlášení“	Otevře se stránka "Odhlášení", kde kliknutím na tlačítko „Odhlásit se “ proběhne odhlášení z Univerzitního informačního systému.

4.2.2 Exekuce testovacího scénáře

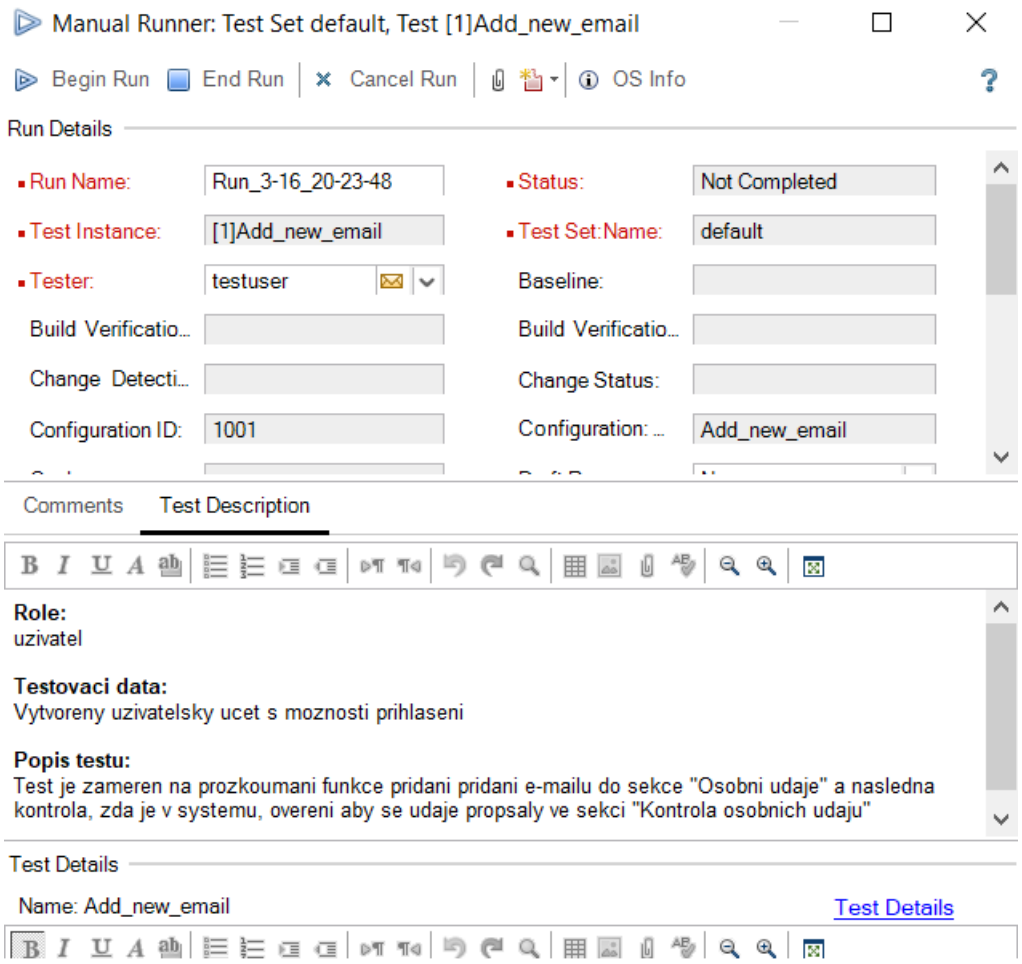
Pro spuštění testovacího scénářů, je potřeba vytvořit nový Test Set v Test Labu přes tlačítko „New Test Set“, který lze naplnit jednotlivými testy vytvořenými v Test Planu.



Obrázek 12: Vytváření nového test setu v Test Labu

Před samotnou exekucí musí tester připravit potřebná test data a splnit všechny uvedené požadavky. Test se spouští pomocí tlačítka „Run Test“.

Testování probíhá v samostatném okně, kde jsou zaznamenány jednotlivé testovací kroky. Než se přejde na spuštění manuálního testu, tester se nejdřív seznámí s popisem testu a splní všechny podmínky testovacího scénáře. Při procházení testovacím scénářem, v případě, že všechno odpovídá funkčnosti aplikace, tester zaznamená stav kroku jako „Passed“. Nejčastěji se lze setkat se stavy „Passed“, „Failed“, „Blocker“ a „N /A “.



Obrázek 13: Exekuce testu

Čas exekuce byl zjištěn pomocí ALMu, který při exekuci zaznamenává čas provedení jednotlivých kroku. Celkem v manuálním testování se zúčastnilo tři testery, kteří se dosud nikdy se systémem nesetkali. Prvnímu testerovi trvala exekuce testu „Add_new_email“ 5 :01 minut, druhému 6 :38 minut a třetímu 4 :11 minut. Test „Studijní plán“ první tester prošel za 5 :11 minut, druhý za 4 :06 minut a třetí za 3 :58 minut. Test „Rozvrh“ trval 3 :04 minut, 1 :58 minut a 2 :41 minut.

4.3 Automatický test

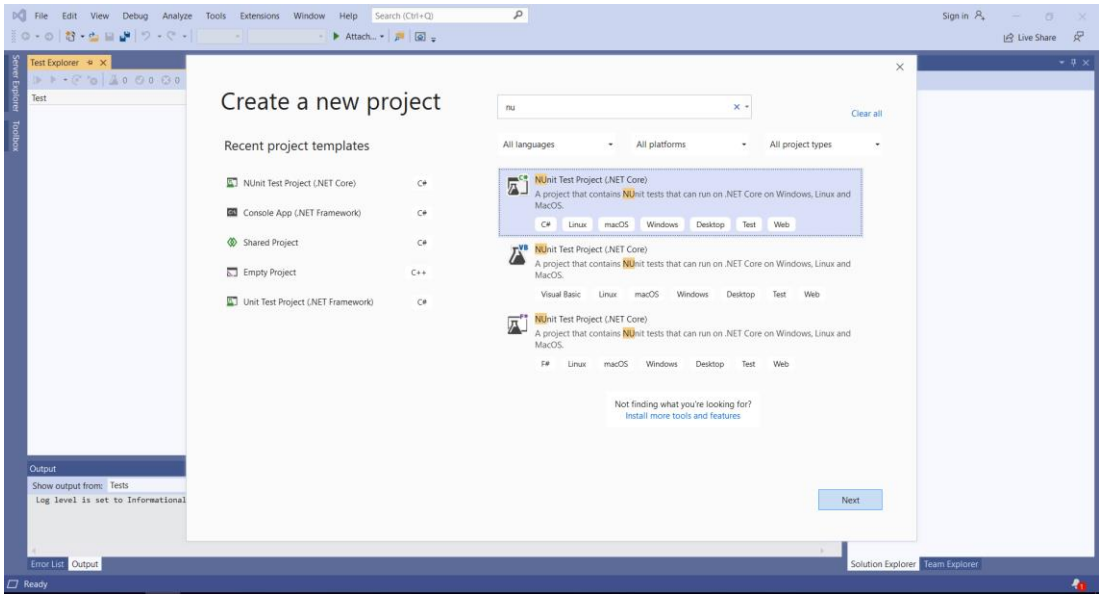
V této části je předpokladem znalost GUI testované aplikace, tester by měl být seznámen s objekty testované aplikace, které se budou využívat při vývoji automatizovaných testů.

Pro tvorbu a exekuci automatického testu byly vybrány nástroje Visual Studio 2019, Selenium WebDriver a NUnit Framework.

4.3.1 Tvorba testovacího scénáře

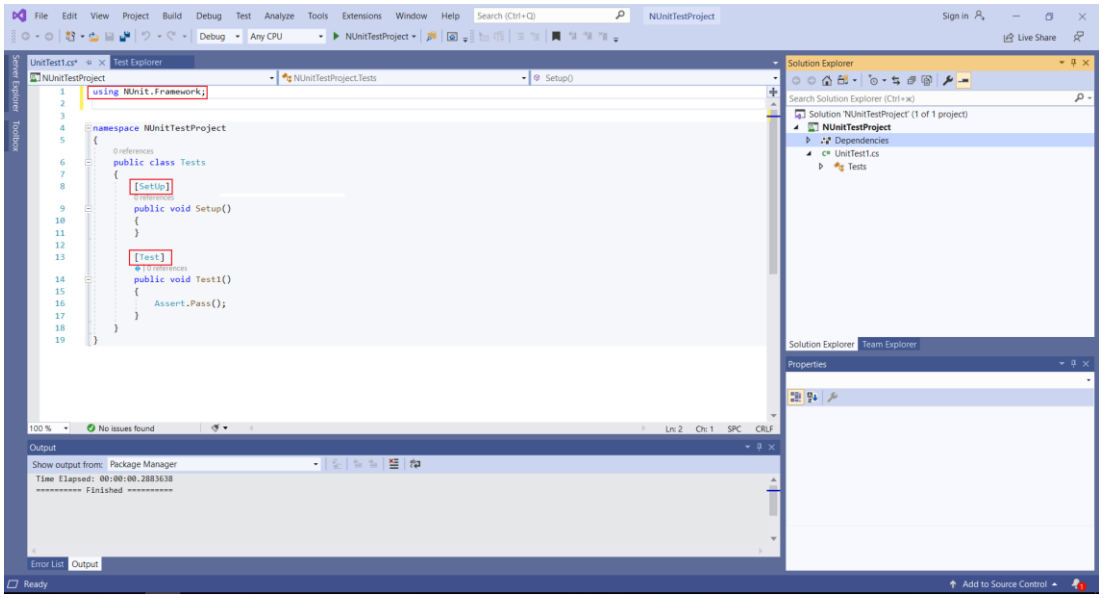
Hned po instalace a spuštění Visual Studio, dalším krokem je vytvoření projektu, ve kterém bude navržen testovací scénář. Stejně jako u všech editorů a sad nástrojů existuje několik způsobů, jak dosáhnout stejného základního cíle, kterým je vytvoření testovacího projektu. Pro cíle práce byl vybrán NUnit test Project v programovacím jazyce C #.

Přechodem na záložku File → New → Project → NUnit Test Project (.Net Core) a kliknutím tlačítka „Next“ je založen nový projekt, který je nutné pojmenovat a indikovat k němu vlastní cestu pro uložení.



Obrázek 14: Založení nového projektu

Nový projekt bude mít připravený základní test s anotacemi NUnit, které budou použity ve zdrojovém kódu.



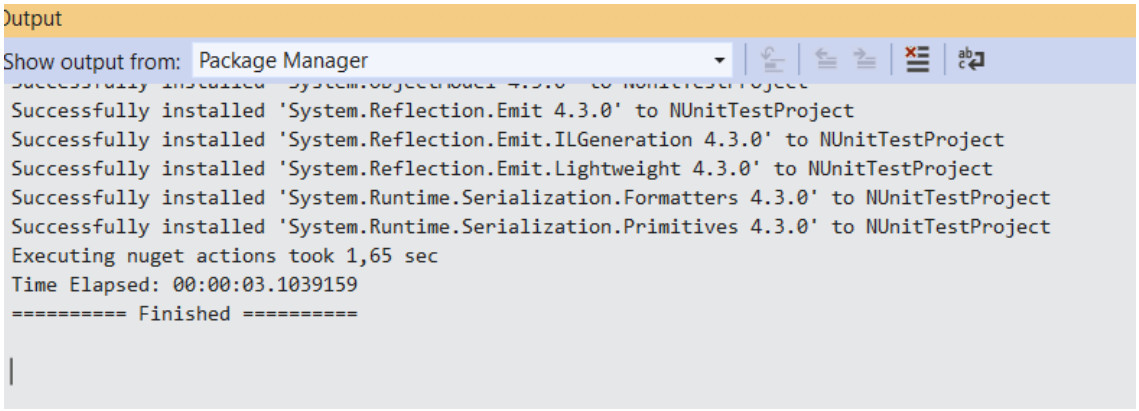
Obrázek 15: Základní test s anotacemi NUnit

V projektu už jsou předinstalovaný balíčky NUnit (3 .12 .0) a NUnit3TestAdapter verzi 3 .16 .1 . Nicméně musíme přidat navíc Selenému WebDriver a Selenium Support. Visual Studio má integrovaný nástroj pro správu závislostí s názvem NuGet. NuGet stahuje závislosti, jako je WebDriver, z centrálních úložišť a přidává je do vybraného projektu.

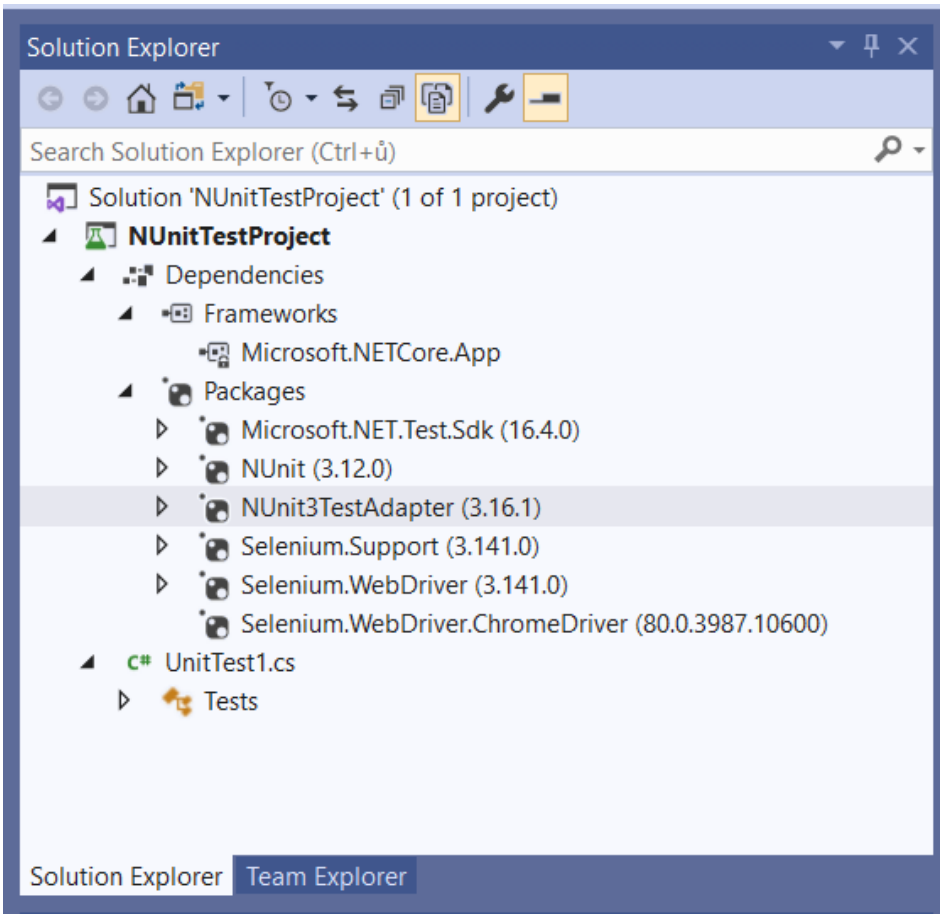
Kliknutím vlevo nahoře na „Browse“ na obrazovce „Správy balíčků“ je možné vyhledat Selenium.WebDriver a Selenium.Support ve verzi 3 .x. Tyto verze jsou preferované především proto, že jsou aktuálními verzemi 3.142.7.

Poté je nutné stáhnout Selenium.WebDriver.ChromeDriver. ChromeDriver je samostatný server nebo samostatný spustitelný soubor, který používá Selenium WebDriver k ovládání prohlížeče Chrome. Bez ChromeDriveru nelze spustit seleniové testovací scénáře v prohlížeči Google Chrome.

Po dokončení instalace všech potřebných balíčků se zobrazí níže uvedená zpráva na obrázku č. 16 a průzkumník řešení měl by vypadat jak na obrázku č. 17.



Obrázek 16: Zpráva o úspěšném dokončení instalace



Obrázek 17: Průzkumník řešení

Když je konfigurace Visual Studio pro Selenium C # framework definovaná (dokončená), lze začít zaznamenávat jednotlivé kroky samotného testovacího scénáře.

Nejprve je nutné si definovat knihovny potřebné pro vytvoření testu. Pro účely této práce projekt bude obsahovat (budou použity):

```
using System;
```

```
using System.Threading;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium.Support.UI;
```

Poté se zapisují jednotlivé kroky pro automat.

Než začne vývoj automatizovaných testů je potřeba vyřešit způsob spouštění aplikace a inicializovat proměnnou, která bude obsahovat instanci webového prohlížeče. SetupTest() je metoda, která se volá před každou metodou. Její opakování tedy záleží na počtu testů a jsou tady deklarovány základní postupy pro běh skriptů. V této metodě je implementována inicializace WebDriveru, také dostává prohlížeč basový URL a URL aplikace, kterou testujeme, dále je zde metoda pro ovládání samotného prohlížeče jako maximalizace prohlížeče, který automat otevře.

Dále následuje krok, který zpracovává přihlášení do aplikace a kontroluje, zda je uživatel na správné stránce pomocí metody Assert.AreEqual(). Zároveň tady je nastavena vlastnost driveru WebDriverWait. Ta slouží k tomu, aby test neselhal, pokud má stránka špatnou odezvu a načítá se pomalu.

```
[SetUp]
0 references
public void SetupTest()
{
    driver = new ChromeDriver();
    baseURL = "https://www.google.com/";
    driver.Manage().Window.Maximize();
    driver.Navigate().GoToUrl("https://is.czu.cz/?lang=cz");
    Assert.AreEqual(true, driver.Title.Contains("Univerzitní informační systém ČZU"), "Title is not matching");
    driver.FindElement(By.XPath("//div[@id='stranka']/div/table/tbody/tr/td[3]/small/a/b/span")).Click();
    driver.FindElement(By.Id("credential_0")).Clear();
    driver.FindElement(By.Id("credential_0")).SendKeys("xpetv022");
    driver.FindElement(By.Id("credential_1")).Click();
    driver.FindElement(By.Id("credential_1")).Clear();
    driver.FindElement(By.Id("credential_1")).SendKeys("Indefferentnoq2");
    driver.FindElement(By.Id("login-btn")).Click();
    WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(30));
    Assert.AreEqual(true, wait.Until(ExpectedConditions.TitleIs("Osobní administrativa")), "Title is not matching");

}
```

Obrázek 18: Metoda SetupTest()

closeBrowser() je metoda, která se zavolá po každé metodě. Její účel je ukončit driver. Je to z toho důvodu, aby měl každý test připravené čisté testovací prostředí a nebyl ovlivněn

předchozím testem.

```
[TearDown]
public void closeBrowser()
{
      driver.Close();
      driver.Quit();
}
```

Dále následují samotné testovací metody, které slouží k spouštění jednotlivých testů a vypadají takto:

```
[Test]
0 references
public void add_new_email()
{
    // add mail and save changes
    driver.FindElement(By.PartialLinkText("Uživatelská nastavení")).Click();
    Assert.AreEqual(true, driver.Title.Contains("Uživatelská nastavení"),
        "Title is not matching");
    driver.FindElement(By.LinkText("Mé osobní údaje")).Click();
    driver.FindElement(By.XPath("//input[@value='Přidat další']")).Click();
    driver.FindElement(By.XPath("(//input[@name='email'])[2]")).Click();
    driver.FindElement(By.XPath("(//input[@name='email'])[2]")).SendKeys("test@gmail.com");
    driver.FindElement(By.Name("ulozit")).Click();
    Assert.AreEqual(true, driver.PageSource.Contains("odsazena"), "Changes wasd not saved"); //Zadané údaje byly úspěšně uloženy.
    driver.FindElement(By.LinkText("Zpět na osobní administrativu")).Click();

    // check if the email was saved and in "Kontrola osobních údajů"  + log out
    driver.FindElement(By.LinkText("Kontrola osobních údajů")).Click();
    Assert.AreEqual(true, driver.Title.Contains("Kontrola osobních údajů"), "Title is not matching");
    Assert.AreEqual(true, driver.PageSource.Contains("test@gmail.com"), "Email was not saved");
    driver.FindElement(By.XPath("//div[@id='ikonky']/a[4]/img")).Click();
    driver.FindElement(By.Name("odhlaseni")).Click();
}
```

Obrázek 19: Testovací scénář add_new_email

```
[Test]
0 references
public void studijni_plan()
{
    driver.FindElement(By.LinkText("Vyhledat")).Click();
    driver.FindElement(By.Name("_appsearch")).SendKeys("Studijní plány");
    driver.FindElement(By.Name("_appsearch")).SendKeys(Keys.Down);
    driver.FindElement(By.Name("_appsearch")).SendKeys(Keys.Enter);
    driver.FindElement(By.XPath("//div[@id='stranka']/div[4]/table/tbody/tr[2]/td/small/a/img")).Click();
    driver.FindElement(By.XPath("//table[@id='tmtab_1']/tbody/tr[2]/td[3]/small/a/img")).Click();
    driver.FindElement(By.XPath("//table[@id='tmtab_1']/tbody/tr/td[4]/small/a/img")).Click();
    driver.FindElement(By.XPath("//table[@id='tmtab_1']/tbody/tr[16]/td[7]/small/a/img")).Click();
    driver.FindElement(By.XPath("//table[@id='tmtab_1']/tbody/tr/td[6]/small/a/img")).Click();
    driver.FindElement(By.XPath("//table[@id='tmtab_1']/tbody/tr/td[3]/small/a/img")).Click();
    driver.FindElement(By.XPath("//div[@id='ikonky']/a[3]/img")).Click();
    driver.FindElement(By.LinkText("Vyhledat")).Click();
    Assert.AreEqual(true, driver.PageSource.Contains("table"), "Timetable was not found");
    driver.FindElement(By.Name("_appsearch")).SendKeys("odhlaseni");
    driver.FindElement(By.Name("_appsearch")).SendKeys(Keys.Down);
    driver.FindElement(By.Name("_appsearch")).SendKeys(Keys.Enter);

}
```

Obrázek 20: Testovací scénář studijni_plan

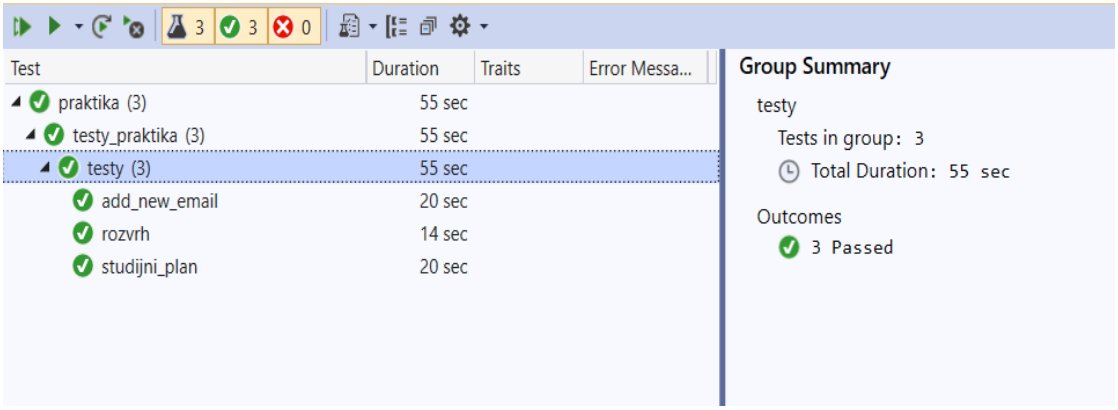
```
[Test]
0 references
public void rozvrh()
{
    driver.FindElement(By.LinkText("Rozvrhy")).Click();
    driver.FindElement(By.Name("ustav")).Click();
    new SelectElement(driver.FindElement(By.Name("ustav"))).SelectByText("Katedra informačního inženýrství (KII PEF)");
    driver.FindElement(By.Name("ustav")).Click();
    driver.FindElement(By.Name("stupen")).Click();
    new SelectElement(driver.FindElement(By.Name("stupen"))).SelectByText("Bakalářský");
    driver.FindElement(By.Name("stupen")).Click();
    driver.FindElement(By.Name("rocnik")).Click();
    new SelectElement(driver.FindElement(By.Name("rocnik"))).SelectByText("3. ročník");
    driver.FindElement(By.Name("rocnik")).Click();
    driver.FindElement(By.XPath("(//input[@name='zobraz'])[2]")).Click();
    driver.FindElement(By.XPath("//div[@id='ikonky']/a[5]/img")).Click();
    driver.FindElement(By.Name("odhlaseni")).Click();
}
```

Obrázek 21: Testovací scénář rozvrh

4.3.2 Exekuce testovacího scénáře

V Microsoft Visual Studio je implementována speciální nabídka příkazů s názvem „Test“ pro práci s Unit testy. Když testovací projekt je sestaven, testy se zobrazí v „Průzkumníku testů“ a lze je tam spustit. Pokud není průzkumník testů viditelný, je třeba zvolit možnost „Test“, pak „Windows“ a „Test explorer“ v nabídce aplikace Visual Studio.

Kliknutím na tlačítko „Run“ či „Run All Tests“ Visual Studio spustí internetový prohlížeč Google Chrome, automaticky projde jednotlivý testovací scénář nebo všechny testy v řešení a zaznamená výsledky. Pokud je test označen zeleným, znamená to, že proběhl bez chyb. Pokud by byl červený, značí to chybu. Existuje ještě třetí možnost: modrý vykřičník, který znamená, že test nebyl spuštěn. Dle obrázku níže automatické testy proběhly v pořádku a v žádném kroku nedošlo k chybě.



Obrázek 22: Report po exekuci automatizovaného testu

Obdobně jako ALM u manuálního testování, i průzkumník testů zaznamenává čas exekuce testu. Na předchozím obrázku je vidět, že celková doba exekucí testů byla pouhých 55 sekund.

5 Výsledky

Manuální testování

Přestože samotný proces absolvování testovacích scénářů neodhalil žádný problém a nebyly nalezeny defekty, v průběhu manuálního testování byly odhaleny drobné nedostatky systému, které avšak můžou výrazně zhoršit celkový dojem o něm. Po absolvování manuálních testů testeři byly požadovány o zpětnou vazbu ohledně testů, funkčnosti webu a taky o tom, jaký mají pocit z UISu obecně. Získaná pomocí dotazníku informace byla analyzována a zahrnuta do práce.

V dotazníku jsem se ptala na následující otázky:

- Byly testovací scénáře srozumitelné?
- Připadají Vám testovací případy dostatečné?
- Můžete uvést hlavní výhody systému, které se Vám líbily?
- Jaké jsou nedostatky systému?
- Jak příjemné je pro uživatele používat design? Jaký máte dojem o systému?

Cílová skupina pro testovaný systém jsou studenti České zemědělské univerzity v Praze. Proto dva uživatelé jsou studenti ČZU, každý ovšem z odlišného studijního oboru. Jeden uživatel zastoupí skupinu studující jinou vysokou školu. Každý tester byl předem seznámen s průběhem a principem testování.

Shrnutí závěrů od testerů

Všichni z testerů odpověděli, že testovací scénáře jsou srozumitelné a testovací případy jsou dostatečné.

Seznam funkcionalit, které uživatelé považují za povedené a chválí je: participantům se nejvíce líbila konzistentnost webu, dostupnost a jasnost informace. Další „Like“ se dal na fulltextové vyhledávání a automatické doplňování slov.

Funkcionality, které uživatelé považují za nepovedené: několik participantů uvedlo, že odkaz na stránku „Osobní administrativa“ není dostatečně viditelný, uživatelé nejčastěji klikali na logo ČZU v pravé části záhlaví, pak zkoušeli menu s odkazy na jednotlivé fakulty. Jeden uživatel uvedl, že na první pohled ho zmátla struktura hlavní stránky, a druhý, že pohybování a hledání je obtížné především proto, že na stránce je příliš mnoho informace a odkazů.

Uživatelé se shodli, že navigace na webových stránkách nepatří do silných stran systému, bylo obtížné procházet složitou strukturou webu a orientovat se v ní. Na panelu, v pravém horním rohu jsou použity „navigační“ ikony, které uživatelé spíše ignorují, a to z toho důvodu, že většině uživatelů přijde jejich design neintuitivní, proto se raději zaměřují na text.

Další problém, který testery měli, byl nalezení vyhledávače – hledali nejdříve vpravo v záhlaví, potom směřovali do levého rohu a očekávali obecné vyhledávací pole, ne textový odkaz. Všem uživatelům tato položka přišla nepřehledná, studenti CŽU se nakonec přiznali, že o možnosti vyhledávání dosud vůbec nevěděli.

Celkový dojem

Participanty hodnotí systém jako dobrý, pokud jde o dostupné informace a o celkové funkčnosti, ale uvedli, že se obtížně používá. Uživatelům vadí na stránkách grafické provedení, všichni se shodli, že nejvíce jim nelíbil vzhled informačního systému a jeho neintuitivnost.

Z výše uvedeného vyplývá, že ke zlepšení User Experience je nutné především zaměřit se na uživatele a vzít v úvahu jeho mentální model. Uživatelské rozhraní by mělo být co nejvíc intuitivním a pochopitelným.

Automatické testování

Spouštění testovacích skriptů proběhlo v pořádku a žádný testovací scénář neodhalil žádný problém, co prokázalo správnou funkčnost jednotlivých komponent a systému celkem.

Níže jsou zobrazeny tabulky ukazující celkové časové náročnosti manuálního a automatického testování

Tabulka 4: Časové náročnosti manuálního testování

Manuální tester	Testovací scénář 1	Testovací scénář 2	Testovací scénář 3	TS1+TS2+TS3
	Čas	Čas	Čas	
Tester 1	5:01	5:11	3:04	13:16
Tester 2	6:38	4:06	1:58	12:42
Tester 3	4:11	3:58	2:41	10:50
Čas celkem				36:48

Tabulka 5: Časové náročnosti automatického testování

Automat	TS1+TS2+TS3
Číslo běhu	Čas
1	0:55
2	0:50
3	0:51
Celkem	2:36

V tomto konkrétním případě vyšlo, že na tento počet opakování manuálnímu testerovi ovšem zabral 36 minut a 48 sekund, přičemž automat tento počet opakování zvládl za 2 minuty a 36 sekund. Z tabulek 4 a 5 je vidět, že automatizované testování má několikanásobnou výhodu před manuálním testováním. V rámci této práci je možné říct, že automatizace může ušetřit spoustu času při testování.

Jak již bylo uvedeno v praktické části, tato práce byla zaměřena především na aktivity zahrnující hlavně grafické uživatelské rozhraní, což je spíše vhodné pro manuální testování, proto někteří si mohou mylně myslet, že automatizace v tomto konkrétním případě byla nadbytečná. Není to ale tak. Automatizace testovacích scénářů má velký potenciál a měla by být součástí životního cyklu vývoje každého softwaru.

Je důležité pochopit, že v praxi testování je často prováděno v několika krocích a není vždy výhodné nadále používat ruční testování. Platí pravidlo, že čím složitější a rozsáhlejší je systém, tím se víc zvyšuje časová a finanční náročnost na jeho testování. Existuje taky lidský faktor, kvůli kterému některé chyby mohou zůstat bez povšimnutí. Ačkoli uživatelské rozhraní lepší testovat ručně, lidé jsou často náchylní k neefektivnosti.

Tomuto problému lze zabránit automatizací testování. Za předpokladu, že se specifikace měnit nebude a produkt je stálý, automatizované testy můžou ušetřit hodně času a peněz při testování GUI v složitějších aplikacích se spoustou funkcí. Praxe ukázala, že oba typy testování mají výhody a nevýhody. Kombinace obou je perfektním způsobem, jak získat maximum z testování.

6 Závěr

Hlavním cílem této bakalářské práce bylo přiblížit čtenáři problematiku testování softwaru a poskytnout systematický návod pro úspěšné zavedení manuálních a automatizovaných testů.

Teoretická část práce se zabývá shrnutím problematiky testování. V této problematice na začátku části byla popsána definice testování společně s jeho cílem. Dále v této části byla popsána struktura a role v testovacím týmu, zmíněny metodiky pro vývoj a testování softwaru. Na to navazuje celkem rozsáhlá kapitola zabývající se softwarovou chybou a samotnou klasifikací testovacich metod. Dále práce má za úkol popsat nástroje určené k testování webových aplikací a další důležitou částí je základní rozdělení druhu testů, které se provádějí během testování.

V praktické části ze zmíněných nástrojů byla vybrána sada nástrojů HP ALM, frameworky Selenium WebDriver v kombinaci s NUnit frameworkem na jednotkové testování, s pomocí kterých byly vytvořeny manuální a automatické testy zaměřené na otestování funkčnosti a grafického rozhraní zadané aplikaci, UISu.

Byl podrobně popsán proces vytvoření jednotlivých kroků testů, které měly kompletně pokrýt testovanou funkčnost, a následně byly spuštěny testy, jejichž získané výsledky byly zhodnoceny. Vypracované manuální testy pomohly k nalezení několika drobných nedostatků grafického rozhraní systému, automatické testy dokázali správnou funkčnost aplikace.

Jako hlavní přínos této práce lze označit obecné shrnutí základů testování, a vytvoření přehledu některých testovacích nástrojů. Dále vytvoření ukázkových testovacích případů s používáním několika nástrojů pro manuální a automatické testování. Podrobně popsán proces tvoření jednotlivých kroků testu, které měly kompletně pokrýt testovanou funkčnost, a následně byly testy spuštěny a získané výsledky byly zhodnoceny.

7 Seznam použitých zdrojů

1. PATTON, Ron. *Testování softwaru*. Vyd. 1 . Praha: Computer Press, 2002. Programování. ISBN 80–722-6636-5 .
2. P . Bourque and R .E . Fairley, eds., *Guide to the Software Engineering Body of Knowledge*, Version 3 .0 , IEEE Computer Society, 2014; Dostupné z : www.swebok.org.
3. HEROUT, Pavel. *Testování pro programátory*. České Budějovice: Kopp, 2016. ISBN 978-80–7232-481-1 .
4. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press, 2013. ISBN 9788025138168.
5. LEWIS, William E ., David DOBBS a Gunasekaran VEERAPILLAI. *Software testing and continuous quality improvement*. 3rd ed . Boca Raton: CRC Press, c2009. ISBN 1420080733.
6. ISTQB [online]. [cit. 30 .02 .2020]. Dostupné z : http://castb.org/wp - content/uploads/2018/09 /ISTQB_CZ _Glossary_20180831.pdf
7. PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. *Jak testuje software Microsoft.* Brno: Computer Press, 2009. ISBN 9788025128695.
8. ALSHAMRANI, A ., BAHATTAB, A . *A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model.* In : International Journal of Computer Science Issues (IJCSI) Volume 12 , Issue 1 , No 1 , January 2015 [online]. [cit. 30 .02 .2020]. Dostupné z : https://www.ijcsi.org/papers/IJCSI-12 -1–1 - 106-111.pdf
9. [online]. Dostupné z: http://test.swtestovani.cz /
10. AMMANN, Paul a Jeff OFFUTT. *Introduction to software testing*. Edition 2 . New York, NY , USA: Cambridge University Press, 2017. ISBN 9781107172012.
11. KRISHNAN, M . S ., *Software development risk aspects and success frequency on spiral and agile model.* Journal of Computer Science Issues (IJCSI) Vol. 3 , Issue 1 , January 2015 [online]. [cit. 30 .02 .2020]. Dostupné z : https://pdfs.semanticscholar.org/6258/87a3a16e3164025cd7c369172b4391c9a626.pdf
12. *Agile Alliance* [online]. [cit. 2018-03–08]. Dostupné z : ttps://www.agilealliance.org
13. JORGENSEN, Paul. *Software testing: a craftsman's approach*. Fourth edition. Boca Raton, [Florida]: CRC Press, Taylor & Francis Group, [2014]. ISBN 1466560681.

14. 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 .
15. ACHARYA S ., PANDYA V . *Bridge between Black Box and White Box–Gray Box Testing Technique* //International Journal of Electronics and Computer Science Engineering [online]. Copyright © [cit. 05 .03 .2020]. Dostupné z : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10 .1 .1 .303.4479&rep=rep1&type=pdf
16. *Testování softwaru* [online]. b.r. [cit. 05.03.2020]. Dostupné z: http://testovanisoftwaru.cz/
17. *HP Application Lifecycle Management User Guide* [online]. Copyright © [cit. 05 .03 .2020] Dostupné z : https://softwaresupport.softwaregrp.com/doc/KM00793981?fileName=hp_man_ALM12.00_User_Guide_pdf.pdf
18. *SeleniumHQ Browser Automation* [online]. [cit. 05.03.2020]. Dostupné z https://www.selenium.dev/documentation/en /
19. *NUnit.org* [online]. Copyright © 2019, Charlie Poole, Rob Prouse. [cit. 5.03.2020]. Dostupné z: https://nunit.org/
20. What is Agile Software Development? - Salpo Technologies. *Salpo Technologies solves business problems with custom software, mobile apps and CRM* [online]. Copyright © Salpo Technologies Ltd 2020 [cit. 5.03.2020]. Dostupné z: https://www.salpo.com/what-Is-agile-software-development
21. Black Box Testing Ranorex Test Automation Tools. *Test Automation for GUI Testing | Ranorex* [online]. Copyright © 2020 Ranorex GmbH. All Rights Reserved [cit. 5.03.2020]. Dostupné z: https://www.ranorex.com/black-box-testing-tools/