Mendelova univerzita v Brně
Provozně ekonomická fakulta

Automatizované testování finanční webové aplikace

Diplomová práce

Vedoucí práce:

Ing. Jaromír Landa Ph.D. **Bc. Jitka Procházková**

Brno 2015

Chtěla bych poděkovat týmu zaměstnanců firmy EmbedIT za ochotu a cenné rady. Dále bych ráda poděkovala Ing. Jaromíru Landovi Ph.D. za vedení diplomové práce, cenné rady a konzultace.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Automatizované testování finanční webové aplikace**
vypracovala samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*

Jsem si vědoma, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 28. prosince 2015 ______________________________

Abstract

Procházková J. Automated testing financial web application. Diploma thesis. Brno: Mendel University, 2015.
This diploma thesis deals with the design and implementation of automated tests in the IT firm EmbedIT., which should present solutions to improve and increase the efficiency of automated tests.

Keywords

Automated testing, unit testing, financial applications, web applications, regress testing, Selenium, design patterns, page object pattern, Java, Maven.

Abstrakt

Procházková J. Automatizované testování finanční webové aplikace. Diplomová práce. Brno: Mendelova univerzita v Brně, 2015.
Tato diplomové práce se zabývá návrhem a implementací automatizovaných testů v dané IT firmě EmbedIT, které by mělo současné řešení vylepšit a zvýšit tak efektivnost automatizovaných testů.

Klíčová slova

Automatizované testování, finanční aplikace, webové aplikace, unit testy, regresní testování, Selenium, návrhové vzory, page object pattern, Java, Maven.

Obsah

1 Úvod a cíl práce 14

1.1 Úvod 14

1.2 Cíl práce 14

1.3 Metodika práce 15

2 Testování softwaru 16

2.1 Definice testování softwaru 16

2.1.1 Kdy přestat testovat 17

2.2 Chyby z testů 17

2.2.1 Životní cyklus chyb 19

2.2.2 Příčiny chyb 20

2.3 Životní cyklus testování 21

2.4 Členění testů 23

2.4.1 Funkční testování (Functional testing) 24

2.4.2 Nefunkční testování (Non-functional testing) 25

3 Automatizované testování 26

3.1 Definice automatizovaného testování 26

3.2 Proces automatizovaného testování 27

3.3 Důvody vedoucí firmy automatizovat testování 29

3.4 Nevýhody, mýty a problémy automatizace 31

3.5 (Ne)vhodné testovací případy k automatizaci 32

4 Analýza stavu automatizovaných testů ve firmě EmbedIT 33

4.1 Proces automatizace testů 33

4.1.1 Příprava testovacích scénářů 35

4.1.2 Implementace řešení automatizace testovacích scénářů 35

4.1.3 Provedení code review 36

4.1.4 Joby spouštějící testy na serverech 37

4.1.5 Vyhodnocení výsledků testů 37

4.1.6 Zadané chyby z automatizovaných testů ... 38
4.1.7 Přidání nové funkcionality ... 38
4.2 Proces automatizace v procesu vývoje softwaru ve firmě ... 39
4.3 Srovnání procesu automatizace ve firmě s těmi obecnými ... 41
4.3.1 Testovací plán ... 42
4.3.2 Kontrolní dokument ... 42
4.3.3 Definování rozsahu automatizace testovacích případů ... 42
4.3.4 Návrh a vývoj testovacích scriptů ... 42
4.3.5 Spouštění testů ... 43
4.3.6 Analýza výsledků testů ... 43
4.3.7 Údržba testů ... 43
4.4 Souhrn zjištěných problémů v procesu automatizace testů ... 43
4.5 Testy Online půjčky ... 45
4.5.1 Výčet testů ... 45
4.5.2 Funkcionallita aplikace, která není obsahem testovacích scénářů ... 47

5 Použité technologie a nástroje pro implementaci řešení automatizace ve firmě Embedit 49

5.1.1 Junit ... 50
5.1.2 Selenium Webdriver ... 52
5.1.3 Maven ... 52
5.1.4 Page Object Pattern ... 53

6 Automatizované testování finanční webové aplikace a její specifika 55

6.1 Specifika webové aplikace a jejich testování ... 55
6.1.1 Webová aplikace a druhy elementů ... 59
6.1.2 Způsoby hledání elementů na stránce ... 60

7 Návrh a implementace řešení procesu automatizace 63

7.1 Zajištění kvality testovacích scénářů ... 63
7.1.1 Změna přístupu k regresním testům ... 63
7.1.2 Stanovení zásad při tvorbě testovacích scénářů ... 63

7.1.3 Zavedení podprocesu test review ... 64
7.1.4 Zavedení podprocesu předání testovacího scénáře ... 64
7.2 Evidence možných dopadů nové funkcionality na tu stávající ... 65
7.3 Evidence změn v testovacím plánu ... 66
7.4 Kontrolované určení rozsahu automatizace testovacích scénářů ... 67
7.5 Evidence testovacího týmu pro automatizaci ... 67
7.6 Kontrola dodržení doporučených změn v rámci CR ... 67
7.7 Automatizace vytváření jobů ... 68
7.8 Paralelní spouštění automatizovaných testů na vzdálených strojích ... 68
7.9 Upravený proces automatizace dle návrhu ... 68

8 Návrh a implementace řešení zjištěných nedostatků v testech Online půjčky 70

8.1 Stávající automatizované testy ... 70
8.1.1 SecondPageCalculationTest ... 70
8.1.2 MinMonthlyInstalmentTest ... 71
8.1.3 MaxMonthlyInstalmentTest ... 71
8.1.4 SmsCodeVerificationTestWrongCaptchaTest ... 71
8.1.5 AlternativesTest ... 72
8.1.6 RejectAlternativeTest ... 72
8.1.7 InsuranceChangeTest ... 73
8.1.8 LoanReject ... 73
8.2 Nové automatizované testy ... 73
8.2.1 CheckTextOnFirstPageTest ... 73
8.2.2 CheckDeliveryTest ... 74
8.2.3 CheckChosenCashAmountInstalmentsTest ... 74
8.2.4 CheckRecapitulationTest ... 75

9 Srovnání řešení 77

9.1 Srovnání řešení procesu automatizace ... 77
9.2 Srovnání řešení automatizovaných testů Online půjčky ... 78

10 Diskuze 79

10.1 Ekonomické zhodnocení 79
10.2 Přínos práce 80
10.3 Možnosti rozšíření práce 80

11 Závěr 82

12 Literatura 83

Přílohy 87

Seznam obrázků

Obr. 1 Náklady na opravu chyb. Zdroj [10] ... 18
Obr. 2 Životní cyklus chyby. Zdroj [32] ... 19
Obr. 3 Životní cyklus testů. Zdroj [2] ... 22
Obr. 4 Proces automatizovaného testování. Zdroj [25] ... 28
Obr. 5 Proces automatizace. Zdroj [26] ... 29
Obr. 6 Diagram procesu automatizace ... 34
Obr. 7 Procesy životního cyklu softwarového produktu firmy ... 41
Obr. 8 Ukázka posuvníků s výší úvěru a počtem splátek ... 55
Obr. 9 Ukázka prvků volitelných služeb ... 56
Obr. 10 Ukázka formuláře ... 57
Obr. 11 Ukázka rekapitulace ... 58
Obr. 12 Ukázka ověrovacích prvků v aplikaci ... 59
Obr. 13 Začlenění evidence NF do procesu vývoje ... 66
Obr. 14 Upravený proces automatizace ... 69

1 Úvod a cíl práce

1.1 Úvod

Testování aplikací neodmyslitelně patří k fázím vývoje softwaru, přičemž jeho smysl je jasný – zajistit, aby výsledný produkt byl ve shodě s funkcionalitou toho zadaného zákazníkem. Je tudíž důležitou součástí procesu, který vede k hotovému produktu.

Testování webových aplikací je spjato s vývojem webových aplikací, kdy se z webu stala prakticky nová platforma, pro kterou lze vyvíjet aplikace. Omezením jsou pak už jen internetové prohlížeče. Různé verze různých prohlížečů totiž zobrazují některé prvky na stránce jinak či vůbec, což je hned jasným námětem pro testery na tvorbu testovacích scénářů.

Každý člověk je ve své podstatě testerem. Dennodenně se setkává s něčím novým a lidská zvědavost nás vede k tomu, abychom takovou to věc okusili, zjistili jeho funkcionalitu, otestovali. Ne každý je však dobrým testerem.

Protože celá společnost stojí čím dál více na výpočetní technice a používá software nejrůznějšího druhu, je třeba, aby tento software byl spolehlivý. Ostatně s touto spolehlivostí se počítá, občas se bere jako samozřejmost.

Než se však software stane spolehlivým, musí projít sadou testů. Již dávno skončila doba, kdy vývoj software byl pouze na vývojářích, kteří sami prováděli testy. Dnes existují celé divize testerů, kteří mají často větší znalosti o fungování celého softwarového produktu, než samotní vývojáři či analytici.

Pokud produkt neprojde sadou testů, může to být velice drahé. Chyby, které se pak objeví v ostré verzi v provozu, mohou stát miliony.

Kvůli tomu, že se stále snižuje doba, za kterou je produkt uveden na trh, je třeba proces vývoje zrychlovat. Týká se to také testů a je to rozhodně jeden z důvodů, proč firmy sáhnou po automatizaci testů.

Zavedení automatizace však vyžaduje poměrně vysoké počáteční náklady. Navíc automatizace nemusí být vždy úplně tím nejlepším řešením. Klíčové je zde vědět, jaké výhody a též náklady to pro firmu znamená. Pak je větší šance, že se firma rozhodne tak, jak je v dané situaci pro ni nejlepší [1].

Ve firmě pracuji jako tester na plný úvazek téměř dva roky a tak jsem měla možnost sledovat utváření a průběh všech procesů vývoje software ve firmě.

Některé obrázky a kód v uvedený v této práci je pozměněn z důvodu ochrany majetku firmy.

1.2 Cíl práce

Cílem práce bude analýza stavu automatizovaných testů vybrané webové aplikace ve firmě EmbedIT a z toho vyplývající návrh a implementace procesu automatizace s cílem odstranit nedostatky a chyby v současné použité metodice či technologiích a tudíž zlepšit efektivitu celého procesu automatizace testů ve firmě.

1.3 Metodika práce

Po seznámení se s problematikou testování i automatizovaného testování, je provedena analýza automatizovaných testů ve firmě. Je popsán celý proces automatizace a jeho zjištěné problémy ve všech podprocesech. Také se tím myslí jaké technologie a nástroje jsou použity.

Na základě zjištěných nedostatků, v celkovém procesu automatizace i jednotlivých testech, navrhnu řešení. Toto řešení implementuji a porovnám jej s řešením firmy.

2 Testování softwaru

Testování zde existuje v podstatě od doby, kdy začal vývoj softwaru. Automatizované testování začalo vznikat v době, kdy systémy byly stále více složitější a vznikaly potřeby po nových přístupech [1].

Nicméně, dle Hayensové spíše než potřeby nových přístupů, je tu potřeba rychlosti. Technologie jsou dnes totiž využívány jako zbraně, kterými se dá bojovat proti konkurenci, je totiž třeba dodat veškeré softwarové produkty včas. Pokud se takovéto produkty dostanou na trh pozdě, mohou ztratit zákazníky, příjmy a podíl na trhu. Tyto hospodářské tlaky pak vedou společnosti k tomu, aby využívali automatizace, která šetří celkový čas uvedení produktu na trh [1].

Jaká je však definice automatizovaného testování? Dá se říct, že automatizované testy, jsou vším, co jsou testy obecně, jen je celý proces automatizovaný? Pro zodpovězení této otázky je nejrpve třeba si definovat, co je to testování softwaru.

2.1 Definice testování softwaru

Podle Zelinky [2] je testování softwaru chápáno jako:„proces získávání informací o stavu a vlastnostech systému za účelem jejich dalšího zpracování“. Testování samo o sobě je celkem jasně o hledání chyb.

Testování je tou jedinou věcí, kterou je možno provést, ke snížení počtub chyb v hotovém produktu [3]. Naproti tomu Dijkstra [4] vysvětluje, že ačkoliv testováním se efektivně ukazuje přítomnost chyb, nevede to k prokázání toho, že chyby v softwaru nejsou. Tedy může dojít k redukci počtu chyb v softwaru, ale provedené testování s výsledkem bez chyb rozhodně není zárukou, že chyby skutečně nejsou přítomny.

Jakákoliv aktivita, která odhaluje chování programu porušující jeho specifikaci může být nazvána testováním [5]. Což se celkem shoduje s verzí od Havlíčkové a Roudenského[6], kteří definují testování jako:„Proces řízeného spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů“.

Můžeme říct, že testování softwaru je aktivita, jejíž cílem je zjistit, zda chování programu splňuje specifikované či implicitní potřeby uživatelů, nicméně není zaručeno, že výsledek testů neobsahující žádné chyby, znamená bezvadnost programu.

Tato definice vytvořená z výše uvedených tvrzení, platí jak pro automatizované testování tak i pro manuální testování. Ačkoliv je třeba si uvědomit, že mezi manuálním a automatizovaným testováním je rozdíl. Dle Hayesové [1] je chybou se domnívat, že automatizované testování je prosté zachycení a přehrání manuálně prováděného testovacího procesu. Ve skutečnosti, automatizacovaný testovací proces je odlišný od toho manuálního a to tím, že řeší jiné problémy a má jiné možnosti.

2.1.1 Kdy přestat testovat

Jak napsal Dijkstra [4], vzhledem k tomu, že nemůžeme dokázat, že již nejsou žádné další chyby v programu, kdy tedy přestat testovat? K určení konce testování je možné použít heuristických argumentů založených na důkladnosti a propracovanosti testerského úsilí a trendy ve výsledcích nacházení chyb vedoucí k víře, že je riziko nalezení dalších chyb menší [5].

Představa jedné obrazovky, která má 15 políček pro vstupní hodnotu a každé pole má 5 různých možných hodnot, vede k více než 30,5 miliardám různých testů. Je nepravděpodobné, že by se umožnil takový počet testů v časovém rámci pro projekt. Na místo toho se použije takový testovací přístup, který zajistí správné množství testů. To se provede srovnáním s testy a s rizikem pro zákazníka, projekt a software. Hodnocení a řízení rizik je jedna z nejdůležitějších aktivit v každém projektu a je zároveň klíčovou aktivitou a i důvodem pro testování [7].

Pro rozhodnutí o tom, kdy s testováním přestat, se tedy musí vzít v potaz úroveň rizika, včetně technických a obchodních rizik spojených s výrobkem a omezením projektu, jako je čas a rozpočet [7].

Ve zdroji [8] je uvedeno, že je těžké přesně určit, kdy přestat testovat, protože testování je nikdy nekončící proces a nikdo nemůže říct, že jakýkoliv software je dokonale otestovaný. Nicméně lze zmínit některé aspekty, které by měly být zváženy při rozhodování kdy skončit testy. Jsou to:

- Blížící se konečný termín testů.
- Dokončení otestování všech testovacích případů.
- Množství chyb je pod určitou hranicí a nejsou žádné chyby s vysokou prioritou.
- Z prostého rozhodnutí managementu.

2.2 Chyby z testů

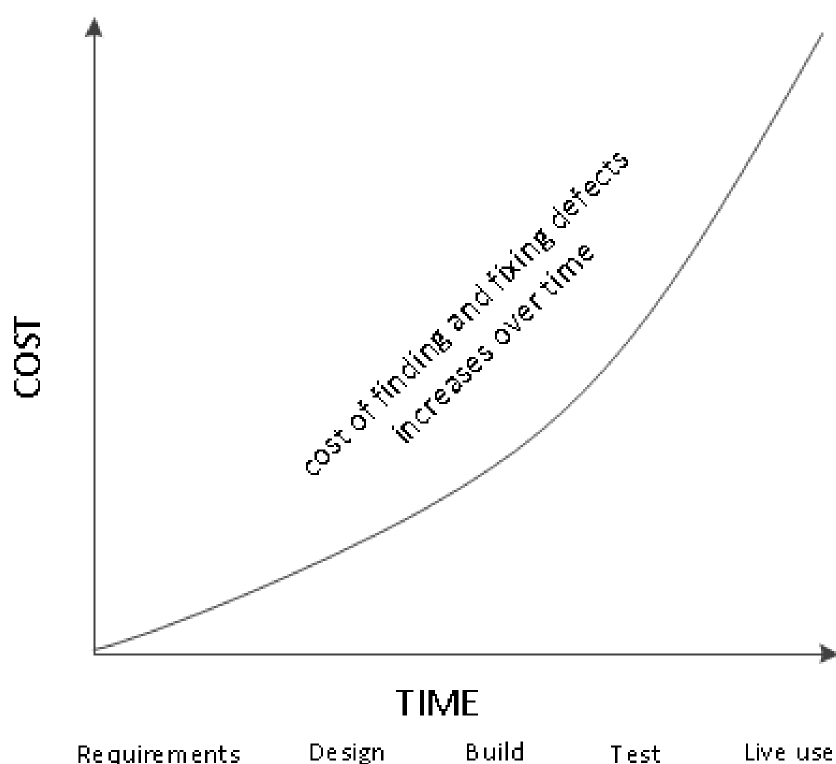
Ať už automatizované či manuální testování, oboje generuje chyby. Patton [9] definuje softwarovou chybu, když nastane jedna z následujících možností:

- Software nedělá něco, co by podle specifikace měl dělat.
- Software dělá něco, co by podle specifikace neměl dělat.
- Software dělá něco, o čem se nezmiňuje specifikace.
- Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale zmiňovat by se měla.

- Software je obtížně srozumitelný, tzn. těžko se s ním pracuje, je pomalý a nebo ho, dle úsudku testera, nebude koncový uživatel pokládat za správný.

Tento popis, vysvětlující co je to chyba, je sice obsáhlejší, nicméně zdroj [10] dále vysvětluje, že když programátor udělá chybu (error), výsledkem toho je chyba (defect, fault, bug) ve zdrojovém kódu softwaru. Pokud je tato chyba (defect) spuštěna, v jistých situacích pak software vyproduktuje špatné výsledky, které způsobí jeho selhání (failure). Ne všechny chyby (defects) však musí nutně způsobit selhání, poruchu (failure) softwaru.

Náklady na zjištění a opravu chyb (defects) výrazně stoupají v průběhu životního cyklu produktu. Je totiž snažší opravit chybu hned, když je to malá chyba, než když ji necháme projít až na produkční systémy, kde může způsobit velké problémy [7].



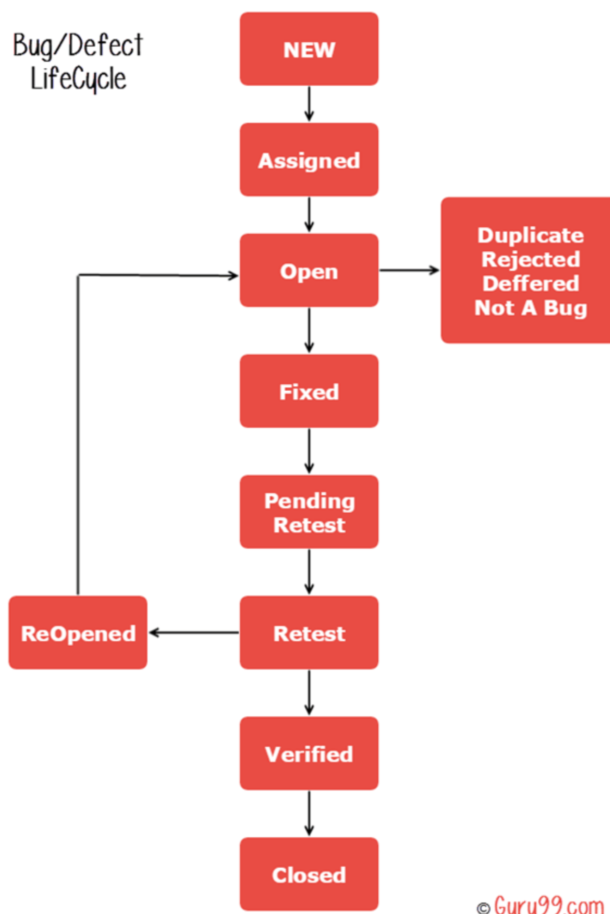
Obr. 1 Náklady na opravu chyb. Zdroj [7]

Na obrázku č.1 je graf, který vysvětluje, že čím později se chyba opraví, tím větší náklady to budou pro firmu. Pokud se totiž chyba odhalí již v zadávání požadavků, ty se pak předělají, aby se chyba nedostala dál. Další fáze, protože se ještě nezačaly realizovat, nejsou nijak ovlivněny. Nejhorší je, pokud se chyba najde až v ostrém provozu (Live use), kdy se musí jak upravit požadavky, tak návrh, kód softwaru a též testy [7]. Je tudíž důležité odhalit chyby co nejdříve.

Jsou však případy, kdy po objevení chyby v testech, nemusí nutně znamenat změnu celého návrhu a požadavků. Nejvíce chyb je totiž přímo v programovém kódu nebo ve vzhledu aplikace [11].

2.2.1 Životní cyklus chyb

Nalezená chyba (defect) má svůj vlastní životní cyklus. Tento cyklus má od nalezení po vyřešení různé fáze, kterými chyba prochází. Ten se liší mezi organizacemi a také mezi projekty, protože je řízen procesem testování a ten samotný je napříč firmami různý. Také na něj má vliv použitý nástroj pro správu chyb [12].



Obr. 2 Životní cyklus chyby. Zdroj [13]

Vše začíná stavem nová chyba (new). Jde o potenciální chybu, která ještě neprošla validací. K tomu je třeba ji napsat (assigned) na vývojáře, který se na zadanou chybu podívá [13].

Ve stavu otevřená (open) chyba ji vývojář začne analyzovat a pracovat na opravě chyby. Jakmile skončí s opravou chyby a provede veškeré potřebné změny v kódu, dá chybu do stavu opraveno (fixed). Chyba je tedy opravená a tak se čeká, až ji tester otestuje – je ve stavu čekání na testy (pending retest). Tester si ji v tomto

stavu převezme a dá ji do stavu retest, kdy jej provede. Pokud však chyba přetrvává i po opravě, dá ji do stavu znovu otevřená (reopen). Když je chyba opravena, přepne chybu do stavu verified (verifikováno) a poté ji uzavře (closed). Pokud se zjistí, že nejde o chybu, zvolí se stav zamítnutá (rejected) a nebo stav není chyba (Not a bug). Když už byla chyba zadaná někým jiným, jedná se o duplikát (duplicate) a tak se přepne do stejného stavu [13].

Některé stavy mohou být nahrazeny stavem jedním. Například přípsání na vývojáře (assign) a otevřená chyba (open) mohou být nahrazeny stavem aktivní (active). Dále čekání na retest (pending retests) jen stavem retest a opraveno (fixed) může být též nahrazeno stavem test [12].

Uvedený životní cyklus chyby v podobě přechodných stavů je pouze ukázkový. Nezobrazuje zrovna ten jediný správný průběh. Jak již bylo řečeno na začátku, jeho podoba závisí na firmě a zvoleném nástroji pro evidenci chyb.

2.2.2 Příčiny chyb

Příčina chyb je celkem jasná, je totiž obecně známo, že lidé nejsou neomylní a dokud budou vytvářet software právě lidé, chyby v nich budou.

Mohou to být chyby (defects) udělané v programovacím kódu, či chyby v dokumentaci nebo již ve specifikaci produktu [7].

Na příčiny chyb se dívá trochu jinak Pradhan [14], který uvádí několik důvodů, proč takové chyby nastávají. Jsou jimi:

- Lidský faktor – jak již bylo zmíněno výše a obecně se ví, člověk dělá chyby

- Chyby v komunikaci – tím se myslí nedorozumění, nedostatek komunikace či chybná komunikace při vývoji softwaru. To pak může vést až k situaci, kdy se programátoři musí vypořádat s problémy, kterým v podstatě nikdo nerozumí

- Nereálný časový rámec pro vývoj – z důvodu nedostatku času pro vývoj se často musí dělat kompromisy, které nejsou vždy šťastným řešením. Není dost času jak na programování tak na testy.

- Chabý návrh – některé návrhy jsou tak komplikované, že vyžadují spoustu brainstormingu. Na návrh se však vždy spěchá, aby se mohlo začít s implementací co nejdříve a tak se co nejvíce zjednodušuje. Navíc se dost často nerozumí celkové problematice navrhovaného produktu, použijí se špatné technologie, což vše vyústí v možné chyby a problémy, jejichž řešení je drahé.

- Špatné postupy v programování – chyby mohou být zavlečeny do softwaru i proto, že se jednoduše programuje špatně. Chybí dostatek validací, řešení vyjímek a podobně.

- Špatná správa verzí – je v případě, kdy se v regresních testech objevují napříč releasy stále stejné chyby a to v pravidelných intervalech.

- Zavedené chyby v nástrojích třetích stran – protože v průběhu vývoje softwaru se používají i nástroje třetích stran, které si firma nevyvinula sama. Ty mohou obsahovat spoustu nepříjemných chyb, které pak zavlečou chyby i do firmou vyvíjeného softwarového produktu.

- Nedostatek kvalifikovaných testů – myslí se tím nedostatky v procesu testování. Ale také sem patří skutečnost, že testeři neberou testy dost seriózně, takže jde i o jejich přístup. Nicméně není to jen o testerech, samotní programátoři dělají jen málo unit testů či vůbec žádné, což může mít za následek špatný programovací kód a proto i větší riziko chyb.

- Změny na poslední chvíli – jsou to změny, které se dělají krátký čas před releasem a tak již není například čas na testy. Mohou to být závěrečné testy kompatibility s operačními systémy či internetovými prohlížeči. Obecně se může jednat o komplexní změny, které vyžadují čas, nicméně se na ně spěchá a tak mohou potenciálně způsobit mnoho chyb ve výsledném softwarovém produktu.

V jiném zdroji [15] zabývající se stejnou problematikou ještě přidávají další důvody. Jsou jimi například přehnaně sebejistí lidé, ať už vývojáři, testeři či analytici, pro které je příliš mnoho věcí „bez problému“ a výsledkem je větší chybovost software. Dále složité automatizované testy, které se neaktualizují a tak nejsou schopny zjistit chyby. Případně opakované testovací případy nejsou automatizované a jsou stále závislé na ručním vyhodnocení testerů. Testeři pak nemají dostatek času na regresní testy, na které se obecně dává málo času, což vede též k větší chybovosti. Provádění těchto regresních, ale i jiných testů, není oprioritizováno.

Dle Pattona [9] je největším důvodem pro vznik chyb špatná nebo neexistující specifikace. Mezi důvody, které způsobují špatnou specifikacii, patří neustále se měnící specifikace, která není dostatečně komunikována s týmem, který vyvíjí produkt, jehož se specifikace týká.

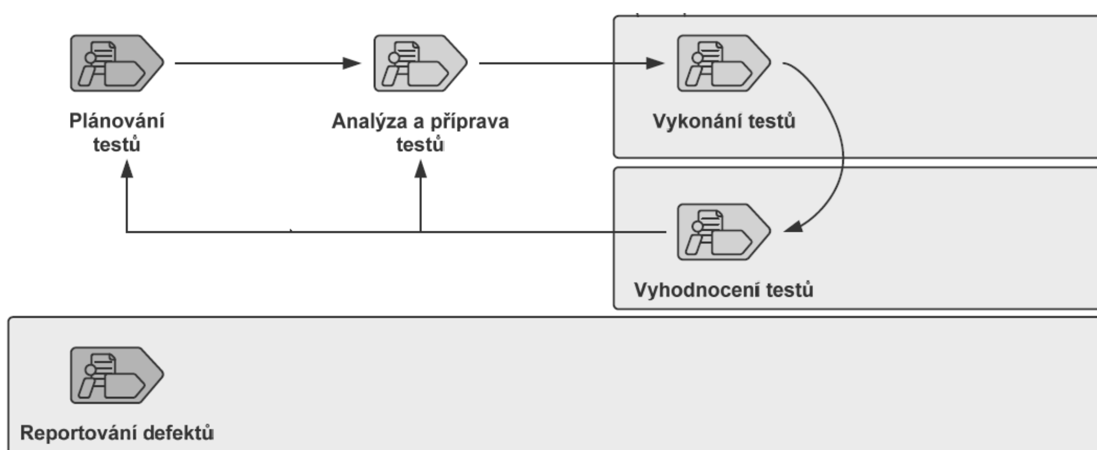
Že jsou chyby ve specifikaci nejčastější příčinou vzniku chyb však přímo rozporuje Borovcová [11], která tvrdí, že kvůli pokroku v softwarovém inženýrství se to změnilo na chyby v programovém kódu nebo vzhledu aplikace. Závisí to však na typu projektu, typu vývoje a na tom, jak schopní lidé jsou ve firmě.

2.3 Životní cyklus testování

Životní cyklus testování je součástí vývojového cyklu. Testování však není jedna aktivita, nýbrž proces, který má vlastní životní cyklus. Různé organizace mají různé fáze životního cyklu testování [16], v závislosti na tom, jak se rozhodne vedení firmy a v závislosti na životním cyklu vývoje softwarového produktu [17]. Pod procesem

testování se skrývá hned několik aktivit. Všechny jsou zobrazeny i v obrázku níže. Dle pana Zelinky [2] jsou jimi:

- Plánování testů – příprava testovacího plánu, která se odvíjí od definice projektu a stanovení toho, co kontrolovat.
- Analýza a příprava testů – v této fázi se připravují testovací data a navrhují jednotlivé testy.
- Vykonání testů – provedení testů.
- Vyhodnocení testů – výhodnocení případných chyb.
- Reportování chyb a jejich další sledování – nahlášení chyb a zadání jich k řešení vývoji. Kontrola jejich počtů, jak moc byly závažné.



Obr. 3 Životní cyklus testů. Zdroj [2]

Obrázek výše popisuje postup jednotlivých procesů. Nejprve se plánují testy, poté se provádí jejich analýza a příprava, následovaná vykonáváním testů. Po proběhlých testech se provádí jejich vyhodnocení, které může vést k úpravám v přípravě testů či k plánování testů. Po celou dobu se se provádí reportování defektů.

Profesor Sathish [10] se na životní cyklus dívá jinak a dle něj se testování skládá i z jiných kroků, než jaké jsou uvedeny výše. Kromě plánování a analýzy testů, jejichž význam je obdobný, má ještě navíc kroky:

- Návrh testů – podoby testovacího plánu a testovacích případů jsou v této fázi revidovány a dokončeny. Také se zde vybírají testovací případy k automatizaci a připřavují testovací data a testovací prostředí.
- Konstrukce a verifikace – zde je splněn testovací plán, dokončen vývoj automatizovaných testů, dokončeny testovací případy. Všechny chyby, které se našly jsou reportovány.

- Testovací cyklus – v této fázi se musí dokončit všechny testovací případy tak, aby již negenerovaly žádné chyby. Pokud ano, provede se znovu spuštění testovacích případů, zadávání chyb, atd.

- Finální testování a implementace – Zde se spouští zbývající výkonostní a stresové testy. Dokumentace pro testy je kompletní.

- Post-implementace – testovací proces je vyhodnocen a plyne z něj poučení a další dokumentace. Je zde snaha o předcházení podobných problémů a nalezených chyb budoucích projektů.

Jiný zdroj [17] uvádí ve fázích životního testovacího cyklu též fáze jako přezkoumání projektu a jeho požadavků. Nastavení testovacího prostředí a spouštění testů (ať už manuálně prováděných či automatizovaných) zde uvádí jako samostatnou fázi, nikoliv součást fáze jiné. Navíc přidává fázi report testů, jejíž výsledkem jsou reporty o výsledcích testů, metrikách chyb a podobně.

V rámci zkoumání požadavků a projektů se dle zdroje [18] myslí spíše zjištění, které z požadavků od zákazníka jsou testovatelné a které ne. Což celkově pomáhá k určení rozsahu testů. Po dokončení testů je fáze, ve které se opět posílají různé reporty, jejichž obsah závisí na tom, komu se tyto reporty posílají. Poslední fází je kontrola dokončení testů a kontrola zda nejsou chyby s vážnými dopady na zákazníka stále otevřené. Také se vytváří dokument, který má vést k poučení a předcházení chyb, stejně jako uváděl profesor Sathish výše.

Jak je vidět, různé zdroje se relativně shodují v tom, co má obsahovat životní cyklus testování za fáze či aktivity. Nicméně, jak již bylo zmíněno [16], přesná podoba životního cyklu může být napříč firmami různá.

2.4 Členění testů

Typy testů představují způsob k jasné definici úkolů či cílů v konkrétní testovací úrovni pro program či projekt [7]. Členit testy lze dle různých ukazatelů, například míry kontroly programového kódu či způsoby provádění testů [2]. Dle míry kontroly programového kódu lze dělit testy na:

- White box – někdy se též používá výraz clear-box či glass-box, zná tester jak principy fungování, tak i strukturu programu a datový model testovaného softwaru [2]. Má tedy přístup k těmto informacím a pomáhají mu s testováním. Příkladem mohou být unit testy.

- Black box – tester zná, co by software měl dělat, ale nemůže se podívat jak to dělá, jakým způsobem operuje. Jen dává nějaké vstupy a kontroluje výstupy [9]. Neví pak nic o logice, která stojí za výslednými výstupy. Patří sem například akceptační testování (User acceptance Testing) [3].

Dle způsobu provádění testů se mohou dělit na:

- Manuání testy – testy provádí a vyhodnocuje tester sám, manuálně. Provádí všechny kroky dle podkladů [2].

- Automatizované testy – testy provádí specializovaný software. Více o automatizovaných testech v kapitole Automatizované testování.

2.4.1 Funkční testování (Functional testing)

Funkční testování se provádí se zacílením na vhodnost, schopnost a ochotu systémů spolupracovat. Též na bezpečnost a přesnost [7]. Funkční testy mají za cíl otestovat všechny funkce, které jsou v aplikaci implementovány a ověřuje se tak, že fungují správně a odpovídají požadavkům zákazníka [19]. Do této kategorie patří:

- Smoke testy – test, který by měl v podstatě zjitit, zda systém funguje [3]. Tento test nám též říká, zda byl kód jednotlivých komponent správně sestaven dohromady [11]. Jedná se tedy o realizaci vybrané podmnožiny vytvořených nebo plánovaných testů s cílem ověřit správnou funkčnost všech kritckých částí systému [20]. Smoke testy jsou obvykle spouštěny po každém sestavení, aby se zjistilo, zda je aplikace v testovatelném stavu [21].

- Regresní testy – v rámci vývoje nové funkcionality či oprav chyb jsou mnohdy některé jeho části modifikovány. Tyto modifikace sebou nesou riziko, že jejich následkem bude do systému zavlečen defekt. Ne vždy je to chyba vývojáře, někdy jde o nedomyšlenosti samotného návrhu. Pro ověření, že nebyla zavlečena žádná chyba, se provádí regresní testy. Ověřuje se tak správnost fungování stávající funkcionality. Takže jde v podstatě o kontrolu oblastí, které zůstaly v programovém kódu nezměněny, ačkoliv na jejich fungování mohl mít dopad vývoj nové funkcionality [2].

- Unit testy – je v podstatě testování jednotlivých softwarových komponent [20]. Tyto testy lze chápat, jako testování nejmenšího stavebního bloku programu. V závislosti na tom, zda je program psán objektově či procedurálně se chápe jako programová jednotka samotná funkce, třída či metoda. Cílem je otestování postupně všech jednotek nezávisle na ostatních a prokázat tak, že její fungování takto v izolaci je korektní. Protože je programová jednotka testována v izolaci, může být třeba přidat simulaci externích prvků. Takovýmto simulovaným externím prvkům se říká stubs či mocks, což lze volně přeložit jako zástupce či imitátory skutečných objektů. Provádění unit testů je výhodné především proto, že jsou opakovatelné a lze je automatizovat, což pak ve výsledku nutí vývojáře se více nad kódem zamýšlet [2]. Zde je jasné, že jde o white-box testování.

- Integrační testy – jde o testování integrace menších jednotek do větších celků, dávající dohromady systém. Jednotky, na rozdíl od jednotkových

(unit) testů, nejsou již testovány samostatně, ale ve skupinách. Cílem je sledovat interakci mezi nimi [3].

2.4.2 Nefunkční testování (Non-functional testing)

Nefunkční testy jsou o testech všech vlastností aplikace, které přímo nesouvisí s jejími funkcemi, ačkoliv jsou důležité pro její správné fungování. Je to například testování výkonu (performance testing), které ověří, zda je aplikace schopna pracovat svižně i pod konkrétní zátěží. Na výslednou bezporuchovost software mají vliv právě nefunkční testy [19].

3 Automatizované testování

V této kapitole je vysvětlen pojem automatizované testování. Dále jsou zde zmíněny důvody pro automatizaci, výhody a nevýhody z toho plynoucí a také kde je zde uveden proces automatizovaného testování.

3.1 Definice automatizovaného testování

Za automatizované testování softwaru se považuje takové, kdy část testovacího procesu či celý proces, je prováděn bez přímého působení člověka a to tak, že se využívá specializovaný software [7].

O tom, že automatizovat lze část či celý testovací proces, se píše i na [24]. A to pro případy, kdy jsou vytvořeny stabilní manuální případy, které se opakovaně vykonávají.

Havlíčková a Roudenský [2] popisují automatizované testování jako kombinaci testovacích nástrojů, které běží bez zásahu lidské ruky a samy provádějí testové případy, vyhledávají chyby a analyzují a zaznamenávají výsledky.

Testovací proces jasně obsahuje provádění testových případů a vyhledávání chyb. Tudíž by se dalo skutečně říct, že pokud je minimálně část testovacího procesu prováděná bez přímého působení člověka, využitím specializovaného softwaru, jedná se o automatizované testování.

Automatizované provádění testových případů v podstatě znamená, že kód vykonává krok po kroku simulaci uživatele pro ověření správné funkcionality konkrétní části aplikace. Pracuje buď přímo s GUI rozhraním či webovými službami a vyhodnocuje reakce systému. Chování systému pak samozřejmě musí odpovídat očekávaným výsledkům [23].

Pod každým stisknutím tlačítka či zadání té či oné hodnoty, se dá očekávat nějaký výsledek, který se má někde zobrazit. Je za tím schovaná funkcionalita obsažená v back-endu, jejíž správné fungování ověříme právě výše zmíněnými kroky na front-endu. Z nadhledu se tak dá říct, že většina funkcí má účel a dobře napsané automatizované testy ověřují, zda byl tento účel naplněn [23].

Aby se dalo pracovat s grafickým uživatelským rozhraním aplikace či s webovými službami, je k tomu třeba použít nějaký nástroj, správné technologie. Což je v podstatě částí definice automatizovaného testování ze zdroje [24]. Jedná se o použití automatizační technologie po celou dobu životního cyklu testování, který má za cíl poskytnout rychlost provádění testů za rozumnou cenu. Výsledky plynoucí z použití různých technologií a metodik k implementaci automatizovaných testů se značně liší.

Automatizace testování má však svůj vlastní životní cyklus [1], který je podobný životnímu cyklu testování.

3.2 Proces automatizovaného testování

Někdy též nazývaný jako životní cyklus automatizace testování [1]. Každá z etap odpovídá určité činnosti a každá etapa má určitý výsledek [25]. Dle Hayesové [1] se jedná o tyto etapy:

- Testovací plán – popisuje kroky potřebné k automatizaci testování. Myslí se tím potřebné testovací knihovny, zdroje. Je zde rozsah testů, určen testovací tým, rozvrhnut průběh automatizace. Testovací plán je stále aktualizován v průběhu testovacího cyklu.

- Testovací případy – stanoví se testovací případy.

- Testovací scripty – oscriptují se stanovené testovací případy.

- Spouštění a údržba testů – spustí se a provede se případná korekce testů.

Dále Hayesová [1] již rozebírá více jen fáze testovacího plánu. Ten má několik podprocesů:

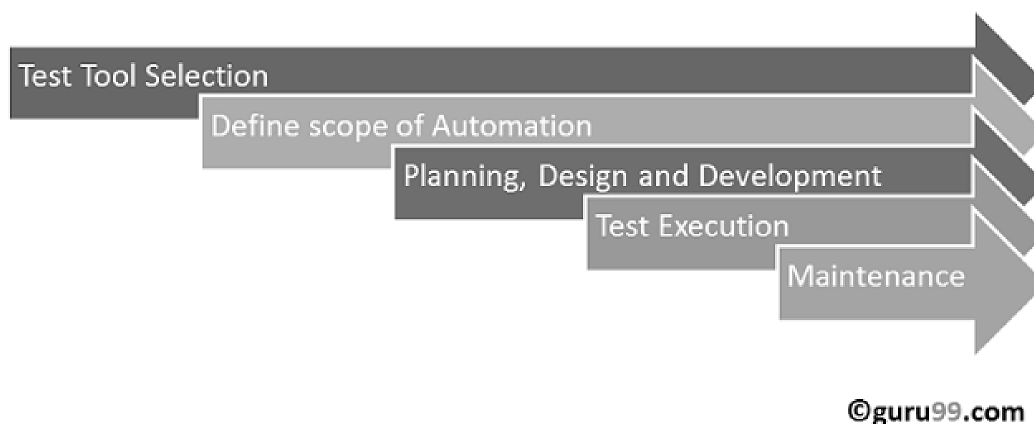
- Kontrolní dokument – používaný pro evidenci všech dodatků a změn v plánu. Jak už bylo zmíněno výše, testovací plán je neustále aktualizován a tak je třeba někde evidovat tyto aktualizace.

- Aplikace – popis testované aplikace.

- Rozsah automatizovaných testů – je důležité stanovit a přesně popsat, co bude a nebude otestováno a kdo bude zodpovědná osoba.

- Testovací tým – tým lidí, kteří budou řešit automatizaci testů.

Jiný zdroj [25] uvádí obdobné kroky jako výše Hayesová, nicméně přidává etapu: výběr testovacího nástroje (Test tool selection). Ten se odvíjí do značné míry na technologii aplikace, která má být testována. Jinak jsou kroky obdobné.



Obr. 4 Proces automatizovaného testování. Zdroj [25]

- Definování rozsahu automatizace (Define scope of automation) – jedná se o definici rozsahu testů softwaru. K jeho určení může pomoci definování si funkcí, které jsou důležité pro business; scénářů, které mají velká množství dat a běžné funkce v aplikaci či složitost testovacích případů.

- Plánování, návrh a vývoj (Planning, Design and Development)– v této fázi je již vybrán nástroj pro automatizaci, návrh frameworku a jeho funkcí. Jsou stanoveny testovací případy spadající do rozsahu automatizace a také jsou určené ty testy, které jsou již mimo rozsah. Také se zde stanoví plán a časová osa programování testů a jejich spouštění.

- Spuštění testů (Test Execution) – během této fáze se provádí spouštění automatizovaných testů.

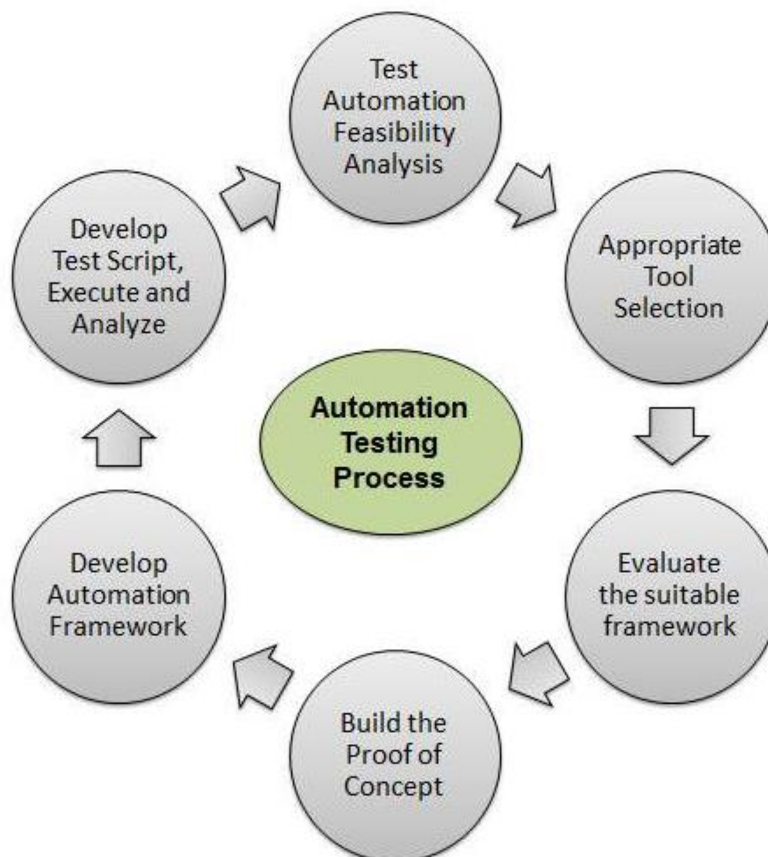
- Údržba automatizovaných testů (Maintenance) – testy musí být udržovány, protože v průběhu vývoje softwaru mohou být přidány nové funkce. Aby byly automatizované testy efektivní, musí se tento krok provádět každý release.

Do procesu automatizace se řadí i další fáze. Zdroj [26] přidává fázi, ve které na začátku zjišťuje, zda je možné testovaný software vůbec automatizovat (Test Automation Feasibility Analysis).

Je obecně známo, že ne každá aplikace se dá automatizovat, kvůli jeho limitům či nepodporovaným technologiím, které nedokážou například odchytávat prvky na stránce [2].

Zdroj [26] dále uvádí fázi, kde se vybírá nástroj pro automatizaci (Appropriate Tool Selection). Kromě nástroje však vybírá též vhodný framework (Evaluate the suitable framework). Speciální fází (Build the Proof of Concept) je, že se pomocí testovacího scénáře, který projde procesem v aplikaci od začátku do konce, ověří, že hlavní funkcionalita aplikace může být skutečně automatizována. Následující fází (Develop Automation Framework) je pak vývoj frameworku pro automatizaci, který

by měl být vyvíjen na základě důkladné analýzy technologie používané testovanou aplikací. Posledním krokem (Develop Test Script, Execute and Analyze) je vývoj testovacích scriptů, jejich spuštění a analýza výsledků testů. Níže je celý průběh graficky vyobrazen.



Obr. 5 Proces automatizace. Zdroj [26][25]

Výše jsou uvedeny různé fáze a různý obsah stejných fází jednoho procesu a to toho automatizačního. Těžko určit ten správný, nicméně ani jeden ze zdrojů netvrdí, že je to právě ten jejich.

3.3 Důvody vedoucí firmy automatizovat testování

Organizace se obecně snaží snížit náklady na testování. Přičemž většina organizací si netroufá snížit množství testů, místo toho se tedy dívají po jiných možnostech. Jednou z nich je právě automatizace testů [3].

Dle Havlíčkové a Roudenského [6] je základním důvodem pro zavedení automatizace testování zvýšení jeho efektivity. Protože obecně je snahou snižovat náklady na testování a přitom dodávat stále kvalitní software, zavádí se automatizace.

Patton [9] se na to dívá jinak. Ve většině firem vyvíjející software je koloběh naprogramování – otestování – oprava provedena několikrát, než je aplikace považována za hotovou. Je třeba ověřit, že chyby, které se nalezly a byly opraveny, nezpůsobily další chyby. I z tohoto důvodu je vhodné ty části aplikace, kterých se oprava týkala, testovat vícekrát. V menších firmách, kde je na projekt navázáno několik tisíc testů a je třeba dělat testy několikrát, je řešením právě automatizace, jinak by to nebylo možné, protože by prostě nebyly kapacity.

Podobně to vidí zdroj [27]. Po každé změně ve zdrojovém kódu se musí testy opakovat. Tudíž i po opravě chyby. Nicméně zde zmiňuje také fakt, že se sníží náklady. Manuální testování všech testů a to několikrát za sebou, je dražší a časově náročnější úkol. V rámci automatizace je pak větší výdej jen na počátku, při jejím zavádění.

I když se změní jen 10% kódu, tak je stále třeba otestovat 100% funkcí, aby se ověřilo, že nikde nebyla zavlečena chyba. Manuální testy tohle množství testů nemohou zvládnout, pokud se nebudou stále zvyšovat počty testerů a čas na testy či pokrytí aplikace testy nebude klesat. Automatizace řeši tuto situaci tím, že umožňuje nahromadit testovací případy a tak může být stále otestována jak stávající funkcionalita, tak i ta nová. V případě, že je málo času na testy, tak se často obětuje čas na regresní testy, aby se stihla otestovat nová funkcionalita [1].

Dalším důvodem pro zavedení AT je snazší reprodukce chyb. Stroj provádí stále stejnou sadu kroků se stále stejnými očekávánými výsledky a tak se přesně ví, co se provedlo a kde nastala chyba. Pak se ušetří čas na zjištění, co a kde je vlastně špatně. Odpadá nejednoznačnost popisu zadání chyby pro její reprodukci od manuálního testera [2].

Automatizace testů, kde není potřeba zásahu lidského faktoru, výrazně snižuje možnost chybného provedení postupu testování softwaru [24] a tím také například snižuje riziko zadávání chyb, které chybami nejsou.

Zavedením automatizovaných testů se tedy snižuje selhání lidského faktoru, kdy se myslí hlavně únava a nepozornost při testování, kdy může ledacos testerovi uniknout. Tento případ nastává i když je třeba stejný test udělat několikrát na více prostředích. U strojů, kteří provádí testy, tohle nehrozí, protože stroj se nikdy neunaví a je mu v podstatě jedno, kolikrát za sebou a kde provede stejný test[28].

Velkou výhodou AT je jejich rychlost. Automatizace redukuje čas uvedení produktu na trh (time to market) a to umožněním spouštěním testů každý den, celý týden. Protože jakmile jsou všechny testy automatizovány, testování je rychlejší a je možno je spouštět stále dokola [1].

Dalším přínosem a důvodem ke zvážení zavedení automatizovaných testů je usnadnění práce manuálním testerům jak při generování dat, tak provádění nezbytných kroků k dosažení otestování nové funkcionality. Příkladem může být vyplnění desítky položek ve formuláři na několika stránkách, aby se tester dostal na konkrétní stránku a zjistil, zda se změnil název tlačítka či aplikace správně zareagovala na zadané hodnoty na předchozích stránkách [2].
Automatizace umožňuje zvětšit rozsah testů aplikace, protože umožňuje testovat i takové testovací případy, které by nebylo zrovna lehké či reálné provést manuálně.

Myslí se tím například zátěžové testy. Jak se aplikace chová, když provádí stovky uživatelů, stovky operací ve stejnou dobu a zda je toto chování pro zákazníka akceptovatelné [27].

3.4 Nevýhody, mýty a problémy automatizace

Není možné brát automatizaci jako řešení všech problémů testování ve firmě. Obvykle platí, že pokud se firma bude snažit vyřešit chybějící organizaci a vymezení procesu testování zavedením automatizace, jen posune problém do jiné roviny, nicméně se ho nezbaví [2].

Očekávat, že automatizované testy celkově nahradí testy manuální, v dnešní době je chybné. Automatizované testy zatím nemohou nahradit inteligentně smýšlejícího člověka, který na základě toho, co vidí v aplikaci (ať už v rámci prvků v GUI či jejich funkcionality), dokáže odvodit potenciálně velké negativní následky na funkčnost aplikace a také vnímat její použitelnost [9]. Zároveň je fakt, že automatizované testy jen s velkou nepravděpodobností naleznou další chybu, kterou již nenalezli v prvním průchodu. Kontrolují totiž jen to, k čemu byly naprogramovány [29].

Zároveň si manuální tester poradí s nestandardním či nespecifikovaným, nicméně správným chováním, který by automatizovaný test považoval za chybu. Kromě toho jsou automatizované testy sice schopny ve zlomku vteřiny ověřit desítky vstupů na stránce, nicméně už je velmi obtížné vytvořit skripty tak, aby rozeznaly, že velikost písma je pro člověka nečitelná, že obrázek na stránce je ve špatné kvalitě či skutečnost, že jsou prvky na stránce rozházené [2].

Potenciálním rizikem v zavádění automatizace mohou být časté a velké změny ve funkcionalitě. Pokud je do software často přidávána nová funkcionalita, která má dalekosáhlé následky ve stávající funkcionalitě, pak se stane údržba automatizovaných testů vysoce nákladnou. Může být pak až méně efektivní než manuální testování [24].

Některé firmy po zavedení automatizovaných testů očekávají, že se jim počáteční vysoké náklady investované do jejich zavedení ihned vrátí. Realitou však je, že návratnost lze očekávat spíše v řádech měsíců či let, přičemž nelze už vůbec očekávat, že by se náklady vrátily již v průběhu projektu, pro který byla automatizace původně zaváděna [1].

Úplné pokrytí testovacích případů automatizovanými testy je dalším chybným očekáváním. Je to stěží realizovatelný a nepraktický cíl, protože to znamená automatizaci i takových testů, které se automatizovat nevyplatí. Jako příklad může být kontrola zjišťování, zda je prvek na stránce dostatečně viditelný či čitelný. Se současnými technologiemi se to implementuje jen velmi složitě [2].

Posledním problematickým faktorem, který ovlivňuje efektivnost AT jsou též lidé. Záleží totiž na návrhu AT, volbě vhodných technologií, ale také dostatečně kvalifikovaných zaměstnancích, kteří jsou schopni psát kvalitní skripty [30].

3.5 (Ne)vhodné testovací případy k automatizaci

Přesnou specifikaci toho, které testovací případy automatizovat a které ne, nelze jednoznačně nadefinovat. Existují však obecná kritéria, která je dobré uvážit při rozhodování, zda testovací případ pro daný druh testu (ne)automatizovat [2].

Jedním z takových kritérií je četnost modifikace testovacího případu. Pokud je příliš vysoká, nemá smysl takto často měněný test automatizovat, protože náklady na údržbu by byly vysoké a tudíž se to firmě nevyplatí [24].

Vhodným k automatizaci je i testovací případ, jehož četnost provádění testu je vysoká – čím častěji je testovací případ třeba provádět, tím spíše se vyplatí jeho automatizace [31].

Vhodné testovací případy k automatizaci jsou i takové, jejichž provádění je obtížné. Je to případ, kdy pro manuálního testera je zbytečně náročné takový test provádět a stroj by tento test provedl za zlomek vteřiny. Jedná se třeba o ověřování správné výše úroků, což nelze ověřit jinak než výpočtem nebo například operace, kdy je třeba přesné načasování [2].

Zdroj [32] uvádí další testovací případy, které se dají automatizovat. Jsou jimi testy, které běží každé sestavení aplikace a testy, které zahrnují malé a opakující se kroky. O čemž se v podstatě výše zmiňuje i zdroj [31]. Dále jsou to testy, které používají mnoho vstupních dat pro stejné akce. Komplexní a časově náročné testy jsou zde též uváděny jako vhodné testovací případy k automatizaci. Naproti tomu zdroj [29] uvádí, že naopak příliš složité testy je třeba dělat manuálně, protože automatizace by byla časově náročná a protože by stroj nemusel podchytit proces celého testu.

Testovací případy lze automatizovat jen v případě, že je dobře napsaný testovací scénář. To znamená, že všechny kroky scénáře jsou napsány tak, aby bylo jasné, co se má v tom kroce stát. Například krok scénáře je kontrola správného zobrazení nějakého elementu. Ale co znamená správné zobrazení? Jak toto zobrazení kontrolovat? Proti čemu ho kontrolovat? Je za tím nějaký výpočet? Pokud jsou všechny otázky nezodpovězeny, pak je jasné, že automatizace nemůže být provedena [34].

4 Analýza stavu automatizovaných testů ve firmě EmbedIT

Technologie, které se používají při vytváření automatizovaných testů jsou ve firmě zavedeny již několik let, kdy firma se nesnažila nijak mapovat trh s nabízenými nástroji pro automatizaci testů. Nezvažoval se zatím ve větším měřítku nákup jiných nástrojů.

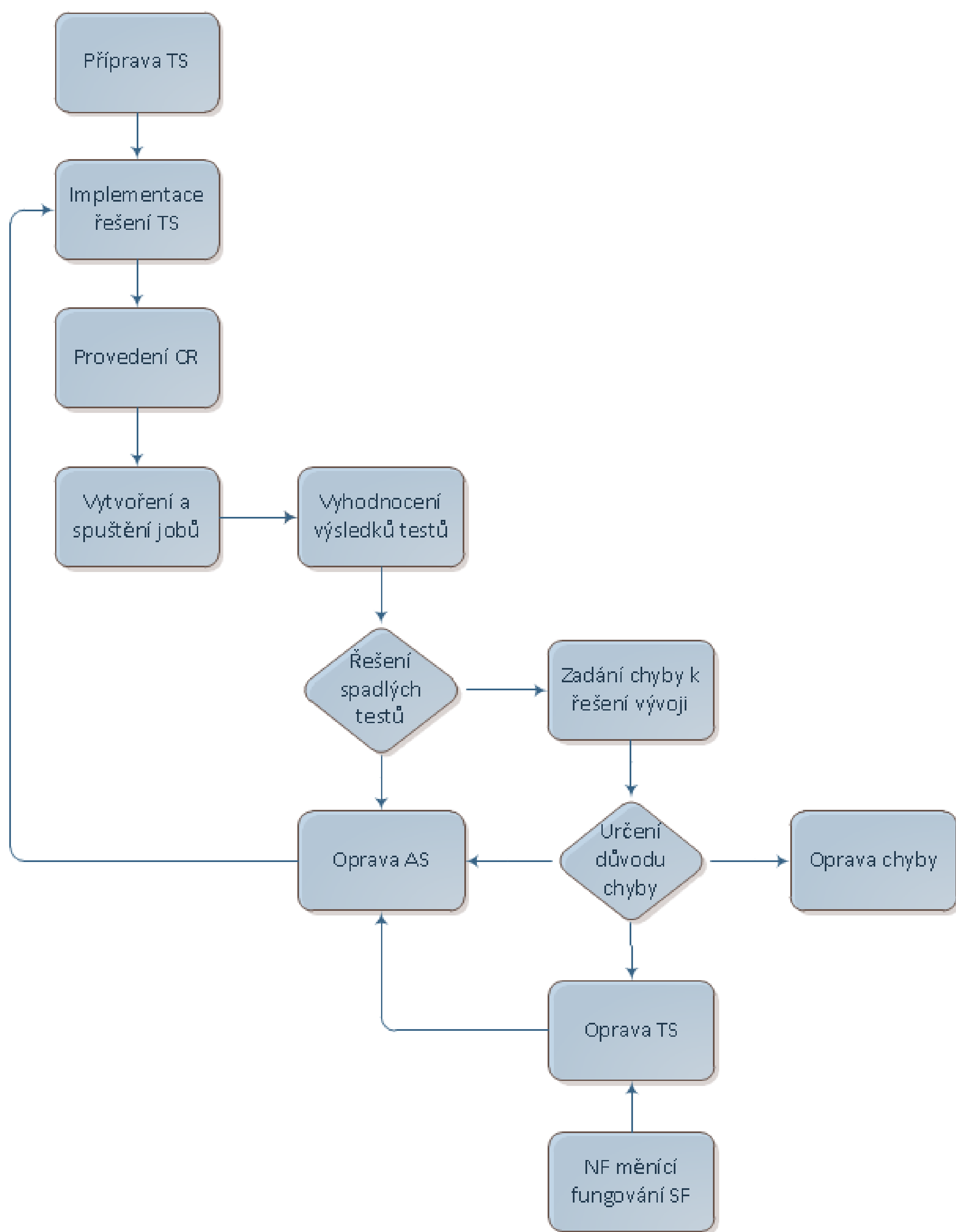
Při zavádění automatizace testů se myslelo hlavně na to, aby byla co nejlevnější, čemuž se podřizovala veškerá řešení.

Firma má automatizované jak regresní, tak smoke testy. Nicméně v této diplomové práci se budu věnovat procesu automatizace regresních testů.

4.1 Proces automatizace testů

Níže je uveden proces automatizace testů tak, jak jdou za sebou jednotlivé podprocesy, které ho dohromady tvoří. Vždy je popsána část procesu a vyplývající problém.

TS je zkratka pro testovací scénáře a AS je zkratkou pro automatizované scénáře. Zkratkou NF se myslí nová funkcionalita a SF stávající funkcionalita.



Obr. 6 Diagram procesu automatizace

4.1.1 Příprava testovacích scénářů

Celý proces začíná přípravou testovacích scénářů. Jednotlivé kroky scénáře jsou automatizovány. Je to tedy ten nejzásadnější proces, protože na něm stojí celá automatizace. Pokud scénáře nejsou správně napsány, nejsou pak ani správně napsány automatizované testy vycházející z těchto scénářů. Takové testy pak ztrácí smysl. Problém je odhalit špatně napsané testy.

Většina testovacích scénářů pro aplikaci Online půjčka vznikla před třemi lety. Jsou psány jak interními zaměstananci, tak brigádníky. Každý napsaný test musel projít revizí provedenou tehdejším test-analytikem. Zda tyto revize probíhaly, není jisté, neprobíhala totiž žádná dohledatelná evidence. Proto není nijak zajištěno, že to, co se v aplikaci testuje, odpovídá tomu, co zákazník chce, aby se testovalo. Takže do teď se nevědělo, zda scénáře pro Online půjčku, obsahují testy pro veškerou, pro zákazníka důležitou, funkcionalitu.

Často se ve scénářích nespecifikují způsoby, jak jsou některé částky vypočteny. Pouze se napíše, že se musí zkontrolovat, že je ta či jiná částka *správně* vypočtena. Co to však znamená není nikde uvedeno a opět se dost často těžko dohledává, co tím vlastně tvůrce myslel a jak to celé funguje. Což je typický příklad testovacích scénářů, které se nemají automatizovat [34].

Jednou z příčin špatně napsaných a neudržovaných testovacích scénářů je však i postoj k regresním testům. Chybí zde motivace provádět regresní testy a opravovat ty, které obsahují nesmysly. V minulosti, kdy se ještě regresní testy neautomatizovaly, byl přístup k regresi ještě horší, vznikala spusta chyb, stále chyběla motivace dělat tuto práci. Když se našla kritická chyba, často se o tom ani mezi testery více nevědělo. Do dnes si testeři plně neuvědomují důležitost testování stávající funkcionality, protože jsou mj. stále vyzdvihovány testy nové funkcionalilty.

Z toho vyplývají problémy:

- Není zjištěno, zda všechny scénáře podchycují veškerou funkcionalitu, kterou chce zákazník, aby se testovala
- Scénáře jsou psány mnohdy nepřesně, neurčitě a dohledávání informací je složité a časově náročné
- Testerům chybí motivace provádět regresní testy a udržovat testovací scénáře aktuální

4.1.2 Implementace řešení automatizace testovacích scénářů

Testovací scénář si vezme technický tester, který ho bude automatizovat. Zde neprobíhá žádný návrh řešení, rovnou se implementuje. Testovací scénář se prochází po jednotlivých krocích, které jsou implementovány.

Pouze pokud se narazí na nějakou nepřesnost, tj. že scénář se odkazuje na některé tlačítko, které v aplikaci není, tak se ptá tvůrců scénáře. Nicméně tito tvůrci

dost často již ve firmě nepracují, takže se složitě dohledává, zda se jedná o chybu či se v rámci vývoje nové funkcionality či úprav toto tlačítko odstranilo.

Takto vyvstává otázka, jak často se již napsané testovací scénáře udržují. Jestli se vůbec v rámci týmů, kteří vyvinuli novou funkcionalitu, šíří informace o tom, že se něco změnilo.

Pokud tedy scénář žádnou takovou nesrovnalost nemá, rovnou je automatizován a víc se neřeší. Techničtí testeři pak často nemají tušení, jak testovaný scénář vlastně funguje, proč se testuje zrovna tohle a jak to souvisí s celým procesem např. založení smlouvy. Některé kroky scénáře jim tak nejsou podezřelé a nepátrají po tom, zda má smysl testovat právě tohle a právě tímto způsobem.

V rámci implementace se musí též řešit příprava testovacích dat či příprava prostředí k tomu, aby nastaly přesné podmínky pro otestování specifického, ale důležitého případu. Z čehož vznikají celkem často chyby, protože některá nastavení nejsou kompletní, či jsou přebita situacemi, které vzniknou na základě nepodchycené výjimky. Následkem jsou pak zbytečně zadávané chyby k řešení vývojem, které chybami nejsou a vedou pak k opravě testovacího a nebo automatizovaného scénáře. Vývojář, který problém řeší a zjišťuje příčinu ztratí tímto čas, který mohl využit k opravě opravdových chyb.

Z toho vyplývají problémy:

- Neprobíhá žádný návrh řešení, který by se případně diskutoval.
- Mezi týmy se nepředávají informace o nových funkcionalitách, které mají dopad na stávající scénáře. Scénáře tak nejsou udržovány.
- Špatná příprava testovacích dat, jejímž důsledkem jsou zbytečně zadávané chyby, které chybami nejsou.
- Techničtí testeři neznají některé zásadní části funkcionality aplikace.

4.1.3 Provedení code review

Po tom, co se implementuje testovací scénář, tzn. že je automatizován, se předají změny v kódu na code review jinému technickému testerovi.

Jeho úkolem je kód projít a ověřit tak, že se dodržely všechny zásady dobře udržovatelného kódu a že implementace odpovídá způsobu, jakým jsou testy ve firmě psány.

Pokud je zde něco, co se testerovi, dělající code review nezdá, přidá poznámku a dokončí tak code review. Technický tester, který implementaci vytvořil, pak dle poznámky reaguje a případně kód upraví.

Po úpravě implementace, dle doporučení kolegy dělající code review, se již nijak nekontroluje a není tudíž ani nijak zaručeno, že doporučení v code review bylo implementováno.

Vyplývající problémy:

- Neověřuje se, zda úpravy v rámci doporučení v code review, proběhly a byly commitnuty.

4.1.4 Joby spouštějící testy na serverech

Automatizované regresní testy se v rámci fáze vývoje, kdy se testuje stávající funkcionalita, testují spouštěním jobu. Není to tedy tak, že by si je tester spouštěl všechny na svém stroji. Testy běží na vzdálených strojích, které se spravuje pomocí nástroje Jenins. Testy běží jeden po druhém, což je nevýhoda oproti tomu, když si je tester pustí u sebe na svém stroji. Může si totiž spustit klidně několik testů najednou.

Tyto joby vytváří technický tester vždy jednou za release, protože i regresní testy probíhají jednou za release. Problém však je, že není přesně určeno, kdo joby vytvoří a tak se na ně dost často čeká. Vytvoření jobů totiž podléhá vytvoření nové větve za předchozí release, aby se všechny opravy commitnuté do trunku týkaly již releasu, na kterém právě probíhají regresní testy. A protože není přesně určené, kdo má vytvářet takové větve, celý proces se prodlužuje.

Joby pro smoke testy jsou vytvářeny již automaticky společně s prostředím. Tento krok byl automatizován, protože vzniká více prostředí v rámci jednoho releasu. Otázkou pak je, proč se nezautomatizoval proces vytváření jobů i pro regresní testy. Spouštění jobů dělá též tester a to v již zmiňovaném nástroji Jenkins.

Po tom, co doběhne job, dojde e-mail informující o této skutečnosti. Nic víc však obsahem e-mailu není, jen že doběhl. Obvykle se statusem *unstable*. Dalším krokem je vyhodnocení výsledku testů.

Vyplývající problémy:

- Testy běží sériově, déle to tak trvá než doběhnou.
- Není jasně určeno, kdo má vytvářet joby a větve.
- Vytváření jobů není automatizované.

4.1.5 Vyhodnocení výsledků testů

Job eviduje všechny testy a *ví* tak, který test spadl a který ne. U každého testu je výpis z běhu testu. Pokud spadne, vypíše se chyba taková, jaká by se vypsala, kdyby se test spustil i v IDE IntelliJ Idea.

Lze zde tedy vidět, který test prošel a který ne i s chybou, na kterou spadl. K některým chybám se i připojí snímek obrazovky, kde chyba nastala. Tato funkcionalita je součástí knihovny *end-to-end-test-support*, kterou vytvořili interní zaměstnanci firmy.

Tester pak retestuje ty testy, které spadly a zjišťuje příčinu. Dle příčiny se pak určí řešení. Zde mohou nastat jen 2 případy, tedy buď je chyba v testu a pak dojde k opravě automatizovaného scénáře a nebo to vypadá na chybu aplikace či prostředí a tak se chyba zadá do systému pro správu chyb, kde se předá k řešení vývojářům.

Při vyhodnocení je relativně těžké určit, proč vlastně test spadl. Ve výpisu nelze poznat, na které stránce test spadl, která metoda se před tím volala. V ideálním případě by tester na první pohled ve výpisu chyby poznal, v čem je chyba. Někdy se tak skutečně stane, není to však ve všech případech. Navíc pro nováčky je těžké se vyznat v typicky javovském výpisu chyby a zjistit z něj příčinu.

Výplývající problémy:

- Těžké určit příčinu toho proč test spadl.
- Až zde se zjišťuje, že je v aplikaci nějaká nová funkcionalita, která ovlivnila běh testu.

4.1.6 Zadané chyby z automatizovaných testů

Vývojář se na zadanou chybu podívá a určí její příčinu. Pokud se jedná skutečně o chybu aplikace, opraví takovou chybu a poté ji dá zpět na retest testerovi. Může se však stát, že se jedná o chybu, která není chybou aplikace, nýbrž chybou testovacího scénáře či automatizovaného scénáře.

Chyba testovacího scénáře může nastat, jestliže k otestování nějaké funkcionality se jde špatným procesem, či se kontroluje něco, co funguje jinak, než je napsáno v testovacím scénáři a podobně.

Pak se taková chyba uzavře s tím, že se o chybu nejedná a tester pak musí opravit testovací scénář. Protože na základě tohoto testovacího scénáře je postaven automatizovaný scénář, tak se musí samozřejmě opravit i ten. Poté se již pokračuje procesem od návrhu řešení změněného scénáře.

Chyba automatizovaného scénáře může například nastat, pokud jsou špatně nastavená testovací data, kdy v důsledku pak dochází k nestandardnímu chování aplikace. V tomto případě se chyba opět uzavře s tím, že se o chybu nejedná a technický tester opravuje automatizovaný test tak, aby k takové chybě již nedošlo.

Vyplývající problémy:

- Opět stav testovacích scénářů může vést k chybě, protože nedávají smysl a k otestování nějaké funkcionality se používá špatný proces.
- Špatně nastavená testovací data, což už je zjištěno z předchozích procesů, důsledkem čehož vývojář stráví čas navíc zjišťováním příčiny u zadané chyby, která, jak se ukáže, chybou není.

4.1.7 Přidání nové funkcionality

Pokud se jedná o úplně novou funkcionalitu, začne se od procesu příprava testovacích scénářů. Dost často však nastávají případy, kdy se pouze změní již stávající funkcionalita, která má menší či větší dopad na stávající funkcionalitu a testovací scénáře.

V takovém případě se musí zjistit, kterých testovacích případů se to týká. Tyto scénáře jsou pak opraveny a musí být pak opraveny i automatizované scénáře, kdy se zase pokračuje od návrhu řešení této úravy a její zakomponování do současného řešení.

Reálně se však tohle děje až ve chvíli, kdy scénář skončí chybou, kvůli přidané nové funkcionalitě. Ačkoliv to není pravidlem. Resp. tohle by ani nemělo nastávat. O změnách by se mělo vědět dopředu, aby se mohly opravit scénáře a automatizované testy a ty pak zbytečně nepadaly při jejich spuštění v rámci regresu.

Dochází totiž tak k tomu, jak už bylo zmíněno výše, že se ztrácí čas navíc, kdy tester zjišťuje, co se vlastně změnilo a kdo tuto novou funkcionalitu vlastně vyvíjel. Takže mu pak vyhodnocení výsledků trvá o dost déle.

Vyplývající problémy:

- Špatné šíření informací o tom, co se kde v aplikaci změnilo, napříč týmy.
- Špatná údržba testovacích scénářů.
- Tester ztrácí čas navíc zjišťováním toho, co se vlastně vše změnilo a kdo tuto NF vyvíjel.

4.2 Proces automatizace v procesu vývoje softwaru ve firmě

Ve firmě probíhají 3 procesy, pričemž 2 z nich probíhají téměř nezávisle na sobě. Vývoj nové funkčnosti (NF) nového release probíhá ve 2. sprintech. Má též svoje procesy, jako jsou:

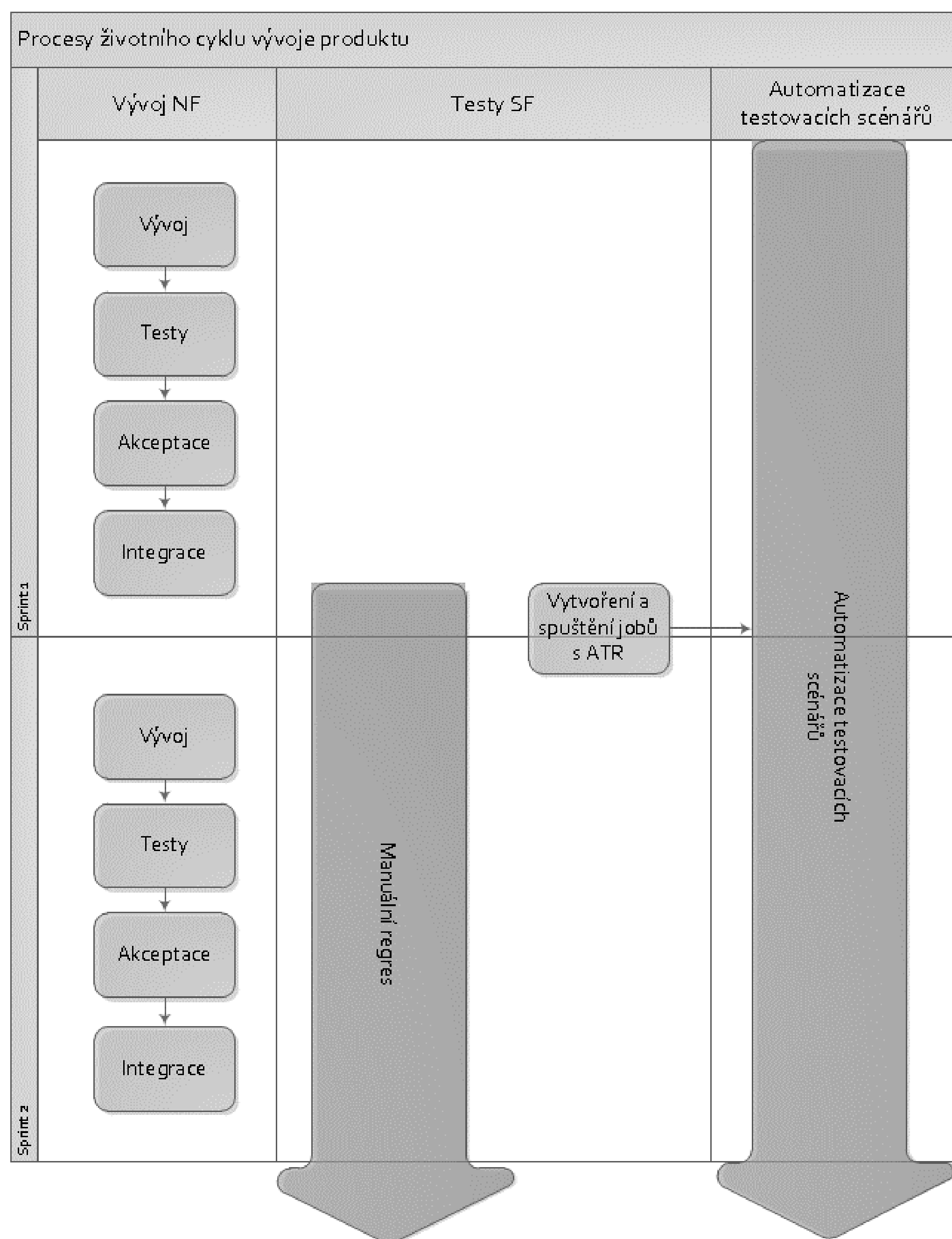
- Vývoj – vývoj NF
- Testy – testy NF
- Akceptace – akceptace NF se zákazníkem, zda odpovídá zadání
- Integrace – integrace NF do stávající SF již na nových prostředích

Zhruba na konci 1. sprintu se začnou dělat regresní testy stávající funkcionality (SF) z předchozího release, který však ještě nebyl nasazen do produkce. Manuální regres probíhá souběžně a nezávisle na vývoji NF nového releasu. Probíhá napříč sprinty a má svůj vlastní konečný termín. V rámci manuálního regresu samozřejmě probíhá zadávání chyb a jejich řešení.

Zároveň se ode dne, kdy se začne manuálně testovat regres, spustí na stejných prostředích automatizované testy pomocí spuštění jobu. Zde se vlastně navazuje na proces automatizace uvedený výše a to od podprocesu vytvoření a spouštění jobů s automatizovanými regresními testy (ATR). Pak je již průběh totožný s výše zmíněným procesem automatizace – provede se vyhodnocení spadlých testů, kdy se pak zadá buď chyba k řešení vývoji či se opraví automatizovaný scénář.

Celkový proces automatizace však probíhá nezávisle na vývoji nové funkcionality či testování stávající funkcionality. Berou se totiž další a další scénáře, které se dříve prováděly manuálně a automatizují se přesně dle kroků uvedených v procesu.

Z výše uvedeného plyne, že za celou vyhrazenou dobu pro regresní testy a opravy chyb z těchto testů se pouští automatizované testy jen jednou a to na jeho začátku. To, že některé opravy mohou mít velký vliv na zbytek funkcionality se nebere dostatečně v potaz. Buď se to nejnutnější otestuje manuálně, což zbytečně vytěžuje další testery, a nebo se větší oprava, pokud to jde (chyba je nižší či střední priority nebo severity), dá na prostředí, která již jsou pro nový release. Takže se oprava vlastně bude týkat až dalšího releasu.



Obr. 7 Procesy životního cyklu softwarového produktu firmy

4.3 Srovnání procesu automatizace ve firmě s těmi obecnými

Aby se daly procesy navzájem srovnat, je nutné namapovat jednotlivé podprocesy z firmy na obecné procesy automatizace uvedené výše. Po spojení všech procesů uvedených v kapitole Proces automatizovaného testování se vytvoří jeden proces, obsahující všechny důležité kroky. Případně se podobné kroky spojí v jeden, aby nedocházelo k duplikacím.

Kroky jako zjištění, zda je aplikace automatizovatelná či výběr a vývoj správného frameworku vynechám, protože automatizace je ve firmě již několik let zavedená a tyto kroky již dávno proběhly.

4.3.1 Testovací plán

Ve firmě se za testovací plán dá považovat dokument, který je průběžně aktualizován a obsahuje některé zmiňované náležitosti, které by testovací plán měl mít. Jeho podoba je taková, že jsou v něm obsaženy všechny scénáře regresních testů a doba, od a do které se mají automatizovat. Zároveň je každý scénář přiřazen nějakému týmu, který by měl automatizaci vykonat. Tudíž je stanoven testovací tým.

Scénáře, které testují aplikaci Online půjčka jsou jasně vyznačeny. Scénáře zároveň určují i rozsah testů aplikace.

K němu se váže další dokument, kde se evidují také všechny regresní scénáře, ale kromě toho i jejich statusy a konkrétní osoby, které scénář automatizují.

4.3.2 Kontrolní dokument

Kontrolní dokument chybí. Není žádné místo, kde by se evidovaly všechny změny, které v testovacím plánu proběhly a nebo teprve proběhnou. Často pak vznikají nejasnosti ohledně scénářů – například proč není přidaný nový scénář k automatizaci a podobně. Též se o některých testovacích scénářích rozhodne, že se již nebudou více testovat a tak nemá smysl je automatizovat.

Zároveň si různé scénáře přehazují týmy mezi sebou a tuto skutečnost je také třeba evidovat. Protože pak opět dochází opět k nejasnostem ohledně toho, který tým testuje které scénáře.

4.3.3 Definování rozsahu automatizace testovacích případů

Definice rozsahu testovacích případů proběhla již v rámci psaní scénářů pro jejich manuální provádění. Určování, které z nich se budou automatizovat probíhá pokaždé, když si technický tester otevře testovací scénář. Měl by být schopen rozhodnout, zda je či není vhodný k automatizaci.

Ty scénáře, které se nakonec automatizovat nebudou, buď obsahují kontrolu externích dokumentů v různých formátech a nebo jsou špatně napsané. Problém je, že špatně napsané scénáře nejsou vždy rozeznány, případně se více neřeší. Neřeší se také to, že něco naopak otestovat chybí. To už je ale řešeno výše v rámci jednotlivých podprocesů automatizace testování.

4.3.4 Návrh a vývoj testovacích scriptů

Návrh automatizace testů v podstatě neprobíhá. Scripty se píší podle kroků ve scénáři a kód se píše dle šablon a návrhového vzoru page object, kterému se věnuje příslušná kapitola níže.

Způsob jak test automatizovat si řeší každý technický tester sám, jen vyjímečně se probírá s kolegy. Ostatně by většinu špatně navržených automatizací scénářů mělo odchytnout prováděné code review.

4.3.5 Spouštění testů

Ať už si je tester spouští na svém stroji či se spouští na vzdálených strojích pomocí jobů, jejich výhoda je, že se dají spouštěť téměř kdykoliv a jakkoliv často. Ve firmě se spouštějí automatizované regresní testy pouze jedinkrát v rámci regresních testů. Nespouští se zde například po tom, co se opraví chyby, jak zmíněno v literatuře v rámci výhod automatizovaných testů. Jejich spuštění a běh by měl být levný, resp. levnější, než kdyby je vykonávali testeři manuálně. Tudíž je otázkou, proč se tak ve firmě neděje. Není využit plný potenciál automatizace.

Ve firmě probíhá spouštění testů pomocí jobů.

4.3.6 Analýza výsledků testů

Ve firmě probíhá fáze analýza výsledků testů, což je jasné už z výše uvedeného procesu automatizace. Tyto výsledky se pak zaznamenávají do speciálního dokumentu, kde se o testech scénářů vede evidence.

4.3.7 Údržba testů

Také na základě analýzy výsledků testů se provádí údržba testů. Ve firmě se provádí i po tom, co se zjistí příčina chyby testu, která není chybou aplikace a ani chybou testovacího scénáře.

V rámci údržby se však myslí i zapracování nové funkcionality jak do testovacích scénářů, tak to testovacích scriptů. K tomu je však třeba šíření informace mezi týmy o nové funkcionalitě zasahující významnou měrou do stávající funkcionality.

Údržba testů ve firmě probíhá každý release, většinou po tom, co doběhnou automatizované regresní testy. Případně se již v rámci vývoje nové funkcionality pro nový release ví, že se v regresních testech právě pro nový release bude chovat něco jinak. A tak proběhne údržba scriptů i mimo regresní testy.

4.4 Souhrn zjištěných problémů v procesu automatizace testů

Největším problémem je kvalita testovacích scénářů. Což je v podstatě osnova, podle které se automatizuje testovací případ. Není jisté, zda všechny scénáře pokrývají veškerou funkcionalitu, která je pro zákazníka důležitá.

Obsah scénářů je psán často velmi nepřesně, kdy se chce po testerovi, aby něco otestoval, ale není napsáno, jak to má otestovat. Takže pak dohledávání informací o tom, jak se mají některé části testu provést, je časově náročné a bývá i složité.

Když je vyvinuta nová funkcionalita, která má dopad na tu stávající, tedy postihne další testovací scénáře, neví se o tom. Nepředají se o tom informace mezi týmy a zjistí se to často až v případě, že na to spadne test. Zjišťování proč test spadl, které následuje, je opět náročné, protože chyba není vždy úplně jasná. Mohou být i

případy, kdy není jisté, zda se jedná o chybu, či o změnu ve stávající funkcionalitě. Problémem tedy je, že scénáře nejsou udržovány.

Protože techničtí testeři, kteří provádí automatizaci testovacích scénářů mnohdy neví, jak některé součásti aplikace fungují, snadno přehlédnou chyby v testovacích scénářích. A protože, jak je uvedeno výše, scénáře nejsou udržovány, jen to přispívá k horší kvalitě výsledných automatizovaných scénářů. Testeři, kteří automatizují scénáře, jsou často odkázáni jen na to, co je ve scénáři, proto musí být co nejkvalitněji provedeny.

Proces automatizace nezahrnuje žádný podproces návrhu řešení implementace testovacího scénáře. U složitějších testů, které vyžadují přípravu testovacích dat, pak nastávají různé případy, se kterými na začátku tester nepočítal.

Příprava testovacích dat je klíčová pro správně proběhnutí testovacího scénáře. Pokud se odbyde, dochází k nestabilnímu průběhu automatizovaného testu. Též dochází k chybám, které nejsou příčinou chyby v aplikaci, nýbrž právě v chybné přípravě testovacích dat.

Špatně nastavená testovací data mají dopad až na vývojáře, kteří teprve zjišťují, že právě v nich je chyba. Výsledkem je neefektivní využití pracovní síly jen proto, že někdo neví, jak přesně má proces probíhat, jaká data jsou k tomu potřeba a jakým způsobem je nastavit.

Poté, co je scénář automatizovaný, vytvoří se code review a připíše se k němu pracovník, který se podívá přidaný test a okomentuje toto řešení, případně doporučí co a proč udělat jinak. Poté se code review vrátí zadavateli a tím proces končí. Již se neprovádí nějaká zpětná kontrola, zda bylo opraveno to, co bylo doporučeno.

Tento laxní přístup je též u tvorby jobů a větví. Není jasně stanoveno, kdo je má kdy vytvářet. Tudíž se proces tvorby jobů a jejich spuštění a tím pádem i vyhodnocení značně prodlužuje. Prodlužuje se čekáním na *někoho*, kdo tyto joby a větve vytvoří.

Joby, které spouští testy, by se mohly vytvářet automatizovaně hned po tom, co vznikne nové prostředí. Dosud se však takto nestalo.

Vytvořené joby spouští na vzdálených strojích jeden test po druhém. Tester sám si však může na svém stroji pouštět i několik testů najednou. Pokud by to stejné zvládl job, mělo by to velký efekt na čas, který si doběhnutí jobu vezme. Otázkou je, zda by nedocházelo k větší chybovosti, protože by se mohlo stát, že by se hůře ovládaly prvky na stránce. Mohlo by pak docházet ke ztrátě focusu na jednotlivá okna, nenacházely by se prvky a podobně.

Chybí nějaký dokument, který by obsahoval informace o všech změnách, které probíhají v testovacím plánu.

Rozsah automatizace testů je dán jen těmi, kdo automatizují testy. Ne všichni mají však dost znalostí určit, který scénář automatizovat a který ne. Zároveň ne všichni techničtí testeři dokáží určit, co v testech aplikace chybí, tj. co vše se netestuje a mělo by.

4.5 Testy Online půjčky

Pro webovou aplikaci Online půjčka, je nyní vytvořených 12 testů. Tyto testy se spouštějí zatím jednou za release v rámci regresních testů. Testy by měly ověřovat důležitou funkcionalitu.

Plánem do budoucna je, že se tyto testy budou pouštět několikrát za release. Důvodem jsou opravy chyb zjištěných v regresních testech, které mohou rozbít to, co již bylo otestováno.

Samotné webové aplikaci se více věnuje kapitola: Automatizované testování finanční webové aplikace a její specifika, kde je vysvětleno, o jakou aplikaci jde, čím se vyznačuje a co by na ní mělo být testováno.

4.5.1 Výčet testů

Každý test by měl testovat něco jiného a měl by být nezávislý na ostatních testech. V této podkapitole je třeba definovat, co vlastně jednotlivé testy testují a mít tudíž lepší představu o tom, do jaké míry je aplikace testována.

- **preliminaryCalculationTest** – kontrola, že jsou prvky s posuvníkem počtu splátek a výše úvěru na první stránce aplikace a lze jimi hýbat. Že se mění jejich hodnoty v závislosti na tom, co je zvoleno na prvním či druhém posuvníku. Kontrola, že posuvníky mají ty správné hodnoty. Zde se testuje maximální částka při zvolení minimálního počtu splátek. Na konci je kontrola výpočtu RPSN. Je to také jediný test týkající se první stránky Online půjčky.

- **secondPageCalculationTest** – nejprve se navolí počet splátek a výše úvěru na první stránce aplikace a pak se přejde na druhou stránku. Nijak se však dále s těmito hodnotami dále nepracuje. Navíc se navolí jiný počet splátek, než je zadáno v proměnné v testu, tedy to nastavení počtu splátek nefunguje. Dále se nevyužije to, že se navolí hodnoty již na prní stránce – nijak se nekontroluje, zda hodnoty zvolené na první stránce si aplikace *pamatuje* a jsou vidět i na stránce druhé. Kromě kontroly toho stejného co v předchozím testu, jen se to testuje na druhé stránce aplikace, se ještě kontroluje, že lze zatrhnout všechny checkboxy a že se po této akci provede přičtení/odečtení hodnoty k vypočtené výši splátky. Nicméně momentálně test spadne na tom, že nemůže najít očekávaný checkbox představující poplatek.Zde se testuje i výše vypočteného RPSN, protože je jednou z hodnot uvedenou na stránce.

- **minMonthlyInstalmentTest** – zde se na první stránce navolí výše úvěru a počet splátek, nicméně na druhé stránce se navolí znovu. Tudíž jsou zde zbytečné a dochází tak k větší časově prodlevě, než se dojde k tomu, co je třeba otestovat Testuje se zde zvolení nejnižšího počtu splátek a jeho kontrola oproti očekávanému minimálního počtu splátek. Měly by se tu zatrhnout všechny checkboxy, nicméně to nefunguje. Zatrhnou se pouze dva.

Nekontroluje se zde, že se přičetly ke splátce. Pak se zvolí specifické pojištění, jako důkaz, že je to možné a aplikace nespadne. Zde má opět smysl počítat RPSN, ale neděje se tak.

- **maxMonthlyInstalmentTest** – navolí se počet splátek a její výše a neproběhne žádná kontrola zadaných hodnot po tom, co se přejde na další stránku. V testech se používá stále stejná metoda *chooseAndSubmitProduct()*, která vrací instanci druhé stránky, na kterou se má přejít. Zde se testuje výše splátky při nejvyšším úvěru a nejnižším počtu splátek. Na konci se ještě kontroluje správný výpočet RPSN, který nyní špatně počítá výslednou hodnotu.

- **smsCodeVerificationTest** – zde už se vyplní celá žádost až k poslední stránce a to je rekapitulace žádosti. Zde na stránce se vyplňuje SMS kód. Dvakrát se klikne na tlačítko a tím se ověří, že pokusy pro zaslání SMS jsou vypotřebované. Neprobíhá kontrola, zda SMS kód zaslaný do databáze souhlasí s tím, který požaduje aplikace.

- **smsCodeVerificationTestWrongCaptchaTest** – opět se vyplní všechny stránky aplikace a na poslední stránce s rekapitulací, se ještě před vyplněním SMS kódu vyplňuje captcha kód. Zadá se špatný kód a kontroluje se, zda na to aplikace patřičně zareagovala.

- **alternativesTest** – vyplní se žádost opět až ke stránce s rekapitulací. Vyplní se kód SMS i captcha a posoudí se žádost. Vybere se některý z alternativních produktů a pokud se jedná o českou lokalizaci a pak se zaklikává možnost odložení splátek a odtrhává se možnost nastavení elektronického výpisu. Pak kontrola na existenci textu ohledně nápovědy k pojištění. Test spadne na neočekávaný výsledek posouzení smlouvy – smlouva se totiž zamítne. Také je nastaveno, že klient žádá o úvěr nižší, než má mzdu, tudíž mu není nic nabídnuto a smlouva se opět zamítne. Problém tedy je v zadávaných datech do formuláře, dle kterých se žádost posoudí.

- **rejectAlternativeTest** – projde se žádost až do konce, odešle se k posouzení, měly by se nabídnout alternativy, nicméně se klikne na odmítnutí nabídek. Dojde k přesměrování na stránku s textem o ukončení žádosti. Kontroluje se vypsaný text na stránce. Zde jsou stejné problémy jak v předchozím testu (alternativeTest).

- **chooseAndSubmitAlternativeTest** – projde se žádostí až k výsledku, kdy se nabízejí alternativy. Opět se kontroluje pojištění a více informací o něm, opět se zaškrtávají checkboxy s možnostmi doplňkových služeb k úvěru. Naproti tomu se zde nezkouší volit různé způsoby zaslání dokumentace, jak je ve scénáři. Poté se zvolí jedna z alternativ a souhlasí se s ní, tedy pošle se žádost opět k posouzení. Kontroluje se text došlého emailu,

ze kterého se vybere specifická část, která se připojí k nově zadané URL adrese do prohlížeče a dojde k ověření žádosti.

- **insuranceChangeTest** – opět se projde žádostí až k rekapitulační stránce, kde se očekává zobrazení boxu s informací o tom, že klient nesplňuje podmínky pro zvolené pojištění. Zde se musí kontrolovat opět nápověda k více informacím o pojištění. V testu kódu se pak zjišťuje, teprve teď, zda je v selectboxu vůbec nějaké vypsané pojištění. Pokud není, test skončí – je tam totiž použit assertTrue. Opět se zjišťuje text pojištění, zda je správně, ale to už odpovídá zadání scénáře. Poté se vyplní zbytek žádosti a kontroluje se výše zmíněný box s informací o nesplnění podmínek. Poté se zaklikne nabízené pojištění, které je akceptovatelné, ale už se nikde nekontroluje, že box zmizel a že došlo k znovunačtení stránky s informací o nově zvoleném pojištění. Změna by se měla projevit do rekapitulace.

- **loanReject** – projde se žádostí a kontroluje se, že výsledek posouzení bylo, že půjčka byla zamítnuta. Dle testu by se měl vypsat konkrétní důvod, proč byla půjčka zamítnuta. Otázkou je, zda je tato funkčnost ještě aktuální. V kódu testu se nic takového neověřuje, jen se hledá specifický text na stránce s výsledkem.

- **sendEmailOfApprovalTest** – žádost se projde až do konce, kdy se očekává, že bude schválena. Měl by dojít email, tedy se kontroluje, nicméně stejná věc už se kontroluje jinde.

4.5.2 Funkcionallita aplikace, která není obsahem testovacích scénářů

V testech aplikace Online půjčky se vůbec netestuje některá funkcionalita, na kterou zadává chyby zákazník. Je tedy třeba její testy dodělat.

Aplikace má několik selectboxů. Jednotlivé hodnoty v některých selectboxech, ovlivňují chování aplikace. Můžou mít i vliv na výsledek posouzení úvěru. Těmito selectboxy jsou:

- Způsob zaslání dokumentace a výplaty úvěru

- Zdroj příjmů

V rámci žádosti o půjčku a pohybu v jednotlivých krocích žádosti, se dá vracet zpět do předchozích kroků. Pokud se v nich něco změní, musí si to aplikace pamatovat. V horní části stránky je stále zobrazena výše úvěru a počet splátek. Kromě těchto dvou prvků je zde též odkaz změnit, který klienta přesměruje na stránku s výběrem produktu, tj. s výší úvěru, počtem splátek, pojištěním, zasláním dokumentace, atd.

Další částí, která se netestuje, je stránka s rekapitulací. Zde jsou zobrazené všechny hodnoty, zadané v průběhu žádosti o půjčku. Všechny změny, které se provedou v rámci vrácení se na předchozí kroky žádosti, se musí projevit v rekapitulaci.

Na úvodní stránce aplikace je spousta prvků, které se též netestují. Ať už je to text či další tlačítka, nejsou obsahem testů.

Popsanou netestovanou funkcionalitu je třeba zahrnout do testovacích scénářů a implementovat je.

Následující tabulka shrnuje zjištěné problémy v testech aplikace Online půjčka. V posledním sloupci je již navrhovaná změna, která je více rozebrána v kapitole Návrh a implementace řešení.

5 Použité technologie a nástroje pro implementaci řešení automatizace ve firmě Embedit

Obecně platí, že je důležitou úlohou vybrat včas a správný nástroj pro automatizaci. Stojí na tom totiž celková úspěšnost automatizace.

Jedním z kritérií, které hrají roli při výběru nástroje, je plná podpora technologie. Nástroj musí být plně kompatibilní s použitou verzí a tudíž je zaručeno, že ji podporuje v nutném rozsahu. Ačkoliv nejde jen o technologii, ale i to, jak je využita. Nástroj musí být schopen pracovat se všemi elementy, které aplikace má. Je proto důležité před pořízením nástroje provést technologické testy na vzorku aplikace, která má být automatizována. Nestane se pak situace, že specifický element, což typicky bývá prvek ovládající uživatelské rozhraní, nemůže být automatizován.

Dalším kritériem je samozřejmě cena. Cenu je třeba chápat jako dlouhodobou investici, kdy kromě samotné ceny nástroje je třeba počítat i cenu případných rozšiřujících modulů.

Posledním stěžejním kritériem je dostupnost podpory a její šíře od výrobce nástroje.

Kromě těchto kritérií je však třeba uvážit i další faktory, které do rozhodování o výběru nástroje vstupují. Je jím například to, zda nástroj umožňuje načítat data z externích souborů, volat jiné programy či konkrétní funkce. Důležité je i zda je možná distribuce provádění testů na více počítačů najednou či analýza výsledků testu z exportů. Analýzou výsledků se myslí například pomocí vizualizace v grafech či porovnání s předchozími běhy testů.

Tyto další faktory si stanovuje firma sama. Každá má trochu jiné potřeby, specifičtější požadavky na nástroj. Tato kapitola je však spíše obecný pohled na to, co se na trhu nabízí, aniž by byly více brány v potaz požadavky firmy Embedit

Automatizované testy jsou psány v jazyce Java. K těmto automatizovaným testům se přistupuje jako k unit testům, proto se používá framework JUnit. Implementace probíhá ve vývojovém prostředí IntelliJ Idea ve verzi 14.1.1. Pro testování se používá nástroj Selenium WebDriver ve verzi 2.34.0.

V jazyce Java je tedy implementován kód. Součástí kódu je zvolená sada testů. Pomocí JUnit se provádí kompilace, jejich spuštění a vyhodnocení testů, na což jsou speciální metody, které budou rozebrány níže.

Selenium WebDriver pak spustí prohlížeč, zadá do něj deklarovanou URL adresu a vyhledává požadované elementy na stránce.

Kromě těchto nástrojů se používá Spring pro řešení připojení k databázi a dále pak Nexus a Maven. Nicméně těmto nástrojům v rámci tématu diplomové práce se nebude dále věnováno. Kromě těchto nástrojů je ještě třeba zmínit Jenkins, na kterém je možné spouštět joby, které spustí jednotlivé testy na vzdálených strojích.

5.1.1 Junit

Jak již bylo zmíněno, JUnit je jednoduchý framework pro tzv. jednotkové testy, které se dají opakovaně spouštět. JUnit je instance xUnit architektury [36].

Všechny automatizované testy jsou tedy jednotkové testy, využívající knihovny JUnit. Testuje se, zda nastal očekávaný stav kontrolované hodnoty či elementu.

Testovací třídy lze třídit do kategorií, např. dle aplikací, pro které jsou testy psány. Speciální runner pak spouští jen takové testy, které náleží do konkrétní kategorie dle anotace *@Category* [37].
Nad testovací metodu, která má anotaci *@Test* se tedy ještě píše *@Category*, určující, kam patří. Testy se pak ve firmě spouštějí pomocí jobu, kterému se nastavuje právě tato kategorie, ze které má brát testy označené konkrétní kategorií pomocí výše zmíněné anotace. K tomu je ale třeba ještě dalších anotací [38]:

- @RunWith – JUnit vyvolá třídu, která je anotována. Případně kategorii tříd
- @Suite.SuiteClasses – suita tříd, které mají být spuštěny s anotací @RunWith
- @Categories.IncludeCategory – určená konkrétní kategorie tříd s testy

V kódu to pak vypadá následovně:

```
@RunWith(Categories.class)
@Suite.SuiteClasses(ILoanTest.class)
@Categories.IncludeCategory(IApplicationTest.class)
public class IApplicationTestSuite {}
```

Výčet tříd zastupující jednotlivé IAplikace je v *@Suite.SuiteClasses*, jak je vidět v kódu.

K určení testovacích metod, tj. že se jedná o unit test, se používá anotace *@Test*. Pro spuštění takové metody, JUnit nejprve vytvoří novou instanci třídy a pak zavolá zmiňovanou testovací metodu. Pokud v průběhu testu nastane jakákoliv výjimka, JUnit vyhodnotí test jako neúspěšný, tj. skončí s chybou [39].

Test skončí také chybou v případě, že dojde k vyjímce typu dělení nulou, odkazování se na nulovou pozici a podobně. Zároveň ale skončí chybou, když se použije jedna z metod k ověřování aktuální hodnoty či elementu k očekávané.

V automatizovaných testech firmy Embedit se počítá s oběma případy. Ačkoliv každý z případů je řešen, odchytáván jinak.

K ověřování hodnoty předpokládané s hodnotou aktuální se používají metody z knihovny JUnit. Jsou to:

- Metody z třídy Assert
- Metody z třídy ErrorCollector

Rozdíl mezi nimi je hlavně takový, že pokud nevyjde předpoklad v *assert*, tak test spadne a část za *assert* už se neprovede. Pokud se však použije *collector*, provede se část i za ním. Chyby se vypíšou až na konci testu, pokud jsou. Níž jsou uvedeny klíčové části kódu reálného testu:

```
@Category(IApplicationTest.class)
@Test
public void PremiumInsuranceTest()throws Exception{

    //část kódu 1

Assert.assertTrue("There is no premium insurance in a select box", de-
tailCalcPage.isInsurancePresentive(getResource("ILOAN_PREMIUM_IN-
SURANCE")));

    //část kódu 2

collector.checkThat("Wrong specific detail of insurance",

detailCalcPage.getInsuranceInfo(),containsString("Premium - (BP)"));

    //část kódu 3

}
```

Jak lze vidět, nejprve je anotace kategorie, do které test patří. Dále je anotace, že se jedná o testovací metodu. Pak následuje název metody s odkazem na výjimky z třídy *Exception*, do které spadají chyby programátora jako dělení nulou a podobně.

Část kódu 1 označuje blok kódu, který se má stát před asercí. V aserci označené jako *Assert.assertTrue* se předpokládá, že je pravdivý předpoklad, že element Insurance, tedy prvek označující pojištění, je přítomný a tudíž je na něj možné kliknout. Pokud by tam takový prvek nebyl, test nemůže pokračovat, protože se celý točí kolem tohoto prvku. Zde se tudíž hodí použít assert.

Po části kódu 2 následuje *collector*, kdy se ověřuje, že konkrétní element obsahuje specifický text. To běh testu nijak nenaruší, pokud tento text nebude obsahovat, nicméně test skončí chybou, pokud by obsahoval text jiný. Výsledek, zda obsahuje či neobsahuje text, se objeví až po doběhnutí testu, tj část kódu 3 proběhne, i kdyby *collector.checkThat* skončil chybou.

Pokud je třeba určité nastavení před každým testem k tomu, aby bylo test vůbec možné provést, označí se tyto metody s operacemi anotací *@Before*. Pak je zajištěno, že před spuštění každého testu proběhne nejprve metoda takto označena. Nastavuje se takto databáze na testovací mód, který je potřeba pro zadávání některých hodnot, protože se dají pak předvídat. Více o tom níže.

Po provedení testu je pak třeba znova nastavit databázi zpět do původního netestového módu. To se provede metodou, která je označená anotací *@After* a je umístěna na konci kódu třídy testů.

5.1.2 Selenium Webdriver

Selenium Webdriver, nebo také Selenium 2.0, se používá na ovládání webového prohlížeče. Selenium 1 je bráno jako Selenium RC, nebo-li Remote Control. Dá se pak říct, že Selenium 1.0 + WebDriver = Selenium 2.0. WebDriver se vyznačuje integrací API. Je navržen tak, aby poskytl jednodušší programovací rozhraní a vyřešil tak některé omezení Selenia RC [40].

Nabízí lepší podporu dynamické webové stránky, kdy se prvky na stránce mění, aniž by došlo ke jejímu znovunačtení. Navíc přidává další funkcionalitu, která nebyla v Seleniu 1.x, jako nahrávání/stažení souboru či pop-up okna. Nyní Selenium 1 není podporaváno a je tudíž zastaralé [41].

Oproti Seleniu RC (Selenium 1.0) provádí Selenium 2.0 přímé volání internetového prohlížeče a dělá to pomocí nativní podpory pro jeho automatizaci [41].

Jak jsou tato přímá volání vytvořena, jak fungují, závisí na prohlížeči, který je právě používán. Ve firmě embedit se defaultně pro všechny automatizované testy používá Mozzila Firefox.

```
public WebElement isElementPresent(By by) {
    try {
        WebElement e = this.webDriver.findElement(by);
        return e;
    } catch (NoSuchElementException var3) {
        return null;
    }
}
```

Výše je zobrazen kód funkce, která vrací instanci třídy WebElement. Zavoláním *findElement()* pomocí instance třídy WebDriver, najde element a vrací ho jako instanci třídy WebElement. Pokud takový element není na stránce nalezen, vyhodí se výjimka. Uvedený kód lze brát jako jednoduchou ukázku použití WebDriveru v praxi.

5.1.3 Maven

Maven je nástroj, který slouží pro řízení a buildění projektů psaných v programovacím jazyce Java a automatizaci tohoto procesu. Maven přináší tři životní cykly, korespondující se třemi hlavními činnostmi, které provádí Maven. Jsou to [42]:

- Sestavení projektu ze zdroje
- Čištění projektových souborů vytvořených sestavením
- Generování webové stránky projektu

Projekt se v Mavenu popíše pomocí Project Object Model, dále již jen POM, který ho popisuje nejen z pohledu závislostí zdrojového kódu, ale i závislostíí na externích knihovnách, procesu buildění a podobně [42].

POM je v podstatě projekt reprezentován jako XML soubor s názvem pom.xml. Obsahuje všechny potřebné informace o projektu, ale také konfigurace pluginů používané v průběhu procesu sestavení [43]. V přílohách je uvedena část pom.xml z projektu automatizovaných testů.
Další částí v xml souboru popisující projekt, jsou moduly. V přílohách jdou vidět celkem 4 moduly. Moduly lze brát jako projekty, které jsou zařazeny do zmíněného pom.xml a jsou spouštěny jako skupina. Jsou v identifikátoru <modules>. Zde na pořadí modulů nezáleží.

Důležitou částí je část mezi identifikátory *<dependencies>* a *</dependencies>.* Je to seznam závislostí. Většina projektů závisí na ostatních do té míry, že aby mohly být vůbec sestaveny a pracovaly správně, musí mít tyto závislosti k dispozici. Maven tento seznam závilostí přečte a automaticky připojí jejich požadované knihovny. Spojí všechny části dohromady a udělá z ní soubor, který lze spustit. Pomocí vytvořených profilů se dá nastavit sestavení pro různá prostředí [42].

V ukázce závislostí je uvedena knihovna *end-to-end-test-support*, kterou si vytvořili zaměstnanci firmy.

5.1.4 Page Object Pattern

Automatizované testy firmy EmbedIT používají Page Object. Tento návrhový vzor se obecně hojně používá v automatizaci testování, protože usnadňuje údržbu kódu a snižuje jeho duplicitu. Page Object je objektově orientovaná třída, která slouží jako rozhraní pro automatizované testy. Testy pak používají metody a elementy stránky, které jsou v této třídě definované [44][42].

Údržbu kódu usnadňuje tím, že pokud se změní uživatelské rozhraní stránky, není třeba opravovat a měnit testy. Stačí tuto změnu zahrnout v kódu stránky. Výsledkem použití tohoto vzoru je, že se odděluje kód testu a kódu samotné stránky a všech operací, které se nad stránkou dají provést. Což je považováno za výhodu, protože pak nejsou prvky stránky a operace nad nimi roztroušeny různě napříč testy, ale jsou na jednom místě [44].

Každá stránka Online půjčky, je reprezentována třídou, mající příponu Page. Na začátku třídy jsou definovány elementy, se kterými se pak provádějí různé operace. Operace jsou definovány jako metody a funkce, které tedy operují s prvky.

V těchto třídách by se neměla dělat aserce či ověřování, zda je nějaká hodnota správná. To už je součástí testu. Jedno ověření se však provádí a to takové, zda je stránka již dostupná a připravená pro test. Ověřuje se tedy, zda je na stránce nějaký kritický prvek, který mj. také ověří, zda jsme na správné stránce – v případech, kdy dojde v rámci testu k přesměrování.

V testech Online půjčky se vždy ověřuje nějaký klíčový prvek, který je jen na té konkrétní stránce a nikde jinde, aby bylo jisté, že je vše připraveno. Dělá se to přes metodu *isPageContentValid()*.

```
public boolean isPageContentValid() {
    return (isElementPresent(By.id("hcrfrag")) != null);
}
```

Ve funkcích, kdy se přechází na novou stránku, tudíž jejich návratovou hodnotou je instance nové třídy, se zavolá funkce *addPageToFactory()*. Tato funkce vrací buď vyjímku, v případě, že *isPageContentValid()* vrací false (tj. že prvek definovaný na stránce nebyl nalezen, jak je vidět výše). A nebo vrátí instanci třídy další, nové stránky.

Metoda *addPageToFactory()* je z třídy GenericPage, která je předkem všech tříd s příponou Page.

```
protected final <T extends GenericPage> T addPageToFactory(Class<T>
pageClass)

{
    GenericPage newPage = (GenericPage)ElementFactory.initEle-
ments(this.webDriver, pageClass);
    newPage.initPageControls();


    if(!newPage.isPageContentValid()) {
        throw new IllegalStateException(String.format("Page has in-
correct content or it is not the page you expected"));
    } else {
        return newPage;
    }
}
```

6 Automatizované testování finanční webové aplikace a její specifika

Finanční webovou aplikací je myšlena aplikace Online půjčka, přes kterou si klienti mohou založit hotovostní úvěr.

Tato aplikace jim vypočítá orientační RPSN, vygeneruje předsmluvní formulář, sesbírá o klientovi data a nabídne mu produkty, případně zamítne jeho žádost.

6.1 Specifika webové aplikace a jejich testování

Protože koncovým uživatelem aplikace je klient, je třeba, aby se na stránce dobře vyznal. Takže když jde klient na zmíněnou aplikaci, je to proto, že si chce založit úvěr. Je tudíž třeba, aby klíčové prvky dominovaly celé stránce.

Klíčovými prvky jsou tu posuvníky s hodnotami výše úvěru a s hodnotami počtu splátek.

Obr. 8 Ukázka posuvníků s výší úvěru a počtem splátek

Tester se zde musí chovat jako klient, který první, co bude zkoušet, bude hýbání posuvníky tam a zpět. Protože každá výše úvěru může mít jinou maximální výši počtu splátek, je třeba, by se to na hodnotách posuvníku počtu splátek projevilo. Přesně tento testovací scénář je již zautomatizován.

Když už je navolena výše úvěru i počet splátek, další důležitou informací k zobrazení je výše splátky. Ta se samozřejmě odvíjí od dvou předchozích hodnot, ale jsou zde připočteny další poplatky a též RPSN.

Klient už má zvolenou výši úvěru, počet splátek a je informován i o výši splátky, je třeba si nyní zvolit způsob zaslání dokumentace k úvěru a též způsob, jakým chce vyplatit úvěr.

Obr. 9 Ukázka prvků volitelných služeb

Na obrázku č.2 lze vidět selectbox s možnostmi pojištění. Na selectbox musí být možné kliknout, po této akci se musí vypsat jeho položky, které by měly odpovídat nastavení. To stejné se týká i ostatních select boxů, tj. *Dokumentaci chci zaslat* a *Úvěr chci vyplatit.*

Ty služby, které jsou zpoplatněné, by se měly po zatrhnutí checkboxu, hned projevit na výši splátky a to o hodnotu, která souhlasí s nastavením této služby. Tudíž je třeba otestovat, že se tak stane a o správnou částku. Samotné checkboxy by mělo být třeba vždy zatrhnout.

Každá jednotlivá volba služby na této stránce, se musí projevit ve výsledném úvěru. Vše musí být uloženo a stane se to tak součástí úvěru. Vše by mělo jít najít v PDF předsmluvního formuláře.

Vypočtené RPSN je třeba ověřit, tj. ověřit, že je vypočteno správně pro danou výši úvěru a počtu splátek. Jeho výše, stejně jako výše úvěru a počet splátek se musí projevit v předsmluvním formuláři.

Posledníkm prvkem je tlačítko pokračovat v žádosti, po kterém musí dojít k přesměrování na další stránku.

Klient už si zvolil produkt a jeho souhrn může vidět v předsmluvním formuláři. Nyní je třeba, aby poskytl nemálo informací o sobě, aby firma mohla posoudit, zda mu produkt schválí.

Dostává se tedy na formulář, kde je třeba zadávat informace o svojí osobě, o zaměstnání a jaké jsou jeho příjmy a výdaje.

Částka: 60 000 Kč Délka splácení: 84 měsíců Měsíční splátka: 1 155 Kč změnit ›

Osobní údaje

Titul: Vyberte ...

Jméno:

Příjmení:

Rodné číslo: /

Datum narození:

Státní příslušnost: Vyberte ...

Místo narození:

Číslo OP:

Platnost OP:

Neomezená platnost

Vzdělání: Vyberte ...

Kontaktní údaje

Mobil: +420

Telefon: +420

E-mail: @

Obr. 10Ukázka formuláře

Ne všechny položky formuláře jsou povinné, je tedy třeba ověřit, že ty položky, které mají být povinné, povinné jsou a ty které nemají být, jimi nejsou.

Nad formulářem se zobrazuje zvolený produkt, který by měl stále odpovídat tomu, který si klient navolil.

Klient zde může zadávat i různé nesmysly, tudíž jsou třeba validace na všechna políčka. Hlavně tam, kde se očekává jen jeden druh znaků, ktedy buď čísla či jen pís-

mena a žádné další znaky. Je jím například rodné číslo či jméno. Pokud se zadá nevalidní hodnota do pole formuláře, musí o tom dát aplikace klientovi vědět. Tato vlastnost musí být otestována.

Předposlední stránkou je pak rekapitulace všech informací, které klient zadal do všech formulářů a též zvolená výše úvěru a počtu splátek.

Částka: 180 000 Kč | Délka splácení: 48 měsíců | Měsíční splátka: 4 899 Kč | změnit ›

Pojištění: Bez pojištění **Volitelné služby:** E-mailová připomínka nadcházející splátky, Elektronický měsíční výpis změnit ›

Osobní údaje

Titul	Ing.
Jméno	Mzlugk
Příjmení	Lquvkenbnh
Rodné číslo	7901083333
Datum narození	8.1.1979
Státní příslušnost	ČR
Místo narození	Ehrvuhw
Číslo OP	NP233601
Druhý identifikační doklad	Cestovní pas
Číslo dokladu	138492777
Vzdělání	Vyučen(a) s maturitou

Kontaktní údaje

Mobil	603386454
Telefon	
E-mail	yyjZYS@EkBbz.cz

Trvalé bydliště

Ulice	Nvsqjkc
Číslo popisné	33
Město	Dapchngq
PSČ	68703
Od kterého roku bydlíte na současné adrese	2009
Druh bydlení	vlastní dům/byt

Rodinné údaje

Rodinný stav	svobodný/svobodná

Zaměstnání

Zdroj příjmu	zaměstnanec
Sektor zaměstnání	státní sektor
Sféra zaměstnání	administrativa
Pracovní pozice	vyšší management (ředitel, manažer, soudce)
Typ pracovní smlouvy	doba neurčitá
Zaměstnán od	1.2.2005
Čistý měsíční příjem	19 500

Zaměstnavatel

Název zaměstnavatele	Dsfs
IČ zaměstnavatele	111111111
Ulice	Dsfdsf
Číslo popisné	sd6f54
Město	Sdfsdf
PSČ	60300
Telefon	519454120

Finanční údaje

Dokumentaci chci zaslat	poslat e-mailem a podepsat SMS kódem
Úvěr chci vyplatit	na běžný účet
Způsob úhrady splátek	poštovní poukázkou
Předčíslí účtu	
Číslo účtu	2052365369
Kód banky	2210 - Evropsko-ruská banka,a.s.
Rok založení běžného účtu	2000

Obr. 11 Ukázka rekapitulace

Je třeba otestovat každou jednu hodnotu, zda odpovídá tomu, co klient zadal původně. Na této stránce se dají též změnit volitelné služby. Tj. po kliknutí na změnit. musí dojít k přesměrování na stránku s volitelnými službami.

Součástí stránky s rekapitulací je zadávání captcha kódu, zda se nejedná o robota a poté zaslání SMS kódu a jeho ověření. Následně se zpřístupní tlačítko *Odeslat žádost*, které vede k výsledku posouzení žádosti.

Obr. 12 Ukázka ověrovacích prvků v aplikaci

Pokusů pro zadání obou kódů je omezený počet. Po vyčerpání pokusů jednoho či druhého zadávání kódu následuje ukončení žádosti. Tester zadává špatné kódy, dokud se nedosáhne reakci aplikace, která musí odpovídat očekávané reakci. Zároveň se ověří počet pokusů, který k této reakci vedl a ověří se, že sedí s parametrem, kde se tento počet pokusů nastavuje.

Poslední stránkou je pak výsledek posouzení. Může mu předcházet ještě volba alternativy, pokud mu aplikace neschválí požadovaný produkt, ale nabídne mu množinu jiných. Pak je třeba ověřit, že všechny potřebné elementy se vyskytují na stránce, že jsou funkční, případně že obsahují očekávané hodnoty.

6.1.1 Webová aplikace a druhy elementů

V souhrnu – druhů používaných elementů v aplikaci Online půjčka je několik a používájí se všude napříč aplikací. Jsou to:

- Tlačítka
- Formulářová pole
- Checkboxy
- Selectboxy s výčtem hodnot
- Labely k formulářovým polím
- Odkazy uvnitř stránky – změnit
- Obrázky
- Text
- Tooltipy s nápovědou
- Lightboxy

Specialní typy pro Online půjčku jsou pak:

- Posuvníky – výše úvěru a počet splátek
- Captcha kód a tlačítko pro jeho ověření
- Tlačítko pro poslání SMS kódu a jeho ověření
- RPSN
- Progressbar

6.1.2 Způsoby hledání elementů na stránce

Každý výše jmenovaný element může mít atributy, které mu přiřadí například identifikátor či název pro spolupráci s jinými prvky. Tyto atributy lze pak využít pro identifikaci prvku na stránce [45].

Jak už bylo zmíněno výše, k automatizaci se používá nástroj Selenium Webdriver. Prvky se tedy hledají buď pomocí metod samotného Webdriveru a nebo pomocí metod třídy WebElement. Instance obou tříd mohou volat metody *Find Element* či *Find Elements*, kdy první vrací objekt třídy WebElement a druhá list objektů třídy WebElement. Tyto dvě metody používají lokátor či dotazovací objekt nazvaný *By [46]*.

- **By ID** – pomocí atributu ID

```
<input id="captchaPassword" class="captcha text" type="text" va-
lue="1234" size="8"></input>
```

```
WebElement tbCaptcha = isElementPresent(By.id("captchaPassword"));
```

- **By Class Name** – pomocí názvu třídy

```
<span class="slidersummary">
```

```
isElementPresent(By.className("slidersummary"))
```

- **By Tag Name** – pomocí názvu použitého HTML tagu

```
<select>

<option value="POST_DOP">
```

```
    poslat poštou doporučeně a podepsat ručně
</option>
<option value="KURYR">
    poslat kurýrem a s ním i ručně podepsat
</option>
</select>
```

```
List<WebElement> options = webElement.findElements(By.tagName("op-
tion"));
```

- **By XPATH** – pomocí Xpath

```
<div class="dijitSliderMoveable dijitSliderMoveableH">
       <div dojoattachpoint="sliderHandle,focusNode"
  wairole="slider"></div>
</div>
```

```
@FindBy(xpath = "//div[@class='dijitSliderMoveable dijitSliderMove-
ableH']/div[@wairole = 'slider']")
private WebElement sdCash;
```

U bodů, kde jsou uvedené příklady, je kód přímo z automatizovaných testů Online půjčky. Vždy je nejprve reprezentace prvku v HTML a za ním podoba kódu v programovacím jazyce Java pro jeho identifikaci. Další možnosti jsou:

- **By Name** – pomocí atributu name
- **By Link Text** – pomocí textu v URL
- **By Partial Link Text** – pomocí části v textu URL
- **By CSS** – pomocí názvru CSS stylu
- **Pomocí JavaScriptu**

Tyto způsoby nejsou využity v automatizaci Online půjčky. Nicméně příklady v kódu lze nalézt ve zdroji [46].

7 Návrh a implementace řešení procesu automatizace

V této kapitole se budu zabývat návrhem řešení problémů zjištěných v rámci analýzy procesu automatizace testů ve firmě EmbedIT.

7.1 Zajištění kvality testovacích scénářů

Základem řešení je změna přístupu k samotnému testování regrese. Regresních scénářů je již přes tisíc a je třeba je všechny, do jednoho, otestovat. Této činnosti se však přiklání kupodivu malá důležitost mezi samotnými testery.

7.1.1 Změna přístupu k regresním testům

Je třeba rozšířit mezi testery význam testování stávající funkcionality a nechat je uvědomit si, že následky, které plynou z chyb, jsou velké a obvykle velmi drahé. Ačkoliv by se mohlo zdát, že jich se to nedotkne, je třeba jim ukázat, že opak je pravdou.

Protože se ve firmě mezi testery berou regresní testy spíš jako „nutné zlo“ než jako zásadní činnost, má vykonávání regresních testů kvalitu testovacích scénářů a počet nalzených chyb v testech. Testy nové funkcionality se vyzdvihují nad vše ostatní a důsledkem toho je na regres pohlíženo jako na něco druhořadého a méně důležitého. Takže jakmile tester dotestuje novou funkcionalitu, vytvoření testovacího scénáře k této nové funkcionalitě, aby se mohla testovat v rámci regresu, je opět bráno jako již něco navíc, co vlastně ani není třeba.

Řešením je připomenutí managementem i zákazníkem význam ověření, že stávající funckcionalita není nijak zasažena změnami z nové funkcionality, tj. že se nic nerozbilo. Říct testerům o výši nákladů, případně ušlém zisku z chyb, které v regresu mohou vzniknout a nebo již vznikly a byly objeveny až na ostré verzi. Jinými slovy je třeba testery motivovat k regresním testům.

Toto připomenutí by proběhlo v rámci meetingu, kde by vystupovali řídící složky firmy i zákazník. Byl by zde prostor pro debatu, ale hlavně by výsledkem měla být změna přistupu testera k testování regresu. Mělo by se testerům připomenout, že tahle práce je pro business také velmi důležitá.

Protože testovací scénář je základem pro automatizaci, bez něj není co automatizovat, je důležité, aby byla zajištěna jeho kvalita. Pokud testeři změní přístup k regresním tesům, které obsahují testovací scénáře, jasně se to odrazí i na kvalitě testovacích scénářů. K jejich vytvoření a údržbě budou přistupovat zodpovědněji, protože budou vědět, že je to důležité a že tahle práce má smysl [35].

7.1.2 Stanovení zásad při tvorbě testovacích scénářů

Aby se testovací scénáře daly automatizovat, je třeba dodržovat několik zásad při jejich tvorbě. Jsou to:

- Opakovatelnost testovacího případu – je třeba psát testovací scénáře tak, aby se pokaždé došlo stejnému výsledku, který se má ověřovat. Takže než se testovací scénář prohlásí za hotový, je třeba ho několikrát vyzkoušet, aby bylo zajištěno, že nenastane nějaká nepředvídatelná situace, která způsobí jiný výsledek. Výsledek musí být pořád stejný.

- Očekávaný výsledek musí být jasně definovaný – specifikování výsledku musí být přesně určené, aby se vědělo, co je třeba kontrolovat. Například, pokud je třeba ověřit nějakou hodnotu, musí být jasně uvedeno, kde tato hodnota je, případně jakým výpočtem se k ní došlo a s čím ji ověřit. Musí být též jasné, o co v tom scénáři jde, tj. co se testuje a ověřuje.

- Kroky testovacího scénáře jsou snadno následovatelné – tím je myšleno, že testování scénáře není podmíněno nějakými kroky či akcemi, které nejsou uvedeny v testovacím scénáři. Nesmí v něm tedy chybět kroky či nesmí být tyto kroky nepřesné. Pokud je například krokem „uživatel založí smlouvu“ pro přípravu testovacích dat, ale není uvedeno kde a jak má smlouvu založit, tak je to přesně ten případ, který nesmí nastat.

7.1.3 Zavedení podprocesu test review

Po tom, co se vytvoří nový scénář, musí proběhnout kontrola, že výše stanovené zásady byly dodrženy. Pokud se zjistí, že scénář nedodržuje výše stanovené zásady, vrátil by se scénář k opravě.

Tím by se měla zajistit kvalita nově vytvořených testovacích scénářů a již by nemělo docházet k tomu, že některé z kroků scénáře nejsou jasně vysvětleny, jsou neproveditelné či nedávají smysl.

Po opravě scénáře, z důvodu procházení kroků scénáře, které skončilo chybou, se dá scénář k test review. Takto bude zajištěno, že oprava scénáře nevygenerovala nějaké nejasnosti.

7.1.4 Zavedení podprocesu předání testovacího scénáře

Testovací scénář je tedy napsaný a je ověřeno, že jsou dodrženy všechny stanovené zásady pro jeho obsah. Není však stále zajištěno, že dohromady testovací scénář je správně pochopen technickým testerem, který ho má automatizovat.

Proto musí dojít k předání informací o tom, čeho se test týká a jak si připravit správná testovací data, mezi testerem, který scénář vytvořil a testerem, který ho bude automatizovat. Často se totiž stává to, že technický tester nerozumí tomu, co se ve scénáři testuje a protože některé scénáře již jsou vytvořeny a již neproběhnou přes krok test review, musí se jeho obsah probrat s členem týmu, který daný testovací případ zná.

Tak se předejde tomu, že se automatizují scénáře, které jsou zbytečné a bezúčelné, protože testují něco, co se již netestuje, co nemůže nastat či se to testuje na jiných testovacích datech.

Také odpadá technickému testerovi dlouhé zjišťování toho, jak scénář vlastně funguje a kdo ho psal, protože tam není nějaký krok jasně definovaný a podobně.

7.2 Evidence možných dopadů nové funkcionality na tu stávající

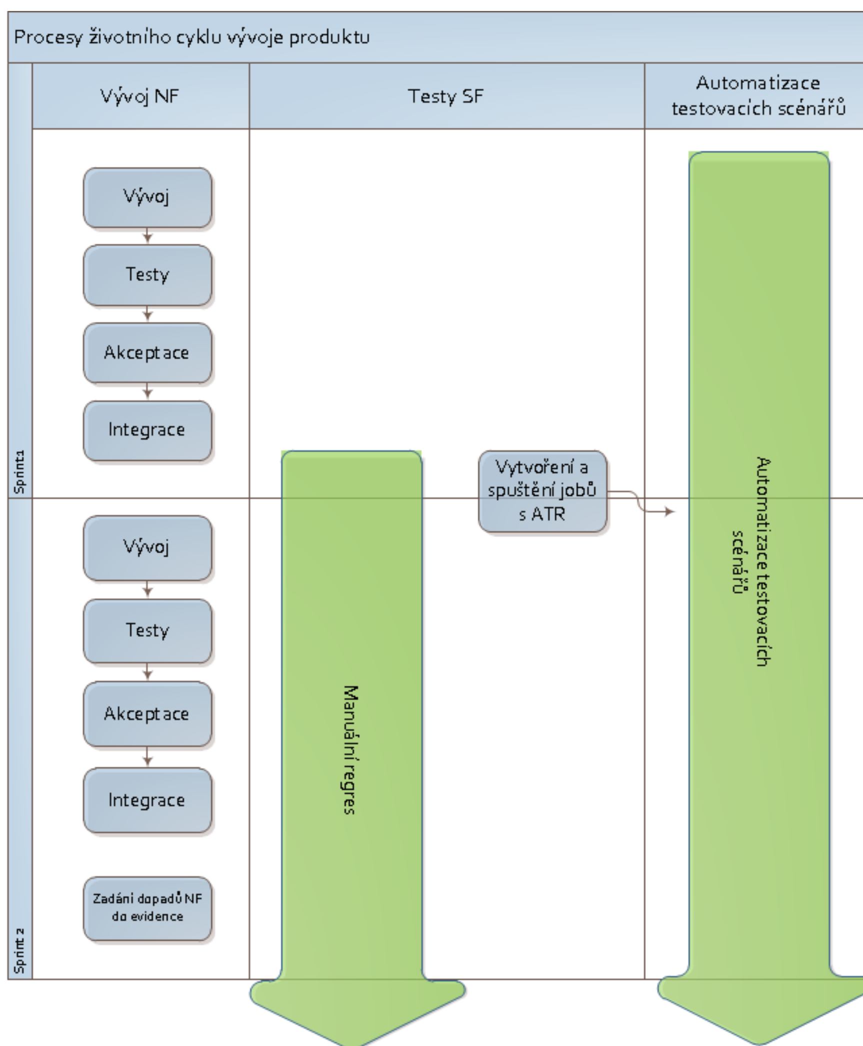
Napříč týmy se vyvíjí nová funkcionalita pro různé aplikace, tedy i pro Online půjčku. Protože je třeba vědět o změnách v rámci nové funkcionality, musí se někde evidovat. Jinak, jak již bylo zmíněno v analýze, dochází k tomu, že jsou scénáře neaktuální.

Všechny týmy, které jsou správci za některé aplikace a jejich testovací scénáře pak mohou každý release kontrolovat, zda neproběhla nějaká změna, kterou je třeba začlenit do svých testovacích scénářů.

Pro evidenci dopadů nové funkcionality na stávající se vytvoří dokument, který bude rozdělný dle jednotlivých aplikací.

Bude v něm jasně uvedeno, o jakou novou funkcionalitu se jedná, její evidenční číslo (číslo user-story), který tým ji vytvářel a kdo přesně ji testoval.

Protože se nová funkcionalita vyvíjí v rámci sprintu, tak se vždy na konci druhého sprintu do tohoto souboru pro evidenci doplní informace za oba sprinty.



Obr. 13 Začlenění evidence NF do procesu vývoje

7.3 Evidence změn v testovacím plánu

Každá změna provedená v existujícím testovacím plánu se musí zaznamenat. To by mělo být v místě, kde je samotný testovací plán, aby bylo vše pohromadě a bylo jasné, co se v rámci plánu vše dělo.

Musí být tedy jasně zaznamenáno kdy se stala změna, o jakou změnu se jedná, tj. její stručný popis a kdo změnu provedl.

Protože nyní je testovací plán vytvořen v aplikaci Excel firmy Microsoft©, je snadné přidat tyto informace. Soubor je sdílen a je lokalizován na firmou sdíleném místě.

7.4 Kontrolované určení rozsahu automatizace testovacích scénářů

V rámci podprocesů test review a předávání scénáře, kdy se s cénářem aktivně pracuje, by se mělo zamýšlet i nad tím, zda tento scénář, testující nějaký testovací případ, stačí. Každý tým, spravující nějakou skupinu scénářů, by se měl takto zamyslet, tj. zda je testy dostatečně podchycena funkcionalita testované aplikace.

V rámci toho by měla též probíhat komunikace se zákazníkem, tj. zda tu stále není něco, co v testech chybí.

Jak už bylo psáno v příslušných kapitolách, co se bude a nebude automatizovat zatím určují v podstatě jen techničtí testeři.

Návrh je takový, že každý scénář, ať už by na první pohled nebyl automatizovatelný např. z důvodu kontroly souboru v externí aplikaci, bude konzultován se zodpovědným týmem, tj. dojde k předání scénáře. Zde by mělo docházet k debatě o tom, co je zde vlastně cílem testovat, zda je vůbec třeba některá data exportovat do externího souboru a podobně.

Zabrání se tak tomu, aby určovali rozsah automatizace pouze techničtí testeři, kteří do testované funkcionality tolik nevidí. Souvisí to nicméně i s kvalitou scénářů, protože pokud by se již při vytváření scénářů myslelo na jejich pozdější automatizaci, psali by se jinak a mohla by se automatizovat větší skupina scénářů.

7.5 Evidence testovacího týmu pro automatizaci

V rámci testovacího plánu se bude evidovat, kdo a co v automatizačním týmu dělá a kdo je za co zodpovědný. Pak se nestane, že se čeká na to, až *někdo* vytvoří joby a podobně. Budou určeni členové týmu zodpovědní za veškeré aktivity v rámci procesu automatizace.

Bude zaznamenáno, kdo spouští joby s automatizovanými testy, kdo je pak vyhodnocuje a provádí maintenance pro každou aplikaci.

7.6 Kontrola dodržení doporučených změn v rámci CR

Protože se ukázalo, že se stává, že doporučené změny v rámci code review (CR) neberou v potaz, tj. změněný kód se dle nich neupraví, je třeba zavést kontrolu, zda se tak děje.

Na každý commit, který se provede, je tudíž třeba udělat code review a ten, co toto CR provádí, se bude muset podívat, zda v některém z dalších commitů, obsahuje i jím doporučené změny.

Commit by pak měl obsahovat komentář, že se jedná o opravu změn v kódu, doporučených v konkrétním code review. To je totiž evidováno přes nástroj.

7.7 Automatizace vytváření jobů

Protože již je automatizováno vytváření jobů pro automatizované smoke testy, které vznikají po vytvoření prostředí, nemám smysl navrhovat nové řešení, jak tohle udělat. Používá se totiž stále stejný proces vytváření jobů, jen několik proměnných je nastaveno jinak.

Proč se nevytvoří i joby s regresními testy při vytváření prostředí je spíše otázka na management. Je zde tudíž třeba tlačit právě tam, aby se to zavedlo. Ti pak budou tlačit na způsobilé a odpovědné týmy za tuto funkcionalitu.

7.8 Paralelní spouštění automatizovaných testů na vzdálených strojích

Jak již bylo psáno výše, spouštění automatizovaných testů se ve firmě provádí za pomoci aplikace Jenkins. Právě v této aplikaci se vytvářejí multi-joby, které obsahují joby, spouštějící jednotlivé testy patřící do konkrétní suity.

Každý job má své nastavení. Mj. se nastavuje též stroj, na kterém má job spustit testy. Každý stroj dává k dispozici několik slotů. Každý slot může využívat jeden job.

Pro paralelní spouštění testů firma není ochotná kupovat speciální nástroje, které to umožňují. Proto se nabízí řešení rozdělit testy na 2 či 3 suity testů. Těmto suitám vytvořit joby, které bude spouštět jeden multijob. Každý z jobů pak bude spouštěný na jiném slotu a to zároveň. Tudíž se najednou budou vykonávat až 3 testy, podobně jako je tomu na lokálním počítači.

Nejprve je třeba vytvořit suity, kde se budou spouštět jen testy Online půjčky. Jedna se jmenuje Iloan1TestSuite a druhá Iloan2TestSuite. Jim jsou pak přiřazovány jednotlivé testy z třídy testů tím, že je přiřadíme do příslušné kategorie pomocí *@Category(Iloan1Test.class)* nebo *@Category(Iloan2Test.class).* To se píše vždy nad implementaci testu. V nastavení jobu se pak určí, ze které kategorie se budou testy spouštět.

Nový job se obvykle tvoří kopií starého, kterému se změní parametry dle požadavků. Zde se v parametrech jobu upraví prostředí, na kterém se budou testy spouštět a zmíněná suita, ze které se budou brát testy.

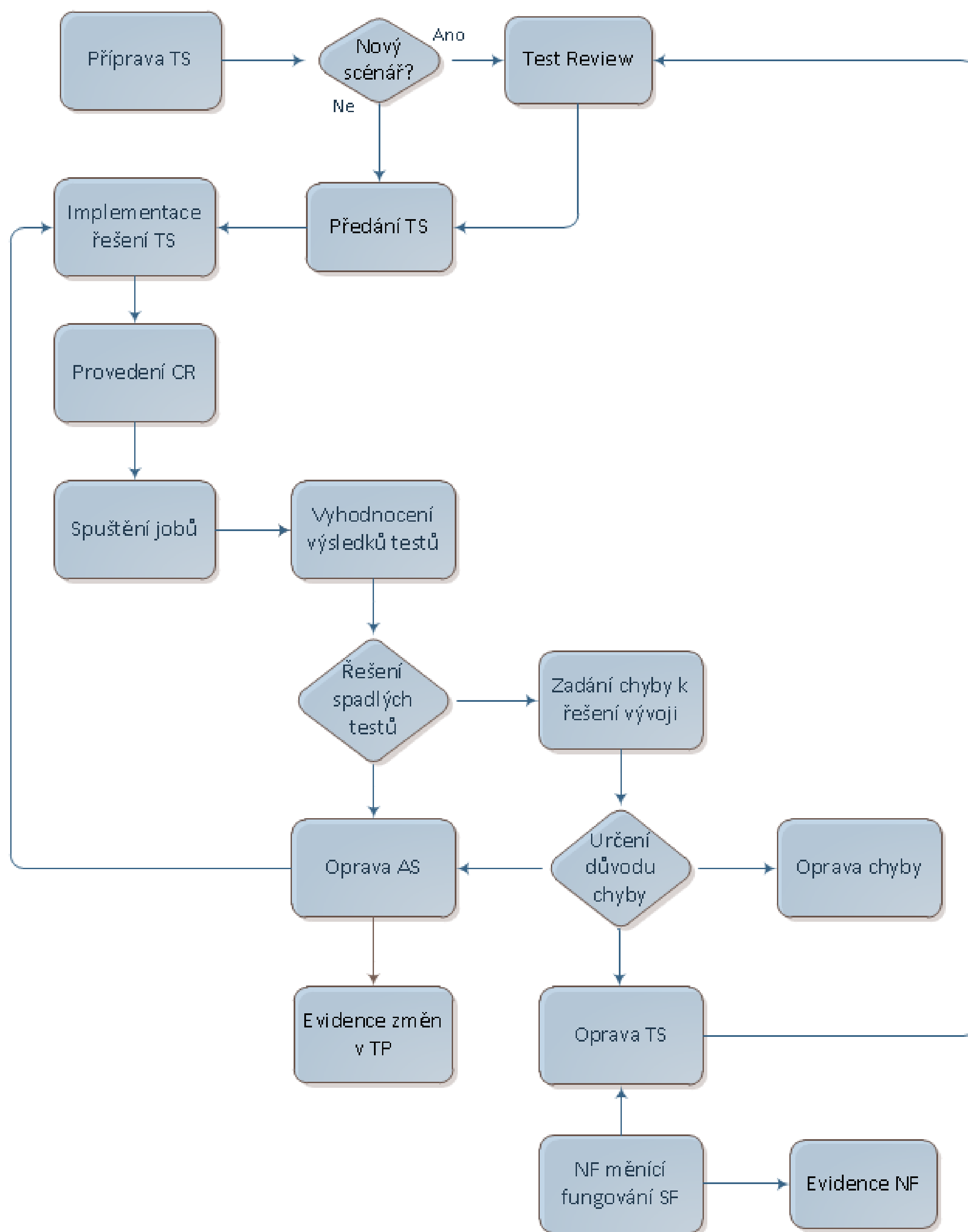
7.9 Upravený proces automatizace dle návrhu

Níže na obrázku je uveden proces automatizace rozšířen o několik podprocesů. V rámci přípravy testovacího scénáře (TS) se rozhodne, zda jde o nový scénář. Pokud ano, provede se nově zavedený podproces test review a teprve poté se testovací scénář předá. Pokud ne, jde se navrhovanému podprocesu předání testovacího scénáře rovnou.

Pokud se určí, že příčinou zadané chyby je chyba v automatizovaném scénáři (AS), povede to k jeho opravě, nicméně, že se tak stalo, se zaznamená v evidenci změn testovacího plánu.

Každá nová funkcionalita (NF) měnící si fungování stávající funkcionality (SF) bude zaznamenána v rámci podprocesu evidence NF.

Po opravě testovacího scénáře se musí provést test review, aby se tedy zajistilo, že scénář neztratil na jeho kvalitě.



Obr. 14 Upravený proces automatizace

8 Návrh a implementace řešení zjištěných nedostatků v testech Online půjčky

Tato kapitola se zabývá návrhem a implementací řešení nedostatků zjištěných v rámci analýzy automatizovaných testů Online půjčky. Níže navržená řešení a implementace vycházejí z analýzy testů v kapitole 4.5 Testy Online půjčky, kde jsou jednotlivé testy popsány.

8.1 Stávající automatizované testy

Testy uvedený v této kapitole jsou ty, které z různých důvodů spadly na vlastní chyby. Aby se tyto chyby odstranily, musely se projít a zjistit, v čem je chyba. Tato kapitola vychází z kapitoly 4.5.1 Výčet testů.

V rámci odstraňování chyb se však často zjistily chyby nové, dosud nezjištěné, protože test se třeba do fáze, kde chyba nastává, nedostal kvůli chybě původní. V této kapitole jsou uvedeny problémy(chyby) v testech a kroky k jejich odstranění.

8.1.1 SecondPageCalculationTest

Nejprve je třeba opravit to, aby se na první stránce správně navolil počet měsíců, po které chce klient splácet. Zjištěný problém byl v tom, že se nejprve volil počet splátek a pak až teprve požadovaná částka úvěru. Každá výše úvěru má totiž jiné rozpětí počtu splátek. Tudíž se mění i pozice původně navoleného počtu splátek. Takže pokud se nejprve navolí 60 splátek, což je momentálně třetí možná pozice táhla na slideru a až teprve teď se nataví výše úvěru, což je v testu 25 000 Kč, tak se automaticky přenastaví i počet splátek, který zůstává na třetí pozici slideru, nicméně je na něm již jiná hodnota a to 48 splátek. Řešením je změnit pořadí nastavení úvěru a počtu splátek. Nejprve se nastaví úvěr a až pak se nastaví doba splácení.

Dále je třeba opravit test, aby nepadal na aserci, kvůli které se nespustí zbytek testu. Aserce totiž předpokládá existenci checkboxu označeného speciálním kódem doplňkové služby. Jak je vidět na obrázku Obr. 9 Ukázka prvků volitelných služeb, jsou zde tyto checkboxy znamenající nabízené doplňkové služby k vybranému produktu. Každá služba má tedy svůj unikátní kód, pod kterým je uložena v databázi. Různé navolené produkty mají různé služby. Tudíž bylo třeba zjistit, jaký produkt z jaké skupiny produktů je v rámci testu vybrán, aby se mu SQL dotazem mohly najít na něj navázané doplňkové služby.

Po zjištění, o který produkt se jedná a z jaké skupiny, byl upraven SQL dotaz, který nyní vrací ty správné kódy doplňkových služeb a test už nepadá na aserci.

Doplněním testu je kontrola, že zvolená výše úvěru a počtu měsíců pro jeho splácení z první stránky aplikace, se propíše do druhé stránky aplikace, kde probíhá detailnější kalkulace.

Nejprve je třeba si pamatovat navolené částky. K tomu jsem použila uložení výše částky a počet splátek do proměnných (klíčů) v resources projektu. Jsou jimi *ILOAN_FIRST_PAGE_CASH* a *ILOAN_FIRST_PAGE_TIME*.

Po tom, co se navolí výše úvěru a počtu splátek, se přejde na další stránku. Zde je zavedena kontrola, že aktuálně navolené hodnoty výše úvěru a počtu splátek v posuvnících jsou stejné, jako byly na první stránce. K tomu je pužit *collector*, volající funkci *checkThat()*, která ověří, zda se skutečně nastavený počet měsíců s původně nastavenou hodnotou shodují. Pokud by to tak nebylo, vypíše se zpráva zadaná v prvním parametru funkce. Ukázka kódu je uvedena v přílohách.

8.1.2 MinMonthlyInstalmentTest

Pro přechod z první stránky na druhou se zde použije funkce *onNextClick()*. Protože se zde nevolí počet splátek a ani výše úvěru. Nedochází tak ke zbytečnému zdržování testu. Jediné, co tato funkce dělá je, že klikne na tlačítko, které vede k přesměrování na další stránku, poté čeká 5 sekund, než se načte další stránka a až pak se ověřuje, zda jsme na správné stránce již zmiňovanou *addPageToFactory()*.

Do tohoto testu je přidána kontrola, zda se cena všech doplňujících služeb přičte k výši splátky. Cena všech služeb je stále stejná, tudíž i suma, která se má přičíst k měsíční splátce bude stejná.

Aby se zatrhly všechny checkboxy pomocí funkce *setAllCheckboxes()*, musí se přidat sekundové čekání než jednotlivé technologie, které mezi sebou komunikují, dokončí potřebné operace. Zatrhnutí checkboxu se totiž nestihlo dost rychle vykreslit a už se pokračovalo s dalším elementem. Proto je třeba přidat *thinkTime()*, které je nastaveno na čekání 1 sekundu. Nyní se již zatrhují tedy všechny checkboxy.

Protože je dopředu známa výše splátky, přičte se k ní součet poplatků, který je též dopředu znám. Tento součet je uložen v resources a pomocí collector a jím volané funkce *checkThat()* se porovnává skutečně zobrazenou výší splátky na stránce.

8.1.3 MaxMonthlyInstalmentTest

Nejprve je třeba změnit pro přechod z první na druhou stránku jinou funkci a to *onNextClick().* Stejně jako v předchozím testu – zabrání se tak zbytečnému nastavení počtu splátek a výše úvěru a tedy i prodloužení doby, po kterou je spuštěný test.

Protože se výše RPSN nepočítá správně, očekávaná hodnota je jiná než skutečná a test končí chybou, bylo třeba ho opravit.

8.1.4 SmsCodeVerificationTestWrongCaptchaTest

Test smsCodeVerification je spojen s tímto testem. Oba testy totiž projdou celou žádostí až do konce, kde se nejprve vyplňuje captcha kód a pak SMS kód. Protože to je jediná odlišnost mezi nimi a přitom na sebe obě funkce navazují, nabízí se právě to spojení do jednoho testu. Ušetří se tím též čas, protože se nemusí procházet celou žádostí až do konce.

Do tohoto testu tedy byla přidána část o tom, že se dvakrát klikne na tlačítko a tím se ověří, že pokusy pro zaslání SMS jsou vypotřebované. Dále byla odstraněna

část, kde se klikne na tlačítko a čeká se na vypsání finálního výsledku posouzení žádosti. Tato část není obsahem testovacího scénáře a často zde test skončil chybou, protože se nestihl vypsat výsledek, na který test čekal.

Chybí zde tedy část o kontrole, zda se aplikace rozpozná a upozorní uživatele chobovou zprávou, že zadal špatný kód SMS, tedy takový kód, který nesouhlasí s tím, co mu byl zaslán. Nejprve se tedy vloží řetězec znaků, který znamená špatný kód, klikne se na tlačítko, po kterém dojde k ověření v aplikaci, zda kódy souhlasí a po té se vypíše chybová hláška. Její zobrazení je třeba kontrolovat.

Přes metodu *fillAndSubmitSmsCode()* se vyplní špatný kód SMS a po té se to opět přes collector a jeho funkci checkThat ověří, že je zobrazena chybová zpráva o špatně zadaném kódu. Přes funkci isErrorMessagePresent() se ověří přítomnost prvku, který představuje chybovou zprávu. Pokud je přítomnost potvrzena, funkce vrátí true a předpoklad je splněn, test projde.

8.1.5 AlternativesTest

Pro snadnější kontrolu nad výsledkem je v tomto testu třeba umožnit zadávání klíčových hodnot pro vyhodnocení žádosti již na úrovni testu. To řeší funkce *fillAllPagesParametrized()*, kteřá má několik parametrů, které se vyplní pak do příslušných políček ve formulářích. Dle hodnot těchto parametrů se dá ovlivnit výsledek schválení smlouvy, tudíž je její použití víceúčelné.

V rámci zmíněné funkce fillAllPagesParametrized() bylo třeba ještě takových funkcí, které by přebraly parametry a postaraly se o vyplnění jednotlivých polí, dle těchto parametrů. Jsou to *fillPersonalPageParametrized()* a *fillIncomePageParametrized()*.

Protože v testech *chooseAndSubmitAlternativeTest* a v *sendEmailOfApprovalTest* je navíc už jen ověření pojištění a také že po přijetí alternativního produktu přišel e-mail a došlo k přesměrování na další stránku s poděkováním klientovi, byly tyto tři testy spojeny. Byl takto ušetřen čas zbytečnou další žádostí se stejným výsledkem alternativ.

Po tom, co se přechází na další stránku s poděkováním, se zde nejprve objeví v okně vždy upozorňění, zda chce uživatel skutečně odejít ze stránky a musí s tímto nejprve souhlasit. Tento souhlas, řeší metoda *acceptAlert()*, která je uvedená v přílohách.

8.1.6 RejectAlternativeTest

Zde je třeba opět funkce *fillAllPagesParametrized(),* díky které se dojde až k nabídnutým alternativám, které se pak příslušným tlačítkem odmítnou. Test již stabilně dostává očekávaný výsledek posouzení žádosti, tudíž mu nic nebrání v tom, aby procházel bez chyby.

8.1.7 InsuranceChangeTest

Nejprve je třeba nahradit funkci se zbytečnou volbou výše úvěru a počtu splátek hned na první stránce, za funkci onNextClick(). Po té se totiž ještě znova volí produkt a také pojištění na stránce druhé.

Nicméně v rámci nastavení výše úvěru a počtu splátek na stránce první, kdy úvěr byl vysoký, se rovnou počítalo s tím, že pojištění se automaticky vyplní a zobrazí se k němu odkaz na nápovědu. Tudíž se musely jednotlivé kroky přeskládat. Po tom, co se přejde na další stránku, hned se vybere produkt a i požadované pojištění a až teprve poté je kontrola na nápovědu k pojištění.

Dále je třeba kontrolovat to, že po akceptaci nového pojištění na poslední stránce s rekapitulací, musí zmizet box s textem a tlačítky, oznamující, že zvolené pojištění si klient nemůže zvolit. Kromě toho také po nastavení probíhá kontrola, zda se propsalo nově akceptované pojištění na rekapitulační stránce.

8.1.8 LoanReject

V rámci zobrazení důvodu zamítnutí, musí být do formuláře vyplněna taková data, aby se specifického zamítnutí dosáhlo. Proto je použita funkce *fillAllPagesParametrized()*, kde se navolí v parametrech požadovaná data do formuláře. Na konci, kde se vypisuje výsledek žádosti, se kontroluje, zda je zobrazen i očekávaný důvod zamítnutí.

8.2 Nové automatizované testy

Tato kapitola vychází z kapitoly 4.5.2 Funkcionallita aplikace, která není obsahem testovacích scénářů. Na základě jí byly vytvořeny níže uvedené testy.

8.2.1 CheckTextOnFirstPageTest

Zde se ověří, že jsou určité elementy, přítomny a že se s nimi dá interagovat. Aby se zajistilo, že žádný z nich nebude chybět. Text elementů musí odpovídat předloze.

V rámci kontroly, že požadovaný text je na stránce, vznikly funkce, které našly jednotlivé HTML prvky jako *<ul>, <p>* či *<li>* a extrahovaly z nich text pomocí funkce *getText()*. Na začátku každé funkce je identifikace iframe, ve kterém prvky jsou. Po extrahování textu do proměnné typu String se musí zase přepnout na původní obsah pomocí přepnutí focus na původní obsah *switchTo().defaultContent().*

Pokud by se nejprve na začátku nepřepínalo do iframe, WebDriver elementy pak *nevidí*, dokud se focus do iframe nepřepne. Musí se však přepnout i zpět na původní obsah, protože jinak by se stále hledalo v posledním přepnutém iframe.

Dále se testuje fungování odkazu, který má přesměrovat na stránky firmy. Nejprve je třeba vytvořit třídu, která bude představovat stránku, na kterou má být přesměrováno. Protože je potomkem třídy GenericPage, což je předek pro všechny Page třídy, musí obsahovat funkci *isPageContentValid(),* protože je tam tato funkce abs-

traktní. Tato funkce byla již zmíněna a vysvětlena. V rámci ní se hledá jedinečný element, který se zobrazí po jejím načtení. Tento element je možné najít pomocí xPath. Jak je ukázáno v přílohách.

Je třeba vytvořit metodu, která klikne na element představující odkaz na stránky firmy.

Jakmile se již našel odkaz, na který se kliklo, vznikla nová instance webového prohlížeče. Protože ji Selenium WebDriver nevidí, je třeba evidovat tato nová a původní okna, aby se mezi nimi dalo přepínat pomocí funkce *switchTo()*. Celý kód řešení je uveden v příloze.

8.2.2 CheckDeliveryTest

V tomto testu se na druhé stránce navolí výše úvěru taková, aby se hodnoty v selectboxu s možnostmi výplaty úvěru změnil. Pod konkrétní výší úvěru jsou zde dvě možnosti, jak úvěr vyplatit a to poštou a posláním na běžný účet. Úvěr nad tu konkrétní výši úvěru již dovolí vyplácet jen na běžný účet klienta.

Pokud se vybere rovnou poštovní poukázkou, musí na stránce, kde se zadávají peněžní údaje o klientovi, být skryta pole pro zadání bankovního účtu. To platí i naopak.

Nejprve se jde tedy hned z první stránky na druhou pomocí *onNextClick()*. Po té tedy se zvolí takový produkt, který umožní výplatu pouze na běžný účet a to pomocí metody *chooseProduct()*. Po té se provede kontrola, zda je v selectboxu pouze hodnota bežný účet. Všechny hodnoty ze selectboxu se dostanou funkcí *getAllRepayments()*.

Dále se přejde na další stránku, kde se vyplňují hodnoty ohledně klienta. Na další stránce, kde klient vyplňuje informace o zaměstnáních a finanční údaje, se provede kontrola, zda jsou zobrazeny pole pro zadání bankovního účtu a kódu banky. Poté se přejde přes odkaz změnit zpět na stránku, kde se volí produkt. Zde se vybere taková výše úvěru, která povolí již i zaslání poštou. Selectbox pro nastavení kam klient chce úvěr vyplatit se nastaví na poštu a pak se přejde na stránku s vyplňováním osobních informací. Zde již je vše vyplněné z předchozích kroků, tak se jde rovnou na stránku o vyplňování finančích údajů. Zde se již provede kontrola, zda je zobrazen bankovní číslo a kód banky.

8.2.3 CheckChosenCashAmountInstalmentsTest

Cílem je otestovat, zda se na všech stránkách zobrazuje původně navolená výše úvěru, počet splátek a výše měsíční splátky. A pokud se změní pomocí odkazu na změnu produktu, která způsobí přesměrování na druhou stránku, kde se produkt detailněji vybírá, musí se tato změna projevit i na příslušném místě ostatních stránek.

Nejprve se tedy navolí výše úvěru a počet splátek pomocí funkce *chooseAndSubmitProduct()*. Dále se předje na další stránku, kde se vyplňují osobní údaje klienta, již zde se nahoře na stránce zobrazuje výše úvěru, počet splátek a výše jedné

splátky. To se porovná s původně zadaným na předchozí stránce. Porovnání se provádí pomocí *collectoru*, volající *checkThat().* Ve funkci *checkThat()* se pak volá funkce *getHeadCashMonth(),* která vrátí zřetězené hodnoty, které jsou zobrazeny na stránce.

Formulář na stránce se vyplní a pokračuje se na další stránku až do rekapitulace, kdy se na každé stránce kontroluje, zda zvolený produkt sedí s tím zobrazovaným.

Na rekapitulační stránce se klikne na změnit produkt pomocí funkce *changeLoan()* a dojde k přesměrování na stránku, kde se detailněji volí produkt. Po té se opět přejde na další stránku s osobními údaji klienta. Zde se kontroluje, zda se na stránce zobrazuje již nově navolená výše úvěru, počet splátek a tudíž i jiná měsíční splátka. Stejná kontrola se provede i na další stránce, kde jsou finanční údaje. Z této stránky je pak pomocí tlačítka *zpět* a funkce *onBackClick()* vráceno na předchozí stránku.

8.2.4 CheckRecapitulationTest

V rámci tohoto testu se otestují na rekapitulační stránce některé zadané hodnoty z předchozích stránek. Tj. ověří se, zda jsou stejné, jako byly zadány.

Pro řešení toho to testu je třeba vytvořit nové třídy, které budou mít jako atributy položky z formulářů. Třídy jsou následující:

- ClientAdressInformation – v této třídě jsou atributy týkající se adresy klienta
- ClientEmployInformation – zde jsou atributy týkající se zaměstnání a zaměstnavatele
- ClientIncomeInformation – zde jsou atributy týkající se příjmů a bankovního účtu
- ClientPersonalInformation – obsahuje atributy týkající se osobních údajů klienta

Protože atributy jsou private, obsahují třídy metody *set()* a *get().* V předkovi třídy, která obsahuje veškeré testy, jsou funkce, které porovnávají jednotlivé elementy na rekapitulační stránce. Těmito funkcemi jsou *checkEmployInfo(), checkAdressInfo(), checkPersonalInfo()* a *checkIncomeInfo()*. Tělo jedné z nich je uvedeno v přílohách.

V testu se nejprve z první stránky rovnou přesměruje kliknutím na tlačítko na další stránku. Po té se zavolá funkce *fillAllPagesParametrized()*, která vrací instanci třídy představující rekapitulační stránku. Tato instance se později použije jako parametr při volání funkcí vracející *true* nebo *false*, porovnávající prvek obsahující hodnotu z rekapitulační stránky s hodnotou uloženou v objektu výše zmíněných tříd obsahující hodnoty z formulářů v atributech. Test je v přílohách.

Protože většina hodnot v selectech se zadávala pomocí metod, které volaly metodu WebDriveru *selectByIndex(),* musely se upravit, aby se v selectboxech volily hodnoty dle viditelného textu, tj. pomocí metody WebDriveru *selectByVisibleText().* Pak se snadno volí hodnoty, které se mají zadat do polí formuláře.

9 Srovnání řešení

V této kapitole stručně srovnám má řešení se současným řešením firmy. Zmíním jeho přínosy pro firmu.

9.1 Srovnání řešení procesu automatizace

Ve stávajícím procesu automatizace není žádný podproces, který by byl mezi přípravou testovacího scénáře a jeho automatizací (Implementace řešení TS). Jak již bylo psáno, není tak zaručena kvalita testovacího scénáře, natož jeho automatizace.

V navrhovaném řešení se zajišťuje kvalita scénářů hned několika způsoby. Prvním je změna přístupu k testování regresních testů. Se špatným přístupem k práci, kdy je obecnou myšlenku, že tyto testy nejsou tolik důležité, se tvoří nekvalitní scénáře a zvyšuje se chybovost. Dalším je stanovení zásad při tvorbě testovacích scénářů. Dosud bylo ve stávajícím procesu dodržováno jen málo zásad a neprobíhalo žádné test review. Což je další z nově zavedených podprocesů – Test review. To zaručí, že testovací scénář vidí ještě někdo další, kdo se ujistí, že scénář tak jak je, dává smysl a dodržuje stanovené zásady.

Důležitým podprocesem, který byl zaveden, je předání testovacího scénáře od týmu, který za něj zodpovídá. Toto předání proběhne mezi technickým testerem a zmíněným týmem. Ve stávajícím procesu si techničtí testeři scénář vezmou a začnou jej automatizovat, což způsobovalo spoustu chyb v testech ale také testování funkcionality, která se již nepoužívá a nebo funguje jinak. Případně se pužívala špatná testovací data. Tahle rizika jsou nyní minimalizována právě předáním scénáře, po kterém by měl technický tester scénáři rozumnět.

V rámci předání scénáře se také dohodne, zda scénář automatizovat či nikoliv. Aby to nebylo pouze na rozhodnutí technického testera, protože některé testy se nakonec neautomatizují jen proto, že pro ně nelze vygenerovat správná testovací data. Což ale týmy, zodpovídající za testovanou funkcionalitu, třeba dokážou obejít.

Dalším problémem ve stávajícím procesu byla nová funkcionalita. Ta totiž často ovlivňovala nebo úplně měnila stávající funkcionalitu a díky tomu padaly testy. Napříč týmy se totiž nedávalo vědět o tom, že se něco změnilo. Tudíž se zbytečně retestovaly spadlé testy a zjišťovaly se příčiny spadlých testů, což klidně zabralo několik mam-hours.

Proto je v návrhu evidovat tuto novou funkcionalitu ve speciálním dokumentu, který bude pro věchny dostupný. Každý si tam kdykoliv může ověřit, že v rámci jeho testy pokryté funkcionality nebyla žádná změna. Faktem je, že tyto kontroly budou prováděny spíše na konci release, před začátkem regresu. Ale stále je to lepší, provést maintanence testů dřív, než se spustí.

Protože často nebylo jisté, kdo vlastně za co odpovídá, byla navržena evidence testovavcího týmu pro automatizaci. Kromě toho také nutnost kontrolovat, zda některý z dalších commitů je oprava kódu, dle zadaných doporučení v code review. To by mělo zajistit lepší orieantaci v tom, kdo je za co odpovědný a na čem právě pracuje, ale také lepší kontrolu nad programovacím kódem testů.

Pro ulehčení v procesu automatizace, byla navržena automatizace vytváření jobů. Tj. že by joby vznikly společně se vznikem prostředí a s joby pro smoke testy. Již by se nečekalo na jejich vytvoření. Bohužel je to zde stále u nátlaku na management, aby ti dali vědět oddělení zodpovědným za automatizaci vytváření jobů pro smoke testy.

Posledním vylepšením oproti stávajícímu řešení je paralelní spouštění automatizovaných testů v rámci jobu. Nyní se testy spouští sériově, jeden za druhým na jednom slotu stroje. Navrhovaná změna spočívá ve vytvoření více suit testů, kdy pro každou z nich se vytvoří job a tak poběží více testů jedné aplikace najednou, protože budou běžet na různých slotech. Jejich běh bude tudíž jasně 2x rychlejší už při použití jedné suity testů navíc (dohromady tedy 2). Musí se však testy skupina testů rozdělit co nejblíže polovině.

9.2 Srovnání řešení automatizovaných testů Online půjčky

Automatizované testy Online půjčky byly původně velmi neefektivní. Všechny testy padaly na chyby testů z různých důvodů. Probíhaly zbytečné operace navíc, které v testu ani nebyly potřeba, což byla například volba výše úvěru a počtu splátek již na první stránce a na další stránce probíhala opět stejná volba.

Testy byly neaktuální a z 12 testů jich vždy alespoň 7 a více spadlo na chyby testu samotných, nikoliv na chyby aplikace. Všechny testy doběhly za 17 minut, kdy běžel jeden test za druhým, tj. běžely sériově. To jsou necelé 2 minuty na každý test.

V rámci řešení byly všechny automatizované testy porovnány s testovacími scénáři, po té byly spuštěny a bylo sledováno, co testy dělají ve skutečnosti. V rámci kódu byly opraveny chyby, kvůli kterým testy spadly. Současný stav je takový, že není test, který by spadl na chybu testu. Nicméně vzhledem k tomu, že Selenium WebDriver není bez vad, bohužel zatím stále nastávají případy, kdy se ztratí focus a WebDriver není schopen najít požadovaný prvek, případně jedna technologie je rychlejší než druhá. To je případ, kdy WebDriver vyplňuje data do položek ve formuláři, nicméně webová aplikace na toto nestihne zareagovat a tak se zadaná položka neprojeví.

Současné testy běží polovinu doby, protože běží paralelně na dvou slotech. Jsou tedy rychlejší. Pokud test spadne, tak ne na chybu testu, tj. nepořešený případ či špatně pochopený a automatizovaný testovací scénář, ale na chybu WebDriveru.

Navíc byly přidány automatizované testy, které testují funkcionalitu, která dosud nebyla testy pokryta. Tudíž oproti původnímu řešení se rozšířil rozsah testované oblasti a tudíž pravděpodobnějšímu předcházení chyb v ostré verzi.

10 Diskuze

Přebrané , již automatizované testy, byly v hrozném stavu. Kromě toho celý proces automatizace byl neefektivní a chybový v mnoha výše popsaných ohledech. Bylo jasné, že se s tím musí dříve nebo později něco dělat, což bylo původcem pro vznik této diplomové práce.

Použití automatizovaných testů ve firmě je však stále na počátku. V jiných firmách používají pokročilejší nástroje a jsou mnohem více využívány napříč všemi testery v průběhu vývoje software.

Navrhovaná řešení mohou být sebevíc efektivní, nicméně je třeba je protlačit do stávajícího procesu ve firmě. To zahrnuje komunikaci s managementem i s jednotlivými pracovníky. Změny v procesech firmy jsou vždy na delší čas, protože firma musí být samozřejmě opatrná a musí se zvážit všechna možná rizika, pokud jsou.

Otázkou je jak se k navrhovanému řešení postaví management. Práce má jasný přínos v předcházení veškerých zjištěných nedostatků v procesu automatizace ústící v chyby, jejichž řešení sebou nese náklady. Nicméně , jak už bylo výše zmíněno, veškeré změny ve větších procesech, jako je proces automatizace, se provádí delší čas, dokud se nezhodnotí všechna rizika a skutečné přínosy.

Opravené automatizované testy, které nyní již nepadají na chyby v nich samotných, jsou velkým přínosem také proto, že usnadňují vyhodnocování doběhlých jobů s automatizovanými testy. Protože to je jedna z činností zahrnutých v mé pracovní náplni, je tento přínos velmi vítaný.

10.1 Ekonomické zhodnocení

Díky tomu, že je nyní proces automatizace efektivnější, se ušetří čas na řešení chyb, které navíc díky změnám mají menší šanci vzniknout.

Protože každý testovaný scénář, včetně přípravy dat, je nyní předáván od týmu, který dané problematice rozumí, je nízká šance zadání chyby, která by vznikla z neporozumnění testovacímu scénáři či špatné přípravě dat.

Šetří se tak čas jak technickým testerům, kteří nemusí řešit, co je to za chybu a test spouštět znova, tak hlavně vývojářům, kteří by museli zjišťovat, proč se daná chyba stala. To může zabrat několik man-hours, které jsou takto ušetřeny. A protože jsou ušetřeny, mohou se využít efektivněji, lépe, například k vývoji nové funkcionality nebo opravě jiných, skutečných chyb.

Protože testy nepadají, nemusí se dělat tolik retestů. Tj. pokud spadne více jak 7 testů, musí se každý z nich pustit znova a to znamená, že kromě času, který si tyto testy vzaly při prvních průchodu, kdy byly spuštěny jobem, si vezmou ještě další čas navíc, v rámci retestů. Technický tester navíc jednotlivé testy musí sledovat, aby věděl, kde přesně spadly a poskytlo mu to další vodítka k zjištění příčiny. Je jasné, že takový čas by se dal využít lépe, ať už k automatizaci dalších testovacích scénářů a nebo k maintenance stávajících.

Zde popsaná řešení tudíž šetří celé man-days, protože retesty spadlých automatizovaných testů a zjišťování příčin zadaných chyb vývojářů se opakují v rámci release.

10.2 Přínos práce

Největší přínos z této práce má samozřejmě firma, pro kterou bylo řešení tvořeno. Toto řešení zmiňovaných problémů se týká metodiky automatizace testů. Také jsou zde upraveny automatizované testy tak, aby byly spolehlivější a fungovaly a zároveň je rozšířen rozsah testované funkcionality přidáním nových testů.

Odstranění chyb automatizovaných testů je dalším přínosem práce. Nejen, že takto upravené testy ušetří čas vyhodnocování, ale rovněž inovace v podobě možnosti spouštět testy paralelně, vede k efektivnějšímu využití celkového času.

Práce rovněž zobrazuje reálné procesy probíhající ve firmě. Tyto procesy popisuje a kriticky hodnotí. Čtenáři se tak naskytne možnost vidět, jak vývoj a proces automatizace probíhá a co obnáší. Protože jsou zde uvedeny i testy webové aplikace, mohou vést k inspiraci vytvoření vlastních testovacích scénářů webové aplikace. Také jsou zde uvedeny technologie, které se k automatizaci používají, což může opět vést k inspiraci k vlastnímu řešení čtenáře.

Dalším přínosem práce je i část, která pojednává o testování a automatizovaném testování. Jsou zde vysvětleny základní pojmy, uvedeny důvody, proč firmy volí automatizaci testů a různé mýty s tím spojené.

10.3 Možnosti rozšíření práce

V práci je uvedeno několik tématických celků, které však mohou být dále rozšířeny. Týká se to hlavně kapitol o testování obecně a také automatizovaném testování. Nabízí se například různé metriky pro měření kvality. Dále je zajímavé vysvětlení, proč by programátoři neměli zároveň i testovat svůj kód, tedy vykonávat roli testerů. Také jak vypadá správný tester, tj. jaké má vlastnosti, které firmy vyhledávají. Další témata mohou být například:

- Podoba testovacích návrhů a jak vlastně přesně vypadají a řeší se testovací scénáře. V různých firmách sice vypadají tetovací scénáře různě, nicméně mají společné prvky, které by bylo možné najít a analyzovat.

- Reportování zadaných chyb, jaké mají důležité atributy. Jaké nástroje se pro zadávání chyb používají a jejich srovnání z různých hledisek.

- Nástroje pro automatizaci testů webových aplikací. Jaké programovací jazyky se nejčastěji používají a proč. Jaké nástroje kromě Selenium WebDriver existují a jejich srovnání, tj. výhody a nevýhody oproti jiným, cenové srovnání, podporované platformy a další.

- Automatizované testy v rámci nefunkčního testování – jaké testy se provádějí, co se testuje a proč. Zmíněné testy Online půjčky byly vše funkční testování. V rámci nefunkčního testování se opět používají různé nástroje, takže se nabízí je popsat a srovnat.

- Vývoj řízený testy – co to je, jak do toho zařadit automatizované testy, jaké jsou výhody a proč ho ne/používat
- Agilní vývoj – dnes velmi moderní způsob vývoje. Co to znamená pro testy, jaké jsou zde role pro testery a jaké nové role v rámci tohoto druhu vývoje vznikají. Ve firmě se tento druh vývoje též používá.

11 Závěr

V této práci je rozebrána problematika testování a automatizace. Jsou vysvětleny oba pojmy i jejich vzájemná provázanost. Práce se věnuje rovněž pojmům jako chyba či životní cyklus testování. Také jsou zde zmíněny důvody, proč firmy zavádějí automatizaci testů a jaké jsou z toho plynoucí výhody a nevýhody.

Je provedena analýza procesu automatizace firmy EmbedIT i již vytvořených automatizovaných testů aplikace Online půjčka. Je zachycen a popsán každý jednotlivý podproces procesu automatizace, tj. od přípravy testovacích scénářů po zadání chyby či přidání nové funkcionality. Jsou zde zmíněny všechny vyplývající problémy v jednotlivých podprocesech, které spolu mnohy napříč podprocesy souvisí.

Dále jsou rozebrány všechny testy aplikace Online půjčky. Je zde popsáno, co zhruba dělají, co je jejich cílem otestovat a kde jsou v nich zjištěné problémy. Tyto problémy byly zjištěny ať už při srovnání s testovacím scénářem, kontrolou kódu a nebo po spuštění.

V rámci analýzy byly zjištěny nedostatky, které má za cíl odstranit návrh řešení a implementaci úprav automatizovaných testů i samotného procesu automatizace. Byly přidány další podprocesy do procesu automatizace a také bylo navrženo řešení pro zrychlení běhu automatizovaných testů v rámci jobů. Též byly přidány a naprogramovány automatizované testy rozšiřující rozsah testované funkcionality aplikace.

Výsledkem by měl být efektivnější proces automatizace, šetřící čas technickým testerům i vývojářům a také spolehlivější automatizované testy.

Dále je provedeno srovnání stávajícího a navrhovaného řešení a jejich efektivnosti.

V kapitole Ekonomické zhodnocení je zmíněn ekonomický přínos pro firmu, hlavně v již zmiňované úspoře času díky spolehlivějšímu průběhu testů a jejich menší chybovosti.

V poslední části je rozebrán přínos této práce, tj. kdo informace, návrhy řešení a implementaci může využít.

12 Literatura

[1] HAYES L. G. *The Automated Testing Handbook* [online]. 2010. [cit. 2015-11-26]. Dostupné z: http://www.softwaretestpro.com/itemassets/4772/automated-testinghandbook.pdf

[2] ZELINKA B. *Unicorn systems: Testování softwaru* [online]. 2013. [cit. 2015-11-26]. Dostupné z: http://d3s.mff.cuni.cz/teaching/commercial_workshops/previous/1213/zelinka-zajisteni_kvality_softwarovych_produktu.pdf

[3] JENKINS N. *A Software Testing Primer: An Introduction to Software Testing* [online]. 2008. [cit. 2015-11-26]. Dostupné z: http://www.nickjenkins.net/prose/testingPrimer.pdf

[4] DIJKSTRA E. W. *The Humble programmer* [online]. 1972. [cit. 2015-11-26]. Dostupné z: *http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF*

[5] HAILPERN B., SANTHANAM P. *IBM Research Report: Software Debugging, Testing,* and Verification [online]. 2001. [cit. 2015-11-26]. Dostupné z: http://cse.yeditepe.edu.tr/~tserif/fall2009/cse544/papers/SoftwareDebuggingTestingandVerification.pdf

[6] ROUDENSKÝ P., HAVLÍČKOVÁ A. *Řízení kvality softwaru: Průvodce testováním.* Computer Press, 2013. EAN 9788025138168

[7] GRAHAM D., VEENENDAAL E., BLACK R., EVANS I.. FOUNDATIONS OF SOFTWARE TESTING. ISTQB CERTIFICATION. [online]. 2015 [cit. 2015-11-27]. Dostupné z: http://www.computing.dcu.ie/~ray/teaching/CA358/dorothy_graham.pdf

[8] *SOFTWARE TESTING TUTORIAL. Tutorials Point*. [online]. 2015 [cit. 2015-11-27]. Dostupné z: http://actoolkit.unprme.org/wp-content/resourcepdf/software_testing.pdf

[9] PATTON R., *Software Testing*. Indiana: SAMS, 2001. ISBN 0-672-31983-7

[10] SATHISH C. G.,. *SOFTWARE TESTING: Basics of Software Testing-I* [online]. 2012. [cit. 2015-11-27]. Dostupné z: http://elearning.vtu.ac.in/12/enotes/Soft_Test/Unit1-SV.pdf

[11] BOROVCOVÁ A., *Testování webových aplikací* [online]. 2001. [cit. 2015-11-27]. ISBN . Dostupné z: http://www.cssi.cz/cssi/system/files/all/si_2011_01_08_Borovcova.pdf

[12] *Defect Life Cycle. Tutorials Point.* [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://www.tutorialspoint.com/software_testing_dictionary/defect_life_cycle.ht

[13] *Defect/Bug Life Cycle – A Tester's Guide. Guru99.* [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://www.guru99.com/defect-life-cycle.htmlm

[14] PRADHAN D. *Top 10 reasons why there are Bugs/Defects in Software!. Software Testing tricks*. [online]. 2008 [cit. 2015-11-28]. Dostupné z:http://www.softwaretestingtricks.com/2008/12/why-are-bugsdefects-in-software.html

[15] *Why Does Software Have Bugs. Software Testing Help*. [online]. 2015 [cit. 2015-11-28]. Dostupné z: http://www.softwaretestinghelp.com/why-does-software-have-bugs/

[16] *Software testing life cycle. SlideShare*. [online]. 2013 [cit. 2015-11-28]. Dostupné z: http://www.slideshare.net/p2cinfotech/software-testing-life-cycle-28441116

[17] *SOFTWARE TESTING LIFE CYCLE (STLC). Software Testing Fundamentals*. [online]. 2012 [cit. 2015-11-28]. Dostupné z: http://softwaretestingfundamentals.com/software-testing-life-cycle

[18] *What is Software Testing Life Cycle (STLC). Software Testing Help.* [online]. 2015 [cit. 2015-11-28]. Dostupné z: http://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc/

[19] HLAVA T. *Funkční a nefunkční testy. Testování softwaru*. [online]. 2011 [cit. 2015-11-28]. Dostupné z: http://testovanisoftwaru.cz/druhy-typy-a-kategorie-testu/funkcni-a-nefunkcni-testy/

[20] VEENENDAAL, VAN E. *Standard glossary of terms used in Software. International Software Testing Qualification Board*. Version 1.3 31.5.2007, 39 s.

[21] *Software Testing Class. Smoke Testing.* [online]. 16.12.2012 [cit. 2015-11-19]. Dostupné z: http://www.softwaretestingclass.com/smoke-testing/

[22] *Základy automatizace testování. SW Testování*. [online]. 2015 [cit. 2015-11-27]. Dostupné z: http://www.swtestovani.cz/index.php/uvod-do-testovani/40-zaklady-automatizace-testovani“

[23] ROWE S. *What is Test Automation?* MSDN blogs. [online]. 19.11.2007 [cit. 2015-11-19]. Dostupné z: http://blogs.msdn.com/b/steverowe/archive/2007/12/19/what-is-test-automation.aspx

[24] *Automatizované testování. Testování softwaru*. [online]. 19.11.2015 [cit. 2015-11-19]. Dostupné z: http://testovanisoftwaru.cz/automatizovane-testovani/

[25] *Automated Testing: Process, Planning, Tool Selection. Guru99.* [online]. 2014 [cit. 2015-11-28]. Dostupné z: http://www.guru99.com/automation-testing.html

[26] *QTP – Automated Testing Process. Tutorials Point.* [online]. 2015 [cit. 2015-11-28]. Dostupné z: http://www.tutorialspoint.com/qtp/qtp_test_automation_process.htm

[27] *Why, How and When to Automate Software Testing. Software Testing Class*. [online]. 2014 [cit. 2015-11-28]. Dostupné z: http://www.softwaretestingclass.com/why-how-and-when-to-automate-software-testing/

[28] *Effective Automated Testing. PerlMonks.* [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://www.perlmonks.org/?node=Effective+Automated+Testing

[29] *Marick B. When Should a Test Be Automated?* [online]. 1998. [cit. 2015-11-30]. Dostupné z: http://www.exampler.com/testing-com/writings/automate.pdf

[30] *To automate or not to automate. Qualitia*. [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://www.qualitiasoft.com/resources/automate-or-not-to-automate/

[31] *10 Tips you should read before automating your testing work. Software Testing Help.* [online]. 2015 [cit. 2015-11-30]. Dostupné z:http://www.softwaretestinghelp.com/10-tips-you-should-read-before-automating-your-testing-work/

[32] *What to automate and what not to automate. Agile Tester*. [online]. 2012 [cit. 2015-11-30]. Dostupné z: https://devmaheshwari.wordpress.com/2012/03/03/what-to-automate-and-what-not-to-automate/

[33] *What Is Automated Software Testing. Innovative Defense technologies*. [online]. 2015 [cit. 2015-11-28]. Dostupné z: http://idtus.com/what-is-automated-software-testing/

[34] *Determining What to Automate. Galmont Colsunting*. [online]. 2015. [cit. 2015-11-30]. ISBN . Dostupné z: http://galmont.com/wp-content/uploads/2013/11/Determining-What-to-Automate-2013_11.13.pdf

[35] RODEN L. *Choosing and Managing the Ideal Test Team. Methods and tools*. [online]. 2005 [cit. 2015-11-30]. Dostupné z: http://www.methodsandtools.com/archive/archive.php?id=30

[36] *Software Testing Class. GitHub.* [online]. 2015 [cit. 2015-11-19]. Dostupné z: https://github.com/junit-team/junit

[37] *Junit-team/junit : Categories*. *GitHub*. [online]. 2015 [cit. 2015-11-19]. Dostupné z: https://github.com/junit-team/junit/wiki/Categories

[38] Java Code Geeks. *JUnit Suite Test Example*. [online]. 2013 [cit. 2015-11-19]. Dostupné z: http://examples.javacodegeeks.com/core-java/junit/junit-suite-test-example/

[39] Vogella. *Unit Testing with JUnit – Tutorial*. [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://www.vogella.com/tutorials/JUnit/article.html

[40] *Selenium WebDriver. Selenium HQ.* [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://docs.seleniumhq.org/projects/webdriver/

[41] *Selenium WebDriver: Introducing WebDriver*. *Selenium HQ.* [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://docs.seleniumhq.org/docs/03_webdriver.jsp

[42] *Introduction. Apache Maven Project.* [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://maven.apache.org/what-is-maven.html

[43] *What is the POM? Apache Maven Project.* [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://maven.apache.org/pom.html#What_is_the_POM

[44] *Test Design Considerations: Page Object Design Pattern*. Selenium HQ. [online]. 2015 [cit. 2015-11-19]. Dostupné z: http://docs.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern

[45] *Obecné atributy. Jak psát web.* [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://www.jakpsatweb.cz/html/obecne-atributy.html#id

[46] *Selenium HQ. Selenium WebDriver: Locating UI Elements (WebElements).* [online]. 2015 [cit. 2015-11-30]. Dostupné z: http://docs.seleniumhq.org/docs/03_webdriver.jsp#locating-ui-elements-webelements

Přílohy

Maven projekt, pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>***</groupId>
        <artifactId>root-project</artifactId>
        <version>1.13</version>
    </parent>

    <groupId>***.cs-web-parent</groupId>
    <artifactId>cs-web-parent</artifactId>
    <version>1.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>cs-web-test</module>
        <module>cs-web-data</module>
        <module>cs-web-dao</module>
        <module>cs-web-infrastructure</module>
    </modules>

    <dependencies>
      <dependency>
         <groupId>***</groupId>
         <artifactId>end-to-end-test-support</artifactId>
         <version>${end-to-end-test-support}</version>
      </dependency>

    </dependencies>
```

Zde jde vidět povinný začátek, pak informace o verzi modelu. Dále jsou identifikátory projektu:

- groupID – unikátní identifikátor organizace a nebo skupiny, která vytvořila projekt
- artifactId – unikátní název primárního artefaktu, generovaného projektem
- version – verze projektu
- packaging – jaký zabalený soubor se má vlastně vytvořit. Pokud je nastaveno pom (jako v tomto případě), pak bude výsledným souborem JAR Přidáním *<packaging>war</packaging>* by byl výsledný balíček typu WAR.

Část kódu z testu **secondPageCalculationTest**:

```
collector.checkThat("Actual chosen time on detail page is not same
like time on first page",detailCalcPage.getSettedTime(), is(Inte-
ger.parseInt(getResource("ILOAN_FIRST_PAGE_TIME"))));
```

Část kódu z testu **smsCodeVerificationWrongCaptchaTest**:

```
recapPage.fillAndSubmitSmsCode("d5d4");
collector.checkThat("SMS code is not recognized as wrong", recap-
Page.isErrorMessagePresent(), is(true));

public boolean isErrorMessagePresent()  {
    return isElementPresentBoolean((By.xpath("//div[@class='inner
clearfix threeStep']/div[@class= 'step2 active']/div[@style='float:
left; color: #FF0000; width:160px;']")));
}
```

Část kódu z testu **chooseAndSubmitAlternativeTest**:

```
protected void acceptAlert()
{
    Alert alert = getWebDriver().switchTo().alert();
    alert.accept();
}
```

Část kódu z testu **CheckTextOnFirstPageTest**:

```
collector.checkThat("Text in how to create loan is not as expected",
loanFirstPage.getTextFromHowToCreateLoan(), containsString(getRe-
source("ILOAN_FIRST_PAGE_HOWTO")));

collector.checkThat("Text in info about SF is not as expected", loan-
FirstPage.getTextFromOnlineLoan(), containsString(getRe-
source("ILOAN_FIRST_OL_SF")));

collector.checkThat("Text in what to prepare is not as expected", lo-
anFirstPage.getWhatPrepare(), containsString(getRe-
source("ILOAN_FIRST_WHATPREPARE")));

collector.checkThat("Text in box with call number is not as expected",
loanFirstPage.getBoxWithCall(), containsString(getRe-
source("ILOAN_CALL")));

loanFirstPage.ClickOnLinkToHC();


String originalWindow = getWebDriver().getWindowHandle();
Set<String> setOfWindows =  getWebDriver().getWindowHandles();
for(String windowW  : setOfWindows)
{
    if(!windowW.equals(originalWindow))
    {
        getWebDriver().switchTo().window(windowW);
        hcpage = loanFirstPage.goToHCpage();
```

```
        getWebDriver().close();
        getWebDriver().switchTo().window(originalWindow);
    }
}
```

Jednotlivé metody volané v collectorech v rámci testu **CheckTextOnFirstPageTest**:

```
public String getTextFromHowToCreateLoan() {
    String text = "";

    WebElement iframe = getWebDriver().findEle-
ment(By.xpath("//div[@class = 'hcaddinfo clearfix']/iframe"));
    getWebDriver().switchTo().frame(iframe);

    text = howToCreateLoan.getText();

    getWebDriver().switchTo().defaultContent();

    return text;
}

public String getTextFromOnlineLoan() {
    String text = "";

    WebElement iframe = getWebDriver().findElement(By.xpath("//div[@id
= 'hclfrag']/iframe"));
    getWebDriver().switchTo().frame(iframe);

    text = onnlineLoanText.getText() + sfInfoLi.getText() + sfIn-
foUl.getText();

    getWebDriver().switchTo().defaultContent();

    return text;
}

public String getWhatPrepare() {
    String text = "";

    WebElement iframe = getWebDriver().findElement(By.xpath("//div[@id
= 'hclfrag']/iframe"));
    getWebDriver().switchTo().frame(iframe);

    text = whatPrepare.getText();

    getWebDriver().switchTo().defaultContent();

    return text;
}

public String getBoxWithCall() {
    String text = "";

    WebElement iframe = getWebDriver().findElement(By.xpath("//div[@id
= 'hcrfrag']/iframe"));
```

```
    getWebDriver().switchTo().frame(iframe);

    text = call.getText();

    getWebDriver().switchTo().defaultContent();

    return text;
}
```

Ukázka definice prvku na stránce v rámci testu **CheckTextOnFirstPageTest**:

```
@FindBy(xpath = "//*[@id='hccalc-fragment']/div[2]/ul")
private WebElement howToCreateLoan;
```

Část kódu z testu **CheckDeliveryTest:**

```
detailCalcPage.selectRepayment(getResource("ILOAN_DELIVERY_POST"));

collector.checkThat("There is more delivery type, and should be only
bank account", detailCalcPage.getAllRepayments(), is(getRe-
source("ILOAN_DELIVERY_BA")));

public ILoanDetailCalcPage selectRepayment(String delivery)
{
    createSelectFromWebElement(slrepayment).selectByVisibleText(deli-
very);
    return addPageToFactory(this.getClass());
}

public String getAllRepayments() {
    String text = "";

    text = slrepayment.getText();

    return text;
}

@FindBy (xpath = "//div[@id = 'repaymentwrap']/label/select")
private WebElement slrepayment;
```

Část kódu z testu **CheckChosenCashAmountInstalmentsTest**:

```
collector.checkThat("Chosen cach and month are not the same on perso-
nal info page and detail page",
        getResource("ILOAN_MIN_INSTALMENT_CASHKC") + getRe-
source("ILOAN_MAX_INSTALMENT_MONTHT") + getResource("ILOAN_MAX_INSTAL-
MENT"), containsString(super.getHeadCashMonth()));



protected String getHeadCashMonth()
{
    String month, cash, instalment;
```

```
    WebElement cashE = getWebDriver().findElement(By.xpath ("//div[@id
= 'container']/div[@id = 'hccalc']/div/div[2]/p[@class ='hcattrs
clearfix']/span[1]"));
    WebElement monthE = getWebDriver().findElement(By.xpath
("//div[@id = 'hccalc']/div/div[2]/p/span[2]"));
    WebElement instalmentE = getWebDriver().findEle-
ment(By.xpath("//div[@id = 'hccalc']/div/div[2]/p/span[3]"));

    cash = cashE.getText();
    month = monthE.getText();
    instalment = instalmentE.getText();

    return cash + month + instalment;
}

@FindBy(xpath = "//input[contains(@id,'backButton')]")
private WebElement btnBack;

public ILoanPersonalInfoPage onBackClick() {

    btnBack.click();

    return addPageToFactory(LoanPersonalInfoPage.class);
}
```

Test **checkRecapitulationTest()**:

```
@Category(***.class)
@Test
public void CheckRecapitulationTest() throws Exception
{
    loanFirstPage = (LoanFirstPage) initBaseTestAndNavigateToMa-
inPage(Constants.DEFAULT_AUTHENTICATION_TIMEOUT);

    recapPage = super.fillAllPagesParametrized(loanFirstPage, ***
phone, "48", clientDao.getIdentInSolus(), ***, getResource(***), ***,
true);

    collector.checkThat("There are some fields in personal page that
not same", super.checkPersonalInfo(recapPage), is(true));

    collector.checkThat("There are some fields in personal page in ad-
ress part that not same", super.checkAdressInfo(recapPage), is(true));

    collector.checkThat("There are some fields in income page in em-
ploy part that not same", super.checkEmployInfo(recapPage), is(true));

    collector.checkThat("There are some fields in income part in in-
come and financial part that not same", super.checkIncomeInfo(recap-
Page), is(true));
```

```
}
```

Funkce z testu **checkRecapitulationTest():**

```
protected boolean checkEmployInfo(ILoanRecapitulationPage recap)
{
    boolean value = true;
    int controll = 0;

    String date = recap.getLiEmployedFrom().getText().replace(".",
"/").substring(2);

    if(!date.equals(employInfo.getEmployFromYear())){
        logger.info("Employed from is not same");
        controll++;
    }

    if(!recap.getLiSector().getText().equals(employInfo.getSector())){
        logger.info("Sector is not same");
        controll++;
    }

    if(!recap.getLiSphere().getText().equals(employInfo.getSphere())){
        logger.info("Sphere is not same");
        controll++;
    }

    if(!recap.getLiEmName().getText().equals(employInfo.getEmployer-
Name())){
        logger.info("Employer name is not same");
        controll++;
    }

    if(!recap.getLiEmStreetNumber().getText().equals(employerAdres-
sInfo.getStreetNumber())){
        logger.info("Employer street number is not same");
        controll++;
    }

    if (controll != 0)
        value = false;

    return value;
}
```