Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Test analýza s ohledem na vhodnost automatických testů

Plzeň 2020 Martin Bouše

Místo této stránky bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 7. května 2020

Martin Bouše

Abstract

Test analysis with regard to the suitability of automatic tests aims for creation of automated tests based on already existing manual tests. Criteria of whether or not a creation of automated test will have any positive impact are suggested first. Then a selection of tests is automated and their efficiency is evaluated in the end.

Abstrakt

Tato bakalářská práce je zaměřena na výrobu automatických testů na základě již existujících manuálních testů. Nejdříve jsou navrhnuta kritéria, dle kterých se lze rozhodnout, má-li vytváření automatického testu smysl. Dále jsou vybrané automatické testy implementovány a na závěr je zhodnocena jejich efektivita výroby, běhu, aj.

Obsah

1 Úvod

Testování je důležitou součástí vývoje aplikací. V naprosté většině případů se výsledné produkty testují ještě před dokončením. Ovšem některé aplikace vyžadují soustavné testování i po jejich dokončení. To je způsobeno přidáváním nové funkčnosti, nebo opravováním chyb, ať těch vzniklých a/nebo neodhalených během vývoje, nebo těch, které byly vytvořeny přidáním nové funkčnosti, či opravou jiné chyby. Chyb se tedy v každém kódu vyskytuje mnoho a nikdy není jasné, kdy se objeví nová chyba a jak bude rozsáhlá.

Testování, zda byla chyba odstraněna, lze provádět manuálně, nebo automaticky. Manuální testování neustále vznikajících chyb je v tomto ohledu náročné jak na pracovníky, tak na finance. Řešením takové situace mohou být automatické testy. Použití automatických testů ale nemusí znamenat snížení nákladů na testování v každém případě. Při zvolení nesprávné strategie může znamenat několikanásobné zvětšení těchto nákladů.

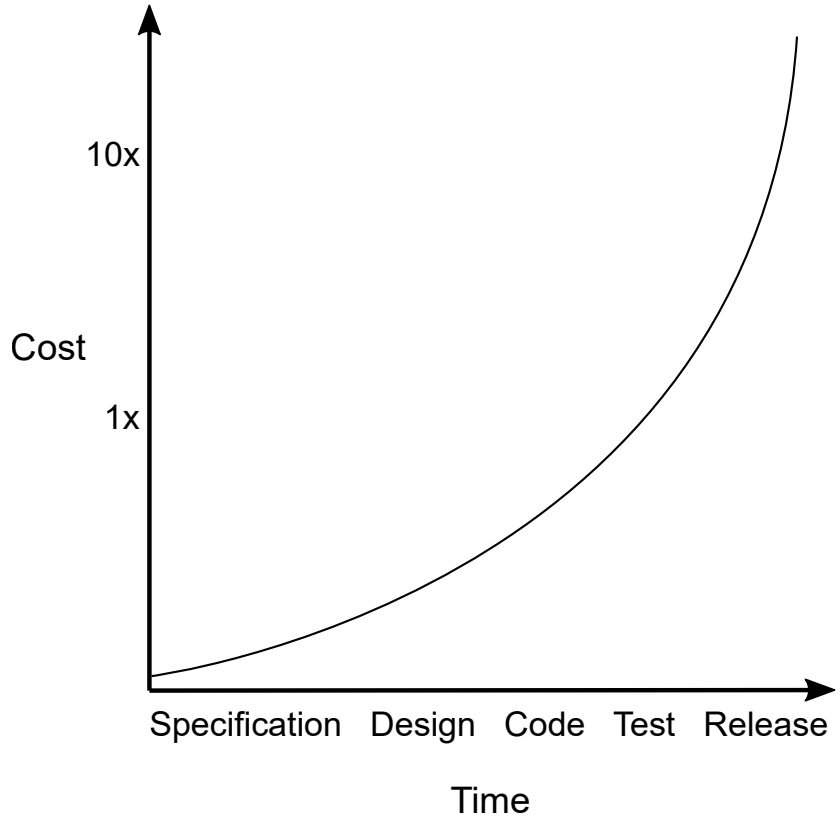
Cílem této práce je, navrhnout kritéria hodnocení přínosu automatických testů, implementovat vybrané automatické testy na základě již existujících manuálních testů a následné zhodnocení efektivity těchto automatizovaných testů z hlediska náročnosti jejich vyrobení, spuštění, případně dalších vhodných hledisek.

2 Testování

Testování je proces, jehož cílem je odhalení defektů (= následek chyby např. ve specifikaci či v kódu), které se v aplikaci nacházejí. Jedná se o úlohu, na kterou bývá spotřebováno 30 až 50 % [4] z celkového rozpočtu na vývoj. Jedná se tedy o finančně náročnou úlohu.

2.1 Proč testovat

Hlavním důvodem jsou náklady. Čím dříve je defekt odhalen, tím menší jsou náklady na jeho opravu [4].



Obrázek 2.1: Náklady na opravu defektu v čase (zalaoženo na [3], [4])

Tato závislost je graficky znázorněna na obrázku 2.1. Např. náklady na opravu defektu odhaleného ve fázi specifikace budou nižší (cena zahrnuje pouze čas potřebný na opravu), než náklady na opravu defektu, který odhalil až zákazník při běžném provozu (zde se konečná cena může skládat nejen

z opravy samotného defektu, ale také z možné náhrady škod, které byly tímto defektem zákazníkovi způsobeny).

2.2 Náklady na testování po dokončení vývoje

Tyto náklady jsou závislé na typu vyvíjené aplikace: krabicové řešení nebo aplikace na zakázku.

2.2.1 Krabicové řešení

Tato aplikace je vytvářena firmou, např. na základě poptávky na trhu, a dále jednorázově prodávána zákazníkům/uživatelům. Kupující zaplatí pouze jednou a může aplikaci plně využívat.

Nejčastěji takováto aplikace není dále rozšiřována a nejsou tedy očekávány náklady na testování po dokončení vývoje. To ale neznamená, že výsledná aplikace je bez defektů. Pokud je nahlášen defekt v aplikaci, měl by být, podle závažnosti, nalezen a opraven, celá aplikace znovu otestována a v neposlední řadě doručena všem zákazníkům (kteří si tuto aplikaci zakoupili). Celý tento proces může znamenat ztrátu reputace a vyžaduje nemalé náklady na uskutečnění, kde část těchto nákladů je využita na testování.

2.2.2 Na zakázku

Taková aplikace je vyvíjena firmou po přímém kontaktování zákazníkem, nebo např. po získání zakázky z veřejné soutěže. Zde je aplikace vytvářena na míru a dá se tedy očekávat požadavek dlouhodobé podpory ze strany zákazníka (pokud není, jedná se vlastně o krabicové řešení). Protože se jedná o požadavek zákazníka, taková podpora je nejčastěji zajištěna formou měsíční paušální platbou, která pokrývá náklady na údržbu a rozvoj aplikace a opravu a testování defektů.

Cena testování takové aplikace každým měsícem narůstá. Aby tento měsíční nárůst byl co nejmenší, je velmi důležitá volba správné testovací strategie již v prvních fázích vývoje.

2.3 Snižování nákladů

Pří vývoji aplikace se firma snaží vyrobit aplikaci v co nejlepší kvalitě za co nejmenší náklady. To samé platí i pro údržbu. Přidat funkce, které zákazník

nebo zákazníci požadují za co nejmenší cenu.

Je zcela jasné, že tato snaha o co nejmenší náklady se týká i testování. Jedná-li se o krabicové řešení, jsou náklady na testování vlastně jednorázové a nevyplatí se investovat do nových možností testování. Ovšem pokud jde o aplikaci na zakázku, kde se testuje neustále a často se jedná o ty samé testy prováděné znovu a znovu, tak se již investovat do způsobu testování vyplatí.

2.3.1 Manuální testování

Manuální testy, jsou testy, které jsou vykonávány člověkem, takzvaným testerem. Nejčastěji se skládají ze sady scénářů (podrobný popis toho, jak má aplikací tester prostupovat - na co kliknout, do jakého pole zadat jaké hodnoty, atd.), podle nichž tester postupně prochází a testuje celou aplikaci, nebo její změněnou část.

Každý takový scénář obsahuje až několik desítek kroků. Vykonávání takového scénáře může trvat i několik hodin. K tomu je nutno započítat přípravu aplikace a potřebných dat pro vykonání samotného testu a pokud příprava zahrnovala např. databázové změny, nebo vytvoření nových entit, měla by být databáze uvedena do stavu před provedením těchto změn.

Někdy jsou ale manuální testy jediným způsobem, jak aplikaci, nebo její část, otestovat. Například testy aplikace na různých platformách. Každá platforma má jiné vlastnosti a chování. Dalším příkladem jsou testy, kde je testovaná aplikace závislá na připojeném zařízení. Toto zařízení může být v mnoha případech nemožné simulovat.

Manuální testy jsou náročné na zdroje, jak lidské tak finanční, ale někdy mohou být levnější, než jiný způsob testování a některé aplikace nemohou být otestovány jinak. U těch aplikací, které jsou firmou dlouhodobě vyvíjené a podporované lze říci, že dlouhodobé manuální testování vede ke ztrátě možného zisku z důvodu náročnosti manuálních testů na zdroje (s každým provedeným testem vzrůstá cena testování). Tuto ztrátu je možné, z dlouhodobého hlediska, snížit pomocí automatických testů. Dále bude o testování mluveno v kontextu aplikace s dlouhodobou údržbou.

3 Automatické testování

Tyto testy jsou často založené na nějakém manuálním testu. Proces vytvoření automatického testu na základu manuálního testu se nazývá automatizace a vzniklý test se nazývá automatizovaný nebo automatický [4]. Takto vzniklý test je test, který byl vytvořen člověkem, ale samotný průběh testu je řízen počítačem.

Jak bylo řečeno dříve, ne všechny manuální testy mohou být automatizované, ale existují typy testů, které jsou ideálními kandidáty na automatizaci [4].

3.1 Ideální kandidáti

3.1.1 Jednotkové testy

Jsou určené pro testy jednotlivých částí aplikace. U malých systémů může jít o test každé metody, což by bylo u rozsáhlých systémů velmi náročné, proto se častěji používají pro testy logických celků. Například test automatického vyplnění pole ve formuláři. Testuje se, zda je vyplněná hodnota očekávanou hodnotou, nikoliv jednotlivé části získávání této hodnoty (dotaz do databáze, parsování, aj.).

Lze je dále rozdělit na pozitivní a negativní testy. Pozitivní testy jsou testu, u kterých jsou zadány validní hodnoty a je očekáván validní výsledek (např. do kalkulačky zadám validní hodnoty „1+2“ a očekávám výsledek „3“). Negativní testy testují reakce aplikace na nevalidní data. Pokud jsou zadána nevalidní data, aplikace by to měla rozpoznat a zobrazit odpovídající zprávu (např. do pole formuláře PSČ vložím text a očekávám hlášku o nevalidním vyplnění pole formuláře).

3.1.2 Integrační testy

Jejich cílem je otestovat, zda všechny moduly aplikace spolupracují tak, jak by měly. Například webová služba poskytuje seznam hodnot. Tento test zjistí, zda-li je poskytnutý seznam hodnot očekávaný seznam hodnot a zda webová stránka volající tuto webovou službu volá správnou webovou služby a jestli je obdržený seznam hodnot správně zpracován.

3.1.3 Regresní testy

Asi největší kandidát na automatizaci. Tyto testy jsou prováděny po každé přidané funkčnosti nebo po každém opraveném defektu. Jejich cílem je zjistit, jestli změnou v aplikaci nebyly ovlivněny jiné části aplikace (tj. přestanou správně fungovat).

3.1.4 Smoke testy

Spouštěny před všemi ostatními testy. Slouží pro kontrolu, že aplikace obsahuje správné verze všech modulů. Pokud tyto testy selžou, je aplikace vrácena programátorům a dále není testována.

3.2 Délka běhu testů

Mohlo by se zdát, že s pomocí automatizovaných testů jsou testy dokončeny během několika sekund, maximálně několika minut. Toto však obecně neplatí. Integrační a regresní testy kvůli množství testů, které je nutno vykonat pro pokrytí celé aplikace a zajištění spolehlivosti, mohou stále po procesu automatizace trvat i několik hodin (pořád se ale jedná o několikanásobné zkrácení doby provádění testů oproti manuálním testům). Oproti tomu u jednotkových testů je rychlý běh očekáván, protože tyto testy jsou nejčastěji spouštěny programátorem testujícím úpravy, které právě provedl a čím dříve obdrží výsledek testu, tím dříve může na základě tohoto výsledku pokračovat v úpravách aplikace.

3.3 Úspora času testera

Automatické testy šetří velké množství času testera v porovnání s vykonáváním manuálního testu. To ale neznamená, že po dobu běhu testů nemá tester co dělat. Během této doby se tester věnuje vytvářením dalších automatických testů, přípravou a úpravou již existujících testů, nebo se věnuje větším a důležitějším problémům vyvíjené aplikace [1].

3.3.1 Znovupoužitelnost testu

Jakmile je automatický test jednou vytvořen, lze jej opakovaně spouštět. To také napomáhá k velké úspoře času. Pokud je potřeba test znovu provést, není nutno procházet celý scénář automatického testu, ale pouze se spustí jeho automatizovaná verze a tester se může věnovat ostatním problémům.

3.4 Simulace dat

Automatizované testy umožňují testování v raných fázích vývoje. Díky tomu je možné zkontrolovat funkce aplikace, které by byly složité až nemožné otestovat pomocí manuálního testování (např. funkce při stisknutí tlačítka, ale samotné tlačítko ještě není implementováno).

3.4.1 Mockování

Podobně lze otestovat funkce, které jsou závislé na jiném modulu, ale tento jiný modul ještě nemusí existovat. Této technice se říká „mockování“ a jedná se o nahrazení skutečného objektu objektem falešným [6]. Tento falešný objekt nejčastěji pouze předstírá požadovanou funkčnost a tím umožňuje testování objektu na něm závislých (např. volání webové služby, pro kterou není vytvořena stránka využívající tuto službu, pomocí mocku lze nasimulovat volání této webové služby a ověřit tak správnost služby).

3.4.2 Izolace testu

Mockování se také používá k tzv. izolaci testu [6]. Pomocí izolace se lze zaměřit přímo na testovaný kód. Není třeba se zaobírat vedlejšími funkcemi, které jsou nepodstatné pro tento test. Např. test zobrazování hodnot seznamu. Cílem testu je zkontrolovat, jestli jsou zobrazeny správné hodnoty. Na konkrétních hodnotách nezáleží, takže mohou být nahrazeny falešnými, které jsou získané ze souboru místo z databáze nebo jsou zapsány přímo v kódu testu.

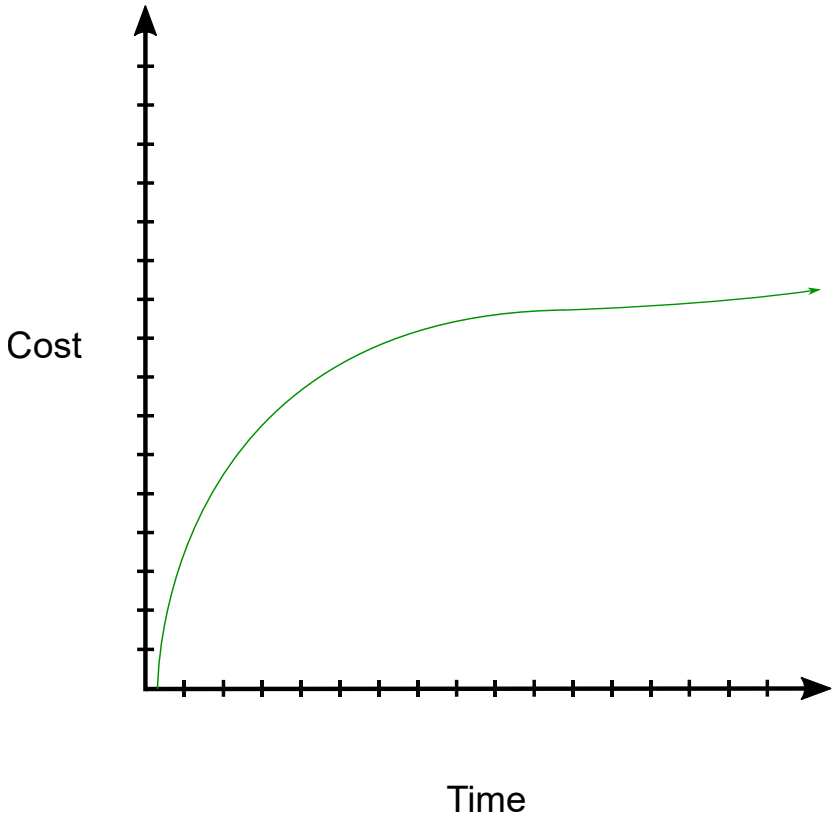
3.4.3 Mnoho testovacích dat

Velmi důležitou součástí testování jsou testovací data, také nazývaná data sety. Jedná se o množinu dat, jejichž podoba zajistí otestování konkrétní funkčnosti. Takových data setů může být pro kvalitní otestování zapotřebí více, např. data pro otestování „happy day scenario“ (tj. průchod aplikací se zadáním pouze validních dat, aplikace by neměla hlásit žádné chyby nebo přestat pracovat), další data pro vyvolání chybových stavů a tím otestovat správné reakce aplikace na tyto stavy, atd. S využitím automatických testů lze aplikaci otestovat nad všemi data sety pouze při jednom spuštění.

Tyto data sety musí tester nejdříve připravit (často nejdelší činnost při výrobě automatických testů), např. do textového souboru a dále zajistit, aby uživatelské vstupy byly simulovány právě skrze načítání hodnot z tohoto souboru.

3.5 Náklady na automatické testování

Na první pohled se zdá, že automatické testování je velmi neefektivní z hlediska nákladů [1]. Tyto vysoké náklady provázejí pouze prvotní fáze testování (viz obr. 3.1) a to proto, protože žádný test ještě není implementován a této implementaci musí testeři věnovat nějaký čas.



Obrázek 3.1: Náklady na automatické testování v čase (založeno na [5])

V pozdějších fázích testování se cena těchto testů začíná snižovat a postupně se ustálí. Toto ustálení je způsobeno tím, že ke konci vývoje je ustálen testovací framework a pravidla testování.

Po dokončení vývoje začnou náklady opět pomalu vzrůstat, a to kvůli přidávání nové funkčnosti, což vede k potřebě implementovat nové automatizované testy.

Tyto testy jsou tedy vhodné především pro aplikace, u nichž je předpokládán dlouhý vývoj nebo je očekávána dlouhodobá podpora po dokončení vývoje.

3.5.1 Význam vs. Cena testu

Dvě základní kritéria při volbě automatických testů nad testy manuálními. Významem testu se rozumí důležitost daného testu. Test je důležitý, pokud se testuje klíčová funkčnost, pokud je funkčnost složitá (např. mnoho funkčních závislostí, nebo větvení kódu), nebo pokud je u testu předpokládána vysoká opakovatelnost. Oproti tomu Cena testu představuje reálné náklady na test.

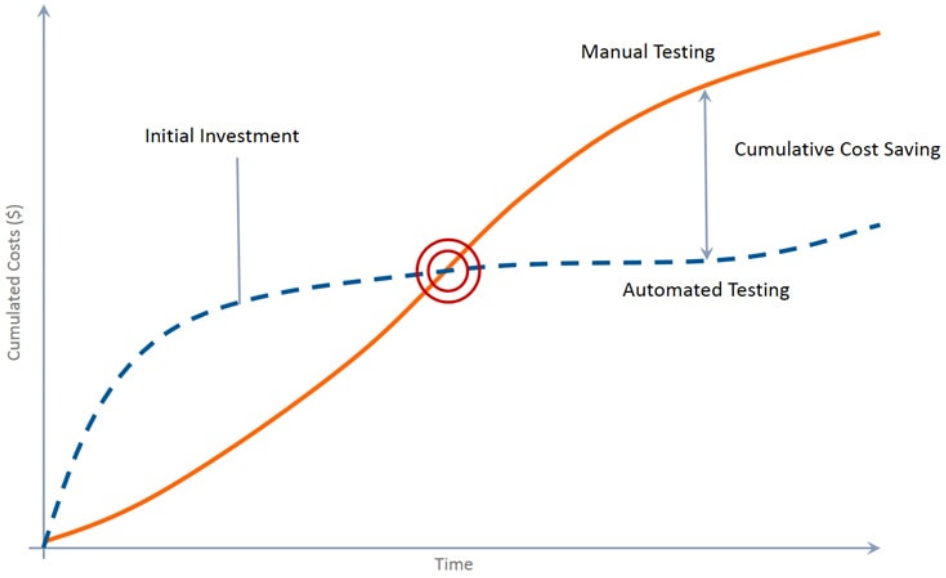
Pouze nízký Význam testu neznamená, že by test neměl být automatizovaný. Jeho Cena může být tak nízká, že vzhledem k Významu se automatizace vyplatí. To stejné platí pro Cenu. Cena testu může být vysoká, ale stejně tak může být vysoký jeho Význam, kvůli kterému se automatizace testu, i navzdory vysoké Ceně, vyplatí. Je proto důležité zvažovat oba tyto faktory zároveň.

3.6 Manuální vs. automatické testy

Největší rozdíl mezi těmito druhy testů je čas potřebný pro vytvoření jednoho testu a následné provádění samotného testu a od toho se odvíjející náklady na celé testování. Vytvoření manuálního testu znamená sestavení scénáře (nebo scénářů) s jehož pomocí je prověřena ověřovaná funkčnost. Příprava tohoto scénáře může trvat několik hodin. Oproti tomu příprava automatického testu může zabrat i několik dní, protože tester se musí nejdříve zorientovat v kódu aplikace, aby věděl, jakou část kódu bude testovat a následně pomocí mockování a dalších technik zajistit co největší nezávislost (izolaci) testu na okolním prostředí.

Provedení manuálních testů je velmi časově náročné, protože celý test musí být proveden testerem, který sedí u počítače a řídí se dodaným scénářem testu. Provedení automatického testu znamená, že tester pouze spustí příslušný test a zatím co je tento test spuštěn, může se věnovat jiným důležitým činnostem.

Náklady na manuální testování jsou tedy zprvu nízké, ale s každým dalším provedeným testem se cena zvyšuje. U automatických testů jsou náklady nejdříve vysoké, ale protože během běhu testu se testeři mohou věnovat jiným činnostem, začne se cena testů snižovat, ale ani v případě automatických testů se nikdy nezastaví na nule. Srovnání nákladů manuálních a automatických testů je graficky znázorněno na obr. 3.2.



Obrázek 3.2: Srovnání nákladů na manuální a automatické testy (převzáno z [5])

4 Kritéria hodnocení přínosu aut. testů

Při výrobě automatického testu na základě již manuálního testu je velmi vhodné zvážit, vyplatí-li se tento test automatizovat. Dále jsou uvedené body, které mají na vhodnost výroby velký vliv.

4.1 Předpokládaná délka vývoje

Také důležitý parametr při rozhodování o využití automatických testů. Při krátké době vývoje může dojít ke zbytečné ztrátě nákladů kvůli vysokým počátečním nákladům automatických testů oprati manuálním (viz obr. 3.2, levá polovina grafu). Proto je vhodnější automatické testování využívat u projektů, jejichž doba vývoje je delší než jeden rok

4.2 Úroveň dekompozice kódu

Jedná se o rozdělení jednoho velkého problému na co nejmenší podproblémy, jejichž postupné vykonání je nutné ke splnění celého problému [2]. Jednotlivé podproblémy jsou zapsány do funkcí a metod tak, že každá metoda nebo funkce vykonává právě jednu takovou činnost. Pomocí dekomponování lze dosáhnout větší čitelnosti a znovupoužitelnosti kódu. Pro vhodně dekomponovaný kód je psaní testů jednodušší. Pokud každá metoda vykonává jednu činnost (např. získávání dat z databáze), lze tuto metodu celou nahradit za testovací data pomocí mockování bez větších problémů.

Špatně dokomponovaný kód může vést ke složitému nahrazování pouze jedné části metody, což vede k prodloužení kódu samotného testu a ke snížení jeho čitelnosti a budoucí upravitelnosti.

Před výrobou samotného automatického testu je tedy potřeba zkontrolovat kód, je-li dostatečně dekomponován. Pokud není, může to vést k prodloužení výroby tohoto testu, tím pádem k větší ceně testu a tím ke snižování jeho hodnoty.

4.3 Četnost testu

Dále je třeba se zamyslet nad tím, jak často bude test spouštěn. Bude-li test spuštěn jednou nebo dvakrát během vývoje, nemá takový test smysl vyrábět, protože jeho hodnota bude velmi malá.

4.4 Množství testovacích dat

Pokud testovaná funkčnost požaduje k jejímu plnému otestování zadávání různých hodnot, řadí se tento test mezi dobré kandidáty pro výrobu automatického testu. Zadávání všech různých hodnot do aplikace při manuálním testování může být zdlouhavé a náročné. Automatické testy umožňují spouštění jednoho testu nad různými data sety.

4.5 Manuální náročnost testu

I když je test automatizovaný, neznamená to, že jeho spuštění probíhá bez zásahu testera. Každý spuštěný test potřebuje ke svému běhu testovací data (např. hodnoty zadané do textových polí), která musí být v testu nastavena (data set). Výhodou je, že test s jedním data setem lze spustit opakovaně. Je-li ale potřeba tato data upravit, často to znamená vytvoření nového data setu a následné dodání těchto dat do konkrétního testu. V závislosti na povaze testu (např. složitost kódu, větvení, nebo nutnost připojení externích zařízení) může být toto nastavení dat do testu velmi složité. Po ukončení běhu testu je potřeba zkontrolovat výsledky testu. Výsledky lze zkontrolovat počítačem nebo testerem. Zde opět záleží na povaze testu a testovacích dat. Při rozhodování o automatizaci testu by proto mělo být přihlíženo na proměnlivost dat, se kterými bude test spouštěn a na složitost jejich vložení do testu.

4.6 Důležitost testované funkce

4.6.1 Klíčová funkce systému

Klíčová funkce je taková funkce, bez které vyvíjená aplikace postrádá smysl a/nebo není schopna provozu (např. u e-mailové aplikace nelze odesílat e-maily). Klíčová funkce musí být dostupná za každých okolností, proto je testována s každou změnou aplikace pro zajištění dostupnosti. To vede k velmi

častému opakování tohoto testu a proto je test klíčové funkce velmi dobrým kandidátem na automatizaci.

4.6.2 Komplikovaná funkčnost

Pokud je funkce důležitou (klíčovou) funkcí aplikace, s vysokou pravděpodobností to znamená velmi složitý kód, mnoho funkčních závislostí napříč aplikací, rozvětvení funkčnosti, aj. To znamená velké množství různých výsledků (testovacích případů) v závislosti na zadaných datech, procházení aplikace, aj.

Pro každý testovací případ je potřeba připravit data a často také speciální test. Díky automatizaci lze vytvořit několik testů, každý využívající jiná data, které lze najednou spustit a ušetřit tak zdlouhavou a opakující se práci manuálního testera.

4.7 Zrychlení vývoje

Při vývoji nebo úpravě funkce, která není vnímána uživatelem (např. přistup do databáze) potřebuje programátor pro ověření správné funkčnosti sestavit celou aplikaci. Aplikaci musí poté spustit, projít aplikací na místo, kde prováděl úpravy a až poté může dojít na samotné ověření (např. jsou zobrazeny správné hodnoty získané z databáze). Tento proces může zabrat i několik minut. Oproti tomu automatizovaný test dokáže připravit pouze prostředí nezbytně nutné pro otestování vyvíjené funkce, čímž předejde sestavování zbytečných částí aplikace (vzhledem k vyvíjené funkci) a tím urychlí její otestování programátorem. Tento test lze poté zařadit do sady testů používaných k otestování celé aplikace samotnými testery.

5 Aplikace ohodnocených testů

U zákazníka (firma CCA) byly vybrány čtyři aplikace a v každé aplikaci bylo vytipováno několik testů vhodných pro automatizaci. Dále jsou uvedeny popisy těchto aplikací.

5.1 KUMUL

Jedná se o aplikaci pro výměnu zpráv nebo dat mezi systémy zákazníka. Zprávy jsou v dohodnutém formátu. Komunikace, kterou aplikace zajišťuje je asynchronní a probíhá tak, že aplikace odesílá zprávy prostřednictvím centrálního serveru a jiná aplikace si pomocí stejného serveru zprávy stahuje.

Celá aplikace běží na pozadí bez interakce s uživatelem. Z toho vyplývá problematické ladění a testování, které vyžaduje vyšší technickou znalost a klade větší nároky na přípravu testovacích dat.

5.2 E-Hotline

Jedná se o komponentu helpdesk systému firmy CCA, která zajišťuje proběhlou elektronickou komunikaci se zákazníky směrem ven a dovnitř. Aplikace je nakonfigurována tak, aby stahovala zprávy z konkrétních e-mailových schránek a dále je zpracovává. Toto stahování probíhá v pravidelných intervalech.

Odesílání zpráv probíhá pomocí fronty. Helpdesk podle aktivity uživatelů zapisuje do fronty zprávy, které mají odejít zákazníkovi. Aplikace v pravidelných intervalech frontu kontroluje a odesílá zprávy na určené adresy.

Formát zpráv i seznam příjemců jsou pro jednotlivé typy zpráv plně konfigurovatelné. Aplikace běží plně na pozadí bez uživatelského rozhraní. Testování je náročné na přípravu dat z důvodu potřeby externích e-mailových schránek.

5.3 eGP

Systém pro přidělování nápadu na organizacích rezortu Ministerstva spravedlnosti. V rámci tohoto systému byly řešeny složité algoritmy spojené s tzv.

nulováním nápadu. Algoritmy byly závislé na řadě různých situací a algoritmus, který rozhodoval, zda se má nulovat nebo ne je velice komplikovaný. Na tomto místě je velké množství funkčních závislostí, které způsobují, že jakákoliv změna aplikace může způsobit chybné chování tohoto algoritmu.

5.4 ISZA

ISZA je systém pro zprávy zákazníků a jejich požadavků, který si pro své interní potřeby vyvinula firma CCA. Systém se neustále rozvíjí. Před několika lety byla do systému přidána možnost automatické notifikace zákazníků a řešitelů tak zvaných hlášení. Tyto notifikace jsou závislé na řadě nastavení a jsou tedy náchylné na chyby. Protože notifikace odcházejí k zákazníkům, existuje velké riziko, že se zákazník o chybě rychle dozví, nebo dokonce aplikace zahltí zákazníka spamem. Nezanedbatelné je také riziko, že zákazník dostane zprávu, která mu nepatří a uvidí tak data jiného zákazníka.

6 Využité technologie a aplikace

6.1 SoapUI

SoapUI je aplikace určena pro testování webových rozhraní. Podporuje dva typy nejčastěji používaných webových rozhraní a to REST (RESTful) Application Programming Interface (API) a SOAP API. Aplikace funguje na principu mockování a umožňuje tak nejen testování strany klienta, ale také strany serveru, který poskytuje webové služby klientské aplikaci.

Pro testování strany serveru, který komunikuje protokolem REST, SoapUI umožňuje vytvoření falešného (mockovaného) klienta. U klienta lze vytvářet dotazy. Při vytváření dotazu postačí zadání URL adresy, na které se webová služba nachází a tento dotaz odeslat. Po odeslání dotazu se v aplikaci zobrazí veškeré informace, které webová služba poskytuje, včetně přenášených dat. Zde může uživatel zkontrolovat, jestli informace a samotná data odpovídají předpokládanému výsledku.

Pro testování klientské aplikace, která komunikuje protokolem REST, lze v SoapUI vytvořit falešný server. Na tomto serveru lze vytvořit webové služby odpovídající různým typům požadovaných dotazů. U každé webové služby je možné připravit odpovědi na dotazy s různými parametry zadanými v URL adrese a s daty odpovídajícími zadaným parametrům. Tento server se dá poté využít v klientské aplikaci a otestovat, jestli se klient zachová tak, jak je očekáváno.

Pro aplikaci komunikující protokolem SOAP je postup vytváření klientských dotazů a webový služeb velmi podobný. Jediný rozdíl je v prvotním vytvoření odpovídajícího projektu. U projektu pro SOAP aplikaci je potřeba zadat cestu k WSDL souboru, který obsahuje definici webového rozhraní jak pro klienta, tak pro server. Při vytváření jednotlivých dotazů a služeb se pomocí WSDL souboru vygeneruje kostra přenášených zpráv, do které uživatel doplní pouze konkrétní data.

6.2 Mockito a PowerMock

Mockito je testovací framework určený pro programovací jazyk Java. Pomocí Mockita lze vytvářet mock objekty, mockovat jejich metody a ty používat při

testování aplikace. PowerMock je framework rozšiřující Mockito. Umožňuje vytvářet mocky objektů a metod, na které je Mockito krátké (např. statické a privátní metody).

```
public void setUpMocks(){
    Mockito.when(aplikace.getData())
        .thenReturn(getMockData());
}

private String[] getMockData(){
    return new String[] {"item1", "item2", "item3"};
}
```

Ukázka 6.1: Příklad mockování v jazyce Java

Mockito poskytuje metodu `when()`, pomocí které lze mockovat metody testované aplikace. V ukázce 6.1 je uveden příklad mockování. Volání metody `aplikace.getData()` je nahrazeno voláním metody `getMockData()`, kterou si vytvořil tester např. za účelem izolování získávání dat od databáze. Pokud je v průběhu testu volána metoda `aplikace.getData()`, jejím výsledkem vždy bude návratová hodnota metody `getMockData()`.

6.3 GreenMail

Při testování aplikací, které pracují s e-mailovým klientem a tedy s velkou pravděpodobností e-maily odesílají, je největším problémem to, že při takovém testu vlastně e-mail odeslat nechceme. Pokud se jedná o kritickou funkci systému, znamenalo by to desítky odeslaných zpráv nic netušícím uživatelům.

Tomu zabraňuje testovací framework GreenMail. S jeho pomocí lze vytvořit mockovaný e-mailový server, který funguje stejně jako běžný e-mailový server. Veškeré odesílání a přijímání zpráv je zajištěno v rámci tohoto falešného serveru a nedochází tak k odesílání nevyžádaných zpráv. Framework umožňuje jednoduché získání odesílatelů, adresátů, textu zprávy, a jiných důležitých údajů, které e-mailová zpráva obsahuje. Je možné i vytvoření jednotlivých uživatelů, kterým jsou testovací emaily odesílány a díky tomu je možné zkontrolovat, jestli uživatel obdržel všechny zprávy, které mu byly odeslány.

6.4 PL/SQL Developer

PL/SQL slouží k tvorbě a údržbě Oracle databází. Pomocí PL/SQL jazyka je možné psát scripty obsahující funkce a procedury, které slouží k manipulaci s entitami. Tyto scripty mohou obsahovat jak logiku ukládání záznamů do databázových entit, tak logiku jejich získávání.

7 Automatizované testy

Všechny testy uváděné v této kapitole jsou přiložené na CD.

7.1 Test aplikace KUMUL

Aplikace je napsána v programovacím jazyce Java a ve stejném jazyce byl psán vytvářený test. Cílem tohoto testu je ověřit správnou funkčnost klienta, který vykonává dvě hlavní činnosti: odesílání zpráv a jejich získávání z centrálního serveru. Proto měl být tento test rozdělen do dvou testovacích případů: test odesílání zpráv a test získávání zpráv. Ale před samotnými testy bylo potřeba izolovat aplikaci od okolních závislostí na databázi a na webovém rozhraní, prostřednictvím kterého je realizována veškerá komunikace klienta s centrálním serverem.

Izolace od webového rozhraní je jednodušší ze dvou úkonů. Z definice webového rozhraní lze získat WSDL soubor obsahující definici rozhraní a jednotlivých přenášených zpráv. Pomocí tohoto souboru se následně v nástroji SoapUI vytvoří falešná webová služba, na kterou by byly směřovány všechny dotazy z vytvořeného testu. Jednotlivým předpřipraveným požadavkům a odpovědím na ně postačí vyplnit potřebná data. Je důležité nezapomenout na to, že požadavek, který vznikne v rámci testu musí mít přesně stejná data, která jsou nastavena ve falešném webovém rozhraní. Pro nástroj SoapUI existuje knihovna, kterou lze zavést do testu a s její pomocí spustit vytvořené falešné rozhraní přímo v rámci testu. Pro zajištění izolace testu od skutečného webového rozhraní stačí přesměrovat požadavky testu na URL adresu vytvořeného falešného webového rozhraní.

Izolace testu od databáze by musela být provedena přímo v testu za využití nástroje Mockito a mocků, které s jeho pomocí lze vytvořit. Zde se ale vyskytl problém, který znemožnil dokončení výroby tohoto testu. Navzdory tomu, jak jednoduchý úkol měl být testován, jeho provedení v kódu aplikace bylo velmi náročné na testování. Kód postrádal vyšší úroveň dekompozice. V rámci jedné metody se získávala data z databáze, vytvářela zpráva k odeslání, vytvořená zpráva se šifrovala, zpráva odesílala a kontroloval výsledek odeslání zprávy. To zcela porušuje základní myšlenku dekompozice: jedna metoda vykonává pouze jednu činnost. Kvůli tomu nemohou být mockovány pouze potřebné metody, ale je nutné projít kód řádek po řádku a mockovat každý přístup do databáze, každé nastavení atributu zprávy, případné šifrování, atd. Aplikace navíc umožňuje komunikaci klientů, kde ne každý klient

má stejnou definici a proto by pravděpodobně musel být test upraven do různých podob v závislosti na typu právě testovaného klienta.

Po analyzování těchto problémů bylo rozhodnuto o ukončení výroby tohoto testu z důvodu velké náročnosti na dokončení testu za stávajícího stavu aplikace.

7.2 Testy aplikace E-Hotline

Aplikace je napsána v programovacím jazyce Java a ve stejném jazyce byly psány vytvářené testy.

7.2.1 Test odesílání e-mailů

Cílem testu je ověřit funkčnost aplikace, která v pravidelných intervalech kontroluje databázovou frontu obsahující zprávy určené k odeslání. Protože se jedná o aplikaci odesílající e-mailové zprávy, bylo zapotřebí využít testovaní framework GreenMail, aby se předešlo odesílání nevyžádaných zpráv (graficky znázorněno na obr. 7.1). Také se tím předejde složité práci s e-mailovým API, které je dostupné pro programovací jazyk Java.

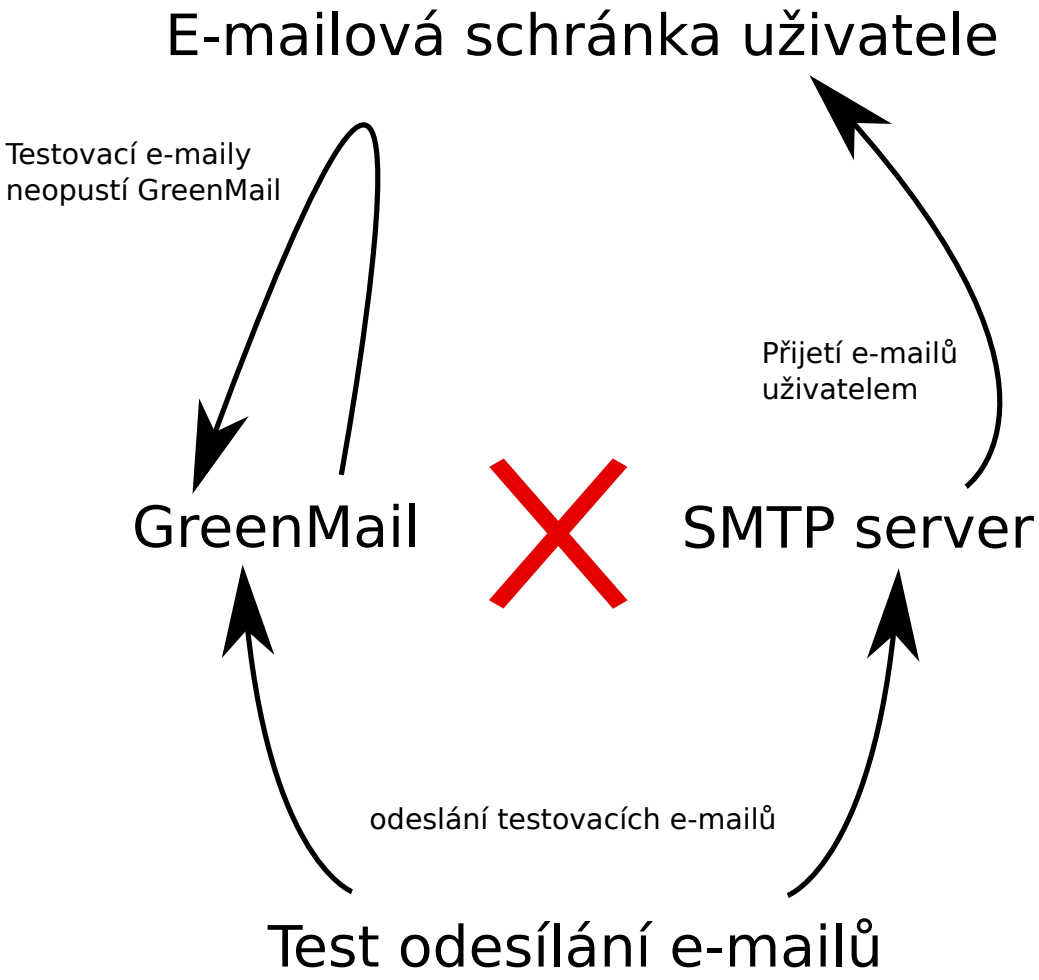
Než se test dostane k vykonání samotného testovacího případu, je nutné zajistit nastavení testovacího prostředí. Zde jsou za tímto účelem využity metody s anotací `@BeforeAll` a `@BeforeEach`. Metoda s anotací `@BeforeAll` (viz ukázka 7.1) je během celého testu spuštěna pouze jednou a to bezprostředně po spuštění testu. V této metodě je zajištěno zprovoznění falešného e-mailového serveru pomocí GreenMailu. Metoda s anotací `@BeforeEach` je spuštěna před každým testovacím případem. Zde je s její pomocí vytvořena e-mailová schránka a jsou nastaveny atributy potřebné pro využití falešného e-mailového serveru, namísto toho reálného.

```
@BeforeAll
public void setUp(){
    greenMail = new GreenMail();
    greenMail.start();
}
```

Ukázka 7.1: Metoda s anotací @BeforeAll

Po nastavení prostředí je spuštěn testovací případ pro odeslání emailů bez příloh[1]. Nejdříve se z pomocné metody získají e-mailové zprávy které mají

[1]Tento testovací případ byl vytvořen dvakrát. Poprvé před revizí testované aplikace a podruhé po revizi. V kapitole je popsán testovací případ po revizi aplikace. Verze před



Obrázek 7.1: Rozdíl při použití GreenMailu a skutečného SMTP serveru

být odeslány. Následně jsou tyto zprávy postupně zpracovány a odeslány testovanou metodou aplikace E-Hotline. Testovací případ pokračuje získáním přijatých e-mailových zpráv na falešný server. Kvůli zjednodušení konečného testování se dále očekává přijetí pouze jedné zprávy. Z přijaté zprávy je získán odesílatel, předmět a tělo zprávy a tyto údaje jsou otestovány vůči očekávaným hodnotám. Otestování je provedeno prostřednictvím metod `AssertAll` a `AssertEquals`, kde první z metod lze předat potřebné množství lambda výrazů v podobě `() -> AssertEquals(value, expectedValue)`.

Po vykonání testovacího případu je potřeba „uklidit“ to, co bylo vytvořeno pro tento případ v rámci metody s anotací `@BeforeEach`. K tomu

revizí je přiložena na CD pod názvem `PosliNotifikaceTestPredRevizi.java`. Na srovnání těchto dvou vypracování je přehledně vidět rozdíl na množství potřebných mocků u aplikací, které jsou dekomponované málo nebo vůbec a těch, které mají nejvyšší úroveň dekompozice.

slouží anotace `@AfterEach` (viz ukázka 7.2) a zde je pouze restartován falešný e-mailový server, čímž je zajištěno to, že do dalšího testovacího případu nebudou zasahovat zprávy odeslané v rámci předešlého testovacího případu. Po vykonání všech testovacích případů je potřeba tento „úklid“ provést také, ale ve vztahu k anotaci `@BeforeAll`. K tomu slouží anotace `@AfterAll` (viz ukázka 7.2), která v tomto testu zastaví falešný e-mailový server.

```
@AfterEach
public void resetGreenMail(){
    greenMail.reset();
}

@AfterAll
public void stopGreenMail(){
    greenMail.stop();
}
```

Ukázka 7.2: Metody s anotací @AfterEach a @AfterAll

Druhý testovací případ je odesílání e-mailových zpráv s přílohou. Metody s anotací `@BeforeAll`, `@BeforeEach`, `@AfterEach` a `@AfterAll` jsou stejné jako v prvním testovacím případě. Samotný testovací případ se od prvního liší pouze tím, že navíc obsahuje získání názvu připojené přílohy a test získaného názvu s očekávaným. Je zde očekávaná pouze jedna připojená příloha.

7.2.2 Test naplnění příloh

Cílem testu je ověřit funkčnost části aplikace zodpovědné za získávání příloh připojovaných k odesílaným e-mailovým zprávám. Příloha může být trojího typu: soubor uložený v databázi, soubor uložený ve firemním cloudu s přímým odkazem, nebo soubor uložený ve firemním cloudu s univerzálním odkazem [2]. Kvůli kompatibilitě s testovacím frameworkem PowerMock musel být u výroby tohoto testu použit testovací framework JUnit 4, oproti předchozímu JUnit 5.

Po spuštění testu je vykonána metoda s anotací `@Before` (ekvivalent anotace `@BeforeEach`), ve které jsou vytvořeny a pomocí mocků nastaveny parametry potřebné pro přístup do firemního cloudu. Také jsou zde vytvo-

[2]Rozdíl mezi přímým a univerzálním odkazem je ten, že přímý odkaz směřuje na umístění souboru. Pokud by byl soubor přesunut do jiné složky, tento odkaz pozbývá platnosti. Univerzální odkaz oproti tomu je navázaný přímo na soubor a tudíž nezáleží na jeho umístění ve firemním cloudu.

řeny dokumenty, které jsou pomocí mocků připojeny k testovací e-mailové zprávě. Připojené dokumenty jsou tři, každý jiného typu.

Poté je spuštěn samotný testovací případ, který obsahuje pouze získání e-mailových zpráv a následné připojení příloh ke každé zprávě. Není zde žádná metoda testující správné připojení příloh. To je způsobeno tím, že v případě příloh, které jsou získané z odkazu na firemní cloud, nelze kódově zajistit, že soubor, který by byl získán v rámci testu za účelem tohoto otestování, bude načten správně.

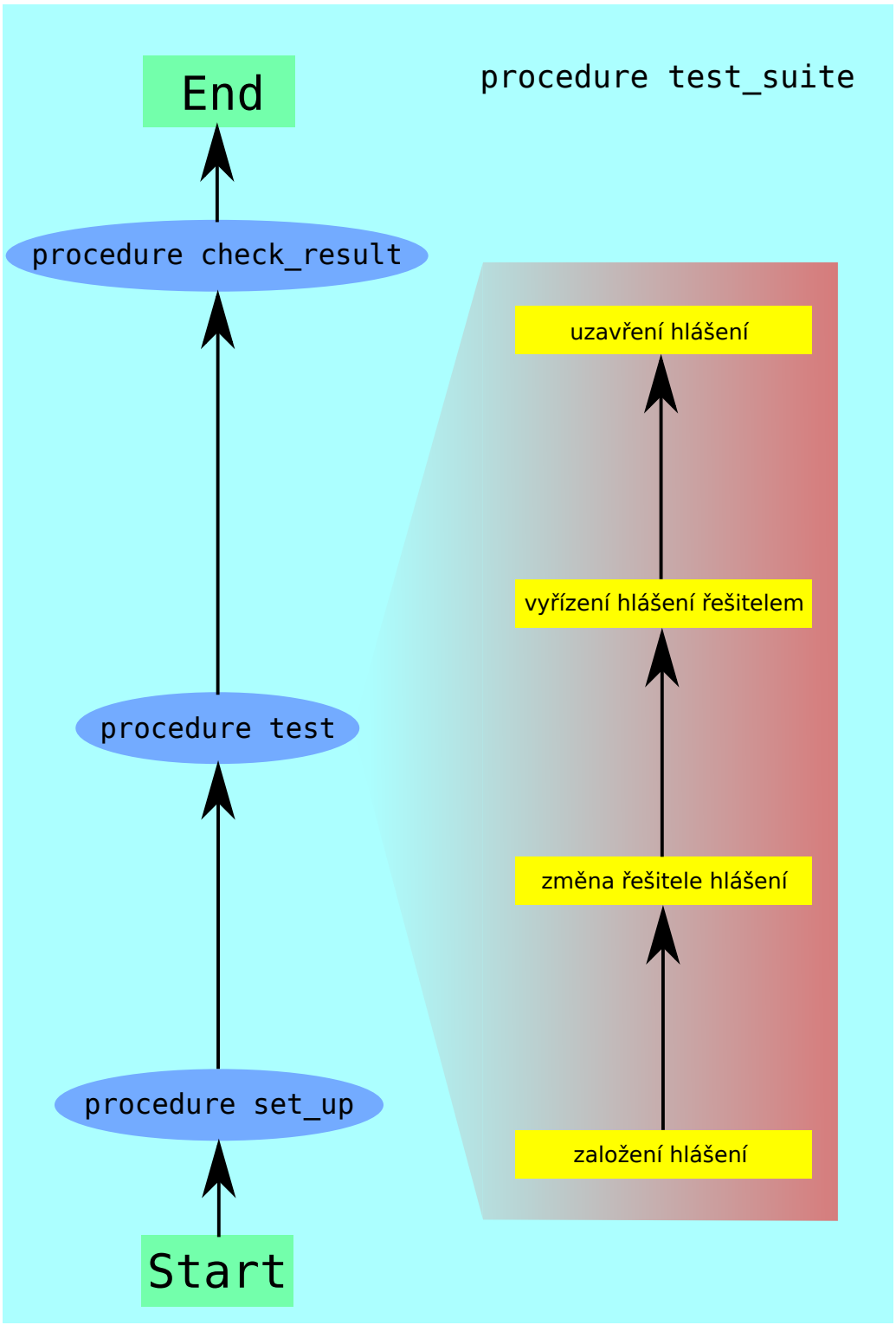
Jednou z možností jak otestovat, že přílohy byly připojeny správně je propojení tohoto testu s testem odesílání e-mailových zpráv (viz 7.2.1), konkrétně s testovacím případem odesílání e-mailových zpráv s přílohou, kterému by byly předány přílohy získané tímto testem. V rámci otestování správnosti připojených příloh by musely být přílohy získány ze serveru GreenMail a porovnány s požadovaným výsledkem.

7.3 Test aplikace ISZA - odesílání notifikací

Aplikace je napsána ve scriptovacím jazyce PL/SQL a ve stejném jazyce byl psán vytvářený test: test notifikací. Cílem tohoto testu je ověření funkčnosti části aplikace zajišťující vytváření notifikací na základě vzniklých událostí. Test testuje událost vzniku tzv. hlášení, změnu osoby řešitele a vyřízení tohoto hlášení řešitelem. Při výkonu tohoto testu není žádná procedura volána automaticky. Všechny potřebné procedury musí být v určitém bodě testu spuštěny uživatelským kódem. Hlavní procedura testu `test_suite` (průchod procedurou je graficky znázorněn na obr. 7.2) postupně zajistí přípravu potřebných globálních proměnných, spuštění samotného testu a následně provede kontrolu výsledků.

První procedura `set_up` uloží nezbytná data do globálních proměnných. Jedná se o proměnnou začátku testu, která je využita při kontrole výsledků a o proměnnou hlášení, s jehož pomocí jsou generované potřebné notifikace. Dále je vykonána hlavní testovací procedura `test`.

Nejdříve je v systému založeno připravené hlášení. Dále je u hlášení změněn jeho řešitel a tato změna je uložena. Následuje vyřízení tohoto hlášení řešitelem. Toho je docíleno nastavením potřebných atributů hlášení tak, aby představovali jeho vyřízení procedurou `vyrid_hlaseni_resitelem`. Stejným způsobem je hlášení kvůli zachování konzistence dat uzavřeno procedurou `uzavri_hlaseni`. Po uzavření hlášení by se mělo ve frontě notifikací nacházet několik zpráv odpovídající změnám prováděných v rámci tohoto testu. Pro případnou manuální kontrolu jsou tyto notifikace v závěru testovací



Obrázek 7.2: Diagram průchodu procedurou test_suite

procedury odeslány adresátům.

Každá odeslaná notifikace má určený typ. Kontrola výsledků tohoto testu probíhá kontrolou přítomnosti notifikací těchto typů a jejich počtem v notifikační frontě procedurou `check_result`. Protože se jedná o frontu, která není využívána pouze tímto testem, ke kontrole je také použit čas začátku testu, aby byly kontrolovány pouze notifikace, které vznikly až po spuštění testu. Použito je také `sessionid`, díky kterému jsou kontrolovány pouze notifikace vzniklé během spuštění hlavní testovací procedury. Pokud neodpovídá očekávaný počet všech notifikací nebo určitého typu notifikací, je zobrazena odpovídající chybová hláška informující o nesprávném počtu notifikací.

7.4 Ověření správnosti výsledků automatizovaných testů

Ověřování automatizovných testů probíhalo tak, že při vývoji automatických testů byly průběžně prováděny manuální testy a tak bylo průběžně ověřováno, že vrací stejné výsledky. Tímto způsobem byly ověřený všechny automatizované testy.

7.5 Zhodnocení efektivity testů

Test aplikace KUMUL byl za stávajícího stavu aplikace příliš náročný na dokončení. Pro jeho dokončení by musela být provedena revize celé aplikace, což je nad rámec této práce a proto není možné zhodnotit efektivitu tohoto testu. Vzhledem k důležitosti funkce bylo firmě CCA doporučeno provést revizi aplikace KUMUL před automatizací tohoto testu.

Při výrobě testů činilo největší problém neznalost všech technologií, které se v testovaných aplikacích využívají. Problémem také byla kompatibilita některých testovacích frameworků.

Pro spuštění každého z testů je potřeba být připojen do infrastruktury firmy CCA. Samotné spuštění testů není náročné a s využitím vhodného vývojového prostředí se tento úkon stává ještě méně náročným.

Doba běhů jednotlivých testů se pohybuje v jednotkách sekund. Příprava prostředí žádného z testů není natolik složitá, aby značně prodloužila běh testu. Testy mají rychlý průběh také díky faktu, že žádný z testů nepotřebuje ke svému průběhu nasazovat aplikační server. Výjimku v běhu testu tvoří test aplikace ISZA - odesílání notifikací, který se pohybuje okolo deseti sekund. Tato delší doba běhu je zapříčiněna nutností založit hlášení, což je velmi

robustní a komplexní úkon. Vyrobené testy jsou plně funkční a je možné pokračovat v jejich rozvoji.

Automatizací těchto testů jsem nabyl dojmu, že výroba automatických testů má skutečně smysl, ale je podmíněna především tím, aby vývojář aplikace myslel na to, že budou na aplikaci vznikat testy, a vývoj tomu uzpůsobil.

8 Ohodnocení testů pomocí kritérií

Implementované testy popsané v kapitole 7 byly ohodnoceny dle zvolených kritérií (viz kapitola 4), která byla rozdělena do dvou skupin: Význam a Cena. Ty jsou dále rozdělena na jednotlivá kritéria, která nabývají hodnot od jedné do tří. Dále jsou uvedena jednotlivá kritéria, jejich rozdělení do zmiňovaných dvou skupin a význam jejich ohodnocení na zvolené stupnici. Ohodnocení testu je přiloženo v tabulkách A.1 a A.2.

8.1 Význam testu

8.1.1 Opakovatelnost

Určuje, jak často je test spouštěn.

1. skoro vůbec
2. pravidelně
3. při každé změně v aplikaci

8.1.2 Náročnost ruční příprava dat

Představuje náročnost přípravy nových dat pro test.

1. hodina
2. do 4 hodin
3. více než 4 hodiny

8.1.3 Důležitost testované funkce

Jak moc je testovaná funkčnost důležitá pro chod aplikace.

1. nefunkčnost způsobí drobné problémy uživateli, ale bude moci dále pracovat
2. některé důležité vlastnosti nebudou dostupné, ale aplikace bude stále dostupná
3. bez této funkčnosti nebude možné aplikaci používat

8.1.4 Manuální náročnost testu

Určuje náročnost manuálního spuštění testu a ruční kontroly výsledků testu.

1. půl hodiny
2. do 4 hodin
3. více než 4 hodiny

8.1.5 Zrychlení vývoje

O kolik byl urychlen vývoj funkce za celou dobu vývoje, především díky připravě prostředí pro ladění programu automatickým testem.

1. vývoj se nezrychlí
2. do 4 hodin
3. nad 4 hodin

8.2 Cena testu

8.2.1 Nekvalita kódu

Míra toho, jak je testovaný kód vhodný pro testování a kolik je potřeba úprav k tomu, aby bylo možné kód otestovat.

1. lze otestovat bez úprav
2. úpravy do 4 hodin
3. úpravy nad 4 hodiny

8.2.2 Očekávané stáří projektu

Předpokládaná délka projektu včetně servisu.

1. nad dva roky
2. od jednoho roka do dvou let
3. méně než jeden rok

8.2.3 Náročnost vyrobení

Určuje, jak dlouhou dobu trvá vyrobení testu.

1. do 2 hodin
2. do 8 hodin
3. nad 8 hodin

8.3 Rozhodnutí o automatizování testu

Každému kritériu je přiřazena jeho váha (uvedena v tabulkách A.1 a A.2, sloupečky váha) a hodnota dle stupnice v bodech 8.1 a 8.2. Pro každý test je na základě váhy a hodnoty u každého kritéria spočtena výsledná hodnota Významu a Ceny pomocí vzorce pro vážený průměr:

$$v = \frac{\sum(váha * hodnota)}{\sum váha}. \tag{8.1}$$

Každý test je tedy určen dvěma veličinami: jeho Významem a Cenou. Konečné rozhodnutí lze určit po odečtení Významu a Ceny:

$$r = Význam - Cena \tag{8.2}$$

a s pomocí tabulky výsledků, viz tabulka 8.1. Výsledné hodnoty rozdílů Významu a Ceny implementovaných testů jsou uvedeny v tabulce A.3.

Výsledek	Rozhodnutí
$r \leq 0$	nevhodné pro automatizaci
$0 < r \leq 0.5$	automatizace testu by měla být konzultována s vývojáři a/nebo s testery
$0.5 < r \leq 1$	test je vhodný pro automatizaci
$1 < r \leq 2$	test by měl být určitě automatizován

Tabulka 8.1: Přehled rozhodnutí na základě výsledků hodnocení testu

8.4 Pouze ohodnocené testy

Kvůli malému počtu implementovaných testů byly vedoucím práce vytipovány další testy, které jsou ve firmě CCA již automatizované, nebo se o jejich automatizaci teprve přemýšlí. Tyto testy byly pouze ohodnoceny podle postupu v kapitole 8. Toto ohodnocení slouží firmě jako potvrzení jejich rozhodnutí nebo doporučení, jakým směrem by se měli v rozmýšlení o automatizaci testů ubírat. Jedná se o následující testy.

8.4.1 Testy eGP

Test generování dokumentu byl vyroben, protože během vývoje bylo nutné celou aplikaci nasazovat na aplikační server při každé změně aplikace, což bylo velmi zdlouhavé, ale generování dokumentu nevyžaduje žádné serverové zdroje. Tímto testem je možné tento proce obejít a zkrátit tak dobu testování během vývoje na minimum.

Test nulování testuje proces, který se spouští v přesně definovaných okamžicích a je ovlivňován celou řadou faktorů. Analýza testu provedena ve firmě CCA identifikovala 66 různých kombinací vstupu, které bylo nutno testovat.

8.4.2 Testy ISZA

Test založení systému testuje jednoduchou operaci, při které uživatel pořizuje data prostřednictvím uživatelského rozhraní. Jde o běžnou CRUD (Create-Read-Upload-Delete) operaci.

Test založení hlášení pouze kontroluje přítomnost identifikačního klíče po založení hlášení v entitě hlášení.

8.5 Výsledky ohodnocení testů

Na základě výsledků ohodnocení implementovaných testů, viz tab. A.3, lze tvrdit, že implementované testy by se z hledisek Významu a nákladů na automatizaci vyplatily firmě CCA automatizovat a zařadit je do používaného testovacího frameworku. Výjimku implementovaných testů tvoří test aplikace KUMUL (viz 7.1). Dle výsledků ohodnocení by byla u tohoto testu doporučena konzultace s vývojáři a testery. Při současné úpravě kódu by byla automatizace bez provedení revize aplikace nejspíše zamítnuta.

Z tabulek ohodnocení také vyplývá zmiňovaná nutnost (viz 3.5.1) posuzovat test jak z hlediska jeho Významu (hodnoty), tak ale zároveň z hlediska jeho Ceny. To lze pozorovat na implementovaných testech aplikací KUMUL a E-Hotline - odesílání e-mailů. Oba tyto testy mají vysokou hodnotu (viz A.1), ale při posouzení s jejich cenou (viz A.2) jsou výsledné rozdíly od sebe vzdáleny dvě úrovně v rozhodovací tabulce 8.1. Na opačnou stranu, při rozhodování podle ceny, ohodnocené testy aplikace ISZA - založení hlášení a založení systému mají relativně nízkou cenou. Pokud ale zvážíme i jejich hodnoty, výsledné rozdíly jsou od sebe vzdáleny jednu úroveň.

9 Závěr

V rámci vypracování této práce jsem byl seznámen s problematikou jak manuálního, tak automatického testování, především v rámci pravidelných schůzí s vedoucím práce. Individuálně jsem se seznámil s problematikou pomocí odkazovaných internetových článků a dokumentů.

Navrhnutá kritéria byla z počátku pouze čtyři, ale v průběhu implementace testů a dalších schůzí s vedoucím práce byl sestaven seznam obsahující osm kritérií, která byla dále použita pro ohodnocení implementovaných testů a rozhodnutí o využitelnosti této implementace.

Implementované testy byly zvoleny vedoucím práce na základě již existujících manuálních testů. Největší potíže při implementaci těchto testů byly způsobeny neznalostí detailního řešení některých aplikací a neznalost některých využitých technologií.

Výsledky implementovaných automatických testů byly ověřeny pomocí výsledků manuálních podob těchto testů. Výsledky obou typů testů se shodují.

Literatura

[1] *What is automated testing* [online]. SmartBear. [cit. 2019/12/29]. Dostupné z: `https://smartbear.com/learn/automated-testing/what-is-automated-testing/`.

[2] *What is decomposition?* [online]. BBC. [cit. 2020/1/17]. Dostupné z: `https://www.bbc.co.uk/bitesize/topics/zkcqn39/articles/z8ngr82`.

[3] *What is the cost of defects in software testing?* [online]. Testing.Guru. [cit. 2019/12/22]. Dostupné z: `https://www.testing.guru/what-is-the-cost-of-defects-in-software-testing/`.

[4] Herout, P. *Přednášky z předmětu KIV/OKS* [online]. 2019. [cit. 2019/12/22]. Dostupné z: `http://www.kiv.zcu.cz/~herout/vyuka/oks/prednasky/oks-2019.pdf`.

[5] Kyryk, I. *What projects need test automation* [online]. QATestLab Blog, 2018. [cit. 2020/1/15]. Dostupné z: `https://blog.qatestlab.com/2018/06/12/when-automate-testing/`.

[6] Padmanabhan, A. *Mock testing* [online]. SmartBear, 2019. [cit. 2020/1/10]. Dostupné z: `https://devopedia.org/mock-testing`.

A Tabulky ohodnocení testů

Význam	Četnost testu		Náročná ruční příprava dat		Důležitost testované funkce		Manuální náročnost testu		Zrychlení vývoje		Vážený průměr
Test	váha	hodnota	váha	hodnota	váha	hodnota	váha	hodnota	váha	hodnota	
KUMUL	2	3	2	2	2	3	2	2	1	1	2.3
EHOTLINE - odesílání	2	3	2	3	2	3	2	2	1	1	2.5
EHOTLINE - přílohy	2	3	2	2	2	2	2	2	1	1	2.1
eGP - generování dokumentu	2	3	2	2	2	1	2	2	1	3	2.1
eGP - test nulování	2	3	2	3	2	3	2	3	1	1	2.7
ISZA - test notifikací	2	3	2	3	2	3	2	3	1	1	2.7
ISZA - založení hlášení	2	3	2	1	2	3	2	1	1	1	1.8
ISZA - založení systému	2	1	2	1	2	2	2	1	1	1	1.2

Tabulka A.1: Ohodnocení testů: Význam

Cena	Nekvalita kódu		Očekávané stáří projektu		Náročnost vyrobení		Vážený průměr
Test	váha	hodnota	váha	hodnota	váha	hodnota	
KUMUL	1	3	1	1	1	2	2.0
E-HOTLINE - odesílání	1	1	1	1	1	2	1.3
E-HOTLINE - přílohy	1	1	1	1	1	2	1.3
eGP - generování dokumentu	1	1	1	1	1	1	1.0
eGP - test nulování	1	1	1	1	1	2	1.3
ISZA - test notifikací	1	1	1	1	1	2	1.3
ISZA - založení hlášení	1	2	1	1	1	2	1.6
ISZA - založení systému	1	2	1	1	1	2	1.6

Tabulka A.2: Ohodnocení testů: Cena

ROZDÍL Test	Význam - Cena
KUMUL	0.3
E-HOTLINE - odesílání	1.2
E-HOTLINE - přílohy	0.8
eGP - generování dokumentu	1.1
eGP - test nulování	1.4
ISZA - test notifikací	1.4
ISZA - založení hlášení	0.2
ISZA - založení systému	-0.4

Tabulka A.3: Ohodnocení testů: rozdíl Významu a Ceny