Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Automatizované testování mobilního softwaru

Bc. Marek Dvořák

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Marek Dvořák

Systémové inženýrství a informatika
Informatika

Název práce

Automatizované testování mobilního softwaru

Název anglicky

Automated testing of mobile software

Cíle práce

Cílem diplomové práce je manuálně a automatizovaně otestovat funkčnost aplikace pro mobilní zařízení a zjistit, který z těchto postupů je výhodnější pro jednotlivé požadavky. Testování se bude konkrétně zabývat mobilním internetovým bankovnictvím, na kterém budou demonstrovány techniky a nástroje pro manuální a automatizované testování. Následně budou porovnány výsledky spolu s finančními náklady.

Metodika

Teoretická část bude sepsána za pomoci dostupných knih a různých internetových materiálů, které budou uvedeny v seznamu použitých zdrojů na konci této práce. V této rešeršní části bude zmíněn postup a druhy vývoje softwaru. Následovat bude podrobnější popis problematiky testování s různými úrovněmi, technikami a postupy. Dále popsání a vysvětlení technik automatizovaného a manuálního testování. Poté budou zmíněny druhy a odlišnosti mobilních aplikací a platforem. Jako poslední v této sekci bude ukázka nástrojů použitelných pro testování.

Praktická část práce bude zaměřena na otestování funkčnosti již konkrétní mobilní aplikace pro internetové bankovnictví. Nejprve budou představeny nástroje, které budou potřeba pro tuto práci. Potom budou specifikovány požadavky a jednotlivé části aplikace, které se budou testovat. Následně vytvoření testovacích případů a jejich jednotlivých kroků. Poté přijde na řadu vytvoření automatizovaných testů na tyto konkrétní funkcionality a jejich spuštění. Po zprovoznění a porovnání s manuálními testy bude následovat vyhodnocení výsledků spolu s finančními náklady.

Doporučený rozsah práce

60-80 stran

Klíčová slova

testování, automatizace, mobilní aplikace, Appium, Selenium

Doporučené zdroje informací

Kshirasagar Naik, Priyadarshi Tripathy – Software Testing and Quality Assurance: Theory and Practice, John Wiley & Sons, Inc, 2008, ISBN: 978-0-471-78911-6

Miroslav Bureš; Miroslav Renda; Michal Doležel; kolektiv – Efektivní testování softwaru, Grada Publishing, a. s, 2016, ISBN: 978-80-247-5594-6

Nishant Verma – Mobile Test Automation with Appium, Packt Publishing, 2017, ISBN:978-1-78728-016-8

Ron Patton – Testování softwaru, Computer Press, 2002, ISBN: 8072266365

Unmesh Gundecha – Selenium Testing Tools Cookbook, Second Edition, Packt Publishing, 2015, ISBN: 978-1-78439-251-2

Předběžný termín obhajoby

2019/20 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 30. 03. 2020

Čestné prohlášení

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

V Praze dne 04.04.2020 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Jiřímu Brožkovi, Ph.D. za vedení mé diplomové práce a mnoho cenných rad, které mi poskytl. Dále bych rád poděkoval rodině, přítelkyni i psovi za to, že se mnou měli po celou dobu psaní této práce i studia trpělivost a podporovali mě v každé situaci.

Automatizované testování mobilního softwaru

Abstrakt

Tato diplomová práce se zabývá manuálním a automatizovaným testováním nativní mobilní aplikace pro bankovnictví.

Teoretická část se věnuje popisu a definici testování. Nejprve zmiňuje samotný vývoj softwaru, do kterého testování patří. Dále představuje testovací tým, potřebnou dokumentaci, jednotlivé úrovně a druhy testování. Poté jsou zde zmíněny i nástroje pro automatizované testování.

Praktická část této práce je již samotná příprava a realizace testování zvolené aplikace na platformě Android. Nejprve je uveden software, který je využíván, dále funkční požadavky, následně testovací případy. Poté jsou vytvořeny a spuštěny jednotlivé manuální i automatizované testy. Nakonec jsou tyto operace zhodnoceny po stránce rychlosti, financí a použitelnosti.

Klíčová slova: testování, automatizace, mobilní aplikace, Appium, Selenium, Java, vývoj softwaru, testovací případ, testovací skript, Android

Automated testing of mobile software

Abstract

This thesis focuses on manual and automated testing of a native mobile banking application.

The aim of the theoretical part is to describe and define testing. It mentions the evolution of the software itself which the testing is part of. Furthermore, it characterizes a testing team, the documentation which is needed for testing, different levels and types of testing. In addition, tools for automated testing are described.

The practical part consists of the preparation and realization of testing the chosen application on the Android platform. Firstly, the software that is used is mentioned, then it focuses on functional requirements and test cases. The next step is the creation and launch of individual manual automated tests. Finally, these processes are evaluated in terms of speed, costs and usability.

Keywords: testing, automation, mobile application, Appium, Selenium, Java, software development, test case, test script, Android

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

SDLC	SYSTEM DEVELOPMENT LIFE CYCLE
HLD	HIGH LEVEL DESIGN
LLD	LOW LEVEL DESIGN
UAT	USER ACCEPTANCE TESTING
TC	TEST CASE
RTM	REQUIREMENTS TRACEABILITY MATRIX
E2E	END TO END
PIN	PERSONAL IDENTIFICATION NUMBER
IDE	INTEGRATED DEVELOPMENT ENVIRONMENT
USB	UNIVERSAL SERIAL BUS
UI	USER INTERFACE

1 Úvod

V dnešní době se setkáváme s čím dál větší popularitou různých aplikací. Tohoto softwaru je všude plno. Aplikace plní různé funkce a požadavky lidí. Některé jsou komplexní, dokážou zastat několik úkonů najednou, jiné jsou určeny pouze pro jednu konkrétní věc. Díky tomuto trendu, mít doslova na vše aplikaci, se rozšířil jejich vývoj. Dnes známe tisíce aplikací, které jsou stále více cíleny na mobilní zařízení jako jsou telefony, tablety nebo chytré hodinky. Mimo jiné pohání různé spotřebiče, domy, ale i třeba montážní linky ve strojírenství. Nalezneme je napříč všemi odvětvími. Od sociálních sítí, kde lidé sdílejí zážitky, fotky, informace a které slouží jako takový komunikační kanál, přes program simulující navigaci, po bankovní aplikace, které nám umožňují sledovat aktuální stav konta nebo posílat platby odkudkoliv. Čím více je daná aplikace komplexnější a složitější, tím ale také více vzniká prostor pro možné chyby. Může se jednat jen o grafické nedostatky, ale také o kritické defekty, které mohou mít fatální důsledky.

Odvětví vývoje, které se stará a kontroluje právě kvalitu a funkčnost, je testování softwaru. Tento často podceňovaný krok vývoje je velmi důležitý a má velký vliv na konečný úspěch či neúspěch konečného softwaru. Testovací tým komplexně kontroluje odvedenou práci vývojářů a poukazuje na chyby a nedostatky, které se musí opravit nebo změnit. Konkrétně v bankovnictví, kde se manipuluje s penězi, se velmi apeluje na kvalitu, funkčnost a hlavně bezpečnost. Banka si nemůže dovolit větší chyby, protože může nenávratně přijít o důvěru svých zákazníků. Testování se tomu snaží zamezit.

Dalším trendem je vydávat aplikaci do produkce („mezi lidi“) bez řádného otestování, tím se ušetří čas a peníze za testování a software se může vyvíjet déle. Tento postup lze aplikovat na software, který není kritický pro daný podnik. V již zmíněném bankovnictví tento postup provést nelze, proto se hledají jiné cesty, jak urychlit fázi testování. Jednou z nich je automatizace testování. Tester zaměřený na tento typ testování napíše testovací skript určený na konkrétní aplikaci a její funkce. Následně stačí skript spustit a počítač dokáže sám provádět proces hledaní chyb.

Tato práce právě proto cílí na tuto problematiku. Vysvětlení toho, co vlastně testování je a co obnáší. Prakticky bude manuálně a automatizovaně otestována konkrétní bankovní mobilní aplikace podle zvolených požadavků. Následně bude zjišťováno a zhodnoceno, jestli tyto postupy jsou rychlé, výhodné a opravdu použitelné při vývoji tohoto softwaru.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této diplomové práce je automatizovaně otestovat zvolené funkce bankovní aplikace pro mobilní zařízení a zjistit, jestli napsané skripty testů budou rychlejší, použitelnější a finančně dostupnější oproti manuálnímu testování na tomto typu softwaru.

Dílčí cíle jsou:

- Zvolit si vhodné nástroje pro testování
- Zvolit si požadavky a funkcionality pro testování
- Napsat testovací případy jednotlivých testovaných funkcionalit
- Manuálně otestovat funkce front-endu mobilní aplikace daného bankovnictví
- Napsat testovací skripty a následně otestovat aplikaci automatizovaně
- Zhodnotit a porovnat manuální a automatizované testování po stránce rychlosti, finančního hlediska a celkové použitelnosti

2.2 Metodika

Teoretická část diplomové práce bude sepsána za pomoci dostupné literatury, která se týká testování softwaru. Práce bude vycházet z dostupných knih, internetových stránek i odborných článků, které řeší tento segment vývoje. Tyto zdroje budou následně uvedeny v seznamu použité literatury na konci této práce. V této rešeršní části bude nejprve zmíněn postup vývoje softwaru, životní cykly a jejich modely, které samotné testování zařazuje do procesu vývoje jakéhokoliv systému. Následovat bude popis testování, kde bude vysvětleno, co to je a proč je důležité. Dále bude zmíněn testovací tým, který samotné testování provádí a k tomu dokumentace, která je nezbytná pro tuto část vývoje. Poté budou popsány jednotlivé úrovně a druhy testování, které jsou používány. V tomto úseku bude také popsáno manuální a automatizované testování. Jako poslední v této sekci budou zmíněny mobilní aplikace a popsány jejich jednotlivé druhy. Na tuto část bude navazovat ukázka používaných nástrojů pro testování mobilních aplikací.

Praktická část práce bude zaměřena na otestování funkčnosti front-endu již konkrétní mobilní aplikace mobilního bankovnictví. Celý postup bude demonstrován na mobilní platformě Android. Nejprve budou představeny jednotlivé nástroje, které budou třeba pro tuto práci. Poté budou specifikovány jednotlivé požadavky na konkrétní funkcionality aplikace, které se dále budou testovat. Následně na jednotlivé tyto části aplikace budou vypracovány testovací případy a jejich jednotlivé kroky. Poté pomocí těchto případů bude mobilní aplikace manuálně otestována. Další na řadu přijde vytvoření automatizovaných testů na tyto konkrétní funkce a následně jejich spuštění. Nakonec, po zprovoznění automatizovaných testů, bude následovat porovnání rychlosti, finanční výhodnosti a použitelnosti oproti manuálním testům na tomto konkrétním softwaru.

3 Teoretická východiska

3.1 Vývoj softwaru

Vývoj softwaru se týká iteračního logického procesu, jehož cílem je vytvořit a naprogramovat požadovaný software tak, aby splňoval jednotlivé cíle. Tohoto cíle je dosaženo nejen vývojářem softwaru, který vytváří počítačový kód, ale i ostatními členy projektového týmu. Proces vývoje zahrnuje také několik kroků, jako jsou výzkum, návrh a tok dat, psaní dokumentace, komplexní testování, ladění a opakované zdokonalování. Tento proces je znám jako životní cyklus vývoje softwaru (SDLC). [1]

3.2 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru (dále už jen SDLC) je definován jako systematický přístup, používaný softwarovým průmyslem při navrhování, vývoji a testování vysoce kvalitního softwaru. Cílem procesu SDLC je vyrábět vysoce kvalitní software, který splňuje očekávání zákazníků. Vývoj systému by měl být dokončen v předem definovaném časovém rámci a nákladech. SDLC se řídí podle podrobného plánu, který vysvětluje, jak plánovat, vytvářet a udržovat konkrétní software. Každá fáze životního cyklu SDLC má svůj vlastní proces a výstupy, které vstupují do další fáze. Celý proces SDLC se dá rozdělit do následujících fází: [2]

1. Plánování a analýza požadavků
2. Definování požadavků
3. Návrh
4. Kódování
5. **Testování**
6. Instalace/ Nasazení
7. Údržba

3.2.1 Plánování a analýza požadavků

Plánování a analýza požadavků je nejdůležitější a základní fází procesu SDLC. Provádí jej zkušení, seniorní členové týmu se vstupy od zákazníka, obchodního oddělení, průzkumů trhu a odborníků v oboru. Tyto informace jsou pak použity k plánování základního projektového přístupu a k provedení studie proveditelnosti produktu v ekonomické, provozní a technické oblasti. V této fázi se také provádí plánování požadavků pro zajištění kvality a identifikace rizik spojených s projektem. Výsledkem studie technické proveditelnosti jsou různé technické postupy, které lze použít pro úspěšnou realizaci projektu s minimálními riziky. [2][3]

3.2.2 Definování požadavků

Po provedení analýzy požadavků je dalším krokem jasné definování a zdokumentování požadavků na produkt a jejich schválení od zákazníka nebo analytiků trhu. Požadavky se dělí na explicitní a implicitní.

- **explicitní** – jedná se o požadavky, které jsou jasně řečené nebo zdokumentované
- **implicitní** – jedná se o požadavky, které nejsou nikde uvedeny (specifikovány), ale přesto se předpokládá, že budou splněny [2][3]

3.2.3 Návrh

Tato fáze slouží jako vstup pro další fázi modelu. V této fázi jsou vytvořeny dva druhy návrhových dokumentů:

- Design na vysoké úrovni (HLD)
    - Stručný popis a název každého modulu
    - Přehled funkčnosti každého modulu
    - Vzájemné vztahy a závislosti mezi moduly
    - Databázové tabulky byly identifikovány spolu s jejich klíčovými prvky
    - Kompletní schémata architektury spolu s technologickými detaily

- Nízkoúrovňový design (LLD)
    - Funkční logika modulů
    - Databázové tabulky, které obsahují typ a velikost
    - Kompletní detail rozhraní
    - Řeší všechny typy problémů se závislostí
    - Výpis chybových zpráv
    - Kompletní vstupy a výstupy pro každý modul [3]

3.2.4 Kódování

V této fázi SDLC začíná samotný vývoj softwaru. Vývojáři začnou stavět celý systém psaním kódu pomocí zvoleného programovacího jazyka. Ve fázi kódování jsou úkoly rozděleny do jednotek nebo modulů a přiřazeny různým vývojářům. Jedná se o nejdelší fázi procesu životního cyklu vývoje softwaru. Vývojáři musí dodržovat pokyny pro kódování definované jejich organizačními a programovacími nástroji. K programování se používají vyšší programovací jazyky, jako jsou např. C, C ++, Pascal, Java a PHP. Programovací jazyk se volí s ohledem na typ vyvíjeného softwaru. [2][3]

3.2.5 Testování

Po dokončení vývoje se software nasadí do testovacího prostředí. Testovací tým začne testovat funkčnost celého systému. To se provádí za účelem ověření, zda celá aplikace funguje podle požadavků zákazníka. Během této fáze může testovací tým najít nějaké chyby, které sdělí vývojářům. Vývojový tým opraví chybu a odešle zpět k novému testu. Tento proces pokračuje, dokud nebude software bez chyb, stabilní a nebude fungovat podle obchodních potřeb daného systému. [2][3]

3.2.6 Instalace/ Nasazení

Jakmile testování softwaru skončí a v systému již nejsou neakceptovatelné chyby, spustí se proces konečného nasazení do produkce. Někdy se nasazení produktu děje ve fázích podle obchodní strategie organizace. Produkt může být nejprve propuštěn v omezeném segmentu a testován v reálném obchodním prostředí (UAT – uživatelské přejímací testování). [2][3]

3.2.7 Údržba

Jakmile je systém nasazen a zákazníci ho začnou používat v jeho vyvinuté formě, dochází k následujícím třem činnostem:

- **Oprava chyb** – chyby jsou hlášeny kvůli některým scénářům, které nebyly vůbec testovány a projevily se později
- **Upgrade** – nahrazení aplikace její novější verzí
- **Vylepšení** – přidání některých nových funkcí do stávajícího softwaru

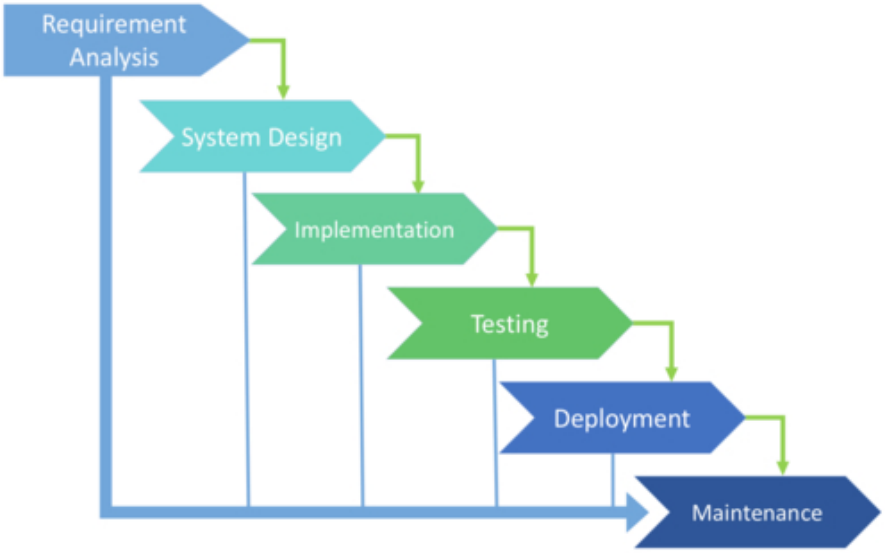
Hlavním zaměřením této fáze SDLC je zajistit, aby systém nadále fungoval podle specifikací a požadavků, které jsou uvedeny v první fázi projektu. [3]

3.3 Modely SDLC

Modely životního cyklu vývoje systému jsou zjednodušeným znázorněním onoho procesu v projektu. Každý model představuje svoje specifikace a rozdílnosti od ostatních. Každý model má své kladné a záporné stránky a každá metoda se hodí použít na jiný typ projektu. Mezi nejznámější metody patří určitě Waterfall a Agile. Neexistují ale pouze tyto dva, je jich celá řada. Zde je výčet těch známějších včetně vysvětlení: [4]

3.3.1 Waterfall

V překladu vodopád je kaskádový model SDLC, který dokonale vystihuje svůj název. Tento model zahrnuje postupné provádění všech fází až do konce a vytváří jakýsi tok, sekvenční přístup. Další fázi vývoje začne, až ve chvíli, kdy je úplně dokončena předešlá operace. Tento proces si klade důraz na přísné zdokumentování a předdefinování vlastností, které očekává pro každou fázi tohoto životního cyklu vývoje softwaru. Jedná se o nejstarší a nejjednodušší z metodik. Tento model vypadá takto: [4][5]



Obrázek 1 - Waterfall model [6]

Výhody

- Jednoduché použití a pochopení
- Ideální pro malé až středně velké projekty, kde jsou jasné a neměnné požadavky
- Snadné určení klíčových bodů ve vývojovém cyklu
- Snadno lze klasifikovat a upřednostňovat úkoly

Nevýhody

- Není to nejlepší volba pro komplexní a objektově orientované projekty
- Nevhodné pro dlouhodobé projekty
- Návrat do dřívějších fází není podporován

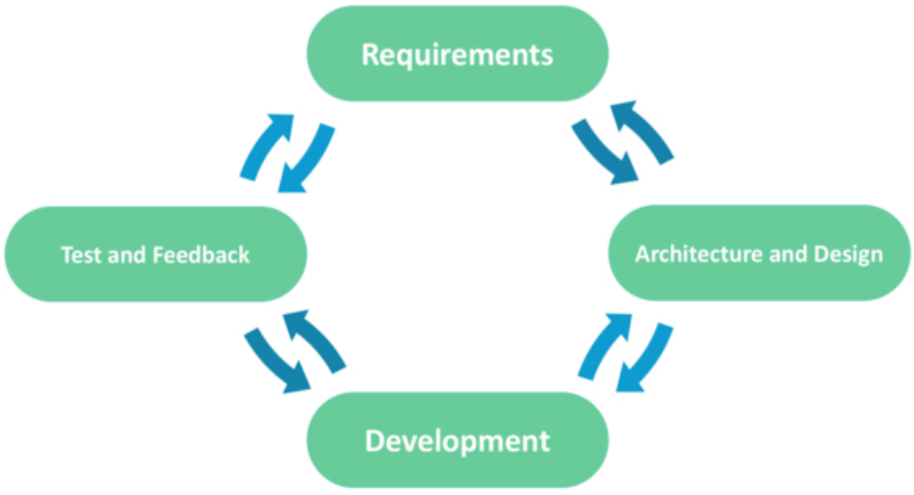
- Problémy se občas projeví až ve fázi testování
- Neposkytuje dostatek zpětné vazby od zákazníka

Doporučení užití metody

- Projekt je kratší
- Projekt má jasné cíle a řešení
- Existence dobře zdokumentovaných požadavků [5]

3.3.2 Agile

Agilní metodiky zohledňují hodnoty agilního manifestu a kladou obvykle důraz na změny směru a priorit, plochou až skoro žádnou hierarchii, získávání informací od zákazníka, neustálé učení se. Tento typ modelu zdůrazňuje interakci mezi právě zákazníky a členy projektového týmu v průběhu celého projektu. Pomocí tohoto modelu je výtvor vydáván postupně, vždy s malými změnami oproti předchozímu verzi. Při každé iteraci je produkt testován testery. V agilní metodologii zákazník vidí po každé iteraci vývoje výsledek a může vyjádřit, zda je s ním spokojen. Agilní metodika je znázorněna na následujícím obrázku. [4][5]



Obrázek 2 - Agile model [7]

Výhody

- Rychlé vydání první verze (části) produktu
- Větší flexibilita při reagování na časté měnící se požadavky
- Rizika jsou minimalizována díky flexibilnímu procesu změn
- Zákazník má přehled během vývoje

Nevýhody

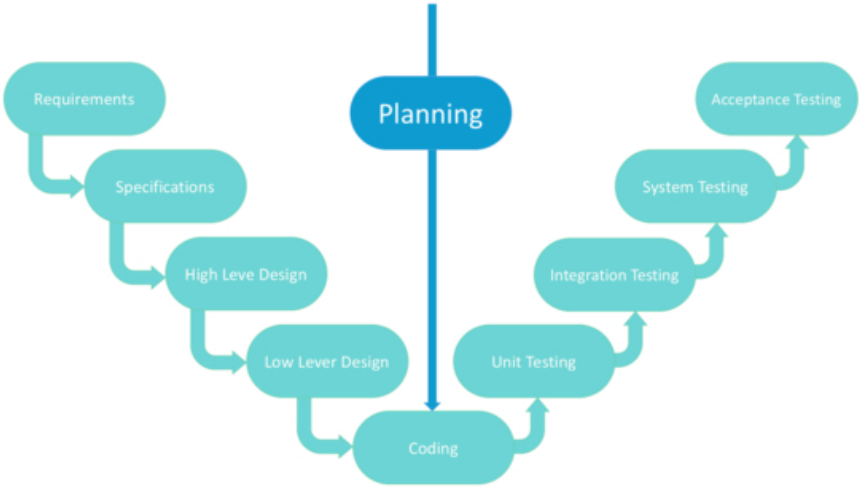
- Menší předvídatelnost
- Větší nároky na vývojáře a zákazníky
- Problémy s měřením konečných nákladů kvůli trvalým změnám
- Nové požadavky mohou být v rozporu s existující architekturou
- Nedostatek potřebné dokumentace

Doporučení užití metody

- Potřeby uživatelů se dynamicky mění
- Nižší cena za provedené změny díky mnoha iteracím
- Pevně definovaný čas projektu [5]

3.3.3 V-Shaped model

Model ve tvaru V je rozšířením vodopádového modelu a je založen na asociaci testovací fáze pro každou odpovídající vývojovou fázi. To znamená, že pro každou jednotlivou fázi vývojového cyklu existuje přímo spojená fáze testování. Jedná se o vysoce disciplinovaný model a další fázi začíná až po dokončení předchozí fáze. Model vypadá takto: [5][8]



Obrázek 3 - V-shaped model [9]

Výhody

- Testování a ověřování probíhá i v raných fázích projektu
- Ideální pro malé projekty se statickými a jasnými požadavky
- Požadavky lze ovšem změnit (např. oproti Waterfall)

Nevýhody

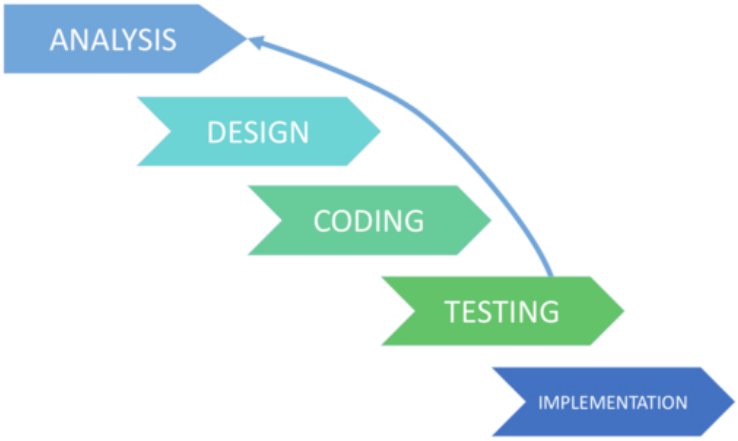
- Relativně velká rizika
- Model není vhodný ani navržený pro krátkodobé projekty
- Nedostatek flexibility

Doporučení užití metody

- Pro projekty, kde je vyžadováno přesné testování produktu
- Pro malé a středně velké projekty, kde jsou požadavky přísně předdefinovány [5]

3.3.4 Iterative model

Jedná se o postup vývoje, který by se dal zařadit pod agilní metodiku. Nepotřebuje úplný seznam požadavků před zahájením projektu. Tento proces se opakuje a umožňuje vytvářet nové verze produktu pro každý cyklus. Každá iterace (která trvá od dvou do šesti týdnů) zahrnuje vývoj samostatné komponenty systému a poté je tato komponenta přidána do dříve vyvinuté funkce. Metoda je graficky znázorněna na obrázku 4. [5][8]



Obrázek 4 - Iterative model [10]

Výhody

- Lze použít paralelní vývoj
- Flexibilita a připravenost na změny požadavků od zákazníka
- Testování a ladění je snadné během každé iterace
- Je snazší kontrolovat rizika, protože vysoce rizikové úkoly jsou dokončeny jako první
- Pokrok je snadno měřitelný

Nevýhody

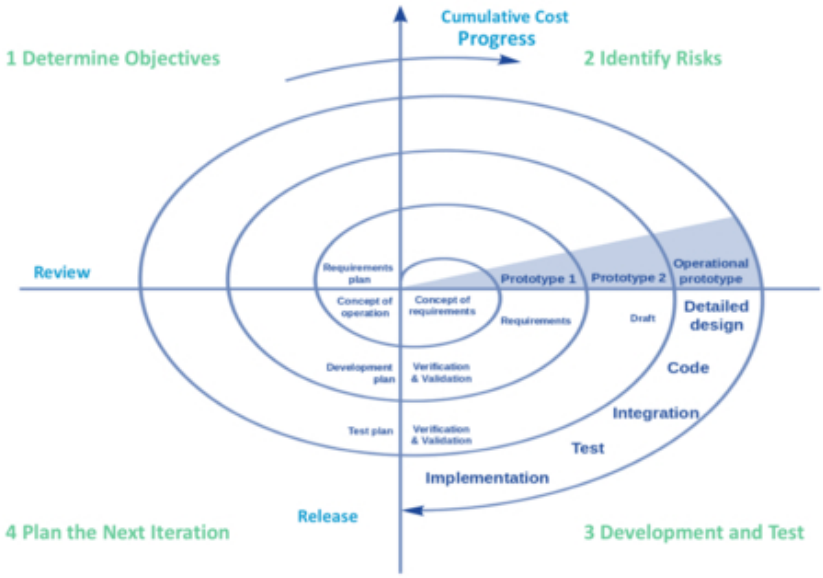
- Mohou se objevit problémy s architekturou nebo designem
- Tento proces je obtížné řídit
- Nevhodná volba pro malé projekty

Doporučení užití metody

- Pro velké projekty
- Lepší kvalita produktu
- Včasná eliminace rizik [5]

3.3.5 Spiral model

Jedna z nejpružnějších metodik SDLC, která do sebe kombinuje výše zmíněné modely Waterfall a Iterative s velkým důrazem na analýzu rizik. Projekt prochází čtyřmi fázemi znovu a znovu ve „spirále“, dokud není dokončen, což umožňuje několik opakování. Tento model umožňuje vytvořit velmi kvalitní a přizpůsobivý produkt. Graficky znázorněná spirála může vypadat takto: [4][8]



Obrázek 5 - Spiral model [11]

Výhody

- Proces vývoje je přesně zdokumentován, ale je možné ho přizpůsobit změnám
- Škálovatelnost umožňuje provádět změny a přidávat nové funkce i v relativně pozdních fázích
- Relativně brzo hotový prototyp
- Uživatelé mohou dříve odhalit nedostatky

Nevýhody

- Projekt se může prodražit
- Neefektivní pro malé projekty
- Velký počet opakování cyklu způsobí nadměrnou dokumentaci

Doporučení užití metody

- Zákazník si není jistý požadavky
- Během vývojového cyklu se očekávají velké úpravy
- Projekty s předpokládaným středním nebo vysokým rizikem
- Nový produkt [4][12]

Modelů SDLC existuje více než těchto pět výše zmíněných. Mezi další používané můžeme zařadit například Big Bang model, Prototype Model, DevOps nebo W-Shaped model, tj. rozšířený V-Shaped model.

3.4 Obecná charakteristika testování

Všeobecně by se dalo říct, že testovat lze úplně všechno. Věci, lidské bytosti i zvířata lze podrobit nějakému testu a zjistit, jak se chovají, nebo zda se chovají správně. Pokud se jedná o lidskou bytost nebo zvíře, testování říká, jaká úroveň znalostí nebo dovedností byla získána. Velmi populárním a známým testem je psychologická diagnostika tzv. Rorschanův test, ve kterém se zjišťuje osobnost člověka pomocí inkoustových skvrn, na které subjekt reaguje. Ještě možná více známý je test osobnosti. Konkrétně například tzv. Myers-Briggs Type Indicator (MBTI). V informatice se zase posuzuje, zdali systém (zařízení) splňuje, překračuje nebo vůbec nesplňuje stanovené cíle. [13][14]

3.5 Testování softwaru

Testování softwaru je proces, který vyhodnocuje funkčnost softwarové aplikace s cílem zjistit, zda vyvinutý program splnil předem stanovené požadavky nebo ne. Dále identifikuje a řeší chyby, aby bylo zajištěno, že produkt je bez kritických vad, a to za účelem výroby kvalitního produktu. Testování softwaru je nesmírně nezbytným a často podceňovaným procesem. Testovat je důležité, protože softwarové chyby mohou být nákladné nebo dokonce nebezpečné. Mohou potenciálně způsobit peněžní a v nejhorším případě i lidské ztráty. Naše historie je plná takových příkladů. [14]

3.6 Testovací tým

Celý projekt je komplexním sdružením několika týmů, které mají rozdílné úkoly. Ačkoliv se jedná o jiné obory, je nutné, aby týmy spolu komunikovaly. Komunikace je jednou ze základních podmínek úspěšného projektu. Jedním z těchto týmů je testing, jehož hlavním úkolem je, aby byl výsledný produkt kvalitní, bez chyb a splňoval požadavky zákazníka. V testovacím týmu existuje, jako v ostatních týmech, jakási hierarchie a jednotlivé role. Zde ovšem také platí to, že se rozložení týmu podřizuje typu a velikosti projektu. Malé projekty mohou mít jen pár členů v testovacím oddělení, zatímco větší a komplexnější projekty jich mají i několik desítek. Není ani neobvyklé, že jeden pracovník zastává více rolí najednou. Pojmenování a počet rolí se také mění s typem projektu. Níže je výčet tří základních rolí v testovacím týmu včetně popisu jejich náplně práce a vlastnostmi. [15][16]

3.6.1 Tester

Na této pozici je pracovník zodpovědný za testování konkrétního produktu podle dokumentace. Role testera vyžaduje důkladné seznámení s aplikací nebo jakýmkoliv jiným softwarem, který bude testování podléhat. Na základě získaných znalostí se podílí na tvorbě testovací strategie, scénářů a návrhu testů. Mezi jeho hlavní činnosti patří technické otestování dané aplikace a hledání chyb, které následně reportuje zpět k opravě vývojářům. Někteří testeři mohou být specializovaní na automatizaci testů. Do jejich zodpovědnosti pak přirozeně spadá nejen provádění testů a zadávání chyb, ale i vytváření automatizovaných skriptů. [16]

Důležité vlastnosti testera:

- Komunikativnost
- Analytické myšlení
- Pečlivost a trpělivost
- Zvídavost a bohatá fantazie
- Inovativnost [16]

3.6.2 Test analytik

Test analytik na rozdíl od testera tolik fyzicky netestuje, ale za to musí testovanému softwaru ještě více rozumět, a to do nejmenších detailů. Hlavní náplní test analytika je vytváření dokumentace a různých podkladů pro testování. Tato role je zodpovědná za počáteční identifikaci a následné definování požadovaných testů. Test analýza, test case, test data, test suit, test script... to jsou všechno odpovědnosti analytika. [16]

Schopnosti a vlastnosti jsou hodně podobné jako u testera s tím, že analytik nese ještě více zodpovědnosti za svoji práci, protože na základě jeho dokumentace se většinou testuje. V malých testovacích týmech je běžné spojení role testera společně s rolí test analytika. U těchto týmů je potom v obou rolích zároveň více osob se stejnými znalostmi a zodpovědnostmi. [16]

3.6.3 Test manager

Úlohou manažera testování softwaru je vést testovací tým, za který nese veškerou odpovědnost. Role zahrnuje neustálou komunikaci s vedoucím vývoje, vedoucím analýzy a projektovým manažerem / vedoucím celého projektu. Test manažer musí obhajovat a reportovat odvedenou práci svého týmu, plánovat a spravovat zdroje pro řešení různých problémů, které mohou brzdit testování. Je zodpovědný za testovací strategii projektu (test plán), harmonogram, vstupy a výstupy z testování... [15][16]

Důležité vlastnosti test manažera:

- Komunikativnost
- Plánování a delegování práce
- Kontrolovat
- Nebát se učinit rozhodnutí [16]

Platí nepsané pravidlo, že pokud test manažer odvedl svojí práci výborně a celý tým pracuje, jak má, není ho skoro během testování potřeba. Hlavní fáze pro test manažera je začátek a konec testingu. Bohužel dnes toto pravidlo v praxi skoro nevidíme, protože

projekty jsou s postupem času čím dál větší a komplexnější, což má za následek spoustu nepřesností a chyb, které se řeší v průběhu projektu. [15]

3.7 Testovací dokumentace

Úspěšné otestování aplikace nezávisí pouze na lidech, kteří testují, ale také na kvalitní dokumentaci, bez které se tento proces neobejde. Dokumentace je nedílnou součástí testingu a na její tvorbě a budoucím udržování závisí úspěch projektu. Ve většině případů se vyvíjí ještě před začátkem testování a v průběhu se jen upravuje a udržuje aktuální. Dokumenty obsahují veškeré informace o vyvíjeném softwaru a pomáhají po celou dobu projektu, a dokonce i po úspěšném vydání. Primárním úkolem dokumentace je vymezení a definování všech postupů, specifikací, plánů, protokolů, analýz aj. Běžně používané dokumenty jsou: [17]

3.7.1 Testovací plán (Test Plan)

Jedná se o stěžejní dokument pro celý proces testování softwaru, který by měl vzniknout úplně jako první. Plán testování popisuje společnou strategii, která bude použita při testování aplikace. Testovací plán může mít několik podob i verzí, ale ve většině případů zahrnuje následující:

- Úvod do dokumentu testovacího plánu
- Předpoklady pro testování aplikace
- Jednotlivé kroky testování
- Seznam testovacích případů pro testování
- Výčet funkcí, které mají/nemají být testovány
- Seznam vstupů a výstupů, které mají být testovány
- Harmonogram testování
- Zdroje, přidělené na testování aplikace
- Úkoly a milníky, které mají být dosaženy
- Možná rizika a odpovědnost [18]

Je důležité, aby tento dokument sepsala zkušená osoba, která ví, co dělá a má zkušenosti. Špatně nastavený plán může způsobit neúspěch celého projektu. [18]

3.7.2 Testovací případ (Test Case)

Testovací případy se píší konkrétně na jednotlivé funkční a nefunkční požadavky aplikace. Klíčovým bodem je jasnost a srozumitelnost jednotlivých kroků (steps). Sdružení několika po sobě jdoucích kroků tvoří právě jeden test case na jednu funkcionalitu. Kroky by měly být atomické a měli by dostatečně vystihovat danou situaci. Testovací případ popisuje, jak otestovat požadovanou funkčnost, to znamená, co má být zadáno na vstupu a co lze očekávat na výstupu. Testovací případy si můžeme rozdělit na:

- **Logický testovací případ** (logical test case) je abstraktní popis toho, co se má otestovat, definuje obor a množinu hodnot.
- **Fyzický testovací případ** (physical test case) je konkrétní popis toho, co se má otestovat a definuje, jaké hodnoty se mají zadat. [19]

3.7.3 Testovací scénář (Test Scenario)

Testovací scénář podrobně popisuje funkčnost testované aplikace. Používá se během průběhu testování a pomocí scénářů lze testovat komplexní aplikační logiku pomocí snadno vyhodnocujících testovacích scénářů. Testovací scénář má přímý vztah k jednotlivým testovacím případům (test cases). Jeden takovýto scénář se váže k jednomu nebo více testovacím případům. Lze samozřejmě testovat jen pomocí scénáře, ale to vyžaduje více zkušeností a větší důslednost daného testera. Testovacímu scénáři by měl být přidělen jednoznačný identifikátor a měl by obsahovat odkaz na Plán testování (ten by zase měl jednoznačně identifikovat testovací scénáře). Obvyklá sktruktura scénáře je:

- Vlastnosti, které budou testovány
- Přístup k testování
- Testovací případy
- Kritéria [20]

3.7.4 Matice sledovatelnosti požadavků (RTM)

Matice sledovatelnosti nebo matice sledovatelnosti softwaru je dokument, který sleduje a mapuje vztah mezi dvěma výchozími dokumenty. Těmito dokumenty v testování většinou myslíme jednotlivé požadavky a testovací případy. Tento dokument je důležitý pro přesvědčení zákazníka, že byly splněny a otestovány všechny jeho požadavky. Ke každému požadavku se váže několik důležitých informací, které pomáhají lepší a přehlednější orientaci. [21]

Výhody použití RTM:

- Plné pokrytí aplikace testováním
- Možnost identifikovat chybějící funkce
- Rychlá identifikace testovacího případu
- Přehledný celkový stav [21]

Samozřejmě mezi dokumentaci určenou k testování musíme zařadit i samotnou test analýzu, podle které se píší test case a scénáře. Dále by se do tohoto výčtu dal zařadit testovací nápad, tedy celá idea testování. Také za zmínku stojí testovací skript, který se používá při automatizaci testování a je potřeba jej někde uchovávat. A v neposlední řadě testovací data, které je potřeba vytvořit a následně udržovat po celou fázi testování. [21]

3.8 Defekt management

Obecně lze správu defektů definovat jako proces detekce chyb a jejich opravy. Je třeba říci, že v procesu vývoje softwaru dochází neustále k chybám. Jsou součástí softwarového průmyslu. Důvodem je skutečnost, že vývoj softwaru je poměrně složitý proces. Každá chyba má svůj životní cyklus, kterým musí projít. Životní cyklus defektu obecně začíná fází, kdy je chyba zjištěna, a končí, když je chyba uzavřena (viz. obrázek 6). Před uzavřením se chyba musí znovu retestovat a zavře se až ve chvíli, kdy je vše v pořádku. Pokud chyba stále přetrvává, vrací se zpátky k vývojářům. [15]

Obrázek 6 - Cyklus defektu [22]

3.8.1 Defekt a jeho atributy

Chyba softwaru je problém, způsobující selhání programu nebo produkování neplatného výstupu. Problém je způsoben nedostatečnou nebo chybnou logikou. Rozdělujeme chybu, defekt, selhání a incident. V praxi se tyto názvy velmi často zaměňují, jehož význam ukazuje na společný fakt – něco nefunguje.

- **Chybu** způsobí člověk, vývojář či analytik při práci na svém artefaktu (fragmentu kódu, kapitole analýzy, diagramu).
- **Defekt** (někdy označovaný jako bug) je pak odchylka aktuálního způsobu fungování softwaru od očekávaného. Defekt je projevem chyby. Je tedy třeba chybu napravit a defekt odstranit.
- **Selhání** (failure) systému může nastat v důsledku výše uvedeného defektu, systém či některá jeho část obvykle selže.

- **Incident** je podle definice ISTQB "jakákoliv událost, která vyžaduje prozkoumání". V praxi se ovšem tento pojem používá hlavně pro označení vady nalezené v produkčním prostředí. [15]

Atributy

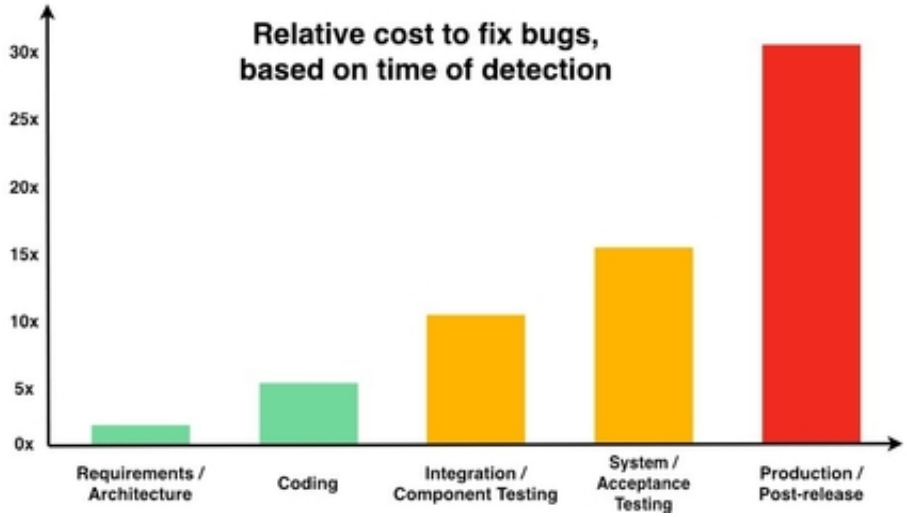
Pro snadnější a přehlednější správu defektů je důležité, aby jednotlivě zadané chyby měly nějaká pravidla a podobu. Požadované údaje o chybě se liší projekt od projektu, firma od firmy nebo program od programu. Zde jsou některé důležité atributy, které by měl obsahovat každý defekt:

- *Datum a čas nalezení* – pomůže při konkrétním odhalování chyby
- *Autor* – samotný tester
- *Adresát* – vývojář nebo pověřená osoba
- *Popis* – jednoduchý popis problému
- *Očekávané chování* – jak by se funkcionalita měla chovat
- *Skutečné chování* – jak se funkcionalita skutečně chová
- *Priorita* – závažnost chyby
- *Stav* – samotný stav defektu, v jaké fázi se nachází
- *Odkaz* – na jinou chybu, test case, požadavek
- *Prostředí* – v jakém testovacím prostředí se chyba vyskytuje
- *Testovací data* – na jakých datech se projevuje
- *Komentáře a historie*
- *Přílohy*… [15]

3.8.2 Rostoucí cena defektu

Náklady na zjištění a odstranění závad v softwaru exponenciálně rostou s časem a fází, ve které se daný produkt nachází (znázorněno na obrázku 7). Je mnohem snazší a levnější odhalit chybu v počátku, kdy vývojář vytváří kód. Jak plyne čas a naprogramovaná aplikace se posouvá o fáze dál, tak je pro vývojáře složitější tuto chybu najít. Mohou se na ní nabalit jiné a společně mohou vážně ohrozit fungování finálního systému. Pokud se

chyba dostane do produkce, cena na opravu je několikanásobná, navíc samotný software bude mít sníženou kvalitu, která může vést i ke ztrátě zákazníků. [23]



Obrázek 7 - Cena defektu [24]

V naší historii nalezneme několik událostí, které kvůli drobné chybě v softwaru stály milióny korun. Zde jsou dva příklady:

Let Ariane 5 501

Zkušební let vesmírné rakety Ariane 5, který měl číslo 501, skončil dne 4.6.1996 neúspěchem. Pár sekund po startu se spustila autodestrukce, a to kvůli selhání počítače, který vychýlil raketu. Chyba spočívala v tom, že ve vyrovnávací funkci došlo k přetečení proměnné. Systém očekával maximálně 16bitové číslo, ale příchozí číslo bylo vyšší. Vývoj Ariane stál kolem 8 miliard dolarů a samotný let, který nesl součásti satelitu, byl vyčíslen na 500 miliónů dolarů. [25]

Y2K Bug

Odhaduje se, že se jedná o nejdražší chybu v historii počítačů. Chyba spočívala v tom, že různé typy softwarů používaly při zápisu roku do data formát posledního dvojčíslí. Vše fungovalo správně, dokud nenastal zlomový rok 2000, ve kterém tyto softwary místo hodnoty 2000 uložily hodnotu 1900. Není v lidských silách dohledat kolik takových

softwarů bylo a kolik to mohlo celkově stát, ale odhaduje se, že tato chyba mohla způsobit škody ve výši 500 miliard dolarů. [25]

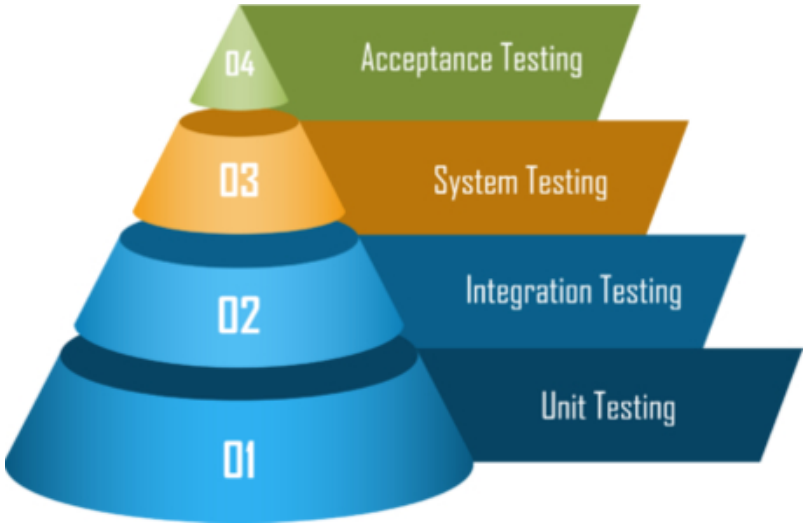
Příkladů z reálného života je celá řada. Opatření proti zbytečným chybám je několik. Zásadní roli v kvalitě softwaru hraje ovšem již několikrát zmíněná fáze testování, na kterou by se nemělo zapomínat. [25]

3.9 Úrovně testování

Testování se provádí na různých úrovních, a to buď po částech, nebo jako celek. Zde opět záleží na typu a velikosti projektu. Vyvíjený software prochází čtyřmi hlavními fázemi testování před jeho skutečným nasazením do produkce. Tyto čtyři úrovně se označují jako:

- Unit testy
- Integrační testy
- Systémové testy
- Akceptační testy

První tři zmíněné úrovně jsou prováděny dodavatelem, a to mezi vývojáři a testovacím týmem. Poslední, akceptační, většinou provádí sám zákazník. Tyto vztahy jsou výstižně znázorněny na následujícím obrázku. [26]



Obrázek 8 - Úrovně testování [27]

3.9.1 Unit testy

Tento typ testování se provádí v nejranějších fázích vývoje a ve většině případech je prováděn přímo vývojáři, kteří software sami vyvíjí. Při testování units (jednotek) testují jednotlivé části kódu. Za jednotku je považována samostatně testovatelná část aplikačního programu. Kontroluje se, zdali jednotlivé postupy, funkce, metody, třídy pracují správně a jsou akceptovatelné pro další fáze vývoje. Jedná se o krok, po kterém je software předán testerům k dalšímu testování. [26]

3.9.2 Integrační testy

Tento druh testování je prováděn již testery. Cílem integračního testování je otestovat různé části softwaru v kombinacích s ostatními. V této fázi jsou různé softwarové moduly kombinovány a testovány jako skupina, aby se zajistilo, že bude integrovaný software funkční a připravený pro systémové testování. [26]

3.9.3 Systémové testy

Testování na úrovni systému zahrnuje široké spektrum testování, zde je výčet některých testů:

- Test funkčnosti (functionality test)
- Test zabezpečení (security test)
- Test robustnosti (robustness test)
- Zátěžový test (load test)
- Test stability (stability test)
- Test hraniční zátěže (stress test)
- Výkonnostní test (performance test)
- Test spolehlivosti (reliability test) …

Testů je opravdu hodně, a proto je testování systému kritickou fází procesu vývoje softwaru. Důležité je splnit přísný časový harmonogram a před datem vydání odhalit

nejlépe všechny chyby. Následně je opravit a znovu otestovat, jelikož se může stát, že se objeví chyby nové. [26]

3.9.4 Akceptační testy

Po úspěšném dokončení všech testů v systémové části je produkt přesunut k otestování zákazníkovi. Zákazník provede vlastní řadu testů se svými testery. Akceptační testování probíhá podle testovacích scénářů na testovacím prostředí u zákazníka. Hlavním účelem UAT (User Acceptance Testing) je zkontrolovat akceptační kritéria a ověřit software před tím, než zákazník definitivně převezme daný produkt. Pokud se objevují chyby, vrací se zpátky k dodavatelským vývojářům k opravě. [26]

3.10 Rozdělení testů

V této kapitole jsou popsány a rozděleny některé známé testy, které se používají při testování softwaru. Každý z nich je něčím specifický a použitelný pro konkrétní situaci, proto je dobré vyvinutý produkt otestovat několika těmito testy. [26][28]

3.10.1 Funkční

Funkční test je typ testování softwaru, při kterém je systém testován na základě funkčních požadavků / specifikací. Jsou testovány především vstupy a výstupy. Funkční testování zajišťuje, že požadavky jsou aplikačně správně splněny. Zde jsou metody, kterými se testuje:

- **Black-box** je testovací technika, při které tester funkčně testuje aplikaci pomocí vstupů a výstupů. Produkt je černou skříňkou, do které se nelze podívat, vidíme jen jak vypadá a jak se chová navenek.
- **White-box** je testování navenek i uvnitř. Oproti Black-boxu se lze „podívat“ dovnitř – tester má přístup ke zdrojovému kódu a k chování uvnitř systému. [26]

3.10.2 Spolehlivosti

Test spolehlivosti je typ testování softwaru, který kontroluje, zda software může provést bezporuchovou operaci po stanovenou dobu v určitém prostředí. Patří sem:

- **Test stability** (stability test) je typ testování softwaru, který měří schopnost softwarového produktu zůstat v provozu při normálních podmínkách, aniž by došlo k jeho selhání.
- **Test hraniční zátěže** (stress test) je typ testování softwaru, který ověřuje stabilitu a spolehlivost systému. Tento test měří hlavně jeho robustnost a schopnost přizpůsobení se podmínkám, které extrémně zatěžují systém.
- **Test robustnosti** (robustness test) je atributem odolnosti, který měří chování systému za nestandardních podmínek. Robustnost je definována jako míra, do které systém pracuje správně za přítomnosti výjimečných vstupů nebo stresových podmínek prostředí. [26][28]

3.10.3 Výkonnosti

Účelem testování výkonu není najít funkční závady, ale odstranit překážky výkonu v softwaru nebo zařízení.

- **Výkonnostní test** (performance test) kontroluje rychlost, dobu odezvy, spolehlivost, využití zdrojů, škálovatelnost softwarového programu při očekávané pracovní zátěži.
- **Zátěžový test** (load test) je typ testování softwaru, který se provádí za účelem porozumění chování aplikace při konkrétním očekávaném zatížení. Zkoušení zátěže se provádí za účelem stanovení chování systému za normálních i kritických podmínek.
- **Benchmark test** je spíše porovnávacím typem testu. Otestuje známou funkcionalitu v softwaru a porovná ji s jinými, s konkurencí [26][28]

3.10.4 Bezpečnosti

- **Penetrační test** (Penetration test) je typ testování softwaru, který odhaluje zranitelnosti, hrozby, rizika v softwarové aplikaci a zabraňuje škodlivým útokům. Účelem bezpečnostních testů je identifikovat všechny možné mezery a slabiny softwarového systému, které by mohly mít za následek ztrátu informací, výnosů a dalšího. [26][28]

3.10.5 Použitelnosti a přístupnosti

Testy použitelnosti měří, jak snadno se dá daný software používat a jak moc je uživatelsky přívětivý. Toto testování se zaměřuje hlavně na snadné používání aplikace, flexibilitu při ovládání ovládacích prvků a schopnost systému splnit uživateli jeho cíle. Princip testování spočívá v tom, že se vyberou náhodní lidé, kteří otestují aplikaci. Testování může probíhat osobně, v nějaké tomu určené laboratoři, nebo vzdáleně.
Testování přístupnosti je definováno jako druh testování softwaru, který zajišťuje, že testovaná aplikace je použitelná pro osoby se zdravotním postižením, jako je sluch, barevná slepota, stáří a další znevýhodněné skupiny. Jedná se o podmnožinu testování použitelnosti. [29]
Lidé se zdravotním postižením používají podpůrnou technologii, která jim pomáhá při provozu softwarového produktu. Mezi tuto technologii patří například:

- Software pro převod mluveného slova na text
- Software pro čtení z obrazovky
- Software pro zvětšení obrazovky
- Speciální klávesnice vytvořená pro uživatele pro snadné psaní [29]

3.10.6 Regresní testy

Regresní testování testuje již existující software, z důvodu ověření, zda se přidáním nebo úpravou nezavlekly nové chyby. Kdykoliv vývojáři změní nebo upraví svůj kód, může mít

i malé vylepšení neočekávané důsledky. Účelem regresí je tedy zachytit chyby, které mohly být implementovány do nového buildu a zkontrolovat, zdali staré chyby se opět neprojevují. Toto testování provádí testeři podle předem připravených scénářů, a to i několikrát za sebou. Při každém novém kole regresí se většinou najde chyba, proto je žádoucí udělat cyklus několikrát. [30]

3.10.7 Smoke testy

Smoke je typ testování, který určuje, zda je nasazená sestava softwaru stabilní, nebo ne. Hlavním účelem smoke testu je projít nejdůležitější funkcionality aplikace a zjistit, zdali fungují a jestli je možné danou aplikaci začít řádně testovat. Obvykle se tento typ testu provádí před začátkem, ale i během testování regresí v pravidelných cyklech. Většinou se jedná o velmi krátké a jednoduché testy. [30]

Příklad: Do webové aplikace vývojář implementuje změnu pro tlačítko přihlášení. Tester tedy udělá jednoduchý smoke test tak, že zkusí funkci přihlášení, jestli nedošlo k nějaké chybě a lze se bez problému přihlásit.

Ostatní chyby typu špatná textace, barva, posunutí komponent... jsou náplní již zmíněných regresních testů. [30]

3.10.8 Free testy

Tzv. volné testování (známé také jako Monkey nebo Ad-hoc testování) je zvláštní druh ověřování funkčnosti aplikace. Oproti výše zmíněným testům se liší jednou zásadní věcí, a to je svobodné rozhodování testera. Nikde není přesná definice, co a jak se má udělat. K tomuto testu nepotřebujeme žádné test case nebo testovací scénáře. Stačí jen libovolně proklikávat aplikaci a zkoumat její chování. Vhodné užití pro komplexní systémy. V praxi často dojde k tomu, že ne každá možnost nebo posloupnost kroků je předmětem testovacích případů. [30]

3.11 Manuální a automatizované testování

Obě tyto hojně rozvinuté oblasti mají svůj směr a využití v rozdílných situacích. V rámci každé této kategorie jsou k dispozici specifické metody testování. V manuálním testování (jak název napovídá) jsou testovací případy prováděny ručně (tj. člověkem) bez jakékoliv podpory nástrojů nebo skriptů. Naproti tomu automatické testování využívá nástroje, skripty a software k otestování předem zvolených případů. [15][31]

3.11.1 Manuální testování

V tomto procesu testeři ručně provádějí testovací případy a generují testovací protokoly bez pomoci jakýchkoliv nástrojů pro automatizační testování softwaru. Je to klasická metoda všech typů testů, která pomáhá najít všechny druhy chyb v systému. Tester prochází jednotlivé kroky a usuzuje, zdali se software chová podle zadání. Jednotlivé nalezené chyby musí ručně zadat a předat vývojářům k opravě. [15][31]

Výhody

- Snadnější přizpůsobení změnám za chodu
- Lepší identifikace grafických chyb
- Lepší identifikace použitelnosti a přístupnosti
- Lepší pro neobvyklé situace

Nevýhody

- Méně spolehlivé výsledky testování (lidé chybují)
- Větší časová náročnost na provedení testu
- Opakování může vést ke "slepotě" a demotivaci testera
- Některé testy nejdou otestovat ručně (Performance test)

Obvyklé využití

- Menší projekty
- Funkční testování nové aplikace
- Průzkumové testování
- Testování použitelnosti
- Free testování [15][31]

3.11.2 Automatizované testování

V tomto odvětví testování softwaru testeři píší kód/testovací skripty, aby byl následně celý proces automatizovaný. Používají vhodné automatizační nástroje k vývoji testovacích skriptů podle zadání. Zde velmi záleží, jak tester daný test napíše. Automat sice projde zkoumanou část aplikace rychleji, ale nezaregistruje nic mimo, proto se hodí spíše na lehčí a jasné scénáře. Report chyb také záleží na systému, jaký si tester ke své práci zvolí. Automatizovat test se vyplatí hlavně v těch částech aplikace, kde zřídka dochází ke změnám, protože při sebemenší změně je potřeba testovací skript upravit. Použít lze například na regresní test. [15][31]

Výhody

- Opakovatelnost
- Rychlé provedení testu
- Důslednost (oproti člověku)
- S každým opakováním výhodnější

Nevýhody

- Vysoké počáteční náklady
- Údržba
- "Slepý" k chybám scénáře
- Použitý nástroj může mít chyby
- Podrobné skripty jsou neudržitelné

Obvyklé využití

- Větší projekty
- Regresní testování
- Časté opakování test case (s jinými daty)
- Zátěžové a výkonnostní testy
- Unit testování [15][31]

3.12 Mobilní aplikace

Jedná se o softwarové programy, které jsou vyvíjeny primárně pro mobilní, přenosná zařízení, jakou jsou telefony, tablety, chytré hodinky a jiné. Jednotlivé programy jsou navrženy tak, aby cílové zařízení mohlo plně využívat své specifické funkce a parametry. Pokud jde o operační systém, tak na telefony a tablety v dnešní době registrujeme dva dominantní velikány: Android a iOS. Mobilní operační systém Windows je již skoro nepoužívaný a většina vývojářů směřuje svůj vývoj ke dvou zmíněným gigantům. Na poli mobilního softwaru rozlišujeme tři základní druhy aplikací. Jsou jimi nativní, webové a hybridní aplikace. [32]

3.12.1 Nativní

Nativní mobilní aplikace je taková aplikace, která je vyvinutá pro konkrétní mobilní zařízení nebo platformu (například Android, iOS nebo i zmíněný Windows pro telefony). Každá tato platforma se programuje ve speciálním programovacím jazyce, který je pro ni určený. Aplikace pro Android často využívá programovacího jazyka Java nebo Kotlin a iOS jazyku Swift nebo Objective-C. Nativní aplikace mají mimo jiné lepší výkon a vyšší spolehlivost, protože používají základní architekturu cíleného systému a jeho vestavěné funkce. Nativní aplikace lze používat jak online, tak i offline, ale většinou pouze na cíleném operačním systému nebo zařízení. Pokud by aplikace měla být využívána jiným systémem, musí se velmi zásadně přeprogramovat nebo vyvinout znovu, což je finančně velmi náročné. [32]

Výhody

- Rychlejší než webové aplikace
- Může pracovat offline
- Zabezpečenější než webové aplikace (pokud je staženo z oficiálního obchodu)

Nevýhody

- Dražší vývoj než webové aplikace
- Nekompatibilní napříč různými platformami
- Vyšší náklady na údržbu a aktualizace [33]

3.12.2 Webová

Mobilní webová aplikace je aplikace přístupná přes mobilní prohlížeč. Snadno se k obsahu dostanete přes vestavěné prohlížeče, například Safari v systému iOS a Chrome v systému Android. Jsou to aplikace primárně vyvinuté pomocí technologií, jako jsou HTML5, CSS nebo JavaScript, které poskytují možnost přizpůsobení obsahu stránky. Jsou tedy v zásadě obsluhovány ze serveru a neukládají se primárně do zařízení. Webové mobilní aplikace mají společnou kódovou základnu a lze k nim přistupovat jako k jakékoliv jiné typické webové aplikaci. Již několik let lze webový obsah přizpůsobovat na menší zařízení díky tzv. responzivnímu designu, který upravuje velikost, rozložení podle velikosti daného zařízení a jeho rozlišení. Mobilní webové aplikace často využívají přístup k vytáčení telefonního čísla a poloze zařízení. K mobilním webovým aplikacím lze přistupovat pouze s platnou sítí (Wifi / 4G / 3G / 2G nebo nově vyvíjenou 5G). [32]

Výhody

- Není potřeba stahovat ani instalovat
- Snadná údržba
- Rychlejší a snadnější vývoj než nativní aplikace
- Nevyžaduje schválení obchodu s aplikacemi

Nevýhody

- Nefunguje offline
- Pomalejší než nativní aplikace
- Méně pokročilé funkce než nativní aplikace
- Kvalita a bezpečnost není vždy zaručena [33]

3.12.3 Hybridní

Hybridní aplikace je směs obou výše zmíněných, tedy nativní a webové aplikace. Jádro aplikace je psáno pomocí webových technologií (HTML, CSS a JavaScript), které jsou pak zapouzdřeny do nativní aplikace. Díky použití pluginů mohou mít tyto aplikace plný přístup k funkcím mobilního zařízení. Místo toho, aby se aplikace zobrazovala v uživatelském prohlížeči, se spouští pomocí nativní aplikace. Spustí se ve vlastním

vestavěném prohlížeči, který je pro uživatele v podstatě neviditelný. Například aplikace pro iOS by k zobrazení naší aplikace použila WKWebView, zatímco v systému Android by k provedení stejné funkce byl použit prvek WebView. [32][34]

3.13 Nástroje pro automatizované testování mobilního softwaru

Na trhu je celá řada nástrojů pro automatizaci testování. Tento druh softwaru pomáhá vývojářům a testerům najít defekty rychle a efektivně. Mimo jiné šetří čas, námahu při samotném vykonávání testů. Zde jsou ty nejlepší a nejpoužívanější nástroje, kterými lze takto testovat aplikace na mobilních zařízeních: [35]

Appium

Jedná se o open-source testovací nástroj používaný na platformách iOS a Android. Vývojáři nebo testeři mohou na tomto softwaru otestovat nativní, webové i hybridní mobilní aplikace. Pro napsání skriptů se používá celá řada programovacích jazyků. Podporuje C#, Javu, Ruby, PHP, JavaScript nebo také v poslední době stále rostoucí Python. Jako nástroj, který je multiplatformní, umožňuje znovu použít zdrojový kód mezi systémy Android a iOS. Appium funguje jako server běžící na pozadí, podobně jako Selenium server (také automatizovaný nástroj). Appium automatizuje Android pomocí knihovny UIAutomator, kterou poskytuje Google jako součást sady Android SDK. Na mobilních zařízeních může ovládat internetové prohlížeče Safari a Chrome. [35]

Robotium

Tento také open-source nástroj slouží pro testování platformy Android ve všech verzích a subverzích. Testuje všechny hybridní a nativní aplikace pro Android. Robotium používá pro psaní testů programovací jazyk Java. Používá se pro automatické testování černé skříňky pro Android aplikace. S testovacími skripty také umožňuje zápis funkcí, systémových a uživatelských akceptačních testovacích scénářů. [35]

MonkeyRunner

MonkeyRunner je speciálně navržen pro testování zařízení a aplikací na framework/ funkční úrovni. Tento nástroj obsahuje úžasné funkce, jako je ovládání více zařízení najednou, regresní testování, rozšiřitelná automatizace, funkční testování aplikací a hardwaru Android. Testy MonkeyRunner jsou psány v Pythonu. Jedinou větší nevýhodou MonkeyRunner je to, že je nutné psát skripty pro každé zařízení a testy vyžadují úpravy pokaždé, když se změní uživatelské rozhraní testovaného programu. [35]

UI Automator

Nástroj na testování UI, tedy uživatelského rozhraní. Tento nástroj plynule spolupracuje se všemi programy a aplikacemi pro Android. Funguje se všemi zařízeními, která podporují platformu Android verze 4.1 a vyšší. Tento software umí mimo jiné uzamknout a odemknout tablet nebo smartphone. Tato funkce rozšiřuje použití právě tohoto nástroje. [35]

Frank

Frank umožňuje testovat pouze aplikace a software pro iOS. Framework kombinuje JSON a Cucumber. Tento nástroj obsahuje inspektor aplikací „Symbioate“, který umožňuje vývojářům získat podrobné informace o spuštěné aplikaci. Je nejvhodnější pro webové aplikace a emulátory. Může být integrován s CI a provádět testy na zařízeních a simulátorech. [35]

MonkeyTalk

MonkeyTalk automatizuje funkční testování aplikací pro Android a iOS. Testování může provádět i osoba, která tolik nerozumí skriptování a programování, protože skripty MonkeyTalk jsou docela srozumitelné a jednoduché. Pomocí tohoto nástroje mohou testeři vytvářet také zprávy XML a HTML. Kromě toho lze také pořizovat snímky obrazovky, když dojde k selhání. MonkeyTalk podporuje různé emulátory. [35]

4 Vlastní práce

Téma "Automatizované testování mobilního softwaru" jsem si vybral, protože již pár let v tomto segmentu vývoje softwaru pracuji jako manuální tester a zajímá mě použitelnost automatizovaného testování v této oblasti. V praktické části budu vycházet z osobních poznatků a zkušeností získaných díky této práci. Testování bude demonstrováno na mobilní platformě Android, která je mi bližší a dostupnější než jiné platformy. Použité nástroje jsou ve většině případů multiplatformní, tudíž samotný výsledek by měl jít přenést s drobnými úpravami i na ostatní mobilní platformy. Testy budou cíleny na funkce front-endu nativní mobilní aplikace, která se zabývá bankovnictvím. Konkrétně se bude jednat o mobilní bankovnictví Smartbanking ČSOB, ve kterém použiji funkci DEMO účet, který by měl být pro testy dostačující a chováním od reálných dat se tolik neliší. S manuálním testováním mi bude pomáhat několik testerů, abych měl co nejreálnější výsledky. Kdybych manuálně testoval mnou napsané testovací případy, mohlo by dojít k velkému ovlivnění výsledků tohoto projektu.

4.1 Použitý software

V této části se pokusím vysvětlit, proč jsem si zvolil daný software pro svou práci a následně stručně popíši jednotlivé kroky nastavení těchto nástrojů nebo knihoven. Veškerý zvolený software je open-source, tudíž není potřeba za něj platit a lze jej zdarma využívat pro osobní účely.

Nejprve je třeba zvolit si vhodný nástroj pro automatizaci. Po zvážení několika možností jsem zvolil open-source nástroj Appium. Jednak protože je asi jeden z nejrozšířenějších, ale také kvůli obrovské komunitě, díky které existuje na internetu spousta návodů a řešených problémů. Velkou výhodou tohoto nástroje je, že lze využít jak pro Android, tak pro iOS. Tudíž napsaný skript by měl být přenositelný mezi těmito platformami. Appium podporuje celou řadu programovacích jazyků. Zde byla zvolena Java a s ním vývojové prostředí Eclipse, které je taktéž open-source (Eclipse IDE for Java EE Developers). Je nutné mít nainstalované Java SE JDK, které lze zdarma stáhnout

z oficiálních stránek Oracle. Dále je nezbytné mít reálné Android zařízení nebo si stáhnout např. Android Studio, které disponuje virtuálními zařízeními od Googlu. Následně je třeba vše naistalovat a propojit. Na internetu je celá řada podrobných návodů, například zde: https://www.guru99.com/introduction-to-appium.html.

K psaní skriptu potom stačí vytvořit nový projekt v IDE, ze kterého se i následně spouštějí testy. Pro testování již napsaného kódu je potřeba připojit reálné/ virtuální zařízení a spustit Appium server. Server se pouští v příkazové řádce, ve které lze následně najít i logy z provedených exekucí testů a určit, zdali byly úspěšné či nikoliv.

Appium disponuje zajímavou funkcí, díky které lze vyscreenovat danou obrazovku aplikace a zjistit konkrétní atributy stránky (třídu, název, ID…). Jedná se o UI Automator Viewer, který byl hojně využíván v této práci.

4.2 Požadavky pro testování

Vývoj softwaru není samoúčelný, ale jeho cílem je uspokojit očekávání daného zákazníka. Proto, než se začne vyvíjet nebo testovat, je nutné si zvolit klíčové požadavky, které by měla daná aplikace splňovat. Požadavků a přání zákazníka může být mnoho. Pro demonstrování této problematiky jsem vybral systémové funkční požadavky, které se mi zdají nejdůležitější pro mobilní bankovní aplikaci, a na kterých si myslím, že se mi bude dobře demonstrovat proces psaní testovacích případů a následné manuální a automatizované otestování. Zvolené požadavky jsou vypsány v následující tabulce:

Číslo	Název	Popis	Priorita
1	Jednorázová platba	Klient žádá zadat a odeslat jednorázovou platbu.	1
2	Zablokování platební karty	Klient se dožaduje dočasně zablokovat platební kartu.	2
3	Dobití kreditu	Klient chce dobít kredit na telefonní číslo.	3

Tabulka 1 - Požadované funkce k otestování

Výše uvedená tabulka funkčních požadavků zobrazuje tři vybrané požadavky s popisy, co se od jednotlivých funkcionalit čeká. Pro znázornění důležitosti (priority) požadavku je nastavena stupnice na 1-3, kde:

- 1 = Vysoká
- 2 = Střední
- 3 = Nízká

Všechny zvolené požadavky budou testy E2E (End to End). Tudíž všechny budou obsahovat samotné spuštění aplikace na začátku testu a následné vypnutí aplikace po dokončení zvoleného požadavku. Základním požadavkem tedy je:

- Možnost spustit aplikaci
- Proklikat úvodní obrazovky a „security check“
- Přihlášení do aplikace
- Odhlášení z aplikace
- Vypnutí aplikace

Níže uvedu jednotlivé body požadované funkce zvolených požadavků.

4.2.1 Požadavek – Jednorázová platba

Prvním požadavkem je jednorázová tuzemská platba, která je základní funkcí všech bankovních aplikací. Zde zákazník očekává:

- Projít celou jednorázovou platbu E2E
- Možnost vybrat si položku z menu aplikace
- Možnost vybrat si platební účet z nabídky vlastních nebo oprávněných účtů
- Možnost vložit číslo účtu příjemce
- Možnost vložit částku
- Možnost vložit libovolnou zprávu k transakci

- Možnost potvrdit transakci pomocí PINu
- Možnost vidět obrazovku s úspěchem/neúspěchem platby

4.2.2 Požadavek – Zablokování platební karty

Dalším požadavkem je možnost dočasně si zablokovat platební kartu. Tato funkce se může hodit při ztrátě nebo odcizení dané karty. Požadovány jsou tyto body:

- Projít dočasnou blokaci platební karty E2E
- Možnost vybrat si položku z menu aplikace
- Mít možnost na svojí kartě stisknout tlačítko pro dočasnou blokaci
- Možnost zvolit důvod blokace (textové pole)
- Možnost ověření pomocí PINu
- Možnost vidět obrazovku s úspěchem/neúspěchem operace

4.2.3 Požadavek – Dobití kreditu

Třetím požadavkem je možnost dobít si kredit na svoje nebo cizí telefonní číslo. Zde se očekávají tyto podmínky:

- Projít celou operaci dobití kreditu E2E
- Možnost vybrat si položku z menu aplikace
- Možnost vybrat si platební účet z nabídky vlastních nebo oprávněných účtů
- Mít možnost si vybrat operátora z nabídky tří hlavních v České republice (T-Mobile, O2 a Vodafone)
- Možnost vložit telefonní číslo
- Možnost vložit částku k dobití
- Možnost vložit zprávu
- Validace telefonního čísla – zobrazení hlášky pro špatné telefonní číslo
- Možnost ověření pomocí PINu
- Možnost vidět obrazovku s úspěchem/neúspěchem operace

4.3 Testovací případy

Na výše uvedené požadavky se budou vázat následující testovací případy, které se budou snažit je ověřit a otestovat. Jednotlivé testovací případy budou napsány tak, aby obsáhly co možná nejvíce zmíněných požadavků, nejlépe všechny. V praxi by byly požadavky o dost rozsáhlejší, složitější a bylo by velmi obtížné je obsáhnout v jednom jediném TC.

4.3.1 TC1 – Jednorázová platba

První test case se zabývá otestováním jednorázové tuzemské platby. Hlavním cílem tohoto případu je najít formulář platby a následně se pokusit vyplnit údaje a přes ověření dojít do zdárného konce odeslání platby. V tomto TC bude vyzkoušeno pole pro výběr účtu, účet příjemce, pole částka, zpráva, přechod na potvrzovací obrazovku a ověření pomocí PINu spolu s výslednou obrazovkou zdárného odeslání.

TC1	Jednorázová platba		
Preconditions:			
DEMO klient: IČ: 11111111 PIN: 11111			
Step	**Step Actions**	**Expected Results**	**Execution**
1	Spusť aplikaci mobilního bankovnictví	Aplikace se spustila a zobrazila se přihlašovací obrazovka	Manual
2	Přihlas se do bankovnictví vyplněním údajů: "IČ" a "PIN" a následně zvol tlačítko: "Aktivovat"	Uživatel je přihlášen na svém účtu	Manual
3	Rozklikni tlačítko: "Menu" (ikonka Hamburger menu)	Menu se zobrazilo	Manual
4	Přejdi na platby a rozklikni položku: "Jednorázová platba"	Zobrazil se formulář pro jednorázovou platbu	Manual
5	Rozklikni pole: "Z účtu"	Zobrazí se nabídka vlastních účtů	Manual
6	Vyber třetí účet	Účet se propsal do formuláře	Manual
7	Vyplň údaje do pole "Na účet": - "Předčíslí": nevyplňuj nic - "Číslo účtu": 3158911123 - "Kód banky": 0800	Údaje vyplněny a jsou zobrazeny ve formuláři	Manual
8	Vyplň pole: "Částka" na 100 Kč	Údaj se vyplnil	Manual
9	Scrolluj dolů a rozklikni položku: "Další pole"	Položka se rozklikla a zobrazila se další pole: - "Konstantní symbol" - "Specifický symbol" - "Zpráva"	Manual
10	Zkus vepsat libovolný text do pole: "Zpráva"	Text lze vložit a zvolený text se zobrazuje ve formuláři	Manual
11	Stiskni tlačítko: "Pokračovat"	Tlačítko přesměruje na "Shrnutí platby"	Manual
12	Scrolluj dolů a vyplň pole: "PIN" a zvol tlačítko: "Potvrdit"	Zobrazila se hláška: "Vaši platbu jsme přijali ke zpracování"	Manual
13	Klikni na tlačítko: "Rozumím" a následně se odhlas z aplikace a aplikaci vypni	Aplikace se ohlásila a vypnula	Manual

Tabulka 2 - TC1: Jednorázová platba

4.3.2 TC2 – Zablokování platební karty

Druhý TC se bude věnovat rychlé dočasné blokaci platební karty. V tomto případě bude ověřeno tlačítko blokace na formuláři přehledu karet. Dále bude ověřen formulář dočasné blokace, konkrétně pole důvodu, pole PIN a následné odeslání a potvrzení požadavku. I zde je kontrolována obrazovka zdárného odeslaní.

TC2	Zablokování platební karty		
Preconditions:			
DEMO klient: IČ: 11111111 PIN: 11111			
Step	**Step Actions**	**Expected Results**	**Execution**
1	Spusť aplikaci mobilního bankovnictví	Aplikace se spustila a zobrazila se přihlašovací obrazovka	Manual
2	Přihlas se do bankovnictví vyplněním údajů: "IČ" a "PIN" a následně zvol tlačítko: "Aktivovat"	Uživatel je přihlášen na svém účtu	Manual
3	Rozklikni tlačítko: "Menu" (ikonka Hamburger menu)	Menu se zobrazilo	Manual
4	Přejdi na položku: "Platební karty"	Zobrazil se formulář s debetní kartou	Manual
5	Zvol tlačítko: "Blokovat"	Zobrazí se formulář dočasné blokace karty	Manual
6	Zkus vepsat libovolný text do pole: "Důvod"	Text lze vložit a zvolený text se zobrazuje ve formuláři	Manual
7	Vyplň: "PIN" a zvol tlačítko: "Potvrdit"	Zobrazí se pop-up okno s hláškou: "Opravdu si přejete provést dočasnou blokaci karty?" a tlačítka: "Odeslat" a "Zrušit"	Manual
8	Zvol tlačítko: "Odeslat"	Zobrazila se hláška: "Dočasná blokace se provedla v pořádku"	Manual
9	Klikni na tlačítko: "Rozumím" a následně se odhlas z aplikace a aplikaci vypni	Aplikace se ohlásila a vypnula	Manual

Tabulka 3 - TC2: Zablokování platební karty

4.3.3 TC3 – Dobití kreditu

U třetího a posledního TC bude zvolen trochu odlišný scénář než u dvou předešlých. Zde se nejprve nepodaří úspěšně dobít kredit a pro špatně vyplněné telefonní číslo se nepovede přejít na další obrazovku a odeslaní požadavku. Zde budou kontrolována pole účtu, operátora, telefonního čísla a částky. Po pokusu pokračovat se zobrazí varovná hláška o nesprávnosti telefonního čísla. Následně bude potřeba upravit číslo na platné a aplikace pustí testera dál k úspěšnému ukončení a odeslání požadavku po zadání PINu.

TC3	Dobití kreditu		
Preconditions:			
DEMO klient: IČ: 11111111 PIN: 11111			
Step	**Step Actions**	**Expected Results**	**Execution**
1	Spusť aplikaci mobilního bankovnictví	Aplikace se spustila a zobrazila se přihlašovací obrazovka	Manual
2	Přihlas se do bankovnictví vyplněním údajů: "IČ" a "PIN" a následně zvol tlačítko: "Aktivovat"	Uživatel je přihlášen na svém účtu	Manual
3	Rozklikni tlačítko: "Menu" (ikonka Hamburger menu)	Menu se zobrazilo	Manual
4	Přejdi na platby a rozklikni položku: "Dobití kreditu"	Zobrazil se formulář pro dobití kreditu	Manual
5	Rozklikni pole: "Z účtu"	Zobrazí se nabídka vlastních účtů	Manual
6	Vyber první účet	Účet se propsal do formuláře	Manual
7	Rozklikni pole: "Operátor"	Zobrazí se nabídka operátorů	Manual
8	Vyber operátora: "Vodafone"	Operátor se propsal do formuláře	Manual
9	Vyplň pole: "Telefonní číslo" na 731 000	Telefonní číslo se vypsalo do formuláře	Manual
10	Vyplň pole "Částka" na 200 Kč	Částka se propsala do formuláře	Manual
11	Potvrď dobití kreditu tlačítkem: "Pokračovat"	Zobrazí se pop-up okno s hláškou: "Vyplň platné telefonní číslo"	Manual
12	Klikni na tlačítko: "Zavřít" a následně oprav pole: "Telefonní číslo" na 731 000 123	Telefonní číslo se vypsalo do formuláře	Manual
13	Potvrď formulář tlačítkem: "Pokračovat"	Tlačítko přesměruje na "Shrnutí příkazu"	Manual
14	Vyplň pole: "PIN" a zvol tlačítko: "Potvrdit"	Zobrazila se hláška: "Příkaz k dobití kreditu byl přijat ke zpracování"	Manual
15	Klikni na tlačítko: "Rozumím" a následně se odhlas z aplikace a aplikaci vypni	Aplikace se ohlásila a vypnula	Manual

Tabulka 4 - TC3: Dobití kreditu

4.4 Manuální testování požadavků

V této části přijde na řadu manuální testování zvolených požadavků. Testovat budou tři zvolení testeři podle napsaných testovacích případů z předešlé kapitoly. Testování bude probíhat ve třech testovacích cyklech a u každého průchodu testem se bude měřit čas a zjišťovat, za jak dlouho daný test zvládli.

Postup zápisu časů bude následující. Testovací případy se budou testovat postupně a následně se sečte čas za celý cyklus, tzn. tester otestuje postupně první, druhý a třetí test case zvlášť a následně se tyto tři hodnoty sečtou. Vznikne tabulka, ve které u každého testera bude ke každému cyklu vždy jeden časový údaj. Následně se hodnoty cyklů u každého testera aritmeticky zprůměrují, až nakonec každý tester bude mít k sobě přiřazen jeden časový údaj.

Exekuce testů (v sec.)	Tester 1	Tester 2	Tester 3
1. Kolo	323	454	346
TC1	143	201	153
TC2	77	97	81
TC3	104	156	112
2. Kolo	300	407	312
TC1	127	179	146
TC2	81	100	71
TC3	92	128	95
3. Kolo	284	368	308
TC1	118	156	136
TC2	74	93	73
TC3	92	119	99
Arit. průměr kol	**302**	**410**	**322**

Tabulka 5 - Exekuce manuálních testů

Z výsledků je parné, že každý tester testoval trochu odlišnou rychlostí a podle toho také vycházejí výsledné hodnoty. Nejrychlejší tester byl Tester 1, který průměrně zvládl celý cyklus za 302 sekund. Naopak nejpomalejší byl Tester 2, který potřeboval průměrně na jedno kolo 410 sekund.

4.5 Automatizované testování požadavků

Po manuálním testování přichází na řadu testování automatizované a samotné napsání testovacích scriptů a jejich následné spuštění z IDE, které byla v tomto případě open source vývojová platforma Eclipse.

Nejprve bylo nutné naistalovat všechny potřebné nástroje a knihovny (viz kapitola 4.1 Použitý software). Následně připojit testovací zařízení. Použil jsem reálné zařízení oproti virtuálnímu telefonu, protože po několika pokusech bylo reálné zařízení, připojené přes USB kabel do počítače, mnohem rychlejší. Poté jsem spustil Appium server a začal jsem psát. Celý napsaný script bude vložen do příloh na konci celé práce. Níže uvedu vybrané části kódu, které popíši.

První třídu programu jsem zvolil k připojení telefonu a instalaci .apk souboru mobilního bankovnictví z počítače. Zde je ukázka:

```
File app = new File("C:\\xxx\\xxx\\xxx\\xxx\\src\\app.apk");

DesiredCapabilities cap=new DesiredCapabilities();
cap.setCapability(MobileCapabilityType.DEVICE_NAME, "MarekD");
cap.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"uiautomator2");
cap.setCapability(MobileCapabilityType.APP, app.getAbsolutePath());
cap.setCapability("appPackage","cz.csob.smartbanking.testing");
cap.setCapability("appActivity",
"com.smartbanking.android.sbank.app.MainActivity");

AndroidDriver<AndroidElement> driver=new AndroidDriver<>(new
URL("http://127.0.0.1:4723/wd/hub"), cap);
```

Zde stačilo stáhnout .apk soubor kamkoliv na svoje úložiště a odkázat absolutní cestou. Následně bylo potřeba nastavit testovací zařízení. Bylo použito reálné Android zařízení připojené přes USB kabel k počítači. Poté stačilo nastavit atributy aplikace a Appium server.

Po spuštění této sekce je již telefon připojený a lze již psát samotný kód pro jednotlivé TC. Pro všechny TC je stejný začátek spuštění aplikace a otevření menu s položkami, kde jsem řešil nejeden problém. Největší problém jsem měl s otevíráním menu, které se zavřelo, pokud aplikace nebyla úplně načtená. Tento problém jsem vyřešil takto:

```
WebDriverWait wait = new WebDriverWait(driver, 5);

wait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath("android
.widget.ImageView")));
driver.findElementById("cz.csob.smartbanking.testing:id/drawer_button").clic
k();
```

V této části kódu bylo třeba naimportovat Selenium knihovnu WebDriverWait, díky které bylo možné provést tuto operaci. Automat v této části čeká, než se zobrazí určitý prvek a až poté provede úkon (kliknutí na tlačítko). Jedná se o metodu tzv. Explicitního čekání, ve které lze nastavit maximální dobu dynamického čekání. Oproti tomu Implicitní čekání pozastaví automat na jasně definovanou dobu.

Další věcí, kterou jsem musel řešit je scrollování na formuláři, toto gesto jsem potřeboval hlavně u TC1, kde bez něj nešlo pokračovat. Tuto situaci jsem řešil tímto kódem:

```
driver.findElement(MobileBy.AndroidUIAutomator("new UiScrollable(new
UiSelector().scrollable(true).instance(0)).scrollIntoView(new
UiSelector().textMatches(\"Display other fields\").instance(0))"));

driver.findElementByXPath("//android.widget.TextView[@text='Display other
fields']").click();
```

Další ukázka kódu je posunutí obrazovky dolů, tedy tzv „scrollování“. Automat po zobrazení stránky začne scrollovat dolů, dokud nenarazí za zvolený element aplikace. Zde šlo o rozkliknutí textu, díky kterému se zobrazila další pole a automat mohl pokračovat.

Psaní scriptu bylo ve většině případů hledání jednotlivých elementů na stránce, které jsem hledal pomocí Xpath nebo Id. K tomuto nalézání jednotlivých elementů bylo důležité

použití nástroje UI Automator Viewer, díky kterému lze lehce najít jména, ID, třídy… konkrétních prvků na obrazovce.

Po napsání a zdokonalení celého testovacího scriptu na jednotlivé TC, byl uložen celý kód na konec této práce (viz 8 Přílohy). Dále bylo zapotřebí provést exekuci jednotlivých částí aplikace, stejně jako tomu bylo v předešle kapitole (4.4 Manuální testování požadavků). Spouštění testů probíhalo v samotné IDE Eclipse a jednotlivé kroky byly sledovány v Appium consoli (příkazové řádce), kde se generovaly jednotlivé logy, včetně následného výsledku testu s časem. Zde byly získány tyto časové výsledky:

Exekuce testů (v sec.)	Automat
1. Kolo	143
TC1	56
TC2	35
TC3	52
2. Kolo	146
TC1	55
TC2	37
TC3	54
3. Kolo	144
TC1	54
TC2	37
TC3	53
Arit. průměr kol	**144**

Tabulka 6 - Exekuce automatizovaných testů

Postup testování byl stejný jako u manuálního, tudíž každý TC byl testován třikrát. Exekuce jednotlivých TC v kole (cyklu) se sečetla a následně se udělal aritmetický průměr z výsledků jednotlivých kol. Výsledkem testu je, že automat v průměru stihl otestovat všechny TC za 144 sekund.

Jednotlivé logy z testování aplikace byly zapisovány v Appium consoli a výsledky na konci vypadaly takto (pokud se jednalo o úspěšný test):

```
[debug] [Instrumentation] .
[debug] [Instrumentation] Time: 56.145
[debug] [Instrumentation]
[debug] [Instrumentation] OK (1 test)
```

Pokud v nějakém případě nastala chyba, tak šlo pomocí logu zjistit, v jaké části test přestal pracovat. Pro názornou ukázku jsem schválně udělal chybu v loginu aplikace, kde jsem pozměnil název elementu. Následný log vypadal takto:

```
NoSuchElementError: An element could not be
located on the page using the given search
parameters.
[HTTP] <-- POST /wd/hub/session/3094cda2-d906-
4e23-bdc5-d2857a40e8ec/element 404 5173 ms -
422
[HTTP]
```

Celková odpověď ze serveru je mnohem obsáhlejší. Zkopíroval jsem jen malou část pro názornou ukázku, která vypovídá o tom, že nebyl nalezen hledaný element, a nakonec celý test skončil neúspěchem.

5 Výsledky a diskuse

Nyní po exekuci všech kol testů a zjištění výsledků je možné zhodnotit a porovnat rozdíly a výhody/ nevýhody mezi manuálním a automatizovaným testováním na této konkrétní aplikaci pro mobilní bankovnictví. Již na začátku práce byly stanoveny tři parametry ke zkoumání. Zhodnocení bude probíhat v těchto aspektech:

- Zhodnocení rychlosti
- Zhodnocení financí
- Zhodnocení použitelnosti

Zhodnocení rychlosti

Rychlost samotné exekuce testů lze lehce vyčíslit a spočítat. Z výsledků jednotlivých testů jsou již známé jednotlivé časy a ty stačí porovnat. Pokud se jedná o automatizovaný test, tak vyšla jedna zprůměrovaná celková hodnota. S manuálním testováním se zabývali tři rozdílní testeři, tudíž byly získány tři rozdílné zprůměrované časy. Pro porovnání je nutné tyto výsledky také zprůměrovat.

Exekuce testů (v sec.)			
Tester 1	302	Automat	144
Tester 2	410		
Tester 3	322		
Průměr	345		144
Poměr	1 : 0,42		

Tabulka 7 - Zhodnocení rychlosti

Z tabulky 7 je jasné, že automatizované testy byly rychlejší. Výsledkem rychlosti tedy je, že automatizované testy byly 2,38krát rychlejší než manuální.

Zhodnocení financí

Cena je asi nejdůležitější parametr, který zajímá snad všechny odpovědné osoby za daný projekt. Proto se pokusím finančně ohodnotit testování zvolených požadavků. Pro tuto operaci je třeba zjistit, kolik stojí práce jednotlivých pracovníků. Pro odhad mezd jsem zvolil webovou stránku jobs.cz (dostupné z: https://www.jobs.cz/). Zde jsem prošel několik inzerátů nabídek práce a na základě těchto informací jsem zvolil nejčastější mzdy testerů a IT/test analytiků, které jsou v následující tabulce:

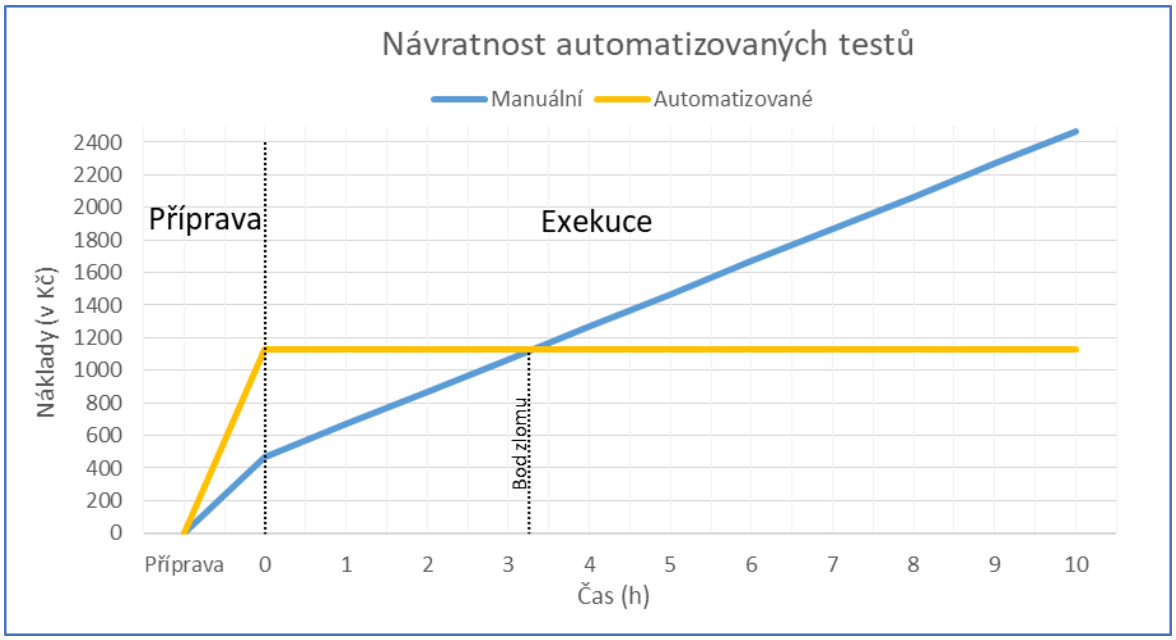
Pozice\mzda	Kč/měsíc	Kč/hodina
Manuální tester	32000	200
Automatizovaný tester	45000	281,25
IT analytik	50000	312,5

Tabulka 8 - Průměrné mzdy pracovníků (2019)

Pro následující výpočty jsem si rozpočítal jednotlivé měsíční mzdy na hodiny (za předpokladu standardní pracovní doby 160 hodin měsíčně). Jednotlivé mzdy se samozřejmě liší podle zkušeností, podniku a kraje, ve kterém daný zaměstnanec pracuje. Vypracování a příprava testovacích případů tohoto rozsahu zabere průměrnému analytikovi kolem hodiny a půl. Pokud se jedná o automatizované testy, tak napsání a vyladění podobných scriptů zabere zkušenějšímu testerovi půlku pracovního dne, tudíž asi čtyři hodiny. Tyto průměrné časové hodnoty jsem získal od lidí, kteří na podobných pozicích pracují.

Následující výpočet a graf je platný za těchto podmínek:

- Analytik napíše a připraví testovací případy pro manuální testery
- Automatizovaný tester napíše a vyladí script a dále jej neupravuje ani neudržuje
- Testovaná aplikace se nemění



Obrázek 9 - Graf – Návratnost automatizovaných testů

Z grafu výše lze vyčíst, že pokud bychom spouštěli testy pořád dokola, tak zlomový bod, ve kterém se automatizovaný test stane finančně výhodnější je cca v 3. hodině a 15. minutě. Pokud bychom to přepočetli na počet opakování, tak zlom se nachází mezi 33. a 34. kolem manuálního testu. To znamená, že manuální test je výhodnější tehdy, je-li počet kol maximálně 33. Více opakování by znamenalo vyšší náklady oproti automatizovanému scriptu. Mimo jiné za 3 hodiny a 15 minut by automatizovaný test dokončoval 85. kolo. Následující tabulka shrnuje cenu za otestování požadavků mobilního bankovnictví podle počtu cyklů (kol).

Cena otestování požadavků (Kč)		
Kolo	**Manuál.**	**Automat.**
0	468,75	1125
1	488,07	1125
10	661,95	1125
20	855,15	1125
30	1048,35	1125
40	1241,55	1125
50	1434,75	1125

Tabulka 9 - Cena otestování požadavků

Zhodnocení použitelnosti

Pokud přijde slovo na použitelnost, tak zde bude velmi těžké rozhodnout, jak moc je daný script použitelný pro testování. Použitelnost bude záviset na faktu, jak se budou dané požadavky testovat. Pokud by šlo o rychlost, tak jak je již výše spočítáno, automat je 2,38krát rychlejší než manuální test, což není závratný výsledek, ale použitelné to rozhodně je. Pokud půjde o cenu, zde záleží, kolik bude opakování. Dalším prvkem by mohlo být pohodlí, protože spouštění z IDE a následné sledování ne moc přehledných logů není pro každého optimální. Velmi také záleží na fázi vývoje softwaru. Pokud aplikace ještě není zcela vyvinutá, tak se provádí časté změny, což s sebou nese i časté úpravy testů. U kompletní a relativně funkční aplikace se již neočekává mnoho velkých zásahů, proto je pro automaty ideální. Tento konkrétní příklad je psaný na mobilní platformu Android, ale podle popisu nástroje Appium a technik, které jsem použil, tak by tento test měl být snadno přenositelný i na jiné platformy (např. iOS).

6 Závěr

Hlavním cílem diplomové práce bylo automatizovaně otestovat zvolenou bankovní aplikaci pro mobilní zařízení. Nejprve jsem ovšem zpracovával teoretickou část, ve které jsem čerpal z dostupných knižních a internetových zdrojů. Psaním této části jsem se dozvěděl spoustu nových věcí a poznatků, které mi dále pomohly v části praktické. Praktickou část jsem začal vysvětlením, proč a jaké softwarové nástroje budu dále používat. Pro samotné testování bylo třeba si zvolit rozsah a požadavky na funkcionality, které budou určeny k testingu. Zvolil jsem tři základní E2E testy na funkce, které se hojně využívají ve většině mobilních aplikací zabývajících se bankovnictvím. Po zvolení požadavků jsem napsal testovací případy, které poté sloužily pro samotné testování. S manuálním testováním mi pomáhali tři dobrovolní testeři, protože kdybych testoval svoje požadavky a testovací případy sám, tak bych dostal zkreslené výsledky a celý následný výzkum by ztrácel smysl. Během exekuce manuálních testů jsem stopoval čas a následně psal jednotlivé časy do tabulky. V další části jsem se pokusil naprogramovat testovací scripty, které by taktéž splňovaly zvolené požadavky. Nad touto problematikou jsem strávil hodně času, protože jsem testovací scripty nikdy předtím nepsal. Nejprve jsem si musel nastudovat používaný nástroj Appium, který se implementoval do IDE Eclipse. Jako programovací jazyk v tomto nástroji byla použita Java. Hojně jsem využíval nástroj UI Automator Viewer, pomocí kterého jsem získával jednotlivé elementy prvků aplikace přímo v telefonu. Zvolil jsem připojení reálného zařízení oproti virtuálnímu, jelikož takto to bylo o něco rychlejší a pohodlnější. Po napsání kódů jsem začal řešit a opravovat chyby. Potom jsem podrobil scripty plnému testu, ve kterém jsem zjistil, že jsou 2,38krát rychlejší než manuální testování. Tento výsledek mě překvapil, čekal jsem, že automat bude mnohem rychlejší, ale po delším zkoumání to není špatný výsledek, protože v mnoha případech musí automat čekat na aplikaci, než zobrazí daný prvek. Následně jsem se pokusil finančně ohodnotit celé testování a výsledkem bylo, že pokud by se měl automatizovaný test vyplatit oproti manuálnímu, musel by celkový test proběhnout více než 33krát. Toto číslo se může zdát vysoké, ale ve skutečnosti se testů provádí opravdu velké množství, aby se zamezilo nečekaným chybám. Jelikož se jedná o bankovní aplikaci, ve které se manipuluje s penězi, tak o to víc by měl být kladen větší důraz na důslednější testování. Poslední jsem zmínil použitelnost daného scriptu a celkově automatizovaného

testování na tomto konkrétním softwaru. Celé testování jsem cílil na mobilní platformu Android, ale podle oficiálních zdrojů Appium, je tento nástroj multiplatformní, tudíž by měl být i napsaný kód ve většině přenositelný na jiné platformy (např. iOS). Testovací script tedy použitelný je, ale jen na tuto konkrétní aplikaci ČSOB Smartbanking, a i tak určitě není dokonalý. Zkušenější automatizovaný tester by nejspíš napsal kód mnohem lépe. Hodlám se automatizaci nadále věnovat a zlepšovat se.

Testování je velmi zajímavá a důležitá část vývoje každého softwaru. Ať už se jedná o mobilní aplikaci nebo informační systém podniku. Bohužel bývá často opomíjena a podceňována kvůli šetření finančních prostředků, což nese za následky tvorbu nekvalitního softwaru, spoustu chyb a také velké náklady v budoucnu.

„Pachuť mizerné kvality zůstává dlouho poté, co zmizí sladká chuť nízké ceny.“

- Benjamin Franklin

7 Seznam použitých zdrojů

1. DOSHI, Kalpesh. Part 1 – Understanding Software Development Process. *Browserstack* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://www.browserstack.com/guide/learn-software-development-process

2. SWERSKY, Dave. The SDLC: 7 phases, popular models, benefits & more. *Raygun* [online]. 2018 [cit. 2020-01-30]. Dostupné z: https://raygun.com/blog/software-development-life-cycle/

3. GURU99. SDLC (Software Development Life Cycle) Tutorial: What is, Phases, Model. *Guru99* [online]. 2020 [cit. 2020-01-30]. Dostupné z: https://www.guru99.com/software-development-life-cycle-tutorial.html#4

4. ELGABRY, Omar. Software Engineering — Software Process and Software Process Models (Part 2). *Medium* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc

5. IMAM, Ali. SDLC Models – A Short Introduction to Different Models. *TestLodge* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://blog.testlodge.com/sdlc-models/

6. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. In: *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://cdn.shortpixel.ai/client/q_lossless,ret_img,w_768/https://existek.com/wp-content/uploads/2017/08/Itterative-SFDC-Model-768x432.png

7. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. In: *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://cdn.shortpixel.ai/client/q_lossless,ret_img,w_768/https://existek.com/wp-content/uploads/2017/08/Agile-SDLC-Model-Scheme-768x432.png

8. HALF, Robert. 6 basic SDLC methodologies: Which one is best? *RobertHalf* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://www.roberthalf.com.au/blog/employers/6-basic-sdlc-methodologies-which-one-best

9. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. In: *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://cdn.shortpixel.ai/client/q_lossless,ret_img,w_768/https://existek.com/wp-content/uploads/2017/08/V-Shaped-SDLC-Model-Scheme-768x432.png

10. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. In: *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://cdn.shortpixel.ai/client/q_lossless,ret_img,w_768/https://existek.com/wp-content/uploads/2017/08/%D0%A1%D0%BB%D0%B0%D0%B9%D0%B41-1-768x432.png

11. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. In: *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://cdn.shortpixel.ai/client/q_lossless,ret_img,w_768/https://existek.com/wp-content/uploads/2017/08/Spiral-SDLC-Model-Scheme-1-768x432.png

12. EXISTEK. SDLC Models Explained: Agile, Waterfall, V-Shaped, Iterative, Spiral. *Existek* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://existek.com/blog/sdlc-models/

13. ROUSE, Margaret. Testing. In: *SearchWinDevelopment* [online]. 2008 [cit. 2020-01-30]. Dostupné z: https://searchwindevelopment.techtarget.com/definition/testing

14. RAJKUMAR. What Is Software Testing – Definition, Types, Methods, Approaches. *Software testing material* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://www.softwaretestingmaterial.com/software-testing/

15. BUREŠ, Miroslav a kolektiv. *Efektivní testování softwaru: Klíčové otázky pro efektivitu testovacího procesu*. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

16. HLAVA, Tomáš. Testovací tým. *Testování softwaru* [online]. 2012 [cit. 2020-01-30]. Dostupné z: http://testovanisoftwaru.cz//?s=role

17. GURU99. Test Documentation in Software Testing. *Guru99* [online]. 2020 [cit. 2020-01-30]. Dostupné z: https://www.guru99.com/testing-documentation.html

18. ČERMÁK, Miroslav. Plán testování. *CleverAndSmart* [online]. 2009 [cit. 2020-01-30]. Dostupné z: https://www.cleverandsmart.cz/plan-testovani/

19. ČERMÁK, Miroslav. Testovací případ. *CleverAndSmart* [online]. 2009 [cit. 2020-01-30]. Dostupné z: https://www.cleverandsmart.cz/testovaci-pripad/

20. ČERMÁK, Miroslav. Testovací scénář. *CleverAndSmart* [online]. 2009 [cit. 2020-01-30]. Dostupné z: https://www.cleverandsmart.cz/testovaci-scenar/

21. RAJKUMAR. Requirements Traceability Matrix (RTM). *Software testing material* [online]. 2020 [cit. 2020-01-30]. Dostupné z: https://www.softwaretestingmaterial.com/requirements-traceability-matrix/

22. GURU99. Defect Management Process in Software Testing. In: *Guru99* [online]. 2020 [cit. 2020-01-30]. Dostupné z: https://www.guru99.com/images/TestManagement/testmanagement_article_4_4.png

23. SANKET. Exponential cost of fixing bugs. *Deepsource* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://deepsource.io/blog/exponential-cost-of-fixing-bugs/

24. SANKET. Exponential cost of fixing bugs. *Deepsource* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://assets.deepsource.io/c3f6fc6/images/blog/cost-of-fixing-bugs/chart.jpg

25. HARLEY, Nick. 11 of the most costly software errors in history. *DEV* [online]. 2018 [cit. 2020-01-30]. Dostupné z: https://dev.to/raygun/11-of-the-most-costly-software-errors-in-history-gbi

26. NAIK, Kshirasagar a Priyadarshi TRIPATHY. *Software testing and quality assurance: theory and practice*. Hoboken, N.J.: John Wiley, c2008. ISBN 978-0-471-78911-6.

27. KALWAN, Anamika. What are the Different Levels of Software Testing? In: *Edureka* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2019/08/levels-of-testing-462x300.png

28. PITTET, Sten. The different types of software testing. *Atlassian* [online]. 2020 [cit. 2020-01-30]. Dostupné z: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

29. BIGBY, Gerenne. Top 25 Awesome Accessibility Testing Tools for Websites. *Dyno mapper* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://dynomapper.com/blog/27-accessibility-testing/246-top-25-awesome-accessibility-testing-tools-for-websites

30. PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002. Programování. ISBN 80-722-6636-5.

31. TOMAS. Manual Testing vs. Test Automation. Pros and Cons – quick overview. *Krone* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://kroneit.com/manual-testing-vs-test-automation-pros-and-cons-quick-overview/

32. VERMA, Nishant. *Mobile Test Automation with Appium: Mobile application testing made easy*. Birmingham: Packt Publishing, 2017. ISBN 978-1-78728-016-8.

33. STEVENS, Emily. What Is The Difference Between A Mobile App And A Web App? *Careerfoundry* [online]. 2018 [cit. 2020-01-30]. Dostupné z: https://careerfoundry.com/en/blog/web-development/what-is-the-difference-between-a-mobile-app-and-a-web-app/

34. GRIFFITH, Chris. What is Hybrid App Development? *Ionicframework* [online]. 2019 [cit. 2020-01-30]. Dostupné z: https://ionicframework.com/resources/articles/what-is-hybrid-app-development

35. RUPARELIYA, Pratik. Top 10 Automated Testing Tools for Mobile Apps. *Medium* [online]. 2017 [cit. 2020-01-30]. Dostupné z: https://medium.com/intuz/top-10-automated-testing-tools-for-mobile-apps-8d9380e1757f

8 Přílohy

V přílohách se nachází celý kód testovacích scriptů nativní mobilní aplikace pro bankovnictví.

1. Třída – Připojení telefonu a nainstalování aplikace

```
import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
import org.openqa.selenium.remote.DesiredCapabilities;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.android.AndroidElement;
import io.appium.java_client.remote.MobileCapabilityType;

public class base {

        public static AndroidDriver<AndroidElement> Capabilities() throws
        MalformedURLException {

        File app = new File("C:\\xxx\\xxx\\xxx\\xxx\\src\\app.apk");

        DesiredCapabilities cap=new DesiredCapabilities();
        cap.setCapability(MobileCapabilityType.DEVICE_NAME, "MarekD");
        cap.setCapability(MobileCapabilityType.AUTOMATION_NAME,
        "uiautomator2");
        cap.setCapability(MobileCapabilityType.APP, app.getAbsolutePath());
        cap.setCapability("appPackage","cz.csob.smartbanking.testing");
        cap.setCapability("appActivity",
        "com.smartbanking.android.sbank.app.MainActivity");

        AndroidDriver<AndroidElement> driver=new AndroidDriver<>(new
        URL("http://127.0.0.1:4723/wd/hub"), cap);
        return driver;
    }
}
```

2. Třída – Třída pro TC1 – Jednorázová platba

```
import java.net.MalformedURLException;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import io.appium.java_client.MobileBy;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.android.AndroidElement;

public class payment extends base{

    public static void main(String[] args) throws MalformedURLException {

        AndroidDriver<AndroidElement> driver=Capabilities();
        WebDriverWait wait = new WebDriverWait(driver, 5);
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
        //Introduction
        driver.findElementByXPath("//android.widget.Button[@text='I
        AGREE']").click();
        driver.findElementByXPath("//android.widget.Button[@text='ALLOW']")
        .click();
        driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
        ]").click();
        //Login page

        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").clic
        k();
        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").send
        Keys("11111111");

        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        click();
        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        sendKeys("11111");
        driver.findElementByXPath("//android.widget.Button[@text='ACTIVATE'
        ]").click();
        //Wait for loading and click on Menu
        wait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath
        ("android.widget.ImageView")));
        driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
        on").click();
        //Menu->payments->quick payment
        driver.findElementByXPath("//android.widget.CheckedTextView[@text='
        Payments']").click();
        driver.findElementByXPath("//android.widget.CheckedTextView[@text='
        Quick payment']").click();
```

```
//Payment form
//Select account
driver.findElementById("cz.csob.smartbanking.testing:id/imageView3"
).click();
//choose 3rd account
driver.findElementByXPath("//android.widget.TextView[@text='ČSOB
Konto v CZK']").click();
//Account number and bank code
driver.findElementByXPath("//android.widget.EditText[@text='Account
number']").sendKeys("3158911123");
driver.findElementByXPath("//android.widget.EditText[@text='Code']"
).sendKeys("0800");
//Amount
driver.findElementByXPath("//android.widget.EditText[@text='0.00
CZK']").sendKeys("100");
//Scroll before find Display other fields and click
driver.findElement(MobileBy.AndroidUIAutomator("new
UiScrollable(new
UiSelector().scrollable(true).instance(0)).scrollIntoView(new
UiSelector().textMatches(\"Display other fields\").instance(0))"));
driver.findElementByXPath("//android.widget.TextView[@text='Display
other fields']").click();
//Scroll to Message, click and fill
driver.findElement(MobileBy.AndroidUIAutomator("new
UiScrollable(new
UiSelector().scrollable(true).instance(0)).scrollIntoView(new
UiSelector().textMatches(\"Message\").instance(0))"));
driver.findElementById("cz.csob.smartbanking.testing:id/edit_text_m
essage").click();
driver.findElementById("cz.csob.smartbanking.testing:id/edit_text_m
essage").sendKeys("Quick payment");
 driver.hideKeyboard();
//Continue
driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
]").click();
//Scroll to confirm
driver.findElement(MobileBy.AndroidUIAutomator("new
UiScrollable(new
UiSelector().scrollable(true).instance(0)).scrollIntoView(new
UiSelector().textMatches(\"PIN\").instance(0))"));
//PIN and confirm

driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").c
lick();
driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").s
endKeys("11111");
driver.findElementById("cz.csob.smartbanking.testing:id/btn_confirm
").click();
//Confirmation screen check
```

```
            try {
                    driver.findElementById("cz.csob.smartbanking.testing:id/succ
                    ess_contentView");
                    System.out.println("Success payment");
            } catch (Exception e) {
                    System.out.println("Failed payment");
            }
            driver.findElementByXPath("//android.widget.Button[@text='AGREE']")
            .click();
            //Logout and turn off the application
            driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
            on").click();

            driver.findElementById("cz.csob.smartbanking.testing:id/icLogout").
            click();
            driver.closeApp();
            driver.quit();

    }

}
```

3. Třída – Třída pro TC2 – Zablokování platební karty

```
import java.net.MalformedURLException;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.android.AndroidElement;

public class card extends base{

    public static void main(String[] args) throws MalformedURLException {

        AndroidDriver<AndroidElement> driver=Capabilities();
        WebDriverWait wait = new WebDriverWait(driver, 5);
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
        //Introduction
        driver.findElementByXPath("//android.widget.Button[@text='I
        AGREE']").click();
        driver.findElementByXPath("//android.widget.Button[@text='ALLOW']")
        .click();
        driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
        ]").click();
        //Login page
        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").clic
        k();
        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").send
        Keys("11111111");
        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        click();
        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        sendKeys("11111");
        driver.findElementByXPath("//android.widget.Button[@text='ACTIVATE'
        ]").click();
        //Wait for loading and click on Menu
        wait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath
        ("android.widget.ImageView")));
        driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
        on").click();
        //Menu -> Payment cards
        driver.findElementByXPath("//android.widget.CheckedTextView[@text='
        Payment cards']").click();
        //Click on block button
        driver.findElementById("cz.csob.smartbanking.testing:id/blockButton
        TextView").click();
        //Block form
        //Fill reason
```

```
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_reason"
        ).click();
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_reason"
        ).sendKeys("I want to block my card!");
         driver.hideKeyboard();
         //Fill PIN
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").c
        lick();
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").s
        endKeys("11111");
        driver.findElementById("cz.csob.smartbanking.testing:id/btn_confirm
        ").click();
         //Confirm
        driver.findElementByXPath("//android.widget.Button[@text='SEND']").
        click();
         //Confirmation screen check
         try {

                driver.findElementById("cz.csob.smartbanking.testing:id/succ
                ess_contentView");
                System.out.println("Card successfully blocked");
         } catch (Exception e) {
                System.out.println("Card blocking failed");
         }
        driver.findElementByXPath("//android.widget.Button[@text='AGREE']")
        .click();
         //Logout and turn off the application
        driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
        on").click();
        driver.findElementById("cz.csob.smartbanking.testing:id/icLogout").
        click();
         driver.closeApp();
         driver.quit();
    }

}
```

4. Třída – Třída pro TC3 – Dobití kreditu

```
import java.net.MalformedURLException;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.android.AndroidElement;

public class credit extends base{

    public static void main(String[] args) throws MalformedURLException {

        AndroidDriver<AndroidElement> driver=Capabilities();
        WebDriverWait wait = new WebDriverWait(driver, 5);
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
        //Introduction
        driver.findElementByXPath("//android.widget.Button[@text='I
        AGREE']").click();
        driver.findElementByXPath("//android.widget.Button[@text='ALLOW']")
        .click();
        driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
        ]").click();
        //Login page
        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").clic
        k();
        driver.findElementById("cz.csob.smartbanking.testing:id/ipid").send
        Keys("11111111");
        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        click();
        driver.findElementById("cz.csob.smartbanking.testing:id/password").
        sendKeys("11111");
        driver.findElementByXPath("//android.widget.Button[@text='ACTIVATE'
        ]").click();
        //Wait for loading and click on Menu
        wait.until(ExpectedConditions.invisibilityOfElementLocated(By.xpath
        ("android.widget.ImageView")));
        driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
        on").click();
        //Menu -> Payments -> Credit recharging
        driver.findElementByXPath("//android.widget.CheckedTextView[@text='
        Payments']").click();
        driver.findElementByXPath("//android.widget.CheckedTextView[@text='
        Credit recharging']").click();
        //Credit recharging form
        //Account
```

```
driver.findElementById("cz.csob.smartbanking.testing:id/imageView3"
).click();

driver.findElementById("cz.csob.smartbanking.testing:id/text_view_a
ccount_name").click();
//Operator
driver.findElementById("cz.csob.smartbanking.testing:id/triple_dots
").click();
driver.findElementByXPath("//android.widget.CheckedTextView[@text='
Vodafone']").click();
driver.findElementByXPath("//android.widget.Button[@text='OK']").cl
ick();
//Phone number
driver.findElementById("cz.csob.smartbanking.testing:id/edt_phone_n
umber").click();
driver.findElementById("cz.csob.smartbanking.testing:id/edt_phone_n
umber").sendKeys("731000");
//Amount
driver.findElementById("cz.csob.smartbanking.testing:id/edt_amount"
).click();
driver.findElementById("cz.csob.smartbanking.testing:id/edt_amount"
).sendKeys("200");
//Message
driver.findElementById("cz.csob.smartbanking.testing:id/edit_text_m
essage").click();
driver.findElementById("cz.csob.smartbanking.testing:id/edit_text_m
essage").sendKeys("I am recharging 200 CZK");
driver.hideKeyboard();
//Continue
driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
]").click();
//Alert -> Bad phone number format
try {
        driver.findElement(By.xpath("//android.widget.TextView[@text
        ='Fill in a valid phone number.']"));
        System.out.println("Correct warning text");
} catch (Exception e) {
        System.out.println("Incorrect warning text");
}
driver.findElementByXPath("//android.widget.Button[@text='CLOSE']")
.click();
//Valid phone number
driver.findElementById("cz.csob.smartbanking.testing:id/edt_phone_n
umber").click();
driver.findElementById("cz.csob.smartbanking.testing:id/edt_phone_n
umber").clear();
driver.findElementById("cz.csob.smartbanking.testing:id/edt_phone_n
umber").sendKeys("731000123");
driver.hideKeyboard();
```

```
        //Valid Continue
        driver.findElementByXPath("//android.widget.Button[@text='CONTINUE'
        ]").click();
        //Fill PIN and confirm
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").c
        lick();
        driver.findElementById("cz.csob.smartbanking.testing:id/edt_pin").s
        endKeys("11111");
        driver.findElementById("cz.csob.smartbanking.testing:id/btn_confirm
        ").click();
        //Confirmation screen check
        try {
                driver.findElementById("cz.csob.smartbanking.testing:id/succ
                ess_contentView");
                System.out.println("Credit recharge successfully accepted");
        } catch (Exception e) {
                 System.out.println("Credit recharge rejected");
        }
        driver.findElementByXPath("//android.widget.Button[@text='AGREE']")
        .click();
        //Logout and turn off the application
        driver.findElementById("cz.csob.smartbanking.testing:id/drawer_butt
        on").click();
        driver.findElementById("cz.csob.smartbanking.testing:id/icLogout").
        click();
        driver.closeApp();
        driver.quit();
    }

}
```