Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

KNIHOVNA PRO AUTOMATIZACE TESTŮ ZALOŽENÁ NA ŠABLONÁCH

Marek Kozlovský

Vedoucí: Ing. Martin Ledvinka
Obor: Softwarové systémy
Studijní program: Otevřená informatika
Květen 2018

Poděkování

Poděkování patří především vedoucímu práce, Ing. Martinu Ledvinkovi, za podnětné připomínky jak k implementační, tak teoretické části.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použíté informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 24. května 2018

Abstrakt

Předmětem této práce je vytvoření Java frameworku pro vykonávání automatických testů na základě textových scénářů. Účelem je umožnit uživateli tohoto frameworku vytvářet testovací scénáře metodikou Behaviour Driven Development, za použití klíčových slov Given-When-Then. Důležitým aspektem je znovupoužitelnost pro různé platformy, pro které existuje implementace rozhraní Selenium WebDriver. Součástí této práce je také aplikace vzniklého frameworku na platformu Android, vytvoření sady exemplárních šablon pro řízení testů a integrace celého frameworku s některým nástrojem pro správu testů typu TestRail TM nebo Squash TM apod.

Klíčová slova: WebDriver, JBehave, Android, Appium, TestRail

Vedoucí: Ing. Martin Ledvinka

Abstract

The major goal of this thesis is to develop a template-based test automation framework. The primary purpose of the framework is to enable developers to create tests using the Behavior Driven Development model with Given-When-Then scenarios. An important aspect of the framework is also its reusability for a variety of different platforms that have their implementation of a Selenium Webdriver interface. Furthermore, another part of the thesis describes the application of the framework on the Android mobile platform, creation of example test templates and integration of the framework with a test management tools, such as TestRail TM or Squash TM.

Keywords: WebDriver, JBehave, Android, Appium, TestRail

Title translation: Template-based Test Automation Framework

Obsah

1 Úvod 1

2 Analýza stávajících přístupů 3

2.1 Základní metody testování software 4

2.1.1 Kategorie testování podle SWEBOK 5

2.2 Tvorba automatických testů 7

2.2.1 Selenium, WebDriver 7

2.2.2 Appium 7

2.3 Behaviour Driven Development . . 8

2.3.1 Cucumber vs. JBehave 10

3 Návrh řešení na základě analýzy 13

3.1 Požadavky aplikace 13

3.1.1 Funkční požadavky 13

3.1.2 Nefunkční požadavky 14

3.2 Návrh řešení 15

3.2.1 Zúčastněné osoby 18

4 Implementace 19

4.1 Struktura aplikace pro spouštění testů 19

4.1.1 Instrukční vrstva 20

4.1.2 Servisní vrstva 20

4.1.3 Řídící vrstva 23

4.1.4 Konfigurace JBehave 26

4.2 Appium 26

4.2.1 Připojení testovacích zařízení 27

4.2.2 Navázání komunikace se zařízením 28

4.3 Inicializační skript 28

4.3.1 Spouštěcí parametry 29

4.3.2 Výběr zařízení 30

4.3.3 Logování průběhu 30

4.4 Rozhraní TestRail TM 31

4.4.1 TestRail API 33

4.5 Použití Jenkins 33

4.6 UIAutomatorViewer 35

5 Analýza možností vytvoření modulu pro správu šablon a jejich implementací **37**

5.1 Realizace 38

6 Závěr **41**

A Literatura **43**

B Obsah přiloženého CD **47**

C Zadání práce **49**

Obrázky

3.1 Diagram aktivit - spouštění testů. 15

3.2 Schéma architektury. 17

4.1 Ukázka scénáře pro přihlášení. . . 25

4.2 Ukázka šablony pro výběr elementu. 25

4.3 Ukázka šablony pro nastavení textu textovému poli. 25

4.4 Ukázka šablony pro ověření správné aktivity. 25

4.5 Adresářová struktura modulu pro spouštění testů. 30

4.6 Ukázka logu z rozhraní Jenkins. 31

4.7 Ukázka scénáře v rozhraní TestRail TM. 32

4.8 Ukázka testovacího běhu před provedením testů. 32

4.9 Ukázka úspěšně provedeného testovacího běhu. 33

4.10 Ukázka úloh Jenkins při provádění testu. 34

4.11 Ukázka testovací šablony. 35

4.12 Ukázka rozhraní UIAutomatorViewer. 35

Úvod

Testování softwaru je nedílnou součástí vývoje moderních aplikací dnešní doby a často se stává klíčovým faktorem, který určuje kvalitu produktu poskytovaného klientovi. Testování softwaru se stalo samostatnou disciplínou, jeho metodami se zabývá široké množství lidí a pro programátora je jeho praktikování významnou denní rutinou. Řádné testování má primárně za úkol odhalit chyby, které při vývoji nejsou na první pohled patrné. Včasné odhalení chyb je klíčem k efektivní práci a naopak zanedbání může vést k časové ztrátě, což je většinou také peněžní ztráta. Defekty v aplikaci se však mohou vyskytovat na mnoha systémových úrovních a jejich odhalení proto nemusí být vždy triviální. Za účelem snížení rizika vzniku těchto chyb existuje celá řada testovacích metod.

Běžnou praktikou při dlouhodobém vývoji a údržbě produktu, jehož stav odráží aktuální trendy moderních aplikací, je neustálá aktualizace použitých metod a technologií. Jedním z rizik, které však časté změny mohou přinést, je vznik nepředvídaných chyb. Metodou, jejímž cílem je odhalení právě takto vzniklých závad, jsou regresní testy. Předpokladem pro vykonání regresních testů, je definice klíčových scénářů, které pokrývají důležité funkce aplikace. V rámci regresních testů jsou tyto scénáře procházeny, přičemž se kontrolují různé atributy uživatelského rozhraní. Velikým přínosem a časovou úsporou je automatizace těchto testů pomocí nástrojů typu Selenium, které však vyžadují určitou znalost programování.

Na vývoj softwaru a jeho efektivitu má nezanedbatelný vliv také projektové řízení, jehož moderními a hojně praktikovanými metodami, jsou metody agilní. Zástupcem těchto metod je ku příkladu Behaviour Driven Development,

dobře známý především díky vlastnímu konceptu psaní use-casů, tedy scénářů podporovaných v aplikaci. Scénáře tohoto typu jsou srozumitelné nejen pro programátora, ale také pro analytiky, testera a zákazníka. Svojí podporu nachází v řadě programovacích jazyků, včetně Javy.

Cílem této práce je spojení konceptu automatických testů a Behaviour Driven Development, tedy vytvoření frameworku pro řízení testů uživatelského rozhraní, jehož základem je textový scénář. Potenciálem této práce je vytvoření nástroje, který znatelně usnadní psaní regresních testů, sníží nároky na zkušenost člověka s programováním a do jisté míry strhne bariéru rozdílů mezi platformami, na které jsou testované aplikace vyvinuty.

Stavebním kamenem automatických testů je rozhraní WebDriver, na kterém tento framework postaví základy. Nezávislost tohoto nástroje na konkrétní implementaci tedy v první řadě umožní přenositelnost knihovny mezi platformami, ale také smaže některé rozdíly při definici jejich testů. Test jedné funkcionality na platformě Android by tedy v naprosto ideálním případě mohl mít stejný scénář jako na platformě iOS.

Součástí výstupu bude také integrace frameworku s některou platformou, exemplární scénář a také napojení na některý nástroj pro správu testů, typu TestRail a Squash TM.

Kapitola 2

Analýza stávajících přístupů

Oborem vývoje softwaru se v dnešní době zabývá množství lidí jak na půdě akademické, tak v komerčním prostředí. Ruku v ruce s pokrokem tohoto oboru roste také obor testování. Tento fakt se odráží např. ve zvyšujícím se množství testovatelných aspektů u produktů softwarového trhu. Každá technologie má určité slabiny a samotná práce programátora nese riziko vzniku nechtěné chyby. Za účelem odhalení chyb ve fázi vývoje existuje řada metod, které se soustředí na různé vlastnosti a potenciální hrozby vyvíjených systémů. Tato kapitola se věnuje seznámení čtenáře s praktikami testování, jejich kategorizací a dalším dělením. Největší pozornost je věnována kategorii automatických testů, s kterými jsou úzce spjaté nástroje, jako Selenium a Appium.

Behaviour Driven Development (BDD), agilní metodika vývoje, která láme bariéru mezi zákazníkem a jazykem programátora. Principem BDD je definování klíčových scénářů aplikace způsobem, který je čitelný i pro člověka neznalého programování a následná implementace těchto scénářů. Tento koncept podporuje řada Java frameworků, např. JBehave a Cucumber. Další význam a využití, rozdíly mezi frameworky také nastíní tato kapitola.

2.1 Základní metody testování software

Jak jsme již řekli, škála kategorií a rozdělení testů, které se při vývoji softwaru používají, je široká. Literatura se v ohledech kategorizace a klasifikace testů liší, my se však nyní budeme řídit mezinárodně uznávaným standardem SWEBOK [1]. Ten rozlišuje dva základní typy testů, které klasifikujeme podle:

- předmetu testování
  - jednotkové testy (unit-testing)
  - integrační testy
  - systémové testy
- cílového efektu testování
  - akceptační testy
  - testování výkonu
  - regresní testy
  - a další...

Přístupů a pohledů do této problematiky existuje rozsáhlé množství, s rychlostí růstu této disciplíny se tomu ani nemůžeme divit. Hrubé linie, které však standard SWEBOK vymezuje, jsou pro náš účel dostatečné a dájí se na nich demonstrovat cíle této práce. V následujících odstavcích popíšeme zmíněné metody jenom stručně, pro širší studie a uvedení do této oblasti je vhodná literatura [2].

Důležitým pojmem, kterému také bude dobré porozumět je pojem Black-box testing:

- **Black box vs. White-box.** Nahlížíme-li na testovaný produkt, rozlišujeme přístupy podle množství znalostí, se kterými při práci s ním operujeme. Pokud mluvíme o black-box testingu, rolí testera je přistoupit pouze k výstupům aplikace bez ohledu na vnitřní strukturu a na ně se zaměřit. White-box testing metoda je případ, kdy implementace testovaného objektu je naprosto známá. Grey-box testing je kompromisem mezi oběma výše zmíněnými přístupy. Pro podrobný popis viz [3].

2.1.1 Kategorie testování podle SWEBOK

Nejlepší cestou k fungující a udržitelné aplikaci je soustavné vytváření testů spolu s implementací jednotlivých částí systému. Ke každému systémovému celku je však žádoucí aplikovat takový test, který správnost řešení verifikuje smysluplně a efektivně. Jednotlivá dělení podle standardu SWEBOK si stručně popišme.

Dělení podle předmětu testování

- **Unit-testy**, nebo-li jednotkové, se ve většině případech zaměřují na malou část kódu, typicky funkci nebo třídu. Smyslem jejich existence je vybudování stabilního podkladu pro tvorbu větších a složitějších celků. Důležitým pravidlem je však to, aby daný test nebyl závislý na ostatních systémových komponentách. Jednu funkci je ve většině případů žádoucí pokrýt množinou use-casů, které by při vykonávání mohly být problémové, jako např. předložení datové typu metodě, na který není připravená nebo případ dělení nulou. Ve spojitosti s unit-testy zpravidla mluvíme o white-box přístupu.

- **Integrační testy** nahlížejí na množinu systémových komponent a zabývají se vztahem a interakcemi mezi nimi. Pokud např. dvě komponenty samostatně pracují výborně, ale výstup jedné z nich neodpovídá očekávání té druhé, nemůže jejich součinnost vést k žádanému výsledku a dostáváme se do problému. Předpokladem pro tento druh testů jsou funkční, spolehlivé jednotky ověřené unit-testy. Integrační testy jsou nejčastěji příkladem white-box přístupu.

- **Systémové testy** mají za úkol ověřit fungování celého systému a splnění klientských požadavků. Také se dá říci, že systémové testy slouží jako výstupní kontrola a v ideálním případě by předpokladem jejich provedení by měly být integrační testy. V rámci procesu testování se simulují zejména kroky, které mohou nastat v interakci přímo s klientem, což je např. v zákaznické aplikaci kritická oblast. U metody systémového testování se nedá jednoznačně rozlišit mezi white, či black-box modelem, některé jejich kategorie mohou nahlížet na problém různými způsoby.

Dělení podle cílového efektu testování

Tento způsob testování může směřovat k různým aspektům daného produktu. Jsou jimi nejen funkční požadavky, ale také ty nefunkční, jako dostupnost aplikace, spolehlivost při vysoké zátěži apod. Cílem těchto testů jsou zkrátka slabiny systému, které se v praxi ukázaly jako problematické, většinou v okrajových případech. Shrneme aspoň ty nejznámější:

- **Akceptační testy** probíhají na straně zákazníka a jejich cílem je kontrola scénářů, které jsou týmem analytiků definované většinou v počáteční fázi vývoje. V případě nalezení nežádoucího chování produktu jsou popsané chyby odeslány týmu vývojářů, který se postará o opravu a zašle správné řešení zpět. Tento cyklus se ke spokojenosti zákazníka může opakovat. Jedná se o typický příklad black-box modelu.

- **Testy výkonu** prověřují schopnost systému reagovat na požadavky v extrémních podmínkách. Tyto podmínky mohou nastat např. při používání systému velkým množstvím uživatelů ve stejnou chvíli, nebo při obsluze paměťové náročných zdrojů. Schopnost systému čelit takovýmto jevům se říká škálovatelnost (*scalability*) a dá se maximalizovat pomocí metody *load-balancingu*, tedy rozprostření zatéže na množství serverů. Odolnost systému však nespočívá pouze v jeho zátěžové kapacitě, měřitelných vlastností je celá řada a zabývá se jimi široké množství literatury, např.[4].

- **Regresní testy** mají za cíl zajistit, aby změny v aplikaci, jako implementace nových funkcí nebo úprav těch stávajících, negativně neovlivnily chod částí, které mají být na změně nezávislé. Provedení takových testů je typicky spjaté s vydáním nové verze aplikace, nebo-li release. Většina scénářů, na začátku definovaných, pokud se jich změny v aplikaci přímo netýkají, zůstávají stejné. Zejména z tohoto důvodu je velice vhodné regresní testy automatizovat (viz 3.2). I v tomto případě mluvíme o black-box testingu.

Dalších testovatelných aspektů je celá řada. Významnými z nich jsou např. zabezpečení, schopnost obnovy po pádu aplikace, rozšiřitelnost apod. Do tohoto tématu poskytuje podrobnější vhled již zmíněná literatura [5].

2.2 Tvorba automatických testů

Testy aplikace na úrovni uživatelského rozhraní se buď provádějí manuálně nebo automaticky, obě metody mají své výhody i nevýhody. V případě manuálního testování je výhodou lidský faktor. Pokud testy provádí člověk, má zpravidla větší šance na nalezení nežádoucího chování aplikace, než v případě vykonávání definovaných kroků a kontrol statických vlastností. Úskalím ruční kontroly je však časová náročnost. Při vhodných podmínkách časovou úsporu přináší automatické testy, které jsou typicky tím příhodnější, o čím větší projekt se jedná. Špatným příkladem užití automatických testů jsou projekty, u kterých se nepředpokládá dlouhodobý vývoj a časté updaty.

2.2.1 Selenium, WebDriver

Nejznámějším frameworkem pro vytváření automatických testů je Selenium, které své použití cílí zejména na webové aplikace. Své testy provádí simulacemi ve webových prohlížečích, z nichž podporuje Google Chrome, Firefox nebo IE a řadu dalších. Obsluha jednotlivých prohlížečů je založena na společném rozhraní WebDriver, což umožňuje psát testy závislé pouze na jednom interfacu a neohlížet se přitom na konkrétní implementaci. Provedení jednoho testu ve více prohlížečích je tedy snadné. Široké nabídce se Selenium těší i v podpoře programovacích jazyků, mezi kterými najdeme Javu, C#, Javascript, ale i PHP a další.

WebDriver je primárně určený k vykonávání automatických testů na zařízení, na kterém jsou testy spouštěny. Ačkoliv je RemoteWebDriver částečnou implementací WebDriveru, ve spojení s ním mluvíme o automatických testech na vzdálených zařízeních. RemoteWebDriver instancuje virtuální server, který odesílá instrukce na vzdálená zařízení a pouze prezentuje výsledky této komunikace.

2.2.2 Appium

Appium je obdobou Selenia a jeho stavebním kamenem je taktéž rozhraní WebDriver. Jeho využití je však primárně cílené na mobilní telefony, což je typ vzdáleného zařízení. Narážíme tedy na zmíněný RemoteWebDriver. Součástí Appium knihoven jsou implementace AndroidDriver a IOSDriver.

Důležité je zmínit, že díky možnosti využití vzdálených zařízení je snadné testy paralelizovat, což vede ke značné časové úspoře.

Dalším materiálem pro ujasnění vztahu WebDriveru a jeho implementací viz. [6].

2.3 Behaviour Driven Development

Jednou z moderních agilních metodik vývoje softwaru je Behaviour Driven Development (BDD). Původní metodou, ze které se tento přístup vyvinul je TDD, tedy Test Driven Development. Standardní součástí analýzy prováděné před začátkem vývoje produktu je definování use-casů, které musí aplikace podporovat. V tomto směru se obě zmíněné metody shodují, rozdíl však nastává v podobě definovaných use-casů. V případě BDD jsou tyto use-casy strukturovány formou sady klíčových slov a definovaných kroků, známých jako stories, které jsou srozumitelné jak pro stranu zákazníka, tak tým vývojářů. V případě TDD je tento krok nahrazen sepsáním sady testů, kterým rozumí jenom programátor. To však nutně není nevýhodou, cílem BDD jsou systémové testy, a naopak cílem TDD testy systémových komponent (unit testy, integrační testy). Behaviour-driven development tedy snižuje riziko nedorozumění mezi oběma stranami, což je v efektivním vývoji klíčové a metoda Test-driven development přispívá k ujasnění vnitřního modelu aplikace a možných komplikací ještě před začátkem vývoje.

Definované příklady užití, stories, musí dodržovat strojem čitelnou strukturu, tedy syntaxi jazyka, který je pro tento účel určený. Populárním jazykem s jednoduchou sémantikou a intuitivním použitím je jazyk Gherkin[7]. Klíčovými slovy jazyka Gherkin, kromě jiných, jsou:

- řídící struktury
  - **Given** specifikuje počáteční podmínky pro průchod scénářem
  - **When** invokuje akci daného use-casu
  - **Then** verifikuje stav aplikace
- informativní / popisné struktury
  - **Feature** slovně popisuje sadu scénářů
  - **Scenario** stručně popisuje průběh scénáře a jeho žádaný výstup

Příkladem textového scénáře tedy může být:

```
Scenario: 2 squared

Given a variable x with value 2
When I square x
Then x should equal 4

Scenario: 3 squared

Given a variable x with value 3
When I square x
Then x should equal 9
```

A mapující třídou v jazyce Java:

```
public class ExampleSteps extends Steps {
        int x;

        @Given("a variable x with value $value")
        public void givenXValue(@Named("value") int value) {
                x = value;
        }

        @When("I multiply x by $value")
        public void whenImultiplyXBy(@Named("value") int value) {
                x = x * value;
        }

        @Then("x should equal $value")
        public void thenXshouldBe(@Named("value") int value) {
                assertTrue(value == x);
        }
}
```

Zdrojem uvedeného příkladu je [8].

Nástrojů, které podporují behaviorální přístup je celá řada. Prakticky každý ze známých, široce používaných high-level programovacích jazyků má svůj framework, který se k k tomuto účelu dá uplatnit. Jsou to např. nástroje:

- Cucumber - Ruby, Java, Groovy atd.
- Easy B - Groovy
- Concordion - Java
- JBehave - Java
- SpecFlow - .Net
- Behat - PHP

Díky nabídce těchto frameworků tedy volba programovacího jazyka není klíčovým aspektem při návrhu celé aplikace. Pokud se však zaměříme na některý z nich, v našem případě jazyk Java, máme možnost výběru mezi několika různými implementacemi. Nejvyšší popularitě mezi Java frameworky se mohou těšit nástroje JBehave a Cucumber, u kterých následně předvedeme výhody i nevýhody a zvážíme jejich použití pro účely této práce.

2.3.1 Cucumber vs. JBehave

Ačkoliv jsou knihovny Cucumber i JBehave na první pohled velice snadno použitelné, za jejich nevinným zevnějškem se skrývá komplexita velice silných nástrojů. Množství možností, jak do počtu klíčových slov při tvorbě scénářů, tak konfigurace aplikačního rozhraní, je silnou stránkou v obou případech.

Ke specifickým rozdílům, výhodám a úskalím těchto nástrojů jsem bohužel nebyl schopný dohledat respektu hodnou literaturu. Toto téma je však předmětem řady veřejně dostupných diskuzí. Na základě sběru subjektivních názorů v nezávislých blozích a debatách jsem dospěl k následující kategorizaci:

1. **Dokumentace.** Důležitým aspektem, speciálně pro uživatele s nízkou znalosti BDD je dokumentace, množství příkladů a webová komunita. V tomto ohledu je podle veřejného mínění JBehave na vyšší úrovni, než Cucumber. Kritické názory poukazují na nedostatečné uvedení nového uživatele do hlavních nástrojů a možností frameworku Cucumber.

2. **Flexibilita parametrů.** Tedy přizpůsobivost scénářových šablon k definici komplexnějších use-casů s potřebou využití např. regulárních výrazů, tabulek atp. Úroveň obou nástrojů je vysoká, malou nevýhodou JBehave je však absence podpory víceřádkových vstupů.

3. **Podpora kompozitních kroků.** Tato funkcionalita umožňuje v rámci anotace javovské třídy využít již definované kroky jako část šablony. JBehave tuto funkci pokrývá jednoduchým způsobem, při použití Cucumber je třeba implementovat zvláštní řešení. Tato funkce však není tak klíčová, jako ostatní.

4. **Podoba výstupních dat.** Výstupní data a podrobně zpracovaný průběh testů je důležitou funkcionalitou. Oba frameworky poskytují rozsáhlý log všech proběhlých testů a jejich statistiky prezentují pomocí dokumentů TXT, XML, HTML apod. Převládajícím názorem je však vyšší kvalita výstupních dat Cucumber.

5. **GivenScenario** je featurou výhradně JBehave a díky němu je možné již definované scénáře použít jako jednu z počátečních podmínek při tvorbě use-casů.

Kapitola 3

Návrh řešení na základě analýzy

Analýza provedená v minulé kapitole nám poskytla široký vhled do dané problematiky a nastínila možná řešení. Nyní sestavíme sadu funkčních a nefunkčních požadavků, podle kterých se při implementaci budeme orientovat. Zejména důležitý pro nás následně bude návrh celé aplikace, jeho architektura a osoby zúčastněné ve výsledném modelu.

3.1 Požadavky aplikace

Požadavky aplikace byly sestaveny na základě analýzy provedené ve spolupráci s odborníkem z praxe tak, aby výsledný produkt byl nejenom akademickou demonstrací několika frameworků, ale aby opravdu našel využití v praxi.

3.1.1 Funkční požadavky

Požadavky na konkrétní funkcionalitu, čili základní mantinely aplikace jsou:

1. Pomocí aplikace bude možné automaticky vykonávat regresní či akceptační testy na úrovni uživatelského rozhraní.

2. Aplikace bude podporovat řízení těchto testů na základě slovních scénářů v podobě jazyka Gherkin a bude odpovídat vzoru Behaviour Driven Development.
3. Výstupní aplikace bude podporovat základní automatické operace, jako kliknutí na prvek, nastavení textu, ověření správnosti textu a další.
4. Provádění testů bude možné paralelizovat.
5. Výsledkem každého testu bude detailní log, který obsahuje informace jak o interpretaci jazykových scénářů, tak činnosti nástroje pro automatizaci. Dále poskytne stručnou statistiku proběhlých testů, jejich úspěšnost, či neúspěšnost.
6. Výsledný framework bude integrován s některým nástrojem pro správu testů. Tedy testy, jejich podoba a spouštění bude možné provádět vzdáleně, přímo z nástroje pro správu testů.
7.

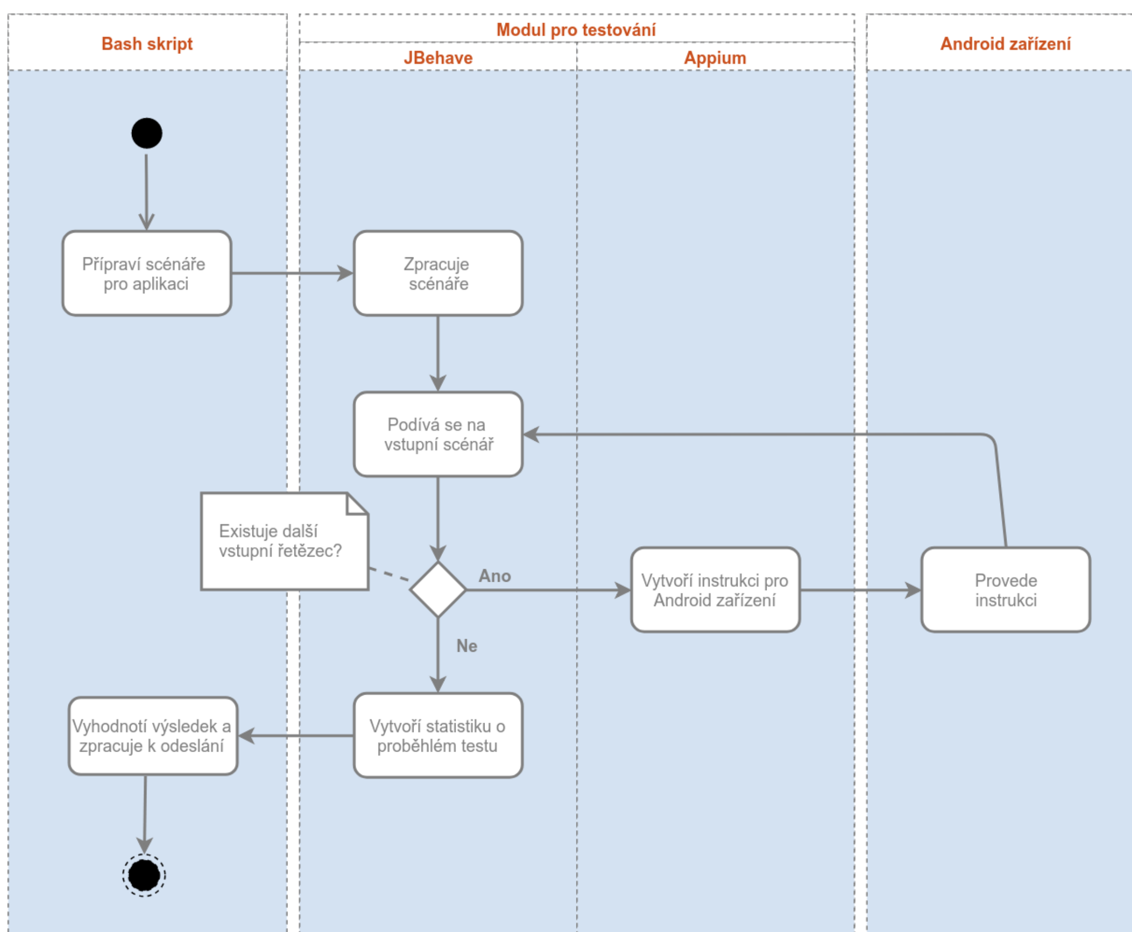
3.1.2 Nefunkční požadavky

1. Součástí výstupní práce bude doporučení vhodného nástroje pro sestavování testů danou knihovnou.

3.2 Návrh řešení

Požadavky aplikace, především ty funkční, nám pomohly ujasnit si priority daného produktu a na jejich základě můžeme postavit návrh celé aplikace. Nyní bodově popíšeme možné řešení a obecný případ užití.

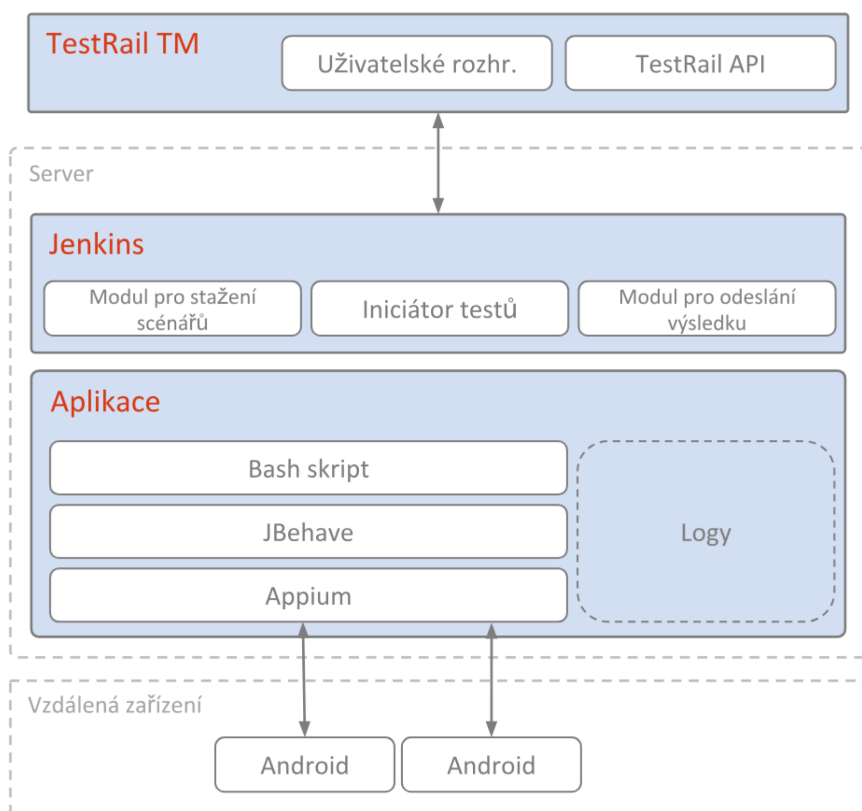
1. Nástrojem pro vytváření testovacích scénářů, jejich mapování na Java třídy a řízení jejich vykonávání bude knihovna JBehave.
2. Automatické testy se budou provádět na mobilních zařízeních systému Android pomocí knihovny Appium.
3. Výše zmíněné kroky bude zaštiťovat Java aplikace jako nezávislý modul.



Obrázek 3.1: Diagram aktivit - spouštění testů.

4. Paralelní vykonávání testů bude realizováno vícenásobným spouštěním dané aplikace pomocí Bash skriptu.
5. Aplikace bude parametrizovaná tak, aby její spuštění cílilo na konkretní zařízení identifikované:
   (a) IP adresou
   (b) jménem zařízení
6. Pro každou sadu testů bude vytvořen adresář, jehož součástí bude seznam všech dostupných zařízení. Dále budou součástí tohoto adresáře podsložky identifikované jménem zařízení, které budou obsahovat:
   - standardní logovací strukturu vytvořenou nástrojem JBehave
   - konzolový výstup Maven úlohy pro spuštění testu
   - činnost knihovny Appium
7. Nástrojem pro správu testů bude TestRail TM
8. Řešení bude využívat koncept Continuous Integration [12] pomocí nástroje Jenkins [13], jehož činností bude:
   a. stažení testovacích scénářů z rozhraní TestRail TM pomocí TestRail API
   b. spuštění testů pomocí testovacího modulu
   c. reporting proběhlých testů zpět do TestRail TM pomocí TestRail API
9. Inicializace automatických testů bude prováděna koncovým uživatelem z rozhraní TestRail pomocí doimplemenovaného tlačítka (TestRail TM umožňuje mírnou úpravu už. rozhraní vlastním způsobem). Takové tlačítko bude odesílat AJAX požadavek do Jenkins a tím spustí jeho činnost.

Schéma 3.2 nám pomůže ucelit si představu o fungování aplikace, její architekturu a moduly, se kterými navržené řešení pracuje.



Obrázek 3.2: Schéma architektury.

3.2.1 Zúčastněné osoby

Tak jako ve většině systémů, i tato aplikace vyžaduje údržbu, updaty a má svého koncového uživatele. Osoby přistupující k této aplikaci dělíme do následujících kategorií:

- **Programátor / Admin.** Úlohou programátora je správa implementovaných šablon, jejich doplňování a úprava v případě potřeby pomocí Gitu. Jeho důležitým úkolem je také péče o aplikaci a připojování testovacích zařízení. Admin musí mít přístup na server, na kterém je aplikace nasazená.
- **Tester.** Pracovní náplní této role je pokrývání use-casů aplikace testovacími scénáři v jazyce Gherkin. Tyto scénáře jsou zadávány do rozhraní TestRail, k čemuž musí mít tester vytvořený účet. Pro efektivní tvorbu testů je také na místě mít samotnou aplikaci na vlastním PC.
- **QA.** Kdokoliv s přístupem do rozhraní TestRail a daného projektu má možnost spustit automatické testy. To je typicky člen QA [2], nejčastěji po výzvě developerem.

Kapitola 4

Implementace

Cílem této kapitoly je seznámit čtenáře s implementačním postupem. Koncept a hrubou architekturu řešení jsme popsali v předchozí kapitole. Nyní je na místě proměnit návrh v implementaci a dílčí části hotového řešení s mírou kritiky rozebrat a popsat.

4.1 Struktura aplikace pro spouštění testů

Aplikace je psaná v programovacím jazyku Java a jejím buildovacím nástrojem je Maven. Výstupní funkcionalitou této aplikace je provádění automatických testů a jejich reporting na jednom konkrétním zařízení. Paralelizace těchto testů na více zařízeních je realizována pomocí Bash skriptu, a proto je důležité, aby se aplikace chovala jako nezávislý modul.

Tak jako je v Javě zvykem, třídy jsou rozděleny do balíčků a jejich struktura se řídí principem vícevrstvé architektury, která mimo jiné koresponduje s pravidly nízké provázanosti [9] a vysoké soudržnosti[9]. Vrstvy aplikace rozdělíme následovně:

- instrukční vrstva
- servisní vrstva
- řídící vrstva

4.1.1 Instrukční vrstva

Tato vrstva se stará o automatické vykonávání instrukcí na vzdáleném zařízení a činí tak pomocí frameworku Appium. Předávání instrukcí je založeno na bázi obousměrné komunikace virtuálního serveru Appium a mobilního zařízení. Komunikačním kanálem a protokolem takového spojení je Android Debug Bridge (ADB)[10]. Na celý proces a integraci Appium se podíváme v kapitole 4.2.

4.1.2 Servisní vrstva

Servisní vrstva je mostem mezi třídami pro mapování textových šablon a knihovnou Appium, tedy mezi řídící a instrukční vrstvou. Její důležitou úlohou je zapouzdření základních instrukcí Appia do větších, lépe použitelných celků, které řídící vrstva volá. Tento princip vede k výraznému zjednodušení používání celé knihovny a podporuje tak nezávislost systémových komponent. Viz [11].

K dalšímu dělení dochází i uvnitř samotné vrstvy, a to pomocí Java balíčků. Významem takové agregace je primárně seskupení funkcí, závislých na stejném rozhraní, a tedy jejich znovupoužitelnost.

- balíček *common*, společný pro různé platformy mobilních zařízení
- balíček *android* specifický pro danou platformu
- balíček *fortuna*, specifický pro danou aplikaci

Důležitý je také systém ukládání grafických prvků aplikace, na kterých voláme akce servisní vrstvy. Viz níže.

Balíček common

Tento balíček je specifický tím, že jeho funkce pracují pouze s rozhraním AppiumDriver. To je důležité zejména proto, že speciálně tato část je znovupoužitelná pro jakoukoliv mobilní aplikaci a obě platformy iOS i Android.

Framework je vyvinut tak, aby s mírnou úpravou dokázal řídit dokonce i desktopové aplikace. Pokud bychom se tedy rozhodli změnit cílová zařízení z androidu na iOS, stačí nám pouze změnit konfiguraci a daný test se provede např. na zařízení iPhone. Výčet a stručných popis implementovaných tříd a jejich metod:

- **ElementActionService** pracuje s prvky aplikačního rozhraní.
  - *clear* - vymaže text prvku
  - *setText* - nastaví text prvku na daný řetězec
  - *getAttributeByName* - vrátí hodnotu atributu prvku
- **ElementSeachService** hledá prvky dostupné v rozhraní spuštěné aplikace pomocí vyhledatelných atributů.
  - *searchForSingleElement* - v aktuálním view aplikace vyhledá element
  - *searchForMultipleElements* - v aktuálním view vyhledá množinu elementů
  - Vyhledatelné atributy:
    - ID elementu
    - řetězec xpath
    - Accessibility ID
    - obsažený text
- **TouchActionService** obstarává operace spojené s dotykem na obrazovce.
  - *actionUponSingleElement* - vyvolá některý typ doteku
    - *TOUCH, PRESS* - stisknutí elementu
    - *RELEASE* - uvolnění stisku
    - *CLICK* - kliknutí na element
    - *SUBMIT* - potvrzení
    - *LONGPRESS* - dlouhý stisk
    - *DOUBLETAP* - dvojité kliknutí na element
  - *dragAndDrop* - podrží prvek a přesune ho do jiné pozice
- **WaitService** a jeho metoda *wait* iniciuje čekání na dostupnost elementů ve view.

Balíček Android

Čím obecnější interface pro provádění automatických akcí používáme, tím širšího uplatnění a znovupoužitelnosti nabývá. Takové pravidlo platí obecně a respektují ho standardní praktiky objektového programování. Je však nutné podotknout, že tento princip s sebou nese i jisté nevýhody. Přistupujeme-li totiž k obecnému rozhraní, připravujeme se tím o funkce vázající se pouze k dané implementaci. Takovým příkladem je i rozhraní AndroidDriver. Implementované funkce v této práci spojené pouze s platformou Android jsou:

- **ActivityService** - třída určená pro zjišťování aktuální android aktivity [14], iniciaci aktivit, či změnu Deeplink URL [15].
  - *getCurrentActivity* - vrátí objekt `Activity`, korespondující s aktuálním stavem aplikace
  - *startActivity* - iniciuje start aktivity
  - *goToUrl* - pomocí systémového volání příkazu ADB změní URL aplikace. Viz DeeplinkDispatch [16].
- **DeviceLockService** - uzamyká, či odemyká zařízení
  - *lock* - zamkne obrazovku
  - *unlock* - odemkne obrazovku

Balíček Fortuna

Píšeme-li automatické testy, je pro nás nežádoucí jakékoliv nepředvídatelné chování, ke kterému při průchodu definovanými use-casy může dojít. V případě mobilních aplikací jsou typickým příkladem vyskakovací okna, aktualizační screeny apod. U aplikace fortuna, na které náš framework testujeme, je nepředvídatelným chováním úvodní obrazovka, jejíž cílem je uživatele seznámit s nejnovějšími možnostmi a funkcemi aplikace. V balíčku *fortuna*, jsme pro ni proto vytvořili následující třídu:

- **WelcomePageGuideResolver.** Jeho metoda *resolve* v případě nutnosti projde úvodní obrazovku pomocí tlačítek "Další".

Ačkoliv

Ukládání prvků

Pro efektivní práci s prvky uživatelského rozhraní je implementován systém ukládání objektů typu `MobileElement`. Třída AndroidAbstractService nabízí úložiště prvků v podobě:

- `Map<MobileElement>` - pro jednotlivé prvky
- `Map<List<MobileElement> >` - pro celé množiny prvků

K daným prvkům se přistupuje pomocí jména, které je jim přiřazeno v rámci textového scénáře, což nám umožní se na pojmenovaný prvek odkazovat později a iniciovat na něm akce. K této funkcionalitě je však třeba přistupovat s vědomím toho, že prvek označený na jedné obrazovce aplikace již není přístupný na obrazovce druhé. Ostražitost v tomto směru předchází nechtěnému chování.

4.1.3 Řídící vrstva

Jak jsme již řekli, pořadí příkazů, vykonávaných automatickými testy, je dáno textovými scénáři v jazyce Gherkin a třídami, které tyto scénáře mapují. K realiziaci této funkcionality používáme knihovnu JBehave a množinu prvků, které v tomto procesu figurují, nazýváme řídící vrstva.

Třídy mapující scénáře, samotné metody a jejich šablony dělíme do balíčků následovně:

Given

- **Finders** - šablony pro vyhledání prvků v aplikačním view a jejich uložení
  - *givenElement* - pokud najde prvek identifikovaný zvolenou vyhledávací metodou, uloží ho

  - *givenElements* - množina prvků, která odpovídá hledanému výrazu se uloží
- **Navigators** - šablony pro přesun v aplikaci
  - *givenLaunchedApp* - zapne testovanou aplikaci
  - *startedActivity* - přesune se do zvolené aktivity
  - *goToUrl* - použije DeepLinkDispatcher pro přesun na URL

When

- **FormHelpers** - šablony pro práci s formulářovými prvky
  - *whenClearNamedElement* - vymaže text pojmenovaného elementu
  - *whenSetTextToNamedElement* - nastaví text pojmenovaného elementu
  - *whenWeWaitForMs* - pozastaví vykonávání testu na dobu v milisekundách
- **Gestures** - šablony spojené s akcemi pro dotyk na obrazovce
  - *whenActionOnNamedElement* - zavolá akci na pojmenovaném elementu
  - *whenActionTwoNamedElements* - zavolá akci na dvou pojmenovaných prvcích
  - *whenClickNamedElement* - klikne na pojmenovaný element
  - *whenSwipeCoordinates* - provede pohyb *swipe*

Then

- **FinderAsserts** - šablony pro ověření přítomnosti prvků ve view apod.
  - *verifyElementCanBeSearchedAndCache* - provede assert na dostupnost prvku v aplikačním view a uloží ho
  - *verifyElementCanBeSearched* - provede assert na dostupnost prvku v aplikačním view
  - *verifyCachedElementReached* - provede assert na dostupnost pojmenovaného elementu v aplikačním view

```
Story: User succesfully logs in.

Given freshly launched app
Given current URL: ftncz://login
When we wait for 3000
Then verify the activity is cz.etnetera.fortuna.activities.LoginActivity

Given element (ID, cz.etnetera.fortuna.cz:id/input_username) as loginTextArea
Given element (ID, cz.etnetera.fortuna.cz:id/input_password) as passwordTextArea
Given element (ID, cz.etnetera.fortuna.cz:id/login_cofirm_btn) as loginButton
Given element (ID, cz.etnetera.fortuna.cz:id/login_visible_pass_btn) as visibilityButton
When we click at visibilityButton
When we clear loginTextArea
When we set text to loginTextArea : testETN5
When we set text to passwordTextArea : *****
When we click at loginButton
When we wait for 5000
Then verify the activity is cz.etnetera.fortuna.activities.HomePageActivity
```

Obrázek 4.1: Ukázka scénáře pro přihlášení.

```
@Given("element ($searchOption, $keyword) as $name")
public void givenElement(
        @Named("searchOption") String searchOption,
        @Named("keyword") String keyword,
        @Named("name") String name) throws Exception {

    MobileElement element;
    element = service.getElementSearchService().searchForSingleElement(searchOption, keyword);
    service.putElement(name, element);
}
```

Obrázek 4.2: Ukázka šablony pro výběr elementu.

```
@When("we set text to $namedElement : $text")
public void whenSetTextToNamedElement(
        @Named("namedElement") String namedElement,
        @Named("text") String text)
                throws ElementNotFoundException, InterruptedException {

    MobileElement element = service.getElement(namedElement);
    service.getMobileElementActionService().setText(element, text);

}
```

Obrázek 4.3: Ukázka šablony pro nastavení textu textovému poli.

```
@Then("verify the activity is $activity")
public void verifyActivity(@Named("activity") String activity) {
    String currentActivity = activityService.getCurrentActivity();
    assertEquals(activity, currentActivity);
}
```

Obrázek 4.4: Ukázka šablony pro ověření správné aktivity.

- **NavigatorAsserts** - šablony pro ověření aktuální polohy v aplikaci
  - *verifyThatHasText* - verifikuje, zda text pojmenovaného elementu je stejný jako zadaný řetězec
  - *verifyThatContains* - verifikuje, zda text pojmenovaného elementu obsahuje zadaný řetězec
  - *verifyUrl* - verifikuje aktuální Deeplink URL

4.1.4 Konfigurace JBehave

S konfigurací JBehave je paradoxně spojená třída AppiumStories (odpovídá doporučené jmenné konvenci), která je také inicializační třídou celého testu. Knihovna JBehave nabízí širokou řadu nastavitelných aspektů, které jsou popsány zde [17]. Konkrétní nastavení v rámci této aplikace je:

- **Výstupní cíle logu:** konzole, adresářová struktura
- **Výstupní formát:** TXT, HTML_TEMPLATE, XML_TEMPLATE
- **Logování na úrovni:** jednotlivých stories, celého testu
- **Výstupní adresář:** závisí na systémovém parametru, typicky složka pojmenovaná podle zařízení, na kterém je daný běh prováděn
- a další...

Součástí konfigurace je také množina souborů, které obsahují dané řídící šablony. Kořenovou složkou pro tyto soubory je *src/test/resources* a jako vstup jsou brány všechny soubory typu *story*.

4.2 Appium

Esenciální složkou instrukční vrstvy naší aplikace je knihovna pro vykonávání automatických akcí. Zvoleným frameworkem pro tento účel je Appium, jehož výhoda spočívá v široké podpoře mobilních platform. Použitá rozhraní a praktiky s nimi spojené, jsou popsaný v servisní vrstvě 4.1.2. Celý framework

Appium nabízí širkový výběr funkcní a konfigurovatelných nastavení, od platformy Android, až po iOS a webové aplikace. Přehled těchto možností je dostupný v oficiální dokumentaci [18]. Tématem této sekce je uvedení testovacích zařízení do stavu, ve kterém můžeme provádět potřebné scénáře.

4.2.1 Připojení testovacích zařízení

Předpokladem pro vykonávání automatických testů na zařízení s platformou Android je následující:

1. na zařízení je aktivovaný vývojářský mód [10] a debugovací režim
2. zařízení je připojeno do stroje, kde je hostována testovací aplikace
   - připojením USB
   - bezdrátově, přes wifi
3. stroj, vykonávající test, je autorizovaný na daném zařízení

Výše zmíněné podmínky nám otevírají cestu ke komunikaci se zařízením pomocí nástroje Android Debug Bridge (ADB) [10], který je součástí standardní sady Android SDK. Je však nutné podotknout, že připojení zařízení bezdrátově je někdy snadné a bezproblémové, jindy však působí potíže a bohužel, chová se nepředvídatelně.

Přehled připojených zařízení je možné vyvolat příkazem v terminálu *adb devices* a vypadá např. takto:

```
$ adb devices

List of devices attached:
ee5f4e15                device
65cb9ec7                device
172.17.1.147:5555                device
172.17.1.156:5555                device
```

Položky v tomto seznamu identifikují zařízení, v podobě UDID připojená přes USB, v podobě IP adresy přeš wifi. Pomocí těchto identifikátorů cílí nástroj ADB instrukce na konkrétní zařízení, čehož ochotně využívá Appium a svým uživatelům v podobě parametru typu *DesiredCapabilities* nabízí tu samou možnost. Naší aplikaci je tento identifikátor předán spouštěcím skriptem, viz 4.3.

4.2.2 Navázání komunikace se zařízením

Jakmile je testovací zařízení připraveno, můžeme začít odesílat instrukce. I odesílání instrukcí se však musí řídit protokolem, jehož součástí je např. navázání obousměrné komunikace a vytvoření session. Standardní praktikou je použití Appium serveru, který se o dodržení protokolu stará sám. Ačkoliv Appium server podporuje souběžnou komunikaci s více zařízeními zároveň, osvědčeným způsobem, který používá i tato práce, je vytvoření vlastní instance pro každé zařízení. Tato instanciace probíhá na úrovni naší Java aplikace.

Součástí první fáze odesílání instrukcí, je vždy kontrola stavu, ve kterém se zařízení nachází. Pokud např. v daném zařízení není na začátku nainstalovaná testovná aplikace, nainstaluje se. Již zmíněné *DesiredCapabilities* [18] některé tyto kroky a celkový průběh ovlivňují, v našem případě to jsou:

- *autoGrantPermission : true* - aplikaci automaticky přiřadí práva
- *automationName : UIAutomator2* - engine pro automatizaci, mírně ovlivňuje vykonávání instrukcí
- *noReset : false* - aplikace se před každým spuštěním resetuje
- a další...

O dalších významných položkách, které jako `DesiredCapabilities` používáme, je zmínka v kapitole 4.3.1.

4.3 Inicializační skript

Aplikaci pro spouštění testu na vzdáleném zařízení jsme si popsali výše. Cílem této práce je však automatické testování provádět na množství zařízeních

souběžně, tedy paralelizovat. Nahlížíme-li na výše zmíněnou aplikaci jako na nezávislý modul, tak její vícenásobné spuštění pomocí Maven cíle *test* není žádný problém. Pro tento účel existuje v kořenové složce projektu Bash skript **executor.sh**.

4.3.1 Spouštěcí parametry

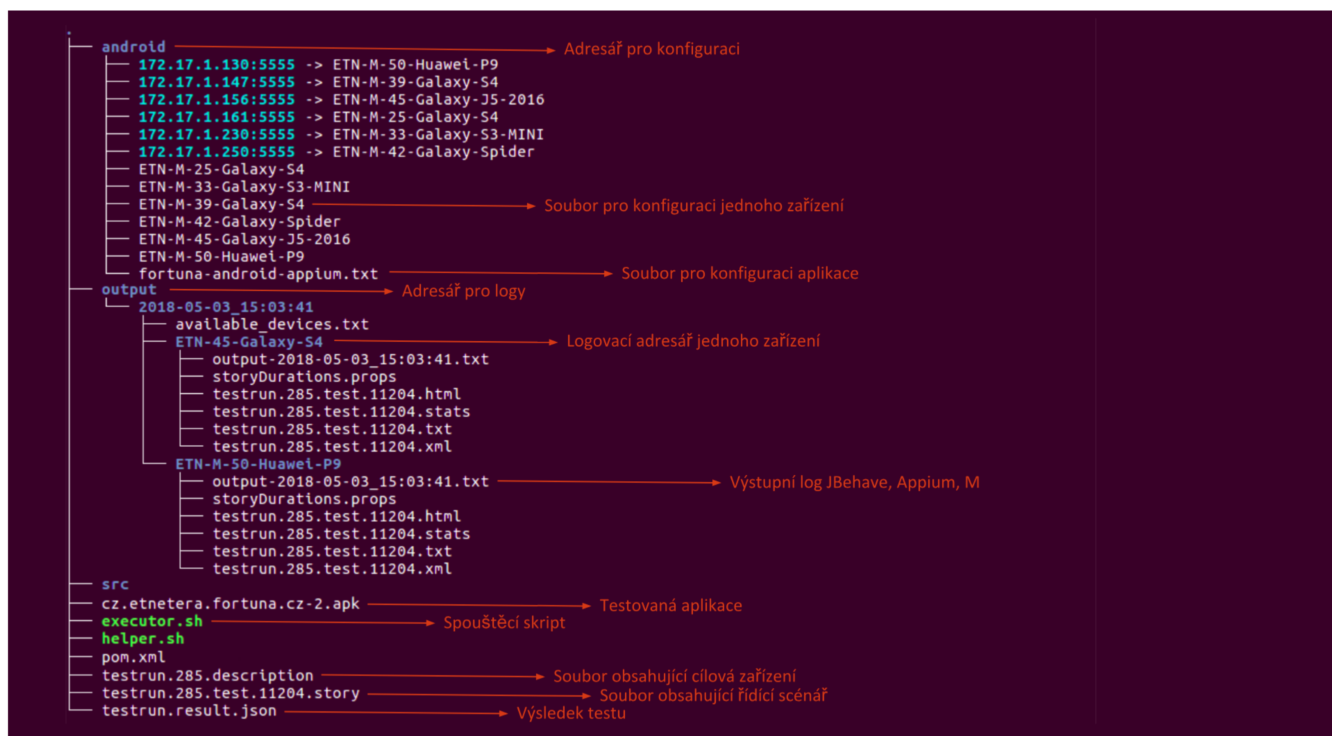
Spuštění se určitě neobejde bez řádné parametrizace, podle které se test orientuje. Existují dva typy konfiguračních souborů, které průběh testu mohou ovlivnit a oba se nacházejí ve složce android. Prvním příkladem je soubor, jehož obsah se týká celé aplikace, na které test běží. Jsou to parametry:

- **app** - jméno aplikace k testování
- **appPackage** - android package, jednoznačně identifikující aplikaci
- **appActivity** - jméno aktivity ke spuštění
- **platformName** - jméno cílové platformy

Druhý typ konfigurace se vztahuje k jednotlivým zařízením, na kterých se testy spouští. Parametry:

- **deviceName** - jméno zařízení
- **udid** - unikátní identifikátor zařízení
- **ipAddr** - IP adresa zařízení
- **port** - unikátní port pro spuštění Appiového serveru

Většina položek z obou konfiguračních souborů je předávána frameworku Appium jako parametry typu `DesiredCapabilities` [18]. Adresář s konfiguračními soubory tedy ve finále může vypadat jako složka *android* na obrázku 4.5. Pro každé zařízení také existuje symbolický link v podobě IP adresy, což usnadňuje jeho výběr uvnitř skriptu.



Obrázek 4.5: Adresářová struktura modulu pro spouštění testů.

4.3.2 Výběr zařízení

Součástí skriptu je také mechanismus na výběr zařízení podle jmen, na kterých by měl test běžet. Předpokladem toho je existence souboru typu *description*, ve kterých je buď seznam názvů, a nebo řetězec ALL_AVAILABLE_DEVICES, čímž se test provede na všech dostupných zařízeních. V obou případech se pracuje s konfiguračními soubory v adresáři *android*, kde musí existovat soubor se jménem, či IP adresou daného telefonu. Pokud daný soubor neexistuje, test nemůže proběhnout.

4.3.3 Logování průběhu

Jedním z důležitých požadavků na celé řešení je poskytnutí detailního logu průběhu testů. Za tímto účelem ve složce *output* je vytvořen adresář identifikovaný časem, ve který se skript zapne, jež dále obsahuje podadresáře se jmény všech zařízení, na kterých se test provádí. Uvnitř této struktury je ve výsledku záznam celého Maven tasku, včetně debugovacího výstupu Appium a JBehave. Průběh chování spouštěcího skriptu je součástí historie Jenkins a může vypadat např. jako na obrázku 4.6.

```
[test-testrail-executor] $ /bin/bash -l /tmp/jenkins72364524524525.sh
TestRunID = 281
TestID = 11202
===== runnig adb devices:
List of devices attached:
ee5f4e15                device
65cb9ec7                device
172.17.1.147:5555               device
172.17.1.156:5555               device

Running the test in background Device 172.17.1.147:5555, ADB port: 8212.
Running the test in background Device 172.17.1.156:5555, ADB port: 8211
=======
Number of available wifi devices: 2
Number of used devices 2

Waiting for all subproces...
Device: 172.17.1.156:5555 : ETN-M-39-Galaxy-J5-2016: success
Device: 172.17.1.147:5555 : ETN-M-39-Galaxy-S4: success
Done.
Generating output to testrun.result.json ...
Cleaning up...
Done.
Finished: SUCCESS
```

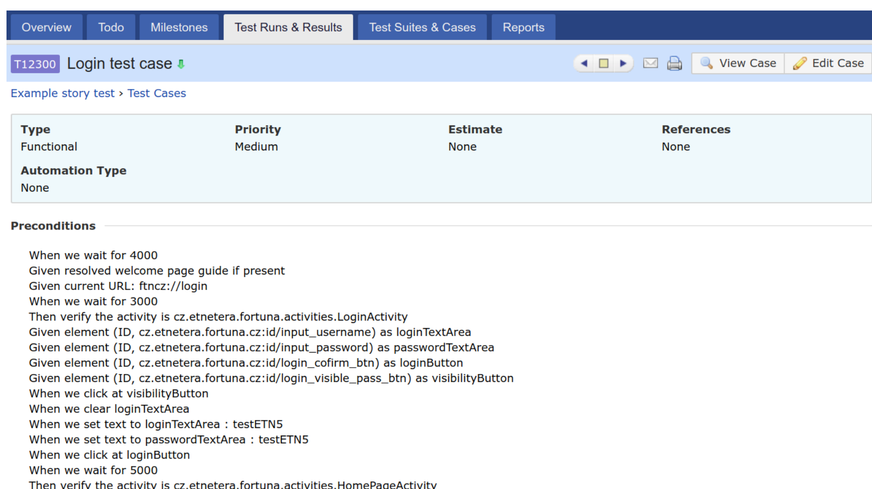
Obrázek 4.6: Ukázka logu z rozhraní Jenkins.

4.4 Rozhraní TestRail TM

Analýza, provedená na začátku této práce, ukazuje široké spektrum testovatelných aspektů softwarových aplikací. Úlohou člena testovacího týmu je s takovými aspekty pracovat a jejich stav zkoumat a zaznamenávat. Efektivní metodou pro vykonávání této činnosti je použití některého nástroje pro správu testů, jež v tomto případě představuje TestRail TM [19]. Organizační strukturou nástroje TestRail je rozdělení na:

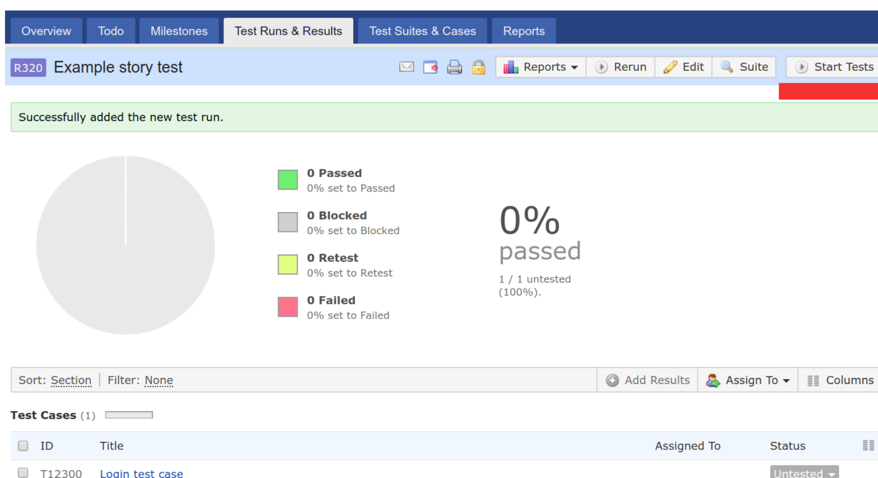
- projekty
  - testovací sady (Test Suite)
    - testované případy (Test Case)

Každý testovací případ obsahuje sadu kroků a očekávaného chování, podle kterého se vyhodnocuje správnost fungování aplikace. Tímto krokem je pro nás testovací scénář v jazyce Gherkin, viz obrázek 4.7.



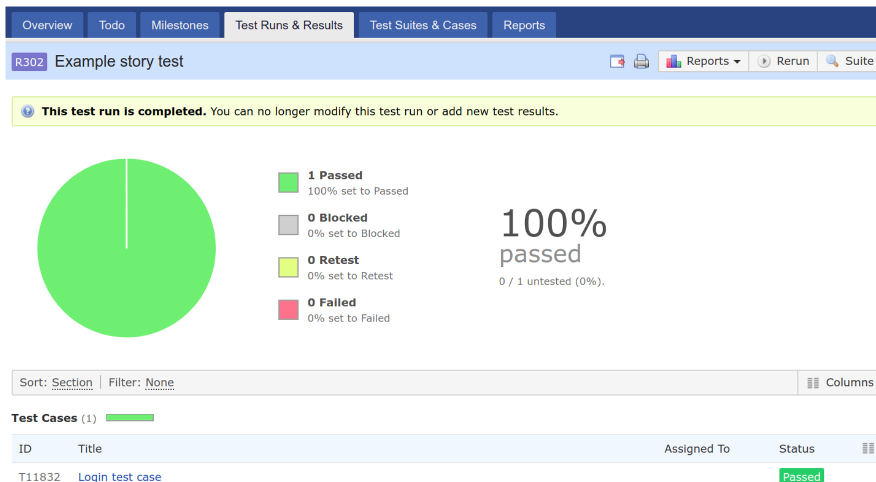
Obrázek 4.7: Ukázka scénáře v rozhraní TestRail TM.

Je však nutné říci, že TestRail a také ostatní nástroje tohoto typu, jsou uzpůsobeny primárně k zaznamenávání výsledků manuálních testů, tedy ke spouštění aplikace třetí strany není uzpůsoben. Na druhou stranu, jednou z funkcí rozhraní TestRail, je také možnost upravit uživatelské rozhraní podle vlastní potřeby, a to přidáním JavaScriptu na vybrané stránky. V našem případě je to přidání tlačítka *Start Tests* do panelu nově vytvořeného testovacího běhu, které při stisknutí odešle Ajaxový požadavek a tím spustí celý cyklus automatického testování. Cílem tohoto požadavku je Jenkins API [21] stroje, na kterém je hostován náš modul pro provádění testů a jeho součástí je pouze unikátní identifikátor prováděného testovacího běhu. Zmíněné tlačítko je červeně vyznačené na obrázku 4.8, který zároveň poskytuje náhled testovacího běhu před jeho provedením.



Obrázek 4.8: Ukázka testovacího běhu před provedením testů.

Další obrázek 4.9 představuje testovací běh po jeho úspěšném provedení.



Obrázek 4.9: Ukázka úspěšně provedeného testovacího běhu.

4.4.1 TestRail API

Aplikace TestRail nabízí i aplikační rozhraní v podobě TestRail API, pomocí kterého je také možné spravovat jednotlivé testovací sady a jejich běhy. Významné je to pro nás tím, že před provedením testu je možné stáhnout testovací scénář přímo do aplikace a po jeho provedení propagovat konečný výsledek zpět.

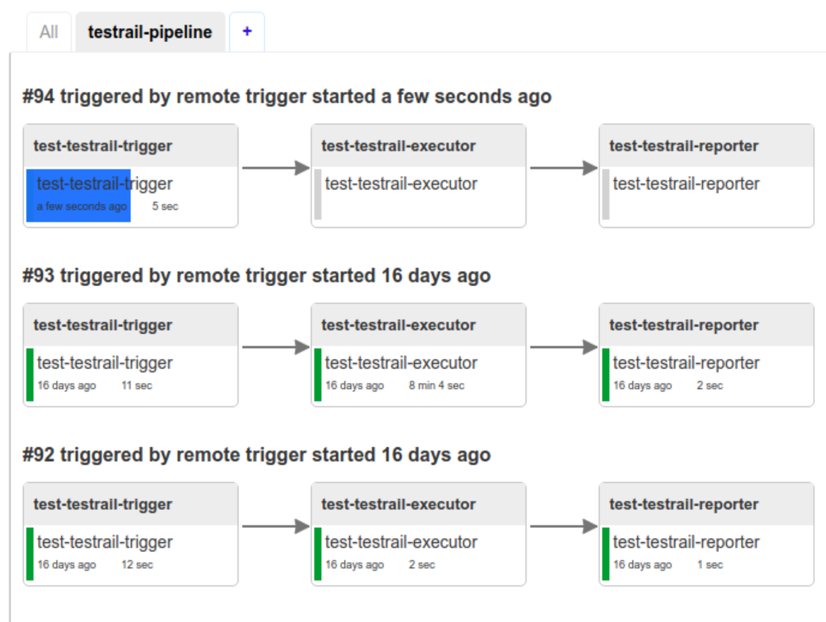
4.5 Použití Jenkins

Koncept Continuous Integration hraje důležitou roli ve správě projektů a jeho přidanou hodnotou je typicky shromáždění klíčovým úloh spojených s projektem, jako např. build aplikace, nasazení, spuštění testů apod. Nástrojem podporujícím tento koncept, který je využitý v případě naší práce je Jenkins.

Ačkoliv to není úplně běžnou praktikou, v celkovém řešení našeho problému, hraje Jenkins vedle spouštěče testů, roli spíše mediátoru informací. Úlohou Jenkins je totiž v první řadě nashromáždění informací o cílovém testu, pak teprve jeho provedení a následná prezentace výsledků. O průběh těchto tří

fázi se starají tři Jenkins úlohy, které jsou spojeny v řídící proud a vykonávají se v určeném pořadí těsně po sobě. Jsou to úlohy:

1. **Trigger.** Spouští se typicky pomocí Jenkins API, požadavkem provedeným z uživatelského rozhraní TestRail. Parametrem jeho spuštění je identifikátor cílového testovacího běhu, který je součástí příchozího požadavku. Jeho úlohou je stažení testovacího scénáře z TestRail pomocí TestRail API [19], k čemuž používá Java třídu *CaseDownloader*.
2. **Executor.** Spouští skript *executor.sh* pro vykonání automatických testů, viz 4.3.
3. **Reporter.** Prezentuje výsledek proběhlých testů zpět do TestRail pomocí Java třídy *ResultPublisher*.



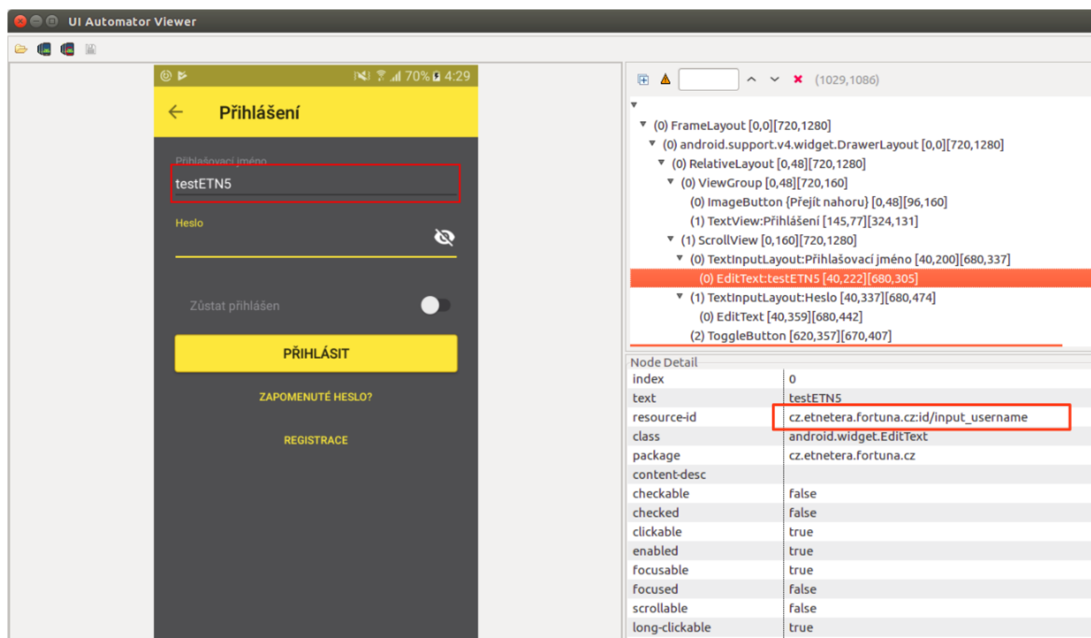
Obrázek 4.10: Ukázka úloh Jenkins při provádění testu.

4.6 UIAutomatorViewer

Díky Appium je práce s prvky uživatelského rozhraní mobilních aplikací snadné. Co však na první pohled není jasné, je způsob identifikace jednotlivých elementů. Nezávisle na platformě, základem každého view, prezentujícího mobilní aplikaci či web, je XML struktura. S tímto faktem je spojena jedna z funkcí Android Debug Bridge, a sice stáhnutí této XML struktury do počítáče. Tato funkce je následně klíčová pro desktopovou aplikaci UIAutomatorViewer [20], jejímž cílem je tuto strukturu prezentovat uživateli. Vlastnosti, se kterými se nejčastěji pracuje jsou *resource-id, text a index*, protože pomocí nich následně přistupujeme k prvkům v textových scénářích. Náhledem použití UIAutomatorViewer je obrázek 4.12, s vyznačeným atributem *resource-id*, jehož použití může vypadat např. jako na na obrázku 4.11.

```
Given element (ID, cz.etnetera.fortuna.cz:id/input_username) as loginTextArea
When we set text to loginTextArea : testETN5
```

Obrázek 4.11: Ukázka testovací šablony.



Obrázek 4.12: Ukázka rozhraní UIAutomatorViewer.

Podstatnou funkcionalitou je také Jenkins podpora verzovacího systému Git, pomocí kterého je implementace nových JBehave šablon usnadněná. Změny provedené v gitu se totiž automaticky updatují i v Jenkins.

Kapitola 5

Analýza možností vytvoření modulu pro správu šablon a jejich implementací

Předmětem této sekce je analýza možností vytvoření modulu pro správu šablon a jejich implementací. Jinými slovy, architektura frameworku, který je předmětem této práce, je navržena způsobem, který nese určitý potenciál ve využití modulárního programování. Rozdělení aplikace na dva moduly, z nichž jeden pouze poskytuje šablony scénářů a druhý tyto šablony používá k vytváření cílových instrukcí, by totiž mohlo přinést značné usnadnění vývoje a nasazování těchto šablon.

V implementační částí této práce je představena třívrstvá architektura, jejímiž prvky jsou instrukční, servisní a řídící vrstva. Třídy řídící části pracují s prvky servisní vrstvy, která tyto požadavky přijímá, transformuje a předkládá instrukční vrstvě. Faktem je, že provázanost těchto vrstev, už díky samotnému konceptu vícevrstvé aplikace, je poměrně malá. Nahlédneme-li např. do tříd jako `ElementSearchService`, která slouží k vyhledávání prvků v uživatelském rozhraní, jedinou další komponentou, se kterou se zde pracuje, je instance třídy `AndroidDriver`. S určitostí můžeme dokonce říct, že podobný princip nastává také u ostatních tříd servisní vrstvy. To je známkou dokonce velmi malé provázanosti s instrukční vrstvou.

Důležitou roli v tomto konceptu hraje také způsob, kterým jsou vybírány třídy, implementující textové scénáře. Spouštěcí třídou celého frameworku je třída AppiumStories, která obsahuje konfigurace nástroje JBehave a spouští cyklus celého testu. Jeho součástí je také vytváření instancí tříd implementující textové scénáře, které jsou vkládány jako parametry konstruktoru `InstanceStepsFactory` následujícím způsobem.

```
new InstanceStepsFactory(configuration(),
        new BeforeSteps(instance),
        new Navigators(instance),
        new Finders(instance),
        new FormHelpers(instance),
        new Gestures(instance),
        new NavigatorAsserts(instance),
        new FinderAsserts(instance),
        new AfterSteps(instance),
        new WelcomePageGuideResolver(instance)
);
```

Nyní je tedy jasné, že framework obsahuje dvě poměrně snadno oddělitelné části, které můžeme nazývat moduly. První částí je řídící a servisní vrstva, druhý částí vrstva instrukční, nebo-li inicializační. K úplnému oddělení je však potřeba provést poslední krok, a sice oddělit zodpovědnost za výběr šablonových tříd od instrukčního modulu. Tedy určování, které třídy se mají použít při párování textových scénářů s implementacemi, by mělo být součástí šablonového modulu. To je možné provést např. těmito dvěma způsoby:

- Vytvořením anotace, která by označovala všechny třídy, které obsahují implementace k šablonám. Pomocí aspektového programování by bylo možné tyto třídy shromáždit a poskytnout instanci `InstanceStepsFactory` i z instrukčního modulu.
- Vytvořením implementace funkcionálního rozhraní, jehož metoda vrací přímo instanci `InstanceStepsFactory` naplněnou vybranými třídami. Tuto implementaci by bylo možné prosadit pomocí konceptu Dependency Injection.

5.1 Realizace

Výše popsaný postup teoreticky rozděluje aplikaci na dva moduly. K realizaci této myšlenky a převedení funkcí do samostatných modulů, je samozřejmě potřeba použít některý framework pro modularizaci. Vhodným kandidátem, který nově nachází svoji integraci ve standardní knihovně Javy 9, je framework Jigsaw. Prvním krokem k jeho využití by bylo logické změnit Java verzi

projektu z 8 na 9. Takové povýšení by také umožnilo použití dalšího velice vhodného nástroje, a sice JLink.

Tradiční spuštění Java aplikace obnáší kompilaci celého projektu, kontrolu závislostí a následné vytvoření Java Runtime Environment. Java 9 v tomto směru přináší poměrně revoluční příštup pomocí nástroje JLink. Použitím JLink je nově možné sestavit běhové prostředí aplikace pouze z kompilátů Java tříd či celých modulů. Taková změna přináší potenciální zefektivnění práce s operační pamětí, ale zároveň sebou nese určitou zodpovědnost. Pro další studium Javy 9 viz [22].

JLink má své použití i v tomto poměrně malém projektu. Za normálních podmínek by k rozšíření frameworku o nové šablony, ať už v modulárním či nemodulárním stavu, bylo třeba nově naimplementované třídy shromáždit ve společném projektu, zkompilovat a teprve tehdy spustit. S Javou 9 je možné preskočit krok společné kompilace a rovnou přikročit k vytváření běhového prostředí. Tato výhoda se na první pohled nemusí zdát zásadní a jejím předpokladem je mít zkompilované Java třídy předem. Pokud však nechceme udržovat několik verzí frameworku pro různé aplikace a kompilovat projekt s přidáním každé šablony, JLink je tím pravým nástrojem.

Kapitola 6

Závěr

Analýza provedená na začátku práce poskytla poměrně široký vhled do metodik testování, přístupu Behaviour Driven Development a vytváření automatických testů. Na základě takto provedené rešerše bylo možné navrhnout komplexní řešení daného problému s poměrně vysokou mírou detailu.

Výstupem této práce je testovací framework, použitelný pro řadu platforem, jako Android, iOS, Windows apod. Součástí tohoto řešení je dokonce jeho aplikace na platformu Android, řada implementovaných šablon a fungující testovací scénáře, které je možné provádět souběžně na několika mobilních zařízeních. Celý tento proces sepsání testovacího scénáře a jeho spuštění je možné ovládat z uživatelského rozhraní TestRail, jehož obsluha nevyžaduje žádnou zkušenost s programováním. Povedlo se tedy vytvořit framework, který je přenositelný mezi více platformami a který snižuje náročnost psaní automatických testů.

Během práce však také vyvstaly problémy a na povrch se dostaly skutečnosti, které je nutné zmínit. Ačkoliv se povedlo vytvořit framework, který je použitelný pro více platforem, nedá se spoléhát na to, že bez další implementace lze využít jeho šablony tak, že vznikne plnohodnotný test na mobilní i desktopovou aplikaci. Jinak řečeno, individuální přístup při vytváření automatických testů na jednotlivých platformách je do jisté míry žádoucí. Implementací rozhraní WebDriver, která je v této prácí použitá, je především AndroidDriver. Přesto, že WebDriver nabízí množství užitečných funkcí, tato implementace pracuje s dalšími atributy a vlastnostmi, specifickými pro právě pro platformu android, které za pouhým rozhraním zůstávají skryté. Konkrétním příkladem je třeba práce s aktivitami, pomocí kterých můžeme

efektivně zjišťovat polohu aplikace. Dalším faktorem je, že cílová zařízení různých platforem, na kterých jsou testy spouštěny, vyžadují naprosto odlišný způsob obsluhy. Na druhou stranu však také platí, že zatímco propast mezi platformami Android a Windows je dost hluboká, rozdíl mezi Android a iOS je poměrně malý a počet funkcí společných pro oba systémy poměrně vysoký.

Vytvořením tohoto nástroje bylo dosaženo cílů, které byly stanoveny na začátku. Pravidelné používání tohoto nástroje v praxi, však teprve odhalí jeho plný potenciál, ale současně jeho slabiny. Souběžné vykonávání testů zatím proběhlo na maximálním počtu 3 zařízení a fungovalo bez problému. Pokryté také byly jenom jednoduché scénáře. Rozšíření nároků na tyto faktory může být v budoucnu možnou překážkou v použití dosavadního řešení. Tedy předpokladem pro další uplatnění této práce může být implementace složitějších scénářových šablon a testování kapacity počtu připojených zařízení.

Literatura

[1] Institute of Electrical and Electronics Engineers, "Guide to the software engineering body of knowledge 2004 SWEBOK [electronic resource]", IEEE 2004. Dostupné z `https://books.google.cz/books?id=bp1wnQAACAAJ`

[2] S. naik and P. Tripathy, Software Testing and Quality Assurance: Theory and Practice, 1st ed., Wiley-Spektrum, 2008, ISBN 978-o-471-78911-6.

[3] Daniel J. Mosley, "A Comparison of Black Box and White Box Text Case Design Strategies", Center for the Study of Data Processing, School of Technology and Information Management, Washington University, 1988

[4] Greg Utas, Robust Communications Software: Extreme Availability, Reliability and Scalability for Carried-Grade systems, 1st ed., Wiley-Spektrum, 2005, ISBN 0-470-85434-0.

[5] Mark Collin, Mastering Selenium WebDriver, Packt Publishing Ltd, 2015, ISBN 9781784397708

[6] Andreas Ebbert-Karroum, JBehave Configuration Tutorial, In: codecentric Blog [online], 16.6.2012 [6.5.2018]. Dostupné z `https://blog.codecentric.de/en/2012/06/jbehave-configuration-tutorial/`

[7] Cucumber Limited, Cucumber: Gherkin. In: github/cucumber [online]. Poslední revize 19.11.2017 [vid. 6.5.2018]. Dostupné z `https://github.com/cucumber/cucumber/wiki/Gherkin`

[8] Mehul Kagathara, Automated Testing using BDT (Behavior Driven Testing), In: © 2018 Infostretch Corporation [online], 13.2.2013 [vid. 7.5.2018]. Dostupné z https://www.infostretch.com/blog/automated-testing-using-bdt-behavior-driven-testing/

[9] B. S. Dhillon, Reliability in Computer System Design, 1st ed., Intellect Books, 1987, ISBN 9780893914127

[10] James Steele, Nelson To, Shane Conder, The Android Developer's Collection (Collection), 1st ed., Addison-Wesley Professional, 2011, ISBN 9780132928618

[11] Arnon Rotem-Gal-Oz, "SOA Patterns". Manning, 2012, ISBN 9781933988269

[12] Sander Rossel, "Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment", 1st ed., Packt Publishing Ltd, 30.10.2017, ISBN 9781787284180

[13] John Smart, Jenkins: The Definitive Guide, 1st ed., "O'Reilly Media, Inc.", 19.7.2011, ISBN 9781449305352

[14] Oficiální dokumentace Android pro vyývojáře: Activity [online], poslední revize 8.5.2018 [vid. 20.5.2018]. Google LLC (c). Dostupné z https://developer.android.com/reference/android/app/Activity

[15] Chris Maddern, A Brief History Of Deep Linking [online], In: Techcrunch. 12.6.2015 [vid. 18.5.2018]. © 2013-2018 Oath Tech Network. Dostupné z https://techcrunch.com/2015/06/12/a-brief-history-of-deep-linking/

[16] Christian Deonier, Felipe Lima, DeepLinkDispatch: A simple, annotation-based library for making deep link handling better on Android [online], In: Medium.com - Airbnb Engineering Data Science. 30.6.2015 [vid. 16.5.2018]. Dostupné z https://medium.com/airbnb-engineering/deeplinkdispatch-778bc2fd54b7

[17] Oficiální dokumentace JBehave [online]. Copyright (c) 2003-2017 jbehave.org. [vid. 20.5.2018]. Dostupné z http://jbehave.org/reference/latest/configuration.html

[18] Oficiální dokumentace Appium [online]. © JS Foundation. [vid. 13.5.2018] Dostupné z https://appium.io/docs/en/writing-running-appium/caps/

[19] Oficiální dokumentace TestRail TM [online]. Copyright © 2003-2018 Gurock Software GmbH. [vid. 15.5.2018]. Dostupné z http://docs.gurock.com/

[20] Oficiální dokumentace Android pro vyývojáře: UI Automator [online], poslední revize 23.4.2018 [vid. 20.5.2018]. Google LLC (c). Dostupné z https://developer.android.com/training/testing/ui-automator

[21] Dokumentace Jenkins API. Poslední revize 20.5.2018 [vid. 20.5.2018]. Dostupné z https://wiki.jenkins.io/display/JENKINS/Remote+access+API

[22] Kishori Sharan, Java 9 Revealed: For Early Adoption and Migration, Apress, 2017, ISBN 9781484225929

Příloha B

Obsah přiloženého CD

```
CD
├── zaverecna_prace.pdf ........................... obsah této práce
└── zadani_prace.pdf ................................... zadání
```

Kód této práce je intelektuálním vlastnictím společnosti Etnetera a.s., a proto nemůže být zveřejněn.

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kozlovský** Jméno: **Marek** Osobní číslo: **423308**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Knihovna pro automatizaci testů založená na šablonách

Název bakalářské práce anglicky:

Template-based Test Automation Framework

Pokyny pro vypracování:

1. Analyzujte možnosti automatizovaného vytváření testů dle uživatelských scénářů.
2. Navrhněte a naimplementujte knihovnu umožňující automaticky generovat a spouštět testy na základě uživatelského scénáře odpovídajícího předdefinovaným šablonám.
3. Integrujte tuto knihovnu s některým z nástrojů pro správu testů (např. Squash TM, TestRail).
4. Vytvořte ukázkové implementace vybraných šablon a demonstrujte na nich funkčnost Vámi navrženého systému.
5. Analyzujte možnosti vytvoření modulu pro správu šablon a jejich implementací

Seznam doporučené literatury:

Matt Wynne, Aslak Hellesoy, The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Bookshelf, 2012
Simon Stewart, David Burns, WebDriver, W3C Candidate Recommendation, 2017
Kent Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, Skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.01.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Martin Ledvinka podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání	Podpis studenta

 © ČVUT v Praze, Design: ČVUT v Praze, VIC