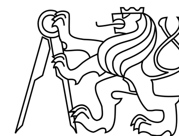
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Moderní testování iOS aplikací |
| **Student:** | Bc. Marek M chura |
| **Vedoucí:** | Ing. Dominik Veselý |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce zimního semestru 2017/18 |

## Pokyny pro vypracování

- Prove te analýzu sou asného stavu integra ních test na platform iOS.
- Porovnejte psaní nativních Apple UI test a n kterého obecného nástroje na bázi Selenia (nap Appium) pro testování nativních i hybridních (html + javascript) mobilních aplikací. Napište ukázkovou sadu test pro všechny postupy.
- Výsledky analýzy a porovnání nástroj zhodno te.
- Navrhn te vhodný postup pro testování UI na mobilních aplikacích za pomocí mockovacích nástroj .
- Napište pomocný nástroj, který usnadní testování UI tester m i automatickým nástroj m pomocí zobrazování jednotlivých obrazovek aplikace jako galerie bez nutnosti prokliku.

## Seznam odborné literatury

Dodá vedoucí práce.

L.S.

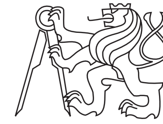
| | |
|---|---|
| Ing. Michal Valenta, Ph.D.<br>vedoucí katedry | prof. Ing. Pavel Tvrdík, CSc.<br>d kan |

V Praze dne 2. b ezna 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství



Diplomová práce

# Moderní testování iOS aplikací

***Bc. Marek Měchura***

Vedoucí práce: Ing. Dominik Veselý

9. května 2016

// Poděkování

Chtěl bych poděkovat Ing. Dominiku Veselému za všechny konzultace, rady a zpětnou vazbu v průběhu práce. Dále bych chtěl poděkovat Daliboru Měchurovi a Janu Nagymu za korektury textu.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2016 . . . . . . . . . . . . . . . . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Marek Měchura. Všechna práva vyhrazena.
*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Měchura, Marek. *Moderní testování iOS aplikací.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

# Abstrakt

Tato práce se zaměřuje na testování v iOS. Konkrétně se zaměřuje na integrační testování. Práce obsahuje analýzu a porovnání dostupných testovacích nástroju pro UI testování na iOS. Důraz je kladen hlavně na nativní testovací nástroje a nástroje založené na bázi Selenia. K těmto nástrojům byly také implementovány sady testů a to pro nativní i hybridní aplikace. Další část práce řeší problematiku mockování při testování na iOS. Poslední část práce řeší implementovaný pomocný nástroj pro testování konkrétních obrazovek aplikace.

**Klíčová slova** testování, iOS, UI Testing in Xcode, Selenium WebDriver, Appium, Calabash, UI Automation, Frank, ios-driver, Mock, Stub

# Abstract

This thesis aims on integration testing on iOS platform. The main part of this thesis analyzes and compares available tools for UI testing on iOS with focus on native testing tools and Selenium based testing tools. For these tools were implemented sets of tests for hybrid and native applications. The next part of this thesis researches mocking and its usage for iOS applications testing. The

last part of the thesis is about implemented helper tool and its features for testing of specific iOS application's screens.

**Keywords** testing, iOS, UI Testing in Xcode, Selenium WebDriver, Appium, Calabash, UI Automation, Frank, ios-driver, Mock, Stub

# Obsah

**Úvod** **1**

**1 Testování softwaru** **3**
1.1 Úrovně testování . . . 3
1.2 Jednotkové testování . . . 5
1.3 Integrační testování . . . 5
1.4 Systémové testování . . . 6
1.5 Akceptační testování . . . 6

**2 Současný stav integračních testů na platformě iOS** **7**
2.1 Testovací nástroje . . . 7
2.2 Testovací aplikace . . . 8
2.3 Nativní vs. hybridní aplikace . . . 8

**3 UI Automation** **11**
3.1 Tvorba testovacího skriptu . . . 11
3.2 Tvorba testovacího skriptu v Automation . . . 12
3.3 Tvorba testovacího skriptu mimo Instruments . . . 12
3.4 Testovací skript . . . 13
3.5 Vyhledávání elementů . . . 13
3.6 Čekání na element . . . 13
3.7 Logování výsledků . . . 14
3.8 UI Automation nahrávání . . . 15
3.9 Kontrola stavů aplikace . . . 16
3.10 Tuneup JS . . . 17
3.11 Příkazová řádka . . . 19

**4 UI Testing in Xcode** **21**
4.1 XCTest . . . 21
4.2 Accessibility . . . 21

4.3 Minimální požadavky pro testování UI v Xcode . . . . . . . . . 22
4.4 UI Testing Xcode Target . . . . . . . . . . . . . . . . . . . . . 22
4.5 UI Testing API . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.6 Syntéza událostí . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.7 UI Recording . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.8 Accessibility Inspector . . . . . . . . . . . . . . . . . . . . . . 30
4.9 Hybridní aplikace . . . . . . . . . . . . . . . . . . . . . . . . . 31

**5 Selenium** **33**
5.1 Selenium RC . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.2 WebDriver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**6 Appium** **37**
6.1 Filozofie Appia . . . . . . . . . . . . . . . . . . . . . . . . . . 37
6.2 Obecný princip fungování . . . . . . . . . . . . . . . . . . . . 38
6.3 Appium na iOS . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.4 Implementace Appia na iOS . . . . . . . . . . . . . . . . . . . 39
6.5 Princip fungování Appia na iOS . . . . . . . . . . . . . . . . . 39
6.6 Appium - životní cyklus . . . . . . . . . . . . . . . . . . . . . 40
6.7 Psaní testů pro Appium . . . . . . . . . . . . . . . . . . . . . 42
6.8 Appium Inspector . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.9 Hybridní aplikace . . . . . . . . . . . . . . . . . . . . . . . . . 45

**7 Calabash** **47**
7.1 Cucumber . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
7.2 Architektura Calabashe . . . . . . . . . . . . . . . . . . . . . . 48
7.3 Integrace s Xcode . . . . . . . . . . . . . . . . . . . . . . . . . 49
7.4 Nastavení prostředí pro práci s testy . . . . . . . . . . . . . . . 50
7.5 Tvorba testů . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
7.6 Ukázka implementace testu . . . . . . . . . . . . . . . . . . . . 53
7.7 Interaktivní mód . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**8 Další testovací nástroje** **57**
8.1 Frank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
8.2 Ios-driver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**9 UI Testing in Xcode vs. Appium** **61**
9.1 Testovací scénář . . . . . . . . . . . . . . . . . . . . . . . . . . 61
9.2 Testovácí aplikace . . . . . . . . . . . . . . . . . . . . . . . . . 62
9.3 Ukázka testů nativní aplikace . . . . . . . . . . . . . . . . . . . 65
9.4 Ukázka testů hybridní aplikace . . . . . . . . . . . . . . . . . . 65
9.5 Pohodlnost . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
9.6 Rychlost . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
9.7 Možnosti testování . . . . . . . . . . . . . . . . . . . . . . . . . 68

9.8 Property Inspector . . . . . . . . . . . . . . . . . . . . . . . . . 69
9.9 Vyhodnocování výsledků . . . . . . . . . . . . . . . . . . . . . . . 70
9.10 Dokumentace . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
9.11 Zhodnocení . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

**10 Zhodnocení testovacích nástrojů** **73**
10.1 Nativní testovací nástroje . . . . . . . . . . . . . . . . . . . . 73
10.2 Nástroje založené na frameworku v Objective-C . . . . . . . . . 75
10.3 Nástroje založené na Selenium WebDriveru . . . . . . . . . . . 76

**11 Integrační testování za pomocí mockovacích nástrojů** **79**
11.1 Základní pojmy . . . . . . . . . . . . . . . . . . . . . . . . . . 80
11.2 Mock vs. Stub . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
11.3 Kdy použít mockování . . . . . . . . . . . . . . . . . . . . . . 82
11.4 Mockování ve Swiftu . . . . . . . . . . . . . . . . . . . . . . . 83
11.5 Mockování UI testů v Xcode . . . . . . . . . . . . . . . . . . . 85

**12 Pomocný nástroj pro testování UI** **87**
12.1 Pomocný nástroj . . . . . . . . . . . . . . . . . . . . . . . . . 87
12.2 Instalace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
12.3 Použití . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
12.4 Možnosti pomocného nástroje . . . . . . . . . . . . . . . . . . 90
12.5 Přínos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**Závěr** **97**

**Literatura** **99**

**A Scénáře pro testování nativní aplikace** **103**
A.1 Testovací scénář 1 . . . . . . . . . . . . . . . . . . . . . . . . . 103
A.2 Testovací scénář 2 . . . . . . . . . . . . . . . . . . . . . . . . . 104
A.3 Testovací scénář 3 . . . . . . . . . . . . . . . . . . . . . . . . . 105
A.4 Testovací scénář 4 . . . . . . . . . . . . . . . . . . . . . . . . . 106
A.5 Testovací scénář 5 . . . . . . . . . . . . . . . . . . . . . . . . . 108

**B Scénáře pro testování hybridní aplikace** **109**
B.1 Testovací scénář 1 . . . . . . . . . . . . . . . . . . . . . . . . . 109
B.2 Testovací scénář 2 . . . . . . . . . . . . . . . . . . . . . . . . . 109
B.3 Testovací scénář 3 . . . . . . . . . . . . . . . . . . . . . . . . . 110
B.4 Testovací scénář 4 . . . . . . . . . . . . . . . . . . . . . . . . . 110

**C Ukázky testů** **113**
C.1 UI Testing in Xcode - nativní aplikace . . . . . . . . . . . . . . 113
C.2 Appium - nativní aplikace . . . . . . . . . . . . . . . . . . . . . 114
C.3 UI Testing in Xcode - hybridní aplikace . . . . . . . . . . . . . 114

C.4 Appium - hybridní aplikace . . . . . . . . . . . . . . . . . . . . . . 114

**D Seznam použitých zkratek** **125**

**E Obsah přiloženého CD** **127**

# Seznam obrázků


2.1 Ukázka obrazovky aplikace zobrazující seznam článků . . . . . . . 8
2.2 Ukázka obrazovky aplikace zobrazující detail článku . . . . . . . . 8

3.1 Ukázka výsledku hierarchického logování v UI Automation . . . . 16

4.1 Stromová hierarchie aplikace . . . . . . . . . . . . . . . . . . . . . 23
4.2 Ukázka Accessibilty inspektoru v Xcode . . . . . . . . . . . . . . . 31

5.1 Znázornění architektury Selenia RC . . . . . . . . . . . . . . . . . 34
5.2 Znázornění architektury WebDriveru . . . . . . . . . . . . . . . . . 35

6.1 Znázornění principu Appia na iOS . . . . . . . . . . . . . . . . . . 40
6.2 Ukázka appium.app . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.3 Ukázka Appium Inspectoru . . . . . . . . . . . . . . . . . . . . . . 45

7.1 Znázornění architektury Calabashe . . . . . . . . . . . . . . . . . . 48
7.2 Znázornění vygenerovaného základu pro Calabash testy . . . . . . 51

9.1 Ukázka obrazovky aplikace zobrazující seznam všech seznamů úkolů 63
9.2 Ukázka obrazovky aplikace zobrazující jednotlivé úkoly ze seznamu úkolů . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
9.3 Ukázka obrazovky aplikace zobrazující úpravu jednotlivých úkolů . 63
9.4 Ukázka obrazovky aplikace zobrazující nastavení aplikace . . . . . 63
9.5 Ukázka výchozí obrazovky aplikace Ionic Beer Explorer . . . . . . 64
9.6 Ukázka obrazovky pro vyhledávání podle jména piva . . . . . . . . 64
9.7 Ukázka obrazovky aplikace zobrazující detail zvoleného piva . . . . 65
9.8 Ukázka obrazovky pro vyhledávání podle pivovaru . . . . . . . . . 65

12.1 Znázornění přilinkování ControllerMocker frameworku . . . . . . . 89
12.2 Ukázka mockovaného kontroleru pomocí ControllerMockeru . . . . 93
12.3 Ukázka druhého mockovaného kontroleru pomocí ControllerMockeru 93
12.4 Ukázka skrytých tlačítek ControllerMockeru . . . . . . . . . . . . . 94

12.5 Ukázka zobrazení seznamu všech mockovaných kontrolerů v ControllerMockeru . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

# Úvod

Raketový rozvoj mobilních technologií a mobilních platforem v posledních letech s sebou přináší stále složitější a sofistikovanější aplikace. S tímto rozvojem komplexnosti, možností a nároků, které jsou na aplikace kladeny, zároveň neustále stoupá potřeba udržovat vývoj těchto aplikací na udržitelné úrovni.

Aby bylo možné z dlouhodobého hlediska aplikaci udržovat a hlavně dále rozvíjet, je nezbytné využívat nějaký mechanismus, který s těmito úkoly pomůže. Tímto mechanismem je právě automatizované testování, které velmi usnadňuje a nepřímo napomáhá dalšímu rozvoji aplikace. Tato práce se konkrétně zaměřuje na testování iOS aplikací.

Testování obecně je velice rozsáhlá oblast. Tato práce se zaměřuje převážně na jednu podoblast testování a to konkrétně na integrační testování na platformě iOS. Integrační testování je v řadě případů na mobilních platformách opomíjené, nicméně se vzrůstající komplexností aplikací je stále důležitější.

Práce si klade za cíl analyzovat dostupné nástroje pro integrační testování na platformě iOS. Zejména se bude zaměřovat na nativní testovací nástroje a nástroje založené na bázi Selenia. Součástí práce je i ukázková sada testů pro testování nativní i hybridní aplikace a to jak pro nativní testovací nástroj, tak i pro nástroj na bázi Selenia. Práce se také zabývá využitím mocků při testování aplikací.

Součástí práce je implementace pomocného nástroje, který usnadňuje testování konkrétních obrazovek aplikace. Pomocný nástroj umožňuje definovat konkrétní obrazovky, které je nutné otestovat a poté mezi nimi umožňuje přepínat podobně jako například v galerii.

Celá práce je strukturována do dvanácti kapitol. První kapitola se zabývá obecným úvodem do testování softwaru. Druhá kapitola se zabývá úvodem do zkoumaných testovacích nástrojů a také uvede testovací aplikaci. Kapitoly tři až osm postupně rozebírají konkrétní testovací nástroje, jejich principy a možnosti.

Kapitola devět se zabývá přímým srovnáním použití konkrétního nativního testovacího nástroje a konkrétního nástroje založeného na bázi Selenia.

V kapitole deset se nachází vyhodnocení testovaných nástrojů. Kapitola jedenáct řeší použití mockování při testování včetně možností mockování testů ve Swiftu. Poslední kapitola pak řeší funkčnost a možnosti implementovaného pomocného nástroje pro testování UI.

Kapitola 1

# Testování softwaru

Software se obvykle skládá z velkého množství nejrůznějších komponent. Jak komplexnost programů roste, roste zároveň množství potřebných komponent a stejně tak roste i potřeba vzájemné spolupráce různých komponent v rámci systému.

Pokud si chceme udržet přehled o programu, jeho funkci, ale souběžně i o jednotlivých komponentách a vazbách mezi komponentami, je nezbytné se na software dívat v různých úrovních v závislosti na tom, jak detailní podrobnosti programu nás zrovna zajímají. Pokud nás například zajímá, jakým způsobem funguje celá aplikace jako globální celek, nebude nás úplně interesovat, že konkrétní funkčnost aplikace řeší $x$ komponent, které spolupracují nějakým konkrétním způsobem.

Obdobně to platí i naopak, jestliže nás zajímá, jak je řešena konkrétní operace aplikace, zajímá nás pouze ta daná komponenta a ne všechny další, které tuto komponentu například využívají ke své funkci.

Úplně stejný princip platí i u testování softwaru. Jakmile máme software, jenž se skládá z komponent, které spolupracují a tvoří větší součásti systému, jež spolupracují s dalšími takovými podobnými částmi systému a ve výsledku tvoří celou aplikaci, je zapotřebí tuto skutečnost reflektovat i při tvorbě testů. To znamená, že při tvorbě testů je nutné myslet na jednotlivé úrovně aplikace a ty postupně testovat.

## 1.1 Úrovně testování

Jak bylo popsáno výše, aplikace se skládá z více různých úrovní, které je potřeba postupně testovat. Podle [1, s. 6] je testování softwaru definováno jako: „Proces vykonávání programu se záměrem nalezení chyb“.

### 1.1.1 Vývoj aplikace a testování

Při vývoji aplikace existuje více přístupů pro implementaci testů. V posledních letech se stále více prosazuje takzvaný Test-driven development (programování řízené testy). Tento způsob implementace aplikace funguje tak, že nejdříve jsou napsány testy pro nějakou konkrétní funkcionalitu a teprve až poté je implementována samotná funkcionalita [2]. Já se zde zaměřím spíše na klasičtější vývoj aplikací, tedy nejprve implementace funkcionality a až poté implementace testů.

Při vývoji aplikace jsou obvykle jako první implementovány menší komponenty systému. Na těchto komponentách a jejich vzájemné spolupráci je obvykle postaven další vývoj aplikace. Z tohoto důvodu je vhodné mít tyto malé komponenty odladěné. Proto je důležité každou samotnou komponentu testovat se záměrem nalezení možných chyb v této komponentě.

Dalším krokem při implementaci aplikace je obvykle spolupráce drobnějších komponent, jež spolu tvoří rozsáhlejší část aplikace. Je samozřejmě také důležité otestovat, že při spolupráci jednotlivých komponent nedochází k chybám, protože i když mám otestované jednotlivé komponenty, ještě to neznamená, že i více komponent spolupracujících spolu bude fungovat bez chyb.

To, že budeme testovat software na různých úrovních, nám pochopitelně ještě nezaručuje, že výsledná aplikace bude bezchybná. Stejně tak nemáme zaručeno, že v testech máme pokryty všechny situace, které mohou nastat. Při použití testů máme ale definovaný nějaký známý stav aplikace. Jestliže po nějaké úpravě aplikace naráz přestane nějaký test procházet, indikuje to, že se něco v aplikaci změnilo, že nějaký očekávaný stav se změnil, a tedy nějaká část aplikace pravděpodobně funguje jinak, než se předpokládá.

Díky použití testů na různých úrovních máme definovaný očekávaný stav aplikace na různých úrovních aplikace, takže se případná chyba odhaluje rychleji a jednodušeji.

### 1.1.2 Výhody testování na různých úrovních aplikace

Nyní si uvedeme ukázku jednoduchého praktického scénáře, proč testovat na všech úrovních. Dejme tomu, že se rozhodneme netestovat jednotlivé komponenty, ale pouze větší části systému, které se skládají z většího počtu komponent. Pokud se k tomuto kroku rozhodnu, jednoznačně se ušetří čas strávený tvorbou testů.

Problém ovšem nastává při vzniku chyby v aplikaci. Jestliže při testu nějaké větší části aplikace dojde k chybě, je zde veliký prostor pro to, kde konkrétně chyba v aplikaci mohla nastat. Chyba mohla nastat v libovolné komponentě aplikace nebo samozřejmě pouze ve spolupráci mezi jednotlivými komponentami. Další možností je kombinace chyby komponenty a vzájemné spolupráce mezi komponentami a tak podobně. Prostor na to, v které části

aplikace mohla nastat chyba, je veliký, a tudíž i čas pro její nalezení je o to větší.

Obecně platí, že ani všechny úspěšné testy nezaručují, že je výsledná aplikace bezchybná. Nicméně přítomnost velkého množství testů, jež testují aplikace na všech úrovních, alespoň zvyšuje šanci nalezení chyby v co nejkratším časovém intervalu, a tím usnadňuje a urychluje její odstranění.

Podle [3, s. 369] se definují čtyři hlavní úrovně testování. Tyto úrovně jsou:

1. Jednotkové testování (Unit Testing)
2. Integrační testování (Integration Testing)
3. Systémové testování (System Testing)
4. Akceptační testování (Acceptance Testing)

## 1.2 Jednotkové testování

Jednotkové testování (Unit Testing) přichází při testování softwaru na řadu jako první. Tento druh testování testuje samostatné jednotky (komponenty) aplikace.

Hlavním účelem tohoto druhu testování je nalezeních chyb jednotlivých komponent. Tyto testy jsou obvykle spouštěny velice často. Jelikož se testuje pouze samostatná jednotka, neběží tyto testy moc dlouho, takže jsou vcelku svižné.

Jednotkové testy jsou většinou pravidelně spouštěny přímo programátorem ještě před tím, než se aplikace dostane na testování k testerovi.

Při testování jednotek se velmi často stává, že testovaná komponenta ke své funkci používá jinou komponentu. Aby bylo možné komponentu otestovat opravdu nezávisle, je v těchto případech nutné použít mocky nebo stuby (více v kapitole 11.1).

## 1.3 Integrační testování

Integrační testování (Integration Testing) slouží na rozdíl od jednotkového testování k testování více spolupracujících jednotek. Tyto jednotky testuje jako skupinu.

Integrační testování slouží ve své podstatě k testování rozhraní mezi spolupracujícími komponentami (jednotkami) [3, s. 370]. Hlavním účelem je najít různé chyby při spolupráci jednotek.

Tento druh testování se používá na různě velké skupiny testovaných jednotek. Minimální počet jednotek nutných k integračnímu testování je dvě jednotky. Po otestování tyto dvě jednotky mohou vytvořit skupinu, kterou je možné testovat s dalšími jednotkami, jež opět vytvoří skupinu a tak to může pokračovat stále dál.

## 1.4 Systémové testování

Systémové testování (System Testing) testuje aplikaci jako celek. Výsledkem systémového testování by mělo být ujištění, že všechny funkce aplikace fungují podle očekávání. Systémové testování testuje také nefunkční požadavky na aplikaci jako jsou výkon, bezpečnost, spolehlivost atd. [3, s. 373].

Tento druh testování obvykle provádí tester, který s vývojem samotné aplikace neměl nic společného. Samotné testovací prostředí by se ideálně mělo co nejvíce blížit produkčnímu prostředí, v jakém aplikace poběží.

Systémové testování by měl být poslední krok před předáním aplikace na testování zákazníkovi.

## 1.5 Akceptační testování

Akceptační testování (Acceptance Testing) je testování, které se odehrává na straně zákazníka. Mělo by přijít na řadu po systémovém testování, tedy hned poté, co tester nebo testerský tým aplikace souhlasí s tím, že je aplikace připravena k předání zákazníkovi.

Hlavním účelem tohoto testování je zjištění, zdali je aplikace připravena na vydání do produkce. Během testování se ověřuje, jestli aplikace dělá to, co by měla. Tento druh testování je velice užitečný hlavně pro zákazníka, který má takto šanci zkontrolovat, zda aplikace funguje podle jeho požadavků.

Kapitola **2**

# Současný stav integračních testů na platformě iOS

Obecně platné pravidlo, že vývoj v IT jde velice rychle dopředu a co je dneska nové, může být zítra zastaralé, platí v mobilních technologiích dvojnásob. Platí to taktéž i o testovacích nástrojích pro mobilní zařízení.

Udržet si přehled o aktuálně používaných technologiích není zcela jednoduché. Přesto jsem se v následujícím přehledu pokusil vybrat, dle mého názoru, v současné době nejpoužívanější testovací nástroje pro integrační testování na platformě iOS.

## 2.1 Testovací nástroje

V následujícím textu a kapitolách budou blíže rozebrány tyto testovací nástroje:

1. UI Automation (viz kapitola 3)
2. UI Testing in Xcode (viz kapitola 4)
3. Appium (viz kapitola 6)
4. Calabash (viz kapitola 7)
5. Frank (viz kapitola 8.1)
6. iOS-driver (viz kapitola 8.2)

Zhodnocení výše uvedených testovacích nástrojů, a to včetně doporučení, kdy je vhodné který nástroj použít, je rozebráno v kapitole 10.

Obrázek 2.1: Ukázka obrazovky aplikace zobrazující seznam článků

Obrázek 2.2: Ukázka obrazovky aplikace zobrazující detail článku

## 2.2 Testovací aplikace

Při analýze testování funkcionality a možností testovacích nástrojů byla použita aplikace pro čtení článků. Dvě hlavní obrazovky jsou zobrazeny na obrázku 2.1 a 2.2.

Aplikace umožňuje čtení článků ze dvou různých serverů. Výpis seznamu článků je vyobrazen na obrázku 2.1. Na tomto obrázku je také vidět tab bar (úplně dole), který umožňuje přepnutí mezi servery. Na druhém serveru vypadá výpis v zásadě stejně. Liší se pouze barvami.

Při kliknutí na článek ve výpisu článků (viz obrázek 2.1) se zobrazí detail článku, který je znázorněn na obrázku 2.2. Jestliže zobrazíme detail článku na druhém serveru, liší se opět pouze barvami.

## 2.3 Nativní vs. hybridní aplikace

V následujících kapitolách se často objeví pojmy jako nativní a hybridní aplikace. Obvykle, pokud nebude řečeno jinak, se jedná o aplikaci nativní.

### 2.3.1 Nativní aplikace

Nativní aplikace je taková aplikace, která je psána v jazyce specifickém pro danou mobilní platformu a využívá nativní komponenty této platformy pro své

fungování. Tyto aplikace mají obvykle vzhled a chování systémových komponent takové, na jaké je uživatel z dané platformy zvyklý. Například nativní aplikace pro iOS jsou psány v Objective-C nebo ve Swiftu.

Jak je asi z předchozího popisu jasné, nativní aplikace běží pouze na dané specifické platformě. Jestliže tedy chceme, aby aplikace běžela na více platformách, musíme stejnou aplikaci implementovat zvlášť pro všechny platformy nebo případně ještě můžeme vytvořit aplikaci hybridní.

Velkou výhodou nativních aplikací je konzistentní vzhled a chování aplikací v rámci dané platformy. Další velká výhoda je to, že tyto aplikace jsou obvykle oproti hybridním aplikacím rychlejší.

### 2.3.2 Hybridní aplikace

Hybridní aplikace je taková aplikace, která běží pouze v rámci WebView. Jedná se tedy o aplikace, jež jsou obvykle vytvořeny za pomoci HTML, CSS a JavaScriptu. Díky tomu, že aplikace je ve své podstatě pouze webová aplikace, která běží v rámci WebView, je možné téměř stejnou aplikaci použít na více různých platformách, které obsahují WebView.

Vzhledem k tomu, že tyto aplikace jsou webové aplikace, nevyužívají nativní systémové komponenty, takže se často může stát, že se chovají a vypadají jinak, než by uživatel na dané platformě čekal.

Největší výhodou těchto aplikací je právě jejich přenositelnost na různé platformy. Mezi hlavní nevýhody pak patří to, že se v řadě případů nechovají takovým způsobem, jakým by uživatel čekal a také často bývají oproti nativním aplikacím pomalejší.

Kapitola 3

# UI Automation

Názvem UI Automation je označován framework pro testování UI od Applu. Až do představení UI Testing in Xcode se jednalo o nativní testovací nástroj od Applu. Po představení UI Testing in Xcode (viz kapitola 4) byl tento framework označen jako zastaralý (deprecated) a v oficiální dokumentaci Applu je doporučeno použít UI Testing in Xcode (viz kapitola 4) [4].

UI Automation je JavaScriptové rozhraní (API), které se využívá ke specifikaci akcí, jež se mají v rámci testu provést. Samotné testy poté běží v Instruments (viz kapitola 6.4.2) a to konkrétně v komponentě Automation. Díky využití Instruments je možné testy integrovat i dalšími nástroji dostupnými v Instruments a vytvořit tak sofistikované testy.

Testovací skripty využívají UI Automation API pro simulaci uživatelských akcí, které se mají provést. Všechny testovací skripty jsou tedy psány v JavaScriptu. Samotné skripty běží mimo testovací aplikaci. Mohou běžet v Instruments, ale také je lze spustit z příkazové řádky.

## 3.1 Tvorba testovacího skriptu

UI Automation umožňuje tvorbu testovacích skriptů dvěma způsoby.

První z nich je za pomocí Instruments přímo v Xcode. V tomto případě je celý skript kompletně tvořen v Automation editoru a odsud lze tento testovací skript spustit.

Druhou možností je tvorba testovacího skriptu mimo Instruments v libovolném textovém editoru. Takto vytvořený testovací skript poté může být buď naimportován do Automation, nebo může být spuštěn pomocí příkazové řádky.

## 3.2 Tvorba testovacího skriptu v Automation

Jak již bylo zmíněno dříve, tak Automation je nástroj z Instruments. Pokud chceme tvořit testovací skript přímo v Automation, musíme jej zapnout.

Zapnout Instruments pro volbu nástroje je možné z Xcode, pokud testovanou aplikaci zkompilujeme pro profilování (lze pomocí klávesové zkratky cmd+i). Po kompilaci se zobrazí okno se všemi nástroji, které Instruments obsahuje. Stačí zvolit nástroj Automation.

Po zapnutí Automation se zobrazí editor, který umožňuje tvorbu a import testovacího skriptu. Automation také umožňuje tvorbu skriptu pomocí nahrávání (viz kapitola 3.8). Výsledný testovací skript je také možné exportovat například pro opětovné použití v jiných projektech.

Implementovaný skript lze průběžně spouštět pouze kliknutím na tlačítko.

### 3.2.1 Nevýhody

Dle mého názoru je použití Automation pro psaní skriptu celkem nepohodlné. Hlavně proto, že editor není příliš chytrý a neumí například vůbec napovídat, takže je potřeba si příkazy pamatovat a napsat celé. Další velkou nevýhodou je velice zvláštní chování při psaní skriptu, jakmile se chceme kupříkladu posunout na konec nebo začátek řádky. Standardní chování cmd + šipka vlevo, respektive vpravo prostě nefunguje. Stejně tak jako obvyklá zkratka pro zakomentování kódu atd.

### 3.2.2 Výhody

Naopak mezi silné stránky psaní testovacích skriptů v Automation je právě spouštění testů. Test se spustí pouze stisknutím jednoho tlačítka. Není potřeba řešit žádné cesty k aplikaci apod., což je nutné při spouštění testu z příkazové řádky. Největší výhoda je však přítomnost nahrávání pro generování testovacích skriptů (viz kapitola 3.8).

## 3.3 Tvorba testovacího skriptu mimo Instruments

UI Automation nevynucuje při tvorbě testovacích skriptů použití Automation. Naopak, UI Automation umožňuje tvorbu skriptů v libovolném textovém editoru a následně umožňuje testovací skript spustit pomocí příkazové řádky.

Vzhledem k ne úplně pohodlnému způsobu tvorby testovacího skriptu v Automation po tomto způsobu tvorby testovacích skriptů sáhne spousta testerů. Testovací skripty jsou psány v JavaScriptu, které v dnešní době zvládá snad každý textový editor, čímž se značně zvýší komfort při psaní testů.

### 3.3.1 Nevýhody

Hlavní nevýhodou při tvorbě skriptů, mimo Automation, je nepřítomnost nahrávání (viz kapitola 3.8) při tvorbě testů. Další poměrně velkou nevýhodou a komplikací je nutnost zadání cesty k testovací aplikaci, což je mnohem složitější, než by se mohlo zdát (viz kapitola 3.11).

### 3.3.2 Výhody

Výhoda je zřejmá na první pohled. Testy lze psát v oblíbeném textovém editoru, takže tester může používat takový editor, na který je zvyklý a nemusí si přivykat na nový, který dle mého názoru navíc není moc dobrý (viz kapitola 3.2.1).

## 3.4 Testovací skript

Jednoduchá ukázka syntaxe testovacího skriptu je znázorněna ve zdrojovém kódu 3.1. Pro lepší představu tento test testuje aplikaci popsanou v kapitole 2.2.

Zdrojový kód 3.1 nejprve nastaví orientaci zařízení. Poté klikne na první článek. Po zobrazení detailu prvního článku se vrátí zpět na výpis článků. Zde se změní výpis článků z jednoho serveru za výpis článků z jiného serveru. Následně pokračuje stejně jako předtím. To znamená, že zobrazí detail prvního článku a vrátí se zpět. Poté opět přepne na výpis článků, který se zobrazuje po zapnutí aplikace.

## 3.5 Vyhledávání elementů

Elementy aplikace tvoří stromovou hierarchii. Tento strom je využíván pří vyhledávání elementů. Automation umožňuje vyhledávání elementů čtyřmi způsoby a to na základě:

1. jména,
2. hodnoty,
3. jiných elementů,
4. rodiče.

## 3.6 Čekání na element

V drtivé většině trošku složitějších testů je čas od času zapotřebí na něco počkat. Například jestliže chceme kliknout na nějaké tlačítko, občas musíme nějakou dobu vyčkat, než se nám tlačítko objeví.

Zdrojový kód 3.1: Ukázka testovacího skriptu pomocí UI Automation

```
var target = UIATarget.localTarget();
var app = target.frontMostApp()
var orientation = UIA_DEVICE_ORIENTATION_PORTRAIT

// set orientation
target.setDeviceOrientation(orientation);

// tap first article
app.mainWindow().tableViews()[0].tap();

// go back to articles listing
app.navigationBar().leftButton().tap();

// tap tab bar to show different articles
app.tabBar().buttons()[1].tap();

// tap first article
app.mainWindow().tableViews()[0].tap();

// go back to articles listing
app.navigationBar().leftButton().tap();

// tap tab bar to show first articles listing
app.tabBar().buttons()[0].tap();
```

Ve výchozím stavu je nastaveno výchozí čekání na element na dobu 5s [4]. Tuto dobu je možné v případě potřeby změnit.

### 3.6.1 Zásobníkový přístup

UI Automation využívá zásobníkový přístup k časovačům. To znamená, že pokud chceme změnit čas čekání na element, musíme nejdříve pushnout na zásobník požadovaný časový interval a poté musíme udělat pop, abychom se vrátili k původnímu času.

Příklad změny časového intervalu je znázorněn ve zdrojovém kódu 3.2.

## 3.7 Logování výsledků

Ve výchozím stavu jsou všechny prováděné příkazy logovány do jednoho logu testu. V tomto logu jsou všechny prováděné příkazy za sebou. Tento styl lo-

Zdrojový kód 3.2: Zásobníkový přístup k časovačům v UI Automation

```
UIATarget.localTarget().pushTimeout(10);
// code with longer time period
UIATarget.localTarget().popTimeout();
```

gování je celkem v pořádku pro krátké testy, ale pro delší a složitější testy se stává log nepřehledný. Pro tyto případy obsahuje UI Automation možnost, jak vytvářet hierarchickou strukturu při logování výsledků.

### 3.7.1 Hierarchické logování

Ve zdrojovém kódu 3.3 je ukázka, jak lze průběh testu logovat hierarchicky. V ukázce jsou opravdové příkazy testu kvůli přehlednosti nahrazeny logováním zprávy.

Zdrojový kód 3.3: Ukázka hierarchického logování v UI Automation

```
// classic command without log group
UIALogger.logMessage("do something");
UIALogger.logMessage("do something 2");
UIALogger.logMessage("do something 3");

// all command between logStart and logPass
// are logged in "My log group"
UIALogger.logStart("My log group");
UIALogger.logMessage("do something in group");
UIALogger.logMessage("do something in group 2");
UIALogger.logMessage("do something in group 3");
UIALogger.logPass();
```

Výsledek logu ze zdrojovém kódu 3.3 je znázorněn na obrázku 3.1.

Z výsledku logování (viz obrázek 3.1) je vidět, že při hierarchickém logování lze jednotlivé skupiny skrýt nebo rozbalit podle potřeby, což na přehlednosti přidá poměrně signifikantně.

## 3.8 UI Automation nahrávání

Nahrávání je jeden z nejužitečnějších nástrojů při tvorbě testovacích skriptů v Automation. Princip je velice jednoduchý, stačí zapnout nahrávání. Po zapnutí nahrávání se spustí testovaná aplikace. Tester pracuje se zapnutou aplikací a nahrávání na pozadí generuje testovací kód.

| | | | |
|---|---|---|---|
| 16 | 02:37:42 CEST | do something | Default |
| 17 | 02:37:42 CEST | do something 2 | Default |
| 18 | 02:37:42 CEST | do something 3 | Default |
| 19 | 02:37:42 CEST | ▼ My log group | Pass |
| 20 | 02:37:42 CEST | do something in group | Default |
| 21 | 02:37:42 CEST | do something in group 2 | Default |
| 22 | 02:37:42 CEST | do something in group 3 | Default |
| 23 | 02:37:42 CEST | | Pass |

Obrázek 3.1: Ukázka výsledku hierarchického logování v UI Automation

V Automation se nahrávání spouští jedním tlačítkem, je to tedy velice jednoduché. Zároveň se tím ušetří spoustu práce s ručním hledáním elementů a psaním různých gest.

### 3.8.1 Tokeny

Další užitečnou věcí je, že když chceme nasimulovat nějakou uživatelskou akci, lze ji obvykle provést různými způsoby. Například vyhledat element, na který chceme kliknout, lze vícero způsoby. Jestliže při nahrávání přistupujeme k nějakému elementu, na který lze přistoupit nejedním způsobem, nabídne nám textový editor více možností, jak k danému elementu přistoupit. Tyto „možnosti“ se nazývají tokeny.

## 3.9 Kontrola stavů aplikace

Kontrola stavů aplikace je důležitá pro elementy, které nelze ověřit pouze nalezením elementu. Pokud máme například načtenou obrazovku aplikace, můžeme chtít ověřit, zdali je titulek dané obrazovky opravdu takový, jaký má být.

Ve výchozím stavu UI Automation neobsahuje porovnávací funkce známé jako asserty. Bez assertů je testování stavů celkem zdlouhavé a také ne moc přehledné. Ukázka kontroly stavů aplikace je znázorněna ve zdrojovém kódu 3.4.

Ze zdrojového kódu 3.4 je vidět, že bylo nezbytné naimplementovat vlastní funkci pro logování stavů aplikace. Nutno dodat, že tato funkce řeší pouze shodu řetězců. Kdyby bylo vyžadováno nějaké jiné porovnání, bylo by zapotřebí implementovat další funkci atd.

Při testování by se podle mě tester neměl zdržovat s implementací něčeho tak elementárního, jako jsou porovnávací funkce. Naštěstí zde příchází na pomoc Tuneup JS (viz kapitola 3.10).

Zdrojový kód 3.4: Příklad kontroly stavů aplikace bez assertů

```
var target = UIATarget.localTarget();
var app = target.frontMostApp()

// check if screen title match screen
checkTitle(
  app.navigationBar(),
  "CDR - články",
  "Check navigation bar title after app is loaded"
)

// go to different screen
target.frontMostApp().tabBar().buttons()[1].tap();

// check if screen title match screen
checkTitle(
  app.navigationBar(),
  "DIIT - články",
  "Check navigation bar title after change listing"
)

function checkTitle(navBar, expectedName, testName) {
  UIALogger.logStart(testName);

  if (navigationBar.name() == expectedName) {
    UIALogger.logPass("Navigation bar title ok");
  }
  else {
    UIALogger.logError(
      "Navigation bar title mismatch.");
  }
}
```

## 3.10 Tuneup JS

Tuneup JS je kolekce JavaScriptových utilit k vylepšení UI Automation. Hlavní přínosy jsou rozčlenění testů a implementované porovnávací funkce (asserty) [5].

Instalace Tuneup JS je možná buď manuální, nebo pomocí CocoaPods. Po instalaci je nutné naimportovat JavaScriptový soubor tuneup.js. Poté je možné psaní testu za pomoci tohoto rozšíření.

Ve zdrojovém kódu 3.5 je přepsaný příklad zdrojového kódu 3.4 pomocí Tuneup Js.

Zdrojový kód 3.5: Příklad kontroly stavů aplikace pomocí Tuneup JS

```
#import "/path/to/Pods/tuneup_js/tuneup.js"

// check if screen title match screen
test(
  "Check navigation bar title after app is loaded",
  function(target, app) {
    var frontApp = target.frontMostApp()
    assertEquals(
      "CDR - články",
      frontApp.navigationBar().name(),
      "Navigation bar title mismatch."
    )
});

// go to different screen
target.frontMostApp().tabBar().buttons()[1].tap();

// check if screen title match screen
test(
  "Check navigation bar title after app is loaded",
  function(target, app) {
    var frontApp = target.frontMostApp()
    assertEquals(
      "DIIT - články",
      frontApp.navigationBar().name(),
      "Navigation bar title mismatch."
    )
});
```

Příklad 3.5 ukazuje, že jednotlivé testy jsou nyní automaticky členěny mezi funkce ***test***. Každá tato funkce má svůj název, který naznačuje, co funkce dělá, a tento název je po testu vidět v logu.

Z příkladu 3.5 je také vidět, že díky Tuneup JS již není nutné psát vlastní porovnávací funkce, jelikož Tuneup JS je implementuje za nás.

## 3.11 Příkazová řádka

V kapitole 3.3 bylo zmíněno, že testovací skripty je možné spouštět pomocí příkazové řádky. Byly také podotknuty nevýhody spojené s tímto způsobem použití (viz kapitola 3.3.1). Na způsob užití a popsané nevýhody bych se zde rád zaměřil.

### 3.11.1 Spuštění testovacího skriptu přes příkazovou řádku

Příkaz pro spuštění testovacího skriptu je zobrazený ve zdrojovém kódu 3.6. Můžeme pozorovat, že pro spuštění je potřebné zadat relativně velké množství parametrů.

Zdrojový kód 3.6: Ukázka spuštění testu pomocí příkazové řádky

```
instruments \
    -w <deviceID> \
    -t </path/to/Automation.tracetemplate> \
    <fullPathToApplication> \
    -e UIASCRIPT <pathToScript.js>
```

Po skončení testu se zobrazí cesta k výslednému souboru, který je známý jako ***trace*** soubor.

### 3.11.2 Trace soubor

Trace soubor je soubor vytvořený po běhu testovacího skriptu přes příkazovou řádku. Tento soubor se automaticky otevírá v Automation.

Trace soubor obsahuje kompletní log testu spolu s informací o výsledcích jednotlivých příkazů. Soubor také zobrazuje časovou osu testu spolu se zařízením, na kterém test běžel.

### 3.11.3 Zjištění ID zařízení pro test

V příkazu pro spuštění testu pomocí příkazové řádky (viz zdrojový kód 3.6), je potřeba zadat ID zařízení, na kterém chceme test spustit.

Nejjednodušší způsob, jak zjistit dostupné zařízení, je spustit testovací skript (viz zdrojový kód 3.6) s prázdným ID zařízení (***deviceID***). Pokud tak učiníme, zobrazí se seznam dostupných zařízení pro testování.

### 3.11.4 Cesta k testované aplikaci

Asi největší nepříjemností při použití příkazové řádky pro testování je nalezení cesty k testované aplikaci. Toto tvrzení ale platí jenom někdy.

Standardně v Xcode máme možnost, při kliknutí pravým tlačítkem na libovolný soubor, tento soubor otevřít ve Finderu, čímž získáme velice jednoduchým způsobem celou cestu k požadovanému souboru. Takový postup můžeme využít i pro zjištění cesty ke zkompilované aplikaci. Problém ovšem nastává ve chvíli, kdy tento způsob přestane fungovat.

Naneštěstí v praxi takovýto způsob zjištění cesty k aplikaci nefunguje velice často. Při požadavku na otevření aplikace ve Finderu skončí tento příkaz často zobrazeným Finderem, který ale nezobrazí cestu k požadovanému souboru. Jedná se pravděpodobně o bug v Xcode.

Pokud otevření ve Finderu selže, je pravděpodobně nejjednodušší použít k vyhledání aplikace příkaz ***find*** v příkazové řádce. Toto vyhledávání není úplně přehledné a to hlavně díky dlouhým a nepřehledným identifikátorům. Každopádně se dle mých zkušeností stále jedná o nejpohodlnější způsob, jak zjistit celou cestu ke zkompilované aplikaci, jestliže Xcode zrovna nespolupracuje.

### 3.11.5 Psaní testů - Automation vs. příkazová řádka

V předchozím textu byly relativně podrobně probrány výhody a nevýhody testování pomocí UI Automation v Automation, respektive příkazové řádce.

Mně osobně nejvíce vyhovovalo si pomocí nahrávání v Automation zjistit cesty k jednotlivým elementům a samotné testy poté řešit mimo Automation a spouštět je v příkazové řádce.

Kapitola **4**

# UI Testing in Xcode

UI Testing in Xcode bylo přestaveno v polovině roku 2015 na WWDC. Jedná se o nástupce UIAutomation, které bylo do té doby používáno jako oficiální doporučený způsob pro testování UI ze strany Applu. Ihned po představení UI Testing in Xcode byl označen UIAutomation jako zastaralý (deprecated).

UI Testing in Xcode stojí na spolupráci testovacího frameworku XCTest a Accessibility.

## 4.1 XCTest

XCTest je testovací framework zakomponovaný v Xcode, který byl představen v Xcode 5. Podpora pro UI testy byla přidána až v Xcode 7.

Díky integraci testovacího frameworku přímo v Xcode je možné využívat všechny výhody Xcode, například autocomplete a debuger. Podporovaný jazyk pro psaní testů je Objective-C a Swift.

Samotné použití XCTest je velice přímočaré a skládá se ze tří jednoduchých kroků.

1. Tvorba podtřídy XCTest.

2. Implementace testovacích metod.

3. Použití Assertions pro validaci stavů aplikace.

## 4.2 Accessibility

Accessibility neboli přístupnost je technologie, kterou Apple využívá pro zpřístupnění aplikace zdravotně znevýhodněným lidem. Toto zpřístupnění by mělo takovým lidem umožňovat stejný uživatelský zážitek při používání aplikací.

Accessibility využívá bohatou množinu sémantických dat o UI, kterou poté může využívat například hlasový syntetizátor pro provedení aplikací u slepých

lidí. Accessibility je přímo integrována v UIKitu, takže využíváním těchto komponent lze získat spoustu přístupnosti zcela zdarma.

## 4.3 Minimální požadavky pro testování UI v Xcode

Aby bylo možné používat UI Testing in Xcode, je potřeba splnit tři podmínky [6]:

1. použít verzi iOS 9 nebo novější,
2. povolit zařízení pro vývoj,
3. zařízení musí být připojeno k důvěryhodnému hostu, na kterém běží Xcode.

## 4.4 UI Testing Xcode Target

Pro Unit testy se v Xcode používá separátní Xcode target. V případě UI testů je situace obdobná.

UI testy mají nový typ targetu, který zajišťuje některé specifické požadavky pro UI testy. Tento target zajišťuje například to, aby byla testovaná aplikace spuštěna v rámci targetu v separátním procesu. Dále například kontroluje oprávnění pro používání accessibility.

## 4.5 UI Testing API

UI Testing API se skládá v zásadě ze tří tříd.

1. XCUIApplication
2. XCUIElement
3. XCUIElementQuery

### 4.5.1 XCUIApplication

XCUIApplication je proxy objekt pro testovanou aplikaci.

UI testy vždy běží v separátním procesu. Pokud spustíme novou instanci aplikace, implicitně se ukončí předchozí instance. Každá testovaná aplikace se musí spustit za pomocí proxy objektu XCUIApplication.

Po spuštění aplikace lze začít pracovat se samotnými elementy a dotazy.

### 4.5.2 XCUIElement

XCUIElement je proxy objekt pro UI elementy v testovací aplikaci.

Každý XCUIElement má svůj typ (například Button, Window, Cell, atd.) a taky má svůj identifikátor.

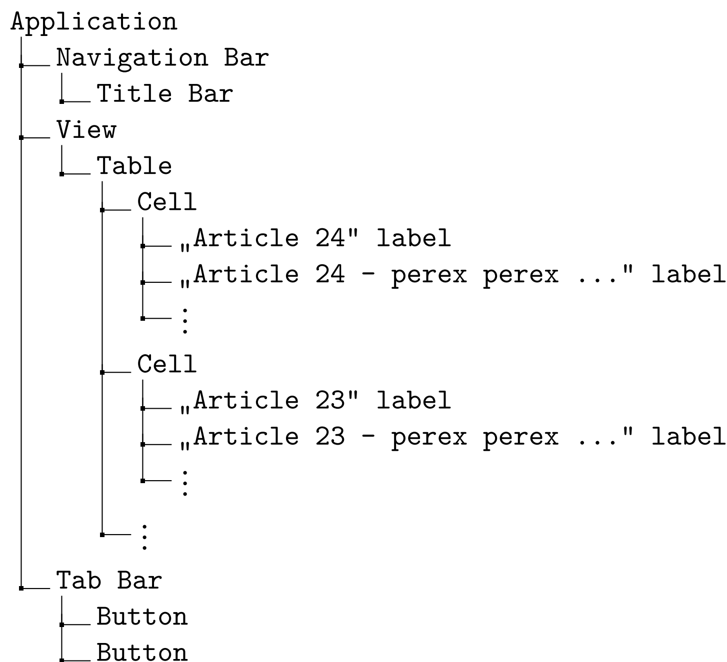
Identifikátorů může být více. XCUIElement může mít accessibility identifikátor, což je řetězec, který tento element identifikuje. Dále může být element identifikován například pomocí labelu, titulku atd.

Při vyhledávání elementů je možné kombinovat typ elementu s jeho identifikátorem. Například můžeme hledat typ elementu „Button" s labelem „Yes".

#### 4.5.2.1 Hierarchie elementů

Všechny elementy aplikace jsou hierarchicky řazeny do stromu. Aplikace (XCUIApplication) je kořen tohoto stromu.

Na obrázku 4.1 je znázorněna stromová hierachie elementů z aplikace zobrazené na obrázku 2.1.




Obrázek 4.1: Ukázka znázornění stromové hierarchie aplikace 2.1

### 4.5.3 XCUIElementQuery

XCUIElementQuery je API, které slouží pro specifikaci elementu. Je schopné vyhledávat pouze elementy, které jsou viditelné pro accessibility. Výsledek do-

tazu vrací množinu nalezených elementů. Při vyhledávání se využívá hierarchie elementů spolu s typem a identifikátory.

### 4.5.4 XCUIElementQuery - vztahy

#### 4.5.4.1 Descendants

Pomocí vztahu Descendants (potomstvo) lze zvolit všechny podelementy daného elementu.

#### 4.5.4.2 Children

Vztah Children (děti) se liší od vztahu Desendants pouze v tom, že vybere jen přímé potomky daného elementu.

#### 4.5.4.3 Containment

Containment (zadržování) je typ vztahu, jenž lze použít u elementů, které jsou hodně obecné, ale obsahují specifický element. Typickým příkladem může být volba buňky, která obsahuje nějaký konkrétní text. Příkladem může být vyhledání buňky (Cell), jež obsahuje label „Article 24“.

### 4.5.5 XCUIElementQuery - filtry

Vyhledávání elementů v UI testech umožňuje filtrování podle:

1. typu - Button, Cell, Menu atd,
2. identifikátoru - accessibility ID, label, title atd,
3. predikátu - hodnota nebo částečná shoda (například „začíná s“) atd.

### 4.5.6 XCUIElementQuery - vyhledávání

Vyhledávání elementů pomocí XCUIElementQuery využívá kombinace vztahů (viz kapitola 4.5.4) a filtrů (viz kapitola 4.5.5).

Pro vyhledávání jsou v API k dispozici 3 konstrukty:

1. descedantMatchingType,
2. childrenMatchingType,
3. containingType.

#### 4.5.6.1 descedantMatchingType

Podle Applu je tento typ dotazu nejpoužívanější ze všech [6]. Vyhledávání funguje podle principu popsaného zde 4.5.4.1. Protože se jedná o nejužívanější typ dotazu, připravil Apple zjednodušené použití. Ukázka syntaxe včetně zjednodušení tohoto dotazu je ukázána ve zdrojovém kódu 4.1.

Zdrojový kód 4.1: Ukázka použití descedantMatchingType

```
// Find all Buttons in app
XCUIApplication().descedantMatchingType(.Button)

// Find all Buttons in app - simplification
XCUIApplication().buttons
```

#### 4.5.6.2 childrenMatchingType

Tento typ dotazu funguje podle principu popsaného v 4.5.4.2. Příklad syntaxe pro vyhledání labelu v Navigation baru je ukázán ve zdrojovém kódu 4.2

Zdrojový kód 4.2: Ukázka použití childrenMatchingType

```
// Find all Cells in table
let navBars = XCUIApplication().navigationBars
navBars.childrenMatchingType(.Label)
```

#### 4.5.6.3 containingType

Jak již bylo popsáno v 4.5.4.3, tento dotaz slouží k vyhledávání elementů, které mají specifický subelement. Ukázka syntaxe pro vyhledání buňky tabulky na základě jejího labelu je ukázána ve zdrojovém kódu 4.3.

Zdrojový kód 4.3: Ukázka použití containingType

```
// Find cell which contains label "Article 24"
cells.containingType(
        .StaticText,
        identifier: "Article 24"
)
```

### 4.5.7 XCUIElementQuery - volba elementu

Pokud vykonáme dotaz, který nevrací jeden unikátní element, ale vrací množinu elementů, musíme být schopni nějakým způsobem z této množiny zvolit konkrétní element. V této situaci máme v zásadě tři možnosti, jak z množiny elementů zvolit jeden určitý.

#### 4.5.7.1 Subscripting

První možností je zvolit element pomocí jeho textové hodnoty. Tento přístup k prvkům obvykle generuje UI Recording (viz kapitola 4.7). Pokud se dotazujeme na element, který je statický a víme, že se nebude jeho text měnit (například konkrétní tlačítko), je tento postup v pořádku.

Problém nastává u dynamicky generovaného obsahu, jako je například seznam článků stahovaných přes internet (viz obrázek 2.1). Pokud bychom ve svém UI testu otestovali zobrazení detailu prvního článku a použili pro výběr tohoto článku jeho titulek, jak to obvykle generuje UI Recording (viz kapitola 4.7), fungoval by tento test jenom do doby, než by se při aktualizaci seznamu článků dostal na první pozici nový článek. Nový článek má samozřejmě jiný titulek, takže by nezvolil první článek.

Pokud má tedy UI test fungovat obecně, tedy například zvolit $x-$tý element, je vhodnější použít index (viz kapitola 4.5.7.2)

#### 4.5.7.2 Index

Volba elementu podle indexu je ideální u množiny elementů, které se dynamicky mění nebo pokud chceme zvolit vždy $x-$tý element.

#### 4.5.7.3 Unikátní prvek

Poslední možnost, jak zvolit konkrétní element z množiny elementů funguje na základě toho, že je volený prvek unikátní. Tuto metodu přístupu k elementu můžeme použít tehdy, jestliže víme, že výsledný prvek bude unikátní.

### 4.5.8 XCUIElementQuery - vyhodnocování dotazů

Všechny XCUIElementQuery dotazy jsou vyhodnocovány na požádání. To znamená, že když si vytvoříme nějaký dotaz, tak dokud ho opravdu nezavoláme nebo nepoložíme dotaz na nějakou jeho vlastnost (property), dotaz se vůbec nevykoná [6].

Pokud vytvoříme dotaz na nějaký neexistující element, ale samotný dotaz nezavoláme, test neskončí chybou.

Pro lepší představu lze použít obrázek 2.1, jenž má tab bar (bar úplně dole), který obsahuje 2 tlačítka. Jestliže vytvoříme dotaz na třetí, tedy neexistující tlačítko, nestane se vůbec nic. Při pokusu o kliknutí na toto neexistující tlačítko skončí test chybou. Tento scénář je znázorněn ve zdrojovém kódu 4.4.

Zdrojový kód 4.4: Ukázka vyhodnocování dotazů na požádání

```
let app = XCUIApplication()
let tabBars = app.tabBars
let tabBarBtns = tabBars.childrenMatchingType(.Button)

// query third (non-existent) button
let thirdBtn = tabBarBtns.elementBoundByIndex(2)

// test fail here because now is query resolved
thirdBtn.tap()
```

Podle Applu by se měly všechny dotazy provést znovu, pokud dojde ke změně UI. V praxi jsem v simulátoru párkrát narazil na problém, kdy se při změně UI dotazy znovu neprovedly. Pravděpodobně se ale jednalo jen o bug v Xcode.

## 4.6 Syntéza událostí

Syntéza událostí neboli Event synthesis je způsob, jakým jsou simulovány uživatelské akce.

Simulace uživatelských událostí se děje na velmi nízké úrovni systému, takže na veškeré akce jsou uplatňovány úplně stejné procesy, jaké jsou uplatňovány na opravdové uživatelské akce [6]. Díky tomu se nemusíme bát, že by UI test, který prošel v pořádku, později nefungoval při testu opravdovým uživatelem.

## 4.7 UI Recording

Z informací o fungování a dotazování se v UI testech by mělo být poměrně zřejmé, jak vlastně testování v Xcode funguje. Z ukázek zdrojových kódů je očividné, že syntaxe není nějak závratně složitá, ale pokud by se měly všechny UI testy psát ručně, zabralo by to pravděpodobně spoustu času a nebyla by to nikterak extrémně zábavná záležitost. Přece jenom, přemýšlení nad tím, jak vybrat konkrétní element v kódu pro testování, když na první pohled nemusí být jasné, o jaký typ elementu se jedná (například UIView se může chovat jako UIButton), může chvilku zabrat. Na pomoc zde přichází UI Recording.

UI Recording je nejjednodušší způsob, jak rychle naklikat UI testy. Dovoluje interagovat s aplikací a zatímco se aplikace normálně používá, UI Recording na pozadí generuje kód toho, co uživatel v aplikaci dělá, aby poté mohl celou akci provedenou uživatelem zopakovat.

Krása UI Recording spočívá v tom, že si lze tímto způsobem naklikat celý UI test, ale zároveň umožňuje rozšířit současný UI test. Jestliže máme například hotový test, do kterého bychom chtěli přidat klik na nějaké nové tlačítko, můžeme pomocí UI Recordingu přidat pouze klik na toto tlačítko a to v místě, kde to logicky v testu dává smysl. UI Recording můžeme také použít pro zjištění identifikátoru elementu.

### 4.7.1 Teorie vs. praxe

Na první pohled se zdá, že UI Recording je vše, co je potřebné pro tvorbu UI testů v Xcode. Většina uživatelů ovšem po pár testech UI Recordingu zjistí, že to není úplně pravda.

### 4.7.2 Klady

Je pravda, že UI Recording funguje skvěle pro statické prvky, které se nemění a také je to skvělý nástroj pro zjištění identifikátoru elementu, pokud vůbec netušíte, jak se k danému elementu dostat.

#### 4.7.2.1 Dynamicky generovaná data

Kde UI Recording mírně pokulhává, bylo již nastíněno v kapitole 4.5.7.1. V dynamicky generovaných seznamech, kde se generuje identifikátor na základě textové hodnoty, je obvykle potřeba ručně identifikátory upravit, aby využívaly indexy.

#### 4.7.2.2 Problém s UITableView

Dalším, a dle mého názoru zásadnějším problémem, je využití UI Recordingu při použití ***UITableView*** s trochu větším množstvím dat. Je obecně známo, že ***UITableView*** je schopno pracovat s obrovským kvantem dat bez větších problémů. Je to možné díky znovupoužití buněk.

Bohužel v případě UI Recordingu je problém v tom, že když test načte obrazovku s tabulkou, potřebuje si udělat snapshot této obrazovky a to celé obrazovky a ne jenom viditelných buněk. To způsobí, že test se při menším počtu dat výrazně zpomalí a při větším množství dat skončí chybou „UI Testing Failure - Failed to get refreshed snapshot“. Tato chyba se projevovala již v Xcode 7.1 a zdá se, že v současné době přetrvává i v Xcode 7.3 beta.

Tato chyba byla reportovaná jako bug již poměrně dávno, takže nezbývá, než si počkat na opravu ze strany Applu.

### 4.7.3 Validace elementů

#### 4.7.3.1 Implicitní validace

Při UI testech nahraných pomocí UI Recording se implicitně validují elementy, se kterými se interaguje. Což znamená, že jestliže má test na něco kliknout, automaticky se validuje, že element existuje. V případě, že tomu tak není, skončí test chybou.

Pokud máme test, o kterém víme, že nějaký prvek v daném okamžiku nemusí existovat (pokud se například zobrazuje až po nějaké akci), lze testovat existenci prvku pomocí property exists.

#### 4.7.3.2 Asserty

Implicitní validace sice automaticky validuje, že daný prvek opravdu existuje v době testu, ale to nemusí vždycky stačit. Typickým příkladem může být zaškrtávací tlačítko (checkbox).

Jakmile budeme mít test, který má zaškrtnout zaškrtávací tlačítko, implicitní test zvaliduje, že zaškrtávací tlačítko existuje. Tato informace nám ale neřekne vůbec nic o tom, zdali v době testu už bylo zaškrtnuté, či nikoliv a také jaký je stav po uživatelské akci.

Jednoduché řešení zde nabízí asserty. Asserty umožňují porovnávat vnitřní stavy elementů k nějaké požadované hodnotě. Když vezmeme příklad zaškrtávacího tlačítka, asserty umožňují nadefinovat porovnání, které zkontroluje stav zaškrtávací tlačítko před a taky po uživatelské akci, takže je možné zjistit, jestli se aplikace chová opravdu korektně.

Je nutné dát pozor na to, že hodnota elementu, kterou lze použít pro asserty, vrací typ AnyObject, takže před porovnáním je potřeba ji přetypovat. Jednoduchá ukázka assertu výše popsaného příkladu je ve zdrojovém kódu 4.5.

Zdrojový kód 4.5: Ukázka assertů v UI testech

```
let myBtn = table.buttons["myBtn"]

// verify that checkbox is not checked
XCAssertEqual(myBtn.value as! String, "0")
// check checkbox
myBtn.tap()
// verify that checkbox is checked
XCAssertEqual(myBtn.value as! String, "1")
```

Asi nejjednodušší způsob, jakým lze zjistit hodnotu pro porovnání v assertu, je pomocí příkazu ***po myBtn.value***, který můžeme zadat do Xcode konzole.

### 4.7.4 Accessibility

Jak již bylo zmíněno v úvodu této kapitoly, UI Testing in Xcode využívá accessibility ke svému fungování. UI Recording ji samozřejmě využívá také.

Může se stát, že při použití UI Recording budeme klikat na nějaký element, ale vygenerovaný kód bude na první pohled vypadat nesprávně (vygeneruje se nějaký nesmysl). Tento problém je obvykle způsoben špatný nastavením accessibility daného elementu. Většinou se objevuje u nestandardních elementů, jako je custom ***UIView*** a podobně, které se ve výsledku chová například jako tlačítko.

Problém s přístupností elementů lze vcelku jednoduše řešit a sice takovým způsobem, že se accessibility u elementu zapne. Zapnutí accessibility lze jednoduše provést jak ve storyboardu, tak i v kódu. Při zapnutí accessibility a nastavení ID je také vhodné nastavit vlastnost, které se nazývá „Trait“.

Pomocí Trait lze určit, jak se má systém k tomuto custom elementu chovat. Jestliže máme například custom ***UIView***, které se chová jako ***UIButton***, nastavením Trait na Button se bude systém k tomuto ***UIView*** chovat jako k tlačítku.

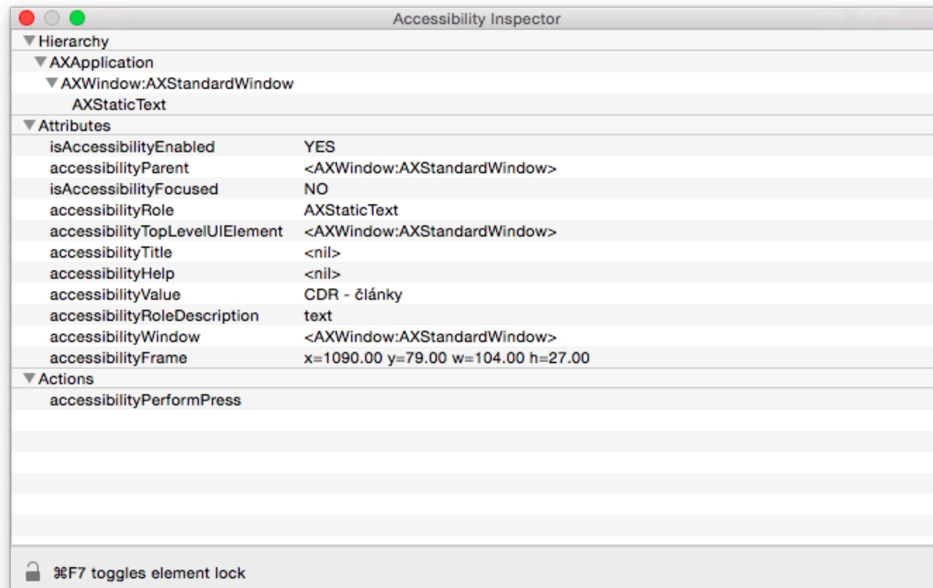
Z předchozího textu je zřejmé, že UI testy nejenom testují aplikaci, ale slouží taktéž jako zpětná vazba k tomu, jak moc je aplikace přístupná pro postižené lidi.

## 4.8 Accessibility Inspector

Velice užitečným pomocníkem při tvorbě UI testů je Accessibility Inspector (inspektor přístupnosti). Tento inspektor funguje tak, že po zapnutí se zobrazí okno, který se automaticky drží nad všemi ostatními. Toto okno zobrazuje všechny informace o přístupnosti elementu, na který ukazuje kurzor myši. Ukázka tohoto inspektoru je na obrázku 4.2. Z obrázku je vidět, že zvolený element je přístupný. Jedná o statický text, který má hodnotu „CDR - články“.

Použití tohoto inspektoru je velice užitečné, jestliže chceme pracovat s elementem, u kterého nevím, jaké má accessibility ID. Velmi se také hodí v případech, kdy používám UI Recording, které po interakci s nějakým elementem vygeneruje nějaký nesmyslný kód nebo pouze komentář s informací, že nebylo možné k dané akci vygenerovat příslušný kód. V těchto případech si člověk nemusí být jistý, zdali se jedná pouze o chybu v Xcode nebo se jedná o problém s přístupností. Při použití inspektoru je hned vidět, jestli je opravdu daný element přístupný, nebo ne.

Accessibility inspektor je dostupný v rámci vývojářských nástrojů od Applu.




Obrázek 4.2: Ukázka Accessibilty inspektoru v Xcode

## 4.9 Hybridní aplikace

Testování v Xcode samozřejmě také umožňuje testování hybridních aplikací. Testování je oproti testování nativních aplikací trochu komplikovanější a neumožňuje takovou flexibilitu testů.

Hlavní rozdíl oproti testování nativních aplikací je v tom, že při výběru elementů v rámci WebView se obvykle hledá pouze na základě textů (statických řetězců), takže není tak jednoduché nadefinovat nějaké obecnější pravidlo pro výběr nějakého elementu.

Kapitola **5**

# Selenium

Na otázku, co je to Selenium, najdeme na oficiálních stránkách krátkou, avšak velmi výstižnou odpověď. Selenium automatizuje prohlížeče [7].

Primárně se Selenium používá pro testování webových aplikací. Dalším častým příkladem použití je automatizace webových administrátorských úkonů.

Z úvodního představení vyplývá, že Selenium je nástroj pro automatizaci prohlížeče a tedy webových aplikací. Důvod, proč je zde tento nástroj podrobněji rozebrán, je ten, že v dnešní době existuje spousta hybridních aplikací, které jsou v zásadě webové aplikace běžící ve WebView. Dalším a zároveň hlavním důvodem je to, že na tomto nástroji je postavený testovací nástroj Appium (viz kapitola 6) a Frank (viz kapitola 8.1).

## 5.1 Selenium RC

Starší verze Selenia je známa jako Selenium 1 nebo Selenium Remote Controler či Selenium RC. Různé názvy, jež odkazují na stejnou aplikaci.

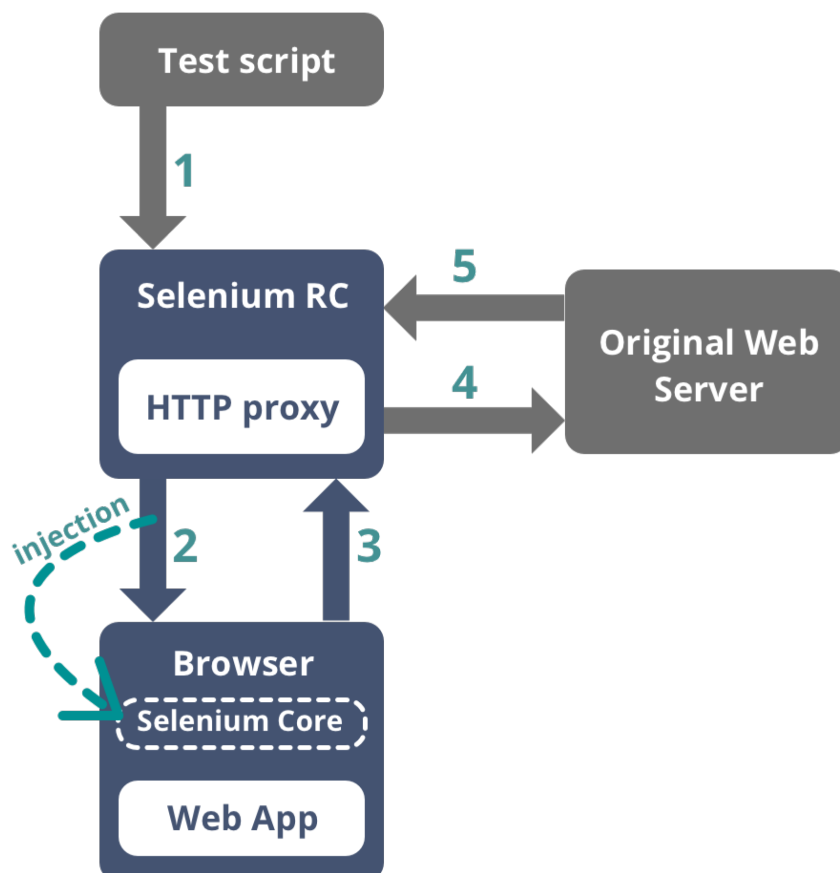
Selenium RC využívalo JavaScript nazývaný Selenium Core k testování aplikací v prohlížeči. Rozhodnutí, aby Selenium fungovalo jako obecný JavaScript, který může testovat aplikaci v libovolném prohlížeči, mohl vždy skončit bezpečností chybou nazývanou Same-Origin Policy.

Ve zkratce a velice zjednodušeně řečeno, když prohlížeč načte nějakou webovou stránku, vytvoří této stránce separátní sandbox pro její JavaScript. Tento sandbox dovoluje vykonávat JavaScript pouze z příslušné domény. Pokud by se pokusil vykonat JavaScript z odlišné domény, bude zablokován prohlížečem.

### 5.1.1 Selenium RC - princip fungování

Zvolená architektura Selenia RC vynucuje překonání bezpečnostního omezení prohlížečů pro správné fungování automatizace. Selenium RC vyřešil tento

problém takovým způsobem, že se chová jako HTTP proxy server. Princip fungování na základě HTTP proxy serveru je znázorněn na obrázku 5.1.




Obrázek 5.1: Znázornění architektury Selenia RC

Jakmile skript žádá spuštění prohlížeče (na obrázku 5.1 znázorněno jako číslo 1), Selenium RC nastartuje prohlížeč, do kterého injektuje Selenium Core (na obrázku 5.1 znázorněno jako číslo 2). Všechny následující požadavky na testování webové aplikace procházejí skrze Selenium RC (HTTP proxy server) ke skutečnému serveru, na kterém je hostována webová aplikace (na obrázku 5.1 znázorněno jako číslo 3, 4 a 5).

Tímto způsobem donutí prohlížeč, aby si myslel, že webová aplikace je hostována na doméně Selenium RC serveru a ne na originálním hostingu. Díky tomu je možné vykonat příkazy ze Selenium Core pro ovládání webové aplikace. Z pohledu prohlížeče je teď aplikace a Selenium Core ve stejné doméně

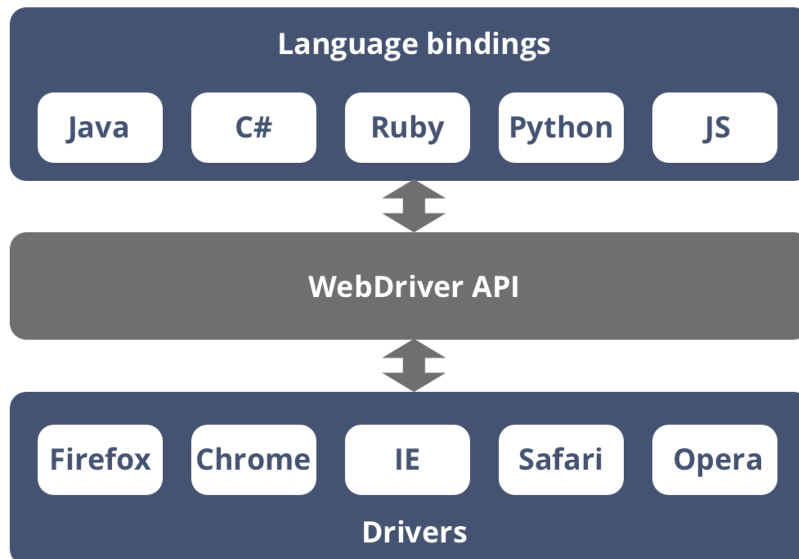
[8, s. 10].

## 5.2 WebDriver

WebDriver, občas nazývaný Selenium WebDriver či Selenium 2, je nová verze Selenia (z roku 2011), která se používá v současné době. Oproti původní verzi Selenia (viz kapitola 5.1.1) se poměrně razantně změnila architektura celé aplikace.

### 5.2.1 Architektura WebDriveru

Architektura WebDriveru je znázorněna na obrázku 5.2.




Obrázek 5.2: Znázornění architektury WebDriveru

Na obrázku 5.2 můžeme vidět, že se architektura skládá ze tří základních částí.

#### 5.2.1.1 Language bindings

Language bindings představují klientské knihovny, pomocí kterých lze psát příkazy, jež se mají v testu provést.

Tyto knihovny jsou psány v různých jazycích. V současné době je jich pět, které oficiálně udržuje přímo tým kolem projektu Selenium. Jsou to jazyky

Java, C#, Ruby, Python a JavaScript (Node). Klientských knihoven udržovaných pomocích třetích stran existuje ještě mnohem víc [9].

#### 5.2.1.2 WebDriver API

WebDriver API představuje prostředníka mezi Language bindings a Drivers.

Pokud chce testovací skript provést nějaký příkaz, musí být napsán za pomoci libovolného jazyka, který implementuje Language bindings. Při vykonávání příkazu se dostane příkaz do WebDriver API. Zde je přijatý příkaz interpretován pro zvolený Driver.

#### 5.2.1.3 Drivers

Drivers (ovladače) jsou určeny k ovládání prohlížeče. Každý Driver vždy implementuje pouze jeden prohlížeč.

Fungování by se dalo přirovnat k rozhraní používaného v objektovém programování. WebDriver rozhraní definuje metody, které musí každý ovladač implementovat. Každý ovladač pak implementuje danou metodu specificky pro daný prohlížeč, což je mnohem čistší způsob, než mít jednu JavaScriptovou implementaci pro všechny prohlížeče, jako tomu bylo ve starší verzi Selenia.

### 5.2.2 Výhody WebDriveru oproti Seleniu RC

- Podpora Driverů ze strany tvůrců prohlížečů. Například Mozilla, Opera a Google se aktivně podílejí na rozvoji Driverů pro své prohlížeče [10].
- Se zavedením Driverů se také zlepšila samotná emulace uživatelské interakce. WebDrivery využívají nativní události pro interakci s prohlížečem [10].
- Jednodušší a kompaktnější API, které je více objektově orientované [10].

Kapitola 6

# Appium

Appium je open source projekt od společnosti Sauce Labs. Tento projekt řeší problematiku multiplatformní automatizace nativních i hybridních aplikací.

Celý projekt je založen na open source projektu pro testování webových stránek, který je známý jako Selenium WebDriver (viz kapitola 5).

## 6.1 Filozofie Appia

Obecně platí, že filozofie do technické literatury nepatří. Nicméně v tomto případě bych rád udělal výjimku. Filozofii Appia je dáván poměrně velký prostor, ať již na webu projektu nebo i při různých prezentacích ohledně Appia.

Samotné Appium bylo na této filozofii založeno a řídí se jí. Z tohoto důvodu bych zde rád filozofii tohoto projektu v krátkosti představil. Pomocí následujících pravidel lze lépe pochopit, proč byl tento testovací framework vytvořen a proč funguje tak, jak funguje.

### 6.1.1 Testujte vždy stejnou aplikaci, kterou budete později publikovat

Hlavní smysl tohoto pravidla je vcelku praktický. Pokud testujeme jinou aplikaci, než kterou budeme později publikovat, nikdy si nemůžeme být jistí, zdali jsou naprosto totožné. Může se tedy stát, že jedna z nich bude mít bug, který ta druhá mít nemusí.

### 6.1.2 Pište testy v jakém jazyce chcete a využívejte k tomu libovolný framework

Appium razí myšlenku, že není důvod, proč by programovací jazyk aplikace měl být totožný s programovacím jazykem pro psaní testů.

Jestliže tedy píši aplikaci například ve Swiftu, ale testy chci psát v Ruby, protože například interpretovaný jazyk dává větší smysl v kombinaci s Jenkinsem, neměl by to být problém.

### 6.1.3 Použijte standardní specifikaci automatizace a API

Tímto pravidlem odkazuje Appium na znovu vynalézání kola. Z tohoto důvodu je Appium založeno právě na Selenium WebDriveru (viz kapitola 5.2) a není psáno celé od začátku.

### 6.1.4 Vybudujte velkou open source komunitu kolem projektu

Open source komunita pomáhá projektu být stále lepší.

## 6.2 Obecný princip fungování

Appium je webový server, který vytváří a zpracovává WebDriver sessions. Jelikož je založen na Selenium WebDriveru, základní princip fungování je v podstatě stejný. To znamená, že Appium přijímá požadavky od klientských knihoven za použití formátu JSON. Tyto požadavky jsou přijímány pomocí HTTP protokolu. Každý požadavek je poté zpracován v závislosti na platformě, kterou momentálně testujeme.

Obecně pro iOS a Android platí, že Appium nastartuje test na simulátoru nebo zařízení tak, že nějakým způsobem spustí server a čeká na požadavky z hlavního procesu Appia.

## 6.3 Appium na iOS

Jelikož Appium funguje jako obecný nástroj pro automatizaci, umožňuje automatizovat různé platformy.

Principiálně funguje na všech platformách v zásadě stejně. Z pohledu uživatele píšícího test je v podstatě jedno, na kterou platformu píší testy, protože testovací API je totožné. Pokud tedy píší testy pro iOS i Android, stačí napsat pouze jeden test, který funguje na obou platformách. Drobný rozdíl je v inicializaci testů.

Je samozřejmě jasné, že i když je testovací API stejné pro více platforem, samotná implementace Appia a princip fungování se liší podle dané platformy. V sekci 6.4 je popsán princip implementace Appia na iOS.

## 6.4 Implementace Appia na iOS

Fungování Appia na iOS zajišťují dvě věci. První je UI Automation (viz kapitola 3) a druhou jsou Instruments (viz kapitola 6.4.2). Appium funguje jako proxy pro příkazy do UI Automation, který beží v Instruments.

### 6.4.1 UI Automation

Popsáno v kapitole 3.

### 6.4.2 Instruments

Instruments je aplikace, kterou poskytuje Apple. Tato aplikace obsahuje spoustu komponent, které umožňují například profilování, hledání memory leaků atd.

Jedna z komponent se jmenuje **Automation**. Tato komponenta umožňuje za pomoci JavaScriptu přistupovat k API nazývanému UI Automation (viz kapitola 3). Toto API tedy umožňuje interagovat s aplikací, jako například kliknout na tlačítko v aplikaci atd.

## 6.5 Princip fungování Appia na iOS

Appium na iOS stojí na třech základních stavebních kamenech.

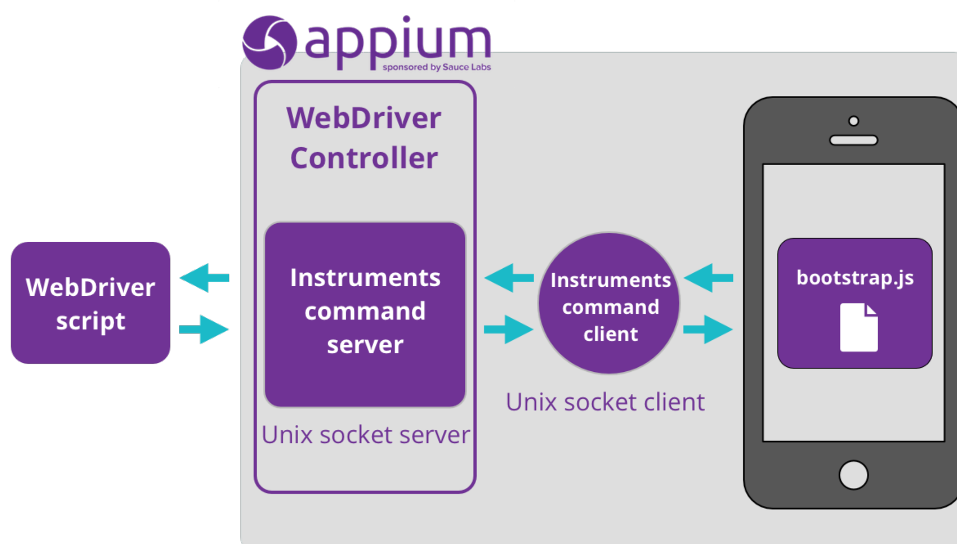
1. Instruments lze zapnout z příkazové řádky.
2. Do Instruments lze předat JavaScriptový soubor, který se má vykonat.
3. Apple umožňuje v UI Automation zavolat shell a spustit libovolný příkaz.

S využitím zmíněných stavebních kamenů je potom celý princip znázorněn na obrázku 6.1.

Jak již bylo zmíněno výše, do Instruments lze předat JavaScriptový soubor. Appium tedy vzalo JavasScriptový soubor, do kterého přidalo nekonečnou smyčku. Při každém průchodu smyčkou se zavolá příkazová řádka.

Dalším stavebních kamenem je možnost v UI Automation zavolat shell a spustit libovolný příkaz. Appium tohoto využívá k tomu, aby nastartovala UNIX socket klienta (na obrázku 6.1 označen jako „Instruments command client“). Tento klient se vzbudí a hledá UNIX socket server (na obrázku 6.1 označen jako „Instruments command server“), který je součástí hlavního procesu Appia.

Pokud má server nějaký příkaz k vykonání, předá jej klientovi a klient zapíše příkaz na standardní výstup. Tento příkaz si poté vezme bootstrap.js (viz obrázek 6.1) a vykoná ho v kontextu Instruments a tudíž i aplikace.




Obrázek 6.1: Znázornění principu Appia na iOS

## 6.6 Appium - životní cyklus

V předchozí kapitole bylo rozebráno, jak Appium funguje na iOS. V této části bych se rád zaměřil na životní cyklus testu. Tedy jak se z příkazu z uživatelského testovacího skriptu dostane do aplikace a jak se výsledek provedeného příkazu dostane zpět do uživatelského testovacího skriptu.

Životní cyklus bude rozdělen do dvou částí. První z nich se zaměří na provedení příkazu z testovacího skriptu v aplikaci (viz kapitola 6.6.1). Druhá část se zaměří na postup opačný, tedy získání výsledku provedeného příkazu zpět do skriptu, aby bylo možné vyhodnotit výsledek příkazu (viz kapitola 6.6.2).

Pokud v následující části nebude řečeno jinak, pod pojmem server je myšlen **Instruments command server** a pod pojmem klient je myšlen **Instruments command client** (viz obrázek 6.1).

### 6.6.1 Předání příkazu z testovacího skriptu do aplikace

Zde popisovaný postup je znázorněn na obrázku 6.1. V této části se budeme pohybovat ve směru šipek doprava.

#### 6.6.1.1 WebDriver

Veškeré testování v Appium vždy začíná ve skriptu WebDriveru. Toto je skript, do kterého se píše posloupnost příkazů, které se mají v aplikaci provést. Tento skript také obsahuje případné validace jednotlivých kroků.

Při spuštění testovacího skriptu se vytvoří HTTP požadavek na Appium server. V době kdy se požadavek posílá, Appium již ví, na jaké platformě bude test probíhat (v tomto případě na iOS) a pošle požadavek na **Instruments command server**.

#### 6.6.1.2 Instruments command server

Po tom, co server obdrží příkaz z WebDriver skriptu, bude čekat, dokud se neprobudí **Instruments command client** a nepožádá o další příkazy. Jakmile klient požádá o příkazy, server mu je předá.

#### 6.6.1.3 Instruments command client

Klient se v pravidelných intervalech ptá serveru (provádí pull), jestli pro něho nemá nové příkazy k provedení. Pokud zrovna neprovádí pull, spí.

Po obdržení nového příkazu jej okamžitě předá do **bootstrap.js**.

#### 6.6.1.4 Bootstrap.js

Bootstrap.js po obdržení příkazu od klienta daný příkaz za pomoci UI Automation provede.

### 6.6.2 Získání výsledku z provedeného příkazu aplikace do testovacího skriptu

Získání výsledku z provedeného příkazu je na obrázku 6.1 znázorněno pomocí šipek směřujících vlevo.

#### 6.6.2.1 Bootstrap.js

Po dokončení požadovaného příkazu vrací bootstrap.js výsledek provedené operace do **Instruments command client**.

#### 6.6.2.2 Instruments command client

Jakmile je klient znovu probuzen a zjistí, že má výsledek příkazu od bootstrap.js, oznámí **Instruments command serveru**, že pro něj má výsledek posledního příkazu.

#### 6.6.2.3 Instruments command server

Server po zjištění, že je k dispozici výsledek operace, tento výsledek vezme. Také zkontroluje, zdali má nějaký příkaz k vykonání, který by měl předat klientovi. Pokud ano, tento příkaz mu zároveň předá.

#### 6.6.2.4 WebDriver

Server po obdržení výsledku příkazu předá tento výsledek zpět do WebDriver skriptu, který originální příkaz vytvořil. Ve skriptu je možné s výsledkem pracovat. Můžeme použít různá porovnání pro ověření, že příkaz skončil žádoucím a očekávaným stavem.

## 6.7 Psaní testů pro Appium

### 6.7.1 Programovací jazyk

Jedna z velkých předností Appia je možnost psát testy v různých programovacích jazycích, jakými jsou například Java, C#, Ruby, Python atd. To je možné díky tomu, že Appium je v jádru jednoduché REST API [11, s. 60].

Jak bylo zmíněno v úvodu této kapitoly, Appium je založeno na Selenium WebDriveru (viz kapitola 5). To znamená, že ke svému fungování využívá WebDriver. Appium rozšiřuje klientské knihovny a přidává extra příkazy pro práci s mobilními zařízeními [11, s. 60]. Při práci s klientskými knihovnami je tedy potřeba použít knihovnu, která je psána pro Appium a nikoliv přímo pro Selenium knihovnu, protože by chyběly příkazy přidané Appiem.

### 6.7.2 Práce s Appiem

S Appiem lze pracovat dvěma způsoby a to buď s použitím příkazové řádky, nebo grafického uživatelského rozhraní.

Před tím, než můžeme začít cokoliv testovat za použití Appia, musíme nejprve nastartovat Appium server a teprve až potom můžeme spouštět jednotlivé testy. Tento krok se liší v závislosti na zvoleném způsobu práce.

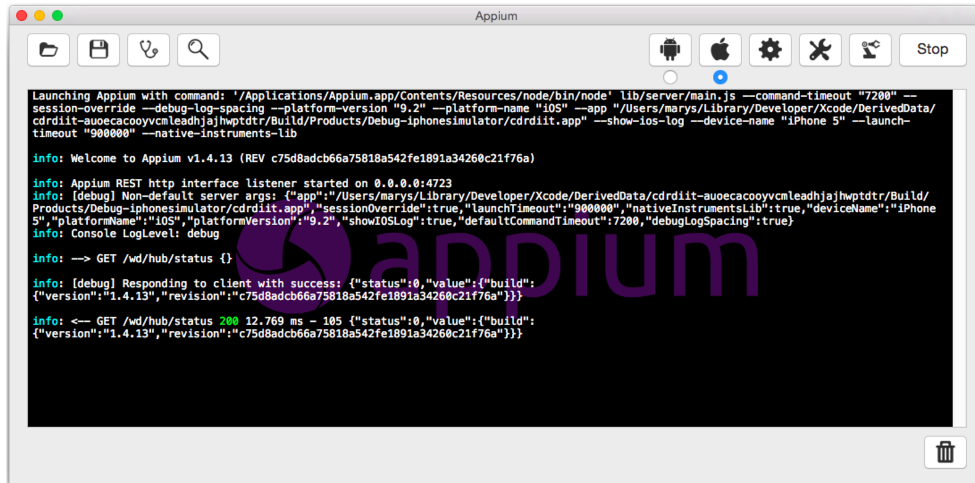
#### 6.7.2.1 Příkazová řádka

Jestliže chceme Appium používat z příkazové řádky, musíme nejprve spustit server, který následně zobrazí informace, jaká verze Appia je spuštěna a na jaké adrese a portu poslouchá.

Po spuštění serveru již lze spustit příslušný test. Spuštění je opět velmi jednoduché a odvíjí se podle použitého programovacího jazyka. Testovací skript se spouští stejně jako jakýkoliv jiný skript v daném programovacím jazyce.

#### 6.7.2.2 GUI

Další možností, jak pracovat s Appiem, je pomocí aplikace **appium.app**, která k Appiu přidává grafické rozhraní. Rozhraní je zobrazeno na obrázku 6.2.




Obrázek 6.2: Ukázka appium.app

Po zapnutí aplikace lze zvolit, kterou platformu chci testovat. Aplikace také umožňuje zkontrolovat, zdali jsou splněny všechny požadavky pro používání Appia, jako je například přítomnost všech potřebných aplikací atd.

Aplikaci **appium.app** je také možné využít jako vyhledávač elementů, se kterými chceme pracovat (více viz kapitola 6.8).

### 6.7.3 Spuštění aplikace v Appiu

Před začátkem psaní testů v Appiu je nutno Appiu říct, co budeme chtít automatizovat. Úvodní nastavení není složité, ale je potřeba ho provést korektně.

Informace nezbytné pro spuštění testů jsou:

- **jméno zařízení (simulátor vs. zařízení)**,
- **cesta k testované aplikaci**.

Vhodné je ještě uvést:

- testovací platformu,
- verzi testovací platformy.

Ve zdrojovém kódu 6.1 je ukázka inicializace testu napsaná v Pythonu.

Zdrojový kód 6.1: Ukázka inicializace testu psaného pro Appium v Pythonu

```
desired_caps = {}
desired_caps['platformName'] = 'iOS'
desired_caps['platformVersion'] = '9.2'
desired_caps['deviceName'] = 'iPhone Simulator'
desired_caps['app'] = '/path/to/my/app/myApp.app'

url = 'http://0.0.0.0:4723/wd/hub'

self.driver = webdriver.Remote(url, desired_caps)
```

### 6.7.4 Ukázka práce s Appium v Pythonu

Každý testovací skript musí začínat úvodním nastavením (viz zdrojový kód 6.1). Po této inicializaci je možné psát samotný test. Ve zdrojovém kódu 6.2 je ukázka vyhledání elementu a následné kliknutí na nalezený element.

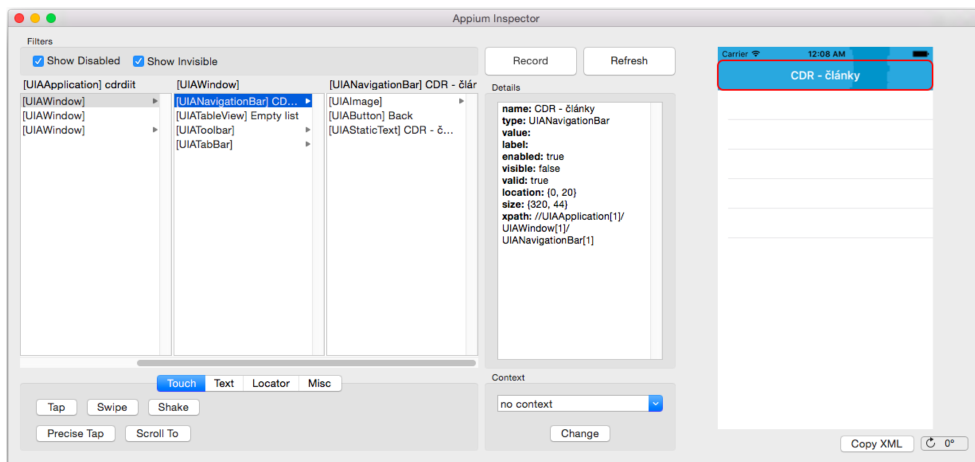
Zdrojový kód 6.2: Ukázka inicializace testu psaného pro Appium v Pythonu

```
window = "//UIAApplication[1]/UIAWindow[1]"
myBtnXPath = window + "/UIATabBar[1]/UIAButton[2]"

# find element
myBtn = self.driver.find_element_by_xpath(myBtnXPath)

# init touch action
action = TouchAction(self.driver)

# show diit articles listing
action.tap(myBtn).perform()
```

## 6.8 Appium Inspector

Součástí aplikace appium.app (viz kapitola 6.7.2.2) je Appium Inspector. Po spuštení vypadá Inspector podobně jako na obrázku 6.3.

Z obrázku 6.3 je poznat, že Inspector se skládá z informací o struktuře elementů, detailních informacích o aktuálně zvoleném elementu a obrazovce aktuálně načtené aplikace. V aplikaci lze také provést základní událost, jako je například kliknutí na element.




Obrázek 6.3: Ukázka Appium Inspectoru

Jak je vidět z obrázku 6.3, obrazovka aplikace není zobrazena korektně. Testovaná aplikace by měla vypadat podobně jako na obrázku 2.1. Na obrazovce úplně chybí spodní tab bar, přestože ve struktuře elementů tyto elementy jsou. Je tedy celkem dobrý zvyk se při vyhledávání elementů nespoléhat pouze na zobrazení obrazovky, ale spíše na zobrazenou strukturu elementů.

### 6.8.1 Record

Record (nahrávání) funguje podobně jako UI Recording při testování v Xcode (viz kapitola 4.7).

Po zapnutí nahrávání se zobrazí aktuální zdrojový kód testu. Při každé další akci v Inspektoru se automaticky generuje na pozadí zdrojový kód provedených akcí. Po skončení je možné vygenerovaný kód uložit do souboru. Takto vygenerovaný soubor je kompletní text, jenž je možné později spustit a nebo jej případně upravit podle potřeby či přidat asserty.

Zdrojový kód vygenerovaný pomocí nahrávání je pochopitelně možné ještě před uložením změnit na požadovaný jazyk (Java, Python atd.). Po změně jazyka se celý nahraný test ihned konvertuje do vybraného programovacího jazyka.

## 6.9 Hybridní aplikace

Appium nativně podporuje automatizaci hybridních aplikací. Jediné, co je zapotřebí, je nastavit příslušné WebView, ve kterém aplikace běží jako moje Window. Appium to vezme jako signál, že máme v plánu automatizovat zvo-

lené WebView. Od té doby píšeme v podstatě klasické webové Selenium testy. Všechny následující příkazy jsou vykonávány v kontextu s WebView.

Kapitola 7

# Calabash

Dalším z nástrojů pro automatizaci je Calabash. Tento nástroj se odlišuje poměrně markantně od nástrojů jako je UI Testing in Xcode nebo UI Automation. Hlavní odlišností tohoto nástroje je způsob psaní jednotlivých testů. Testy napsané pro Calabash jsou na první pohled v podstatě jednoduché věty v angličtině, které se dají spustit. Calabash k tomu používá nástroj zvaný Cucumber (viz kapitola 7.1).

Calabash je open source projekt (Eclipse Public License), který je založen na Cucumberu (viz kapitola 7.1). Testy fungují jak v simulátoru, tak i na fyzickém zařízení. Standardně je podporováno testování nativních i hybridních aplikací. Platformy podporované Calabashem jsou iOS a Android.

Jedna z výhod tohoto projektu je možnost, v případě potřeby, zakoupit komerční podporu, případně využít testovací cloud, jenž otestuje aplikaci na uživatelem zvolených fyzických zařízení.

## 7.1 Cucumber

Cucumber je nástroj běžící v příkazové řádce. Po spuštění si načte specifikace testů ze souborů. Těmto specifikacím se říká **features**. Po načtení vyhodnotí jednotlivé **scénáře** (scenarios). Každý scénář se skládá z jednotlivých **kroků** (steps). Tyto kroky musí Cucumber všechny postupně vyhodnotit.

Vzhledem k tomu, že jednotlivé kroky se na první pohled skládají z jednoduchých anglických vět, je nezbytné dodržovat nějaká základní syntaktická pravidla, aby vše fungovalo, jak má. Samotná pravidla jsou definována v jazyce nazývaném **Gherkin**.

Pro správnou funkčnost testů je také nutné implementovat definici jednotlivých kroků. Tyto definice jsou opravdové implementace toho, co se má provést. Následně se tyto implementace mapují na business pravidla definovaná v rámci kroků v testovacím scénáři [12, s. 7].

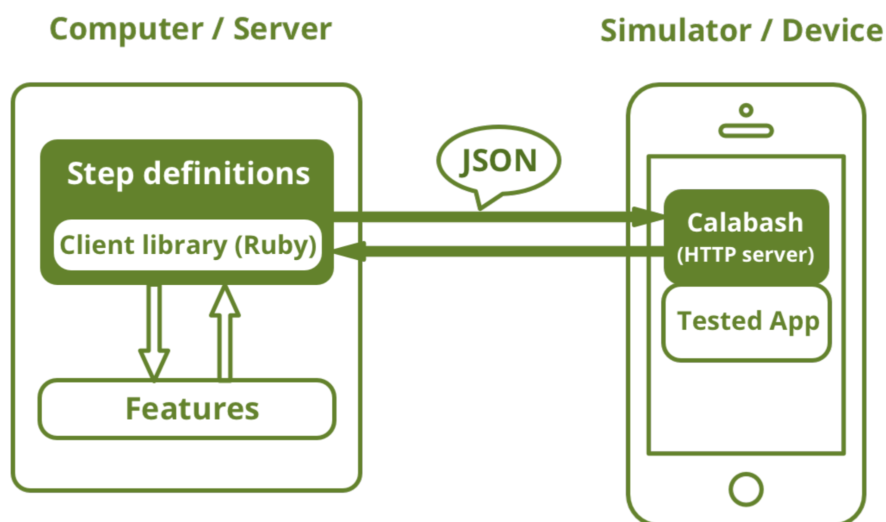
Ukázka syntaxe Cucumberu je ve zdrojovém kódu 7.1.

Zdrojový kód 7.1: Ukázka Cucumber syntaxe

```
Feature: MyFeature
  My comment for this feature.

  Scenario: My Scenario
    Given I am somewhere
    When I do something
    Then I should see something
```

## 7.2 Architektura Calabashe

Princip fungování Calabashe je znázorněn na obrázku 7.1. Z obrázku je evidentní, že se Calabash skládá ze tří základních stavebních kamenů. To jest z definice kroků, features a HTTP serveru.




Obrázek 7.1: Znázornění architektury Calabashe

### 7.2.1 Features

Jedná se o soubory, které v sobě obsahují jednotlivé definice textů v Cucumberu (viz kapitola 7.1) na business úrovni.

### 7.2.2 Definice kroků

Definice kroků obsahují jádro veškerého testování, jelikož obsahují kód, který provádí jednotlivé testy definované na business úrovni. Provádí se zde mapování z business pravidel na definice testů.

Druhy a příklady definice kroků jsou rozebrány v kapitole 7.5.1.

### 7.2.3 HTTP server

HTTP server slouží ke komunikaci mezi testy a aplikací. Aby tato komunikace byla možná, je nezbytné integrovat Calabash s Xcode. Do Xcode 6 to byla záležitost jednoho příkazu v příkazové řádce. Od Xcode 6 je celý proces poměrně zkomplikován (více viz kapitola 7.3).

Principiálně to funguje tak, že Calabash přidá k aplikaci (ne do aplikace) vlastní HTTP server, pomocí které poté komunikuje s aplikací. Aby toto bylo možné, je zapotřebí do Xcode přilinkovat Calabash framework, který tuto logiku řeší. Klientská knihovna (Client library viz obrázek 7.1) poté s tímto HTTP serverem komunikuje [13].

## 7.3 Integrace s Xcode

Integrace Calabashe se od Xcode 6 relativně zkomplikovala. To, co dříve řešil jeden příkaz v příkazové řádce, je teď nutné řešit mnohem složitějším nastavováním.

V současné době existují tři způsoby, jak přidat Calabash do Xcode. Všechny jsou naštěstí velice pěkně zdokumentované a popisují celý postup krok po kroku, včetně screenshotů.

### 7.3.1 Přilinkování Calabashe pro Debug konfiguraci

Přilinkování Calabashe pro Debug konfiguraci je pravděpodobně nejjednodušší a nejrychlejší způsob, jak rozchodit Calabash v Xcode. Stačí pouze v nastavení projektu v Xcode upravit nastavení a vše obvykle funguje na první pokus.

Hlavní nevýhodou tohoto přístupu je nutnost použití síťové frameworku **CFNetworku**, i když ho v aplikaci nepoužíváme. Další nevýhoda je v tom, že při každém spuštění aplikace se zároveň nastartuje i HTTP server [14]. Tuto nevýhodu lze aspoň částečně kompenzovat tím, že je možné v Xcode vytvořit nový target a jen tento target bude linkovat Calabash, takže pouze kompilace v tomto testovacím targetu bude startovat HTTP server. Nicméně i toto je další práce navíc.

U této metody přidání Calabashe do Xcode je také potřeba dát pozor, aby se Calabash nedostal do produkční verze aplikace, která se bude později nahrávat do App Storu, protože by aplikace byla zamítnuta. Calabash používá symboly a metody, které neumožňují přijetí do App Storu [14].

### 7.3.2 Přilinkování Calabashe v nastavení

Tento způsob přidání Calabashe je kompromis mezi ostatními dvěma možnostmi (viz 7.3.1 a 7.3.3). Nastavení je trochu složitější než 7.3.1, ale po následování oficiálního tutoriálu to není zase takový problém.

Hlavní výhoda tohoto přístupu je v možnosti použití preprocesorového makra, které se aplikuje pouze při kompilaci s Calabashem [15].

Já osobně jsem měl s tímto způsobem přidání Calabashe do Xcode velké problémy. Samotné přidání Calabashe do Xcode sice proběhlo v pořádku. Nicméně problémy nastaly s frameworky třetích stran, které byly nainstalovány přes CocoaPods (viz kapitola 12.2.1). Po kompilaci s Calabashem naráz Xcode naprosto ignoroval ostatní frameworky a zobrazovala se pouze chyba, že dané knihovny Xcode nezná.

### 7.3.3 Přilinkování Calabashe pomocí cal Targetu

Při použití Calabashe touto metodou dojde k vytvoření nového targetu, který se jmenuje **název_targetu-cal**. Tento target je vytvořen duplikací stávajícího targetu. Oba targety jsou v zásadě totožné, akorát nový target navíc linkuje Calabas framework [16]. Díky tomu se bude Calabash HTTP server zapínat pouze při spuštění nového targetu, takže původní target zůstane nedotčený.

Oproti předchozím způsobům přidání Calabashe do Xcode (viz 7.3.1 a 7.3.2) je tento způsob jednoznačně pracnější a komplikovanější.

## 7.4 Nastavení prostředí pro práci s testy

Po úspěšné integraci Calabashe s Xcode (viz kapitola 7.3) už nic nebrání tvorbě prvních testů. Nejjednodušší a také doporučovaný způsob pro tvorbu a běh testovacích skriptů je použití Calabash sandboxu.

### 7.4.1 Calabash sandbox

Calabash sandbox je předem nakonfigurované prostředí, které je nastaveno pro potřeby fungování Calabashe. Díky použití tohoto sandboxu odpadá problém s řešením verze Ruby na testovacím stroji a další podobné problémy.

Samotná instalace sandboxu je velice jednoduchá, jedná se o jeden řádek v příkazové řádce, díky kterému odpadá spousta problémů. Po instalaci tohoto sandboxu stačí tento sandbox zapnout a poté již můžeme začít tvořit a spouštět testy.

### 7.4.2 Struktura testů

Pokud chceme začít psát testy, je ideální začít tím, že necháme Calabash, ať vygeneruje základní strukturu testů. Součástí takto vygenerované struktury

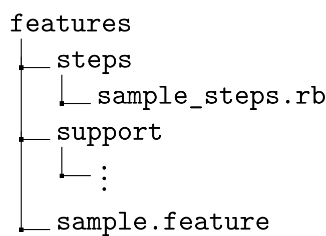
je také příklad jednoduchého testu, který může sloužit jako startovací bod pro další testy.

Vygenerovaná struktura je znázorněna na obrázku 7.2. Jak vidno, vygenerovaný adresář se jmenuje **features**. Tento adresář by měl obsahovat veškeré testy. Jednotlivé scénáře (viz kapitola 7.1) přímo v adresáři **features**. Vždy mají název **název.feature**.

Další adresář z vygenerované struktury, který nás z pohledu psaní testů zajímá, je adresář **steps**. V tomto adresáři jsou všechny soubory, které obsahují implementace k jednotlivých scénářům (viz kapitola 7.5.1).

V podadresáři **support** jsou podpůrné soubory, které z hlediska psaní testů nejsou až tak zajímavé.

Z vygenerovaného příkladu je vidět, že by se měla dodržovat konvence, kdy by soubor se scénáři (**sample**.feature) měl mít stejný základ jména se souborem, který obsahuje implementaci (**sample_**steps.rb).




Obrázek 7.2: Znázornění vygenerovaného základu pro Calabash testy

### 7.4.3 Spuštění testů

Po instalaci a zapnutí Calabash sandboxu (viz kapitola 7.4.1) můžeme konečně začít pracovat.

Pro spuštění nadefinovaných testů v Calabashi je potřebné se v příkazové řádce přesunout do adresáře, kde je uložený Xcode projekt, který chceme testovat. Jakmile jsme v adresáři s testovaným projektem, stačí v příkazové řádce test spustit (viz zdrojový kód 7.2). Ve zdrojovém kódu 7.2 je ukázáno, jak test spustit. Také je vidět jakým způsobem test spustit na konkrétním zařízení a jak zjistit všechna dostupná zařízení pro testování.

## 7.5 Tvorba testů

Jak bylo zmíněno v úvodu této kapitoly, tvorba testů se skládá ze dvou částí. První z nich je definice testovacích scénářů na úrovni businessu (viz kapitola 7.1). Druhá část se stará o implementaci business pravidel v kódu (viz kapitola 7.5.1).

Zdrojový kód 7.2: Ukázky možnosti spuštění Calabash testů

```
# run all test
cucumber

# run all test on iPhone 5 with iOS 9.2
DEVICE_TARGET="iPhone 5 (9.2)" cucumber

# list all available devices for testing
xcrun instruments -s devices
```

### 7.5.1 Implementace kroků scénářů

Kroky (steps) ke scénářům jsou implementovány v Ruby. Samotné mapování kroků na jejich implementaci je řešeno pomocí regulárních výrazů. Ukázka syntaxe implementace kroků k ukázkovému scénáři (viz zdrojový kód 7.1) je znázorněna ve zdrojovém kódu 7.3.

Zdrojový kód 7.3: Ukázky mapování implementace kroků

```
Given(/^I am somewhere$/) do
  # here goes what to really do
end

When(/^I do something$/) do
  # here goes what to really do
end

Then(/^I should see something$/) do
  # here goes what to really do
end
```

### 7.5.2 Předdefinované kroky

Calabash již v základu přichází s velkým množstvím již definovaných kroků. Tyto kroky lze používat i ve svých testech. Při používání těchto kroků je velice důležité mít na paměti, že tyto kroky by se měly používat jen a pouze v případě, kdy to v rámci business pravidel dává smysl.

Předdefinované kroky dávají smysl tam, kde se daný krok přímo trefí do business pravidla (kroku scénáře). Jestliže máme tedy pocit, že pravidlo sice dělá to, co potřebujeme, ale to, jakým způsobem je napsáno, smysl nedává,

je jednoznačně lepší, a dokonce doporučeno, vytvořit si pravidlo nové, aby se business pravidla nemusela přizpůsobovat a dávala smysl.

Příklad předdefinovaných kroků je ve zdrojovém kódu 7.4. Kompletní přehled definovaných kroků je v dokumentaci [17].

Zdrojový kód 7.4: Příklad předdefinovaných kroků

```
# tap ok button
Then I touch ok

# scroll down
Then I scroll down

# check if on current screen is "expected text"
Then I see the text "expected text"
```

### 7.5.3 Kroky na míru

Pokud nestačí předdefinované kroky (což je většina případů), je možné si definovat vlastní kroky. Tyto kroky vypadají tak, jak je znázorněno ve zdrojovém kódu 7.3.

Jednotlivá pravidla se tedy skládají z regulárního výrazu a bloku. Regulární výraz slouží k mapování kroku a jeho implementaci. Blok kódu se píše v Ruby a slouží k implementaci daného kroku.

## 7.6 Ukázka implementace testu

Veškerá základní potřebná teorie již byla v rámci této kapitoly rozebrána, můžeme se tedy podívat na konkrétní příklad testu. Příklad bude realizován na testovací aplikaci (viz kapitola 2.2).

### 7.6.1 Tvorba scénáře

Testovací scénář je jednoduchý, v testovací aplikaci (viz kapitola 2.2) si zobrazíme seznam všech článků. Poté chceme kliknout na první článek, který by se měl otevřít. Po otevření detailu článku máme v úmyslu vrátit se zpět na seznam článků. Odtud se přepneme na články z jiného serveru a uděláme to samé. Popsaný scénář je znázorněn ve zdrojovém kódu 7.5.

Po přečtení zdrojového kódu 7.5 je dle mého názoru každému hned jasné, co tento scénář má testovat. Vzhledem k tomu, že se scénář skládá v podstatě z anglických vět, měl by tento scénář být zřejmý i lidem, kteří nejsou úplně technicky zdatní, což velice usnadňuje domluvu ohledně testování například se zákazníkem.

Zdrojový kód 7.5: Ukázky testovacího scénáře pro zobrazení detailu článku

```
Feature: Articles
  Test works with articles.

  Scenario: Show first article
    Given I am on Articles listing with
          title "CDR - články"
    When I go to the first article
    Then I should see chosen article's detail
    And I go back

    Then I switch to next server

    Given I am on Articles listing with
          title "DIIT - články"
    When I go to the first article
    Then I should see chosen article's detail
```

### 7.6.2 Spuštění testu bez implementace kroků

Calabash se snaží tvorbu testů co nejvíce zjednodušit, a proto když spustíme scénář bez implementovaných kroků, Calabash vygeneruje kostru implementace těchto kroků. Tento vygenerovaný kód potom stačí zkopírovat do příslušného souboru v podadresáři **steps** (viz obrázek 7.2).

### 7.6.3 Implementace kroků scénáře

K implementaci kroků testovacího scénáře (viz 7.6.1) je nejjednodušší použít kostru vygenerovanou při spuštění testovacího scénáře bez implementovaných kroků.

Při implementaci jednotlivých kroků je dobré vědět, jak jsou vlastně jednotlivé kroky vyhodnocovány. Princip vyhodnocování je velice prostý. Pokud krok nevyhodí výjimku, je automaticky bráno, že daný test prošel. V opačném případě test skončí s chybou.

Implementace testovacího scénáře (viz 7.6.1) je ve zdrojovém kódu 7.6.

## 7.7 Interaktivní mód

Obecně při psaní testů platí, že je nezbytné průběžně zkoušet, zdali test opravdu dělá to, co potřebujeme. Takový postup může časem přinášet spoustu zdržení, jelikož s časem obvykle roste počet testů a s tím i doba běhu testů. Z tohoto důvodu zavedl Calabash interaktivní mód.

Zdrojový kód 7.6: Ukázky implementace kroků scénáře

```
Given(/^I am on Articles listing with title "(.*?)"$/)
    do |listing_title|
  # store title of first article
  $textOfArticle = query(
    "tableViewCell index:0 label", :text
  ).at(1)
end

When(/^I go to the first article$/) do
  # tap first row
  touch "tableViewCell index:0"
end

Then(/^I should see chosen article's detail$/) do
  # wait for element to be properly loaded
  wait_for_elements_exist(["label index:2"])

  # check if element with given text exists
  element_exists("label text:'#{$textOfArticle}'")
end

Then(/^I switch to next server$/) do
  # tap second tab bar button
  touch "tabBarButton index:1"
end
```

Interaktivní mód umožňuje dotazování a posílání příkazů na testované zařízení. Takže odpadá nutnost po každé úpravě testu spuštění všech testů. Díky interaktivnímu módu lze požadovaný dotaz či příkaz odladit tak, aby dělal přesně to, co vyžadujeme a poté jej lze jednoduše zkopírovat do testovacího skriptu.

Pro zapnutí interaktivního módu stačí v sandboxu spustit Calabash konzoli.

### 7.7.1 Vyhledávání elementů

Pro vyhledávání slouží v Calabashi příkaz **query**. Jedná se o obecný příkaz, který umí vyhledat cokoliv. K vyhledávání elementů se využívá accessibility identifikátoru (viz kapitola 4.2). Příklady vyhledávání jsou ve zdrojovém kódu 7.7.

Zdrojový kód 7.7: Ukázka vyhledávání elementů v Calabashi

```
# find all cells in table
query("tableViewCells")

# find first cell in table
query("tableViewCells index:0")

# find button with accessibility ID "XXX"
query("button marked:'XXX'")
```

### 7.7.2 Nahrávání

Calabash umožňuje nahrávání uživatelských akcí jako většina nástrojů. Calabash to ovšem pojal trochu odlišným stylem než například UI Automation (viz kapitola 3.8) nebo UI Testing in Xcode (viz kapitola 4.7).

Nahrávání v Calabashi totiž negeneruje uživatelské akce, jak je tomu u většiny nástrojů. Respektive Calabash si interně ukládá uživatelské akce a poté je schopen tuto sekvenci akcí zopakovat. Uživatel může spustit celou nahranou sekvenci, ale nevidí přímo vygenerovaný kód jednotlivých akcí v rámci sekvence.

Každá nahraná sekvence má své jméno. Pomocí tohoto jména můžeme nahranou sekvenci akcí zopakovat.

### 7.7.3 Interpolace nahrané akce

Pokud máme nahranou nějakou akci, Calabash umožňuje tuto akci spustit znovu. To ale není vše, co Calabash dokáže. Velice zajímavou možností, kterou Calabash poskytuje, je takzvaná interpolace nahrané akce.

Dejme tomu, že máme nahranou vlastní akci, která přesune element z místa $a$ do místa $b$. Tuto nahranou akci můžeme v případě potřeby přehrát znovu.

Často se ale tester dostane do situace, kdy by potřeboval provést v podstatě stejnou akci, jenom s mírně odlišnými parametry. Přesně k tomu účelu slouží interpolace. Zavoláme stejnou akci, akorát jí upravíme parametry, mezi kterými se má daná akce provést.

Jestliže nahraná akce například přesune element z konkrétního místa $a$ do konkrétního místa $b$, pomocí interpolace můžeme z této konkrétní akce udělat akci obecnou, kterou je možné aplikovat na přesun z různých míst do libovolných dalších míst.

Kapitola **8**

# Další testovací nástroje

## 8.1 Frank

Frank je testovací nástroj, který funguje velice podobně jako Calabash (viz kapitola 7). Jedná se tedy také o nástroj, který funguje tak, že do testované aplikace vloží malý HTTP server, jenž je napsaný v Objective-C. Testovací skript poté posílá příkazy na tento server, čte aktuální stav aplikace a provádí různé akce [18].

### 8.1.1 Instalace

Samotná instalace a následné zprovoznění Franka je velice jednoduché. Frank se nainstaluje pomocí příkazu v příkazové řádce.

Po instalaci Franka je potřeba Franka přidat k projektu, který chceme testovat. Přidání Franka k projektu je opět realizováno přes příkazovou řádku. Po spuštění příkazu pro přidání Franka k projektu v adresáři s testovaným projektem se vygeneruje adresář s názvem **Frank**, který obsahuje vše potřebné pro testování pomocí Franka. Adresář také obsahuje podadresář **features**, který má stejný význam a použití jako v Calabashi (viz kapitola 7.4.2).

Pokud máme úspěšně vygenerovaný adresář, který Frank potřebuje ke své funkci, můžeme aplikaci zkompilovat s Frankem. Pokud kompilace proběhne v pořádku, stačí aplikaci pustit s Frankem (HTTP serverem) a testování už nic nebrání. Kompilace a spuštění s Frankem je vyobrazeno ve zdrojovém kódu 8.1.

### 8.1.2 Psaní testů

Vzhledem k tomu, že Frank je založen na Cucumberu (viz kapitola 7.1), funguje tvorba testů stejně jako v případě Calabashe (viz kapitola 7.5).

Podoba s Calabashem je opravdu veliká, takže zde například funguje i stejný interaktivní mód jako u Calabashe (viz kapitola 7.7).

Zdrojový kód 8.1: Ukázka kompilace a spuštění aplikace s Frankem

```
# build app with Frank
frank build

# launch app with Frank HTTP server
frank launch
```

### 8.1.3 CocoaPods

Při testování Franka jsem velice narazil, když jsem se Franka snažil přidat k aplikaci, která využívá CocoaPods (viz kapitola 12.2.1). Jestliže používáme CocoaPods a pokusíme se aplikaci zkompilovat pomocí příkazu znázorněného ve zdrojovém kódu 8.1, skončí kompilace chybou.

I po velmi dlouhém hledání a úpravě snad všech kompilačních parametrů, které Frank poskytuje, se mi bohužel aplikaci nepodařilo zkompilovat. Bohužel ani po přidání dotazu do oficiálního repozitáře Franka na GitHubu jsem nedostal žádnou nápovědu, jak tento problém vyřešit. Z mého pohledu je tedy tento nástroj nepoužitelný pro aplikace s CocoaPods.

## 8.2 Ios-driver

Ios-driver je testovací nástroj, který je založen na Selenium WebDriveru (viz kapitola 5.2). Umožňuje testování nativních, hybridních i webových aplikací. V současné době podporuje běh pouze v simulátoru a ne na fyzických zařízeních [19].

Vzhledem k tomu, že je aplikace postavená na Selenium WebDriveru, funguje principiálně v podstatě stejně jako Appium (viz kapitola 6). To znamená, že pro testování je nutné zapnout nejdříve server a poté je možné testovat aplikaci. Aplikaci není potřeba kompilovat, stačí mít k dispozici již zkompilovanou aplikaci.

### 8.2.1 Instalace

Instalace ios-driveru je velice jednoduchá. Stačí ze stránek ios-driver ([19]) stáhnout server napsaný v Javě. Při startu serveru je nezbytné zadat cestu k testované zkompilované aplikaci a také port, na kterém má server běžet. Ukázka spuštění serveru je ve zdrojovém kódu 8.2. Tato ukázka předpokládá, že se v příkazové řádce nacházíte v adresáři se staženým serverem.

Zdrojový kód 8.2: Ukázka spuštění serveru ios-driveru

```
java -jar ios-server-standalone-0.6.6-SNAPSHOT.jar \
-aut /path/to/tested/app.app -port 5555
```

### 8.2.2 Testovací skript

Po zapnutí serveru (viz 8.2.1) je potřeba vytvořit testovací skript. Vzhledem k tomu, že API ios-driveru je plně kompatibilní s tím WebDriverovým, psaní skriptů je v zásadě téměř totožné s Appiem.

Na začátku testovacího skriptu je vždy nutné tento skript správně inicializovat, aby mohl testovací skript navázat komunikaci se serverem. Ukázka inicializace testovacího skriptu nativní aplikace je znázorněna ve zdrojovém kódu 8.3. Ukázka inicializace je v Pythonu.

Zdrojový kód 8.3: Ukázka inicializace testovacího skriptu ios-driveru v Pythonu

```
desired_caps = {}
desired_caps['device'] = 'iphone'
desired_caps['CFBundleName'] = 'myapp'
desired_caps['variation'] = 'Regular'
desired_caps['language'] = 'en'
desired_caps['locale'] = 'en_GB'
desired_caps['simulator'] = 'true'

driver = webdriver.Remote(
  "http://localhost:5555/wd/hub",
  desired_caps
)
```

### 8.2.3 Funkčnost

Při testování ios-driveru jsem po spuštění serveru a správné inicializaci testovacího serveru narazil na problém při spouštění testovacího skriptu. Spuštění testu vyžaduje přítomnost nástroje nazvaného **instruments-without-delay**.

Standardně UI Automation (viz kapitola 3) má funkci, která spustí nějakou akci s definovaným časovým intervalem. Ve výchozím stavu vždy trvá jednu sekundu, než tato funkce odpovídá a to bez ohledu na to, jak rychle se příkaz provedl. Nástroj **instruments-without-delay** odstraňuje toto čekání, čímž

celé testování výrazně urychluje. Tento nástroj byl vytvořen Facebookem a v současné době již není dále vyvíjen [20].

Problém je to, že poslední commit u nástroje **instruments-without-delay** byl cca před rokem, takže chybí veškerá podpora pro poslední verzi Xcode. Vzhledem k tomu, že poslední podporovaná verze Xcode je verze 6, není tento nástroj funkční v poslední verzi Xcode. Spuštění testu v ios-driveru je tedy celkem problematické.

Kapitola **9**

# UI Testing in Xcode vs. Appium

Tato kapitola si klade za úkol porovnat psaní nativních UI testů a psaní testů v testovacím nástroji založeném na Seleniu (viz kapitola 5). Jako zástupce testovacího nástroje založeném na Seleniu byl zvolen nástroj Appium (viz kapitola 6).

V současné době existují dva oficiální nástroje pro testování UI ze strany Applu a to UI Automation (viz kapitola 3) a UI Testing in Xcode (viz kapitola 4). Jako nástroj pro nativní testování ze strany Applu byl zvolen UI Testing in Xcode. Volba tohoto nástroje byla hlavně z toho důvodu, že tento nástroj je budoucnost pro UI testování vzhledem k tomu, že UI Automation je momentálně označen jako zastaralý.

Samotné porovnání je psáno na základě testování dvou aplikací. Jedna aplikace je nativní a druhá hybridní. Ke každé aplikaci je napsán testovací scénář, ke kterému jsou také implementovány jednotlivé testy.

Aby mělo porovnání co největší vypovídací hodnotu, jsou ke každé aplikaci napsány vždy stejné scénáře a testy pro oba testovací nástroje. Testy jsou tedy jednou napsány pomocí UI Testing in Xcode a podruhé pomocí Appia. Díky tomu lze v celku jednoduše na první pohled vidět, jak lze napsat stejný test pomocí různých nástrojů a zároveň se při psaní testů projeví silné a slabé stránky daného nástroje.

Jako testovací aplikace byly zvoleny dvě aplikace, které jsou volně ke stažení na GitHubu (viz kapitola 9.2.1 a 9.2.2).

## 9.1 Testovací scénář

Veškeré testy, které budou v rámci této kapitoly popsány, vychází z testovacích scénářů (Test Scenarios), testovacích případů (Test Cases), jednotlivých kroků (Steps) a očekávání (Expectations). Používané zkratky jsou vypsány v kapitole 9.1.2.

Každý testovací scénář se skládá z testovacích případů a každý testovací případ se skládá z jednotlivých kroků a očekávání. Jednotlivé testy mohou mít

také předpoklady (Preconditions), se kterými se v testu počítá.

Ukázka používáného formátu pro psaní scénářů je znázorněna v kapitole 9.1.1.

### 9.1.1 Formát testovacího scénáře

**TS 1** Validace přihlašovací stránky.

- **TC 1** Vložení špatných přihlašovacích údajů.
  - **S 1** Vložte uživatelské jméno „admin“ a heslo „heslo“.
  - ***Ex 1*** *Vyplněné pole jméno a heslo.*
  - **S 2** Klikněte na „ok“.
  - ***Ex 2*** *Zobrazení hlášky o neúspěchu přihlášení.*
- **TC 2** Vložení správných přihlašovacích údajů.
  - **S 1** Vložte uživatelské jméno „admin“ a heslo „heslo123“.
  - ***Ex 1*** *Vyplněné pole jméno a heslo.*
  - **S 2** Klikněte na „ok“.
  - ***Ex 2*** *Zobrazení hlášky o úspěšném přihlášení.*

### 9.1.2 Používané zkratky pro testovací scénáře

**TS** Test Scenarion (Testovací scénář)

**TC** Test Case (Testovací případ)

**S** Step (Krok)

**Ex** Expectation (Očekávání)

**P** Precondition (Předpoklad)

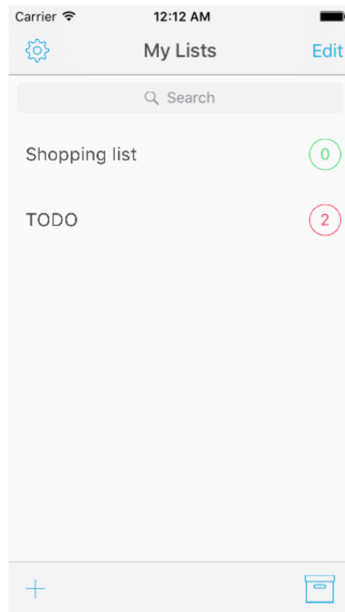
## 9.2 Testovácí aplikace

### 9.2.1 Nativní aplikace

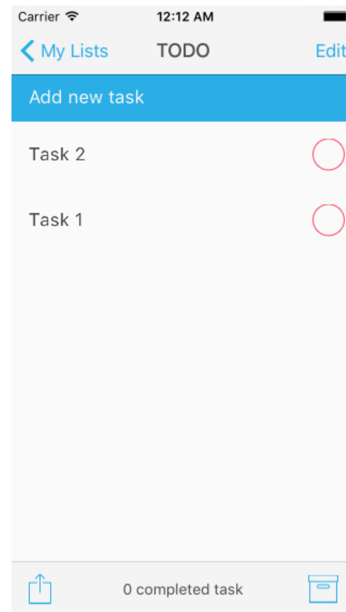
Jako testovací nativní aplikace byla zvolena aplikace s názvem **TinyLog**. Aplikace je dostupná volně ke stažení na GitHubu ([21]) a také je k dispozici v App Storu.

TinyLog je odlehčená aplikace na přidávání a správu úkolů. Aplikace je znázorněna na obrázcích 9.1, 9.2, 9.3 a 9.4.

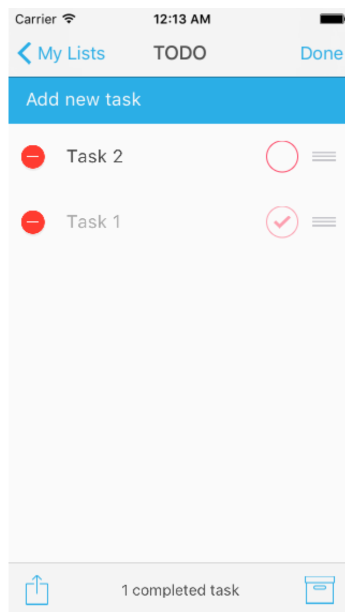
Hlavní smysl aplikace je správa úkolů. Aplikace umožňuje tvorbu listů úkolů. Každý list se skládá ze samotných úkolů. Jednotlivé úkoly je možné označit jako hotové. Úkoly je možné také archivovat a poté případně úplně smazat. Archivace a mazání funguje úplně stejně u listů úkolů. Aplikace umožňuje jednoduché nastavení (viz 9.4). Jednotlivé úkoly a listy úkolů je samozřejmě možné upravovat atd.



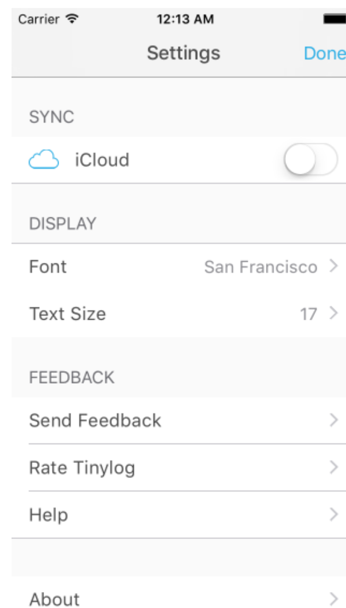

Obrázek 9.1: Ukázka obrazovky aplikace zobrazující seznam všech seznamů úkolů



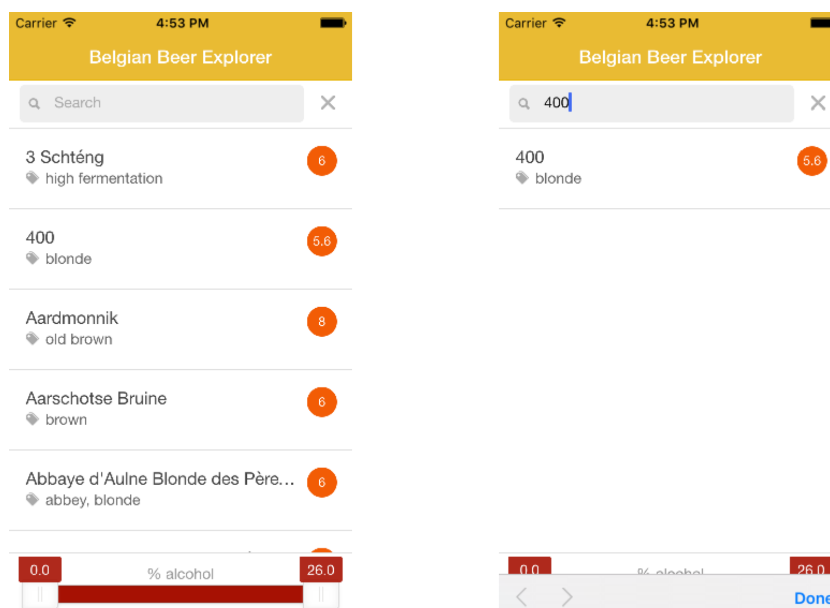

Obrázek 9.2: Ukázka obrazovky aplikace zobrazující jednotlivé úkoly ze seznamu úkolů




Obrázek 9.3: Ukázka obrazovky aplikace zobrazující úpravu jednotlivých úkolů




Obrázek 9.4: Ukázka obrazovky aplikace zobrazující nastavení aplikace




Obrázek 9.5: Ukázka výchozí obrazovky aplikace Ionic Beer Explorer

Obrázek 9.6: Ukázka obrazovky pro vyhledávání podle jména piva

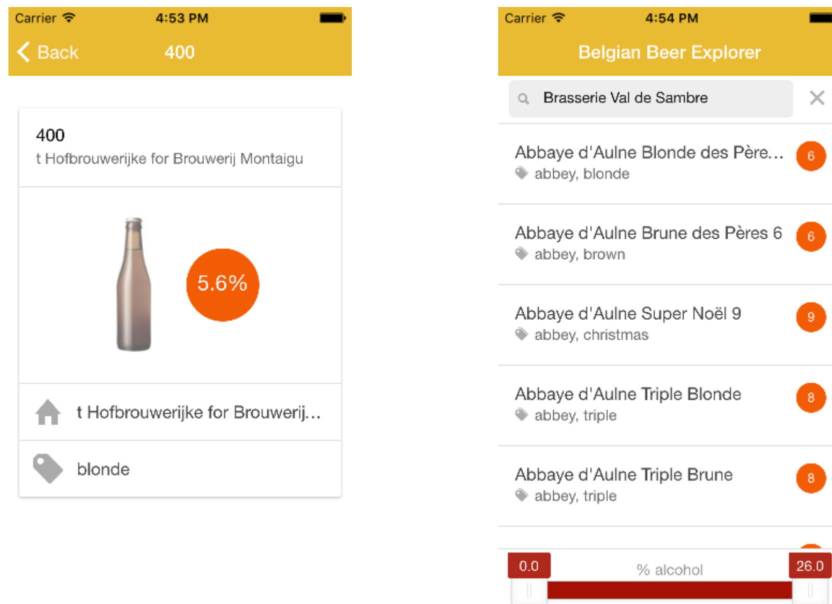
### 9.2.2 Hybridní aplikace

Hledání hybridní aplikace pro testování je o poznání složitější než u aplikace nativní. Hybridních aplikací existuje spousta, ale těch volně je stažení není mnoho. Obvykle jsou to pouze ukázkové aplikace, které toho mnoho neumí.

Nakonec jsem zvolil aplikaci nazvanou **Ionic Beer Explorer**. Aplikace je dostupná na GitHubu [22]. Tato aplikace slouží jako průzkumník belgických piv (viz obrázek 9.5). Umožňuje piva vyhledávat podle názvu (viz obrázek 9.6), pivovaru (viz obrázek 9.8) nebo vlastností. Také umožňuje filtrovat piva podle procent alkoholu. V detailu piva jsou pak uvedeny bližší informace o daném pivu (viz obrázek 9.7).

#### 9.2.2.1 Ionic

Aplikace Ionic Beer Explorer je založena na frameworku **Ionic**. Ionic je HTML5 SDK, které usnadňuje tvorbu hybridních aplikací za pomoci HTML, CSS a JavaScriptu [23]. Ionic ke své funkčnosti využívá nástroje jako Cordova, PhoneGap nebo Trigger.io. Tyto nástroje slouží k finální kompilaci aplikace, aby fungovala na požadované mobilní platformě.




Obrázek 9.7: Ukázka obrazovky aplikace zobrazující detail zvoleného piva

Obrázek 9.8: Ukázka obrazovky pro vyhledávání podle pivovaru

## 9.3 Ukázka testů nativní aplikace

Při výběru testovacího nástroje je potřeba brát v potaz spoustu různých aspektů. Jedním z těchto aspektů je, dle mého názoru, i struktura a syntaxe samotného testu, tedy to, jak přesně vypadá konkrétní test v konkrétním testovacím nástroji. Z tohoto důvodu je v příloze ve zdrojovém kódu C.1 ukázán příklad implementace testovacího scénáře A.1 pro nativní aplikaci (viz 9.2.1) implementovaný pro UI Testing in Xcode.

Pro porovnání je v příloze ve zdrojovém kódu C.4 znázorněn test pro stejný testovací scénář (viz A.1), akorát napsaný pro běh v Appiu. Tento test je konkrétně implementován v Pythonu.

Kompletní implementace testů a také všechny ostatní implementace testovacích scénářů (viz příloha A) jsou k dispozici na přiloženém médiu.

## 9.4 Ukázka testů hybridní aplikace

Stejně jako v kapitole 9.3, i v této kapitole půjde o porovnání konkrétních implementací testů. Opět půjde o porovnání Xcode a Appium, akorát se v tomto případě bude jednat o hybridní aplikaci (viz 9.2.2).

Implementace testovacího scénáře B.3 pro UI Testing in Xcode je znázorněna v příloze ve zdrojovém kódu C.7.

Ve zdrojovém kódu C.10 je ukázána implementace stejného testovacího scénáře jako v předchozím případě (testovací scénář B.3), ovšem tentokrát pro Appium.

Kompletní implementace testů a také všechny ostatní implementace testovacích scénářů (viz příloha B) jsou k dispozici na přiloženém médiu.

## 9.5 Pohodlnost

Od kvalitních testů, které budou testovat pokud možno co nejvíce kódu, je vyžadováno, aby se psaly pohodlně a intuitivně. Těmito aspekty se zabývá tato část srovnání.

### 9.5.1 UI Testing in Xcode

Velkou výhodou psaní testů v Xcode je integrace téměř všeho potřebného pro testování v rámci jednoho editoru. Jediné, co není přímo integrováno v Xcode, je Accessibility Inspector, který neběží přímo v Xcode, nicméně lze jej z Xcode velice jednoduše zapnout.

Při použití UI Testing in Xcode máme automaticky dostupné našeptávání tříd a metod, což je v dnešní době bráno jako samozřejmost, dokud u tuto možnost nepřijdete, jak se mi to stalo u psaní testů pro Appium. Také lze pouze jedním kliknutím spustit UI Recording (viz kapitola 4.7), takže je přepínání mezi psaním testu a nahráváním vcelku rychlé a přirozené.

S ohledem na to, že testy musí být psány v Objective-C nebo Swiftu, působí psaní aplikace a testů velice konzistentním dojmem. Samozřejmě lze využívat všech výhod Swiftu jako je například Extensions pro rozšíření testovacích tříd o custom funkcionalitu.

Samozřejmostí je také spouštění testů přímo z Xcode.

### 9.5.2 Appium

Při psaní testů pro Appium je možné použít libovolný editor. Já osobně jsem zkoušel pro psaní testů editory Sublime text 2, Netbeans a také Visual Studio Code. U všech těchto editorů jsem narazil, jakmile jsem chtěl u Appia s využitím Pythonu použít automatické našeptávání tříd a metod. Je možné, že jsem někde něco vždycky nastavil špatně, každopádně už jenom ta nutnost něco takového řešit zbytečně zdržuje od užitečné práce.

Dalším nepříjemným zjištěním při psaní testů pro Appium je použití appium.app spolu s příkazovou řádkou. Jestliže pro psaní testů pro Appium používáme příkazovou řádku a chceme si vyhledat nějaký element nebo použít nahrávání, musíme použít aplikaci appium.app. Abychom mohli použít tuto aplikaci, musíme vypnout Appium server běžící v příkazové řádce, protože appium.app si zapne vlastní. Po zjištění potřebných aplikací je potřeba apli-

kaci zase vypnout a zapnout server. Toto přepínání je po nějaké době opravdu otravné.

Nahrávání v Appiu funguje celkem spolehlivě, kromě výše zmíněné nepříjemnosti s přepínáním. Oproti Xcode se samozřejmě kód negeneruje přímo do testovacího skriptu, takže je nezbytné po dokončení nahrávání tento kód ručně kopírovat do rozpracovaného testovacího skriptu.

Co je ovšem velice nepříjemné, je časté padání Appium serveru. Konkrétně se mi stávalo téměř pořád, že jakmile běžel test, který neskončil úspěchem, Appium server ukončil session, takže bylo nutné server restartovat. Občas toto chování nastávalo i po úspěšné testu, dělo se to tedy celkem často.

## 9.6 Rychlost

Při psaní testů i při jejich běhu je důležité, aby běžely svižně, jinak by se mohlo stát, že by testy nebyly spouštěny tak často, jak by měly.

### 9.6.1 UI Testing in Xcode

Při psaní testů v Xcode a jejich testování jsem byl velice mile překvapen, jak svižně testy běží. To samé platí i při použití UI Recording, které pro jednoduché úkoly reaguje v podstatě okamžitě. Pokud samozřejmě při nahrávání použijete více akcí naráz, čas od času se stane, že se Xcode sekne, ale obecně vzato je UI Recording velice dobře použitelné.

Velmi příjemná při psaní testů přímo v Xcode je možnost spustit pouze konkrétní test a tedy ne celou sadu testů. Tato možnost je velice užitečná zvláště při vývoji samotných testů a jejich průběžnému testování korektnosti.

Mírně nepříjemné při vývoji testů v Xcode je to, že při častějším spouštění testů se obvykle po nějaké době sekne simulátor, takže je potřeba ho restartovat.

### 9.6.2 Appium

Při testování Appia jsem byl na rozdíl od Xcode velice nepříjemně překvapen, jak pomalu všechny testy běžely. I velice jednoduché testy trvají oproti Xcode velice dlouho a jsou na pohled velice utahané. To platí jak pro samotný běh testů, tak i pro nahrávání.

O nahrávání se v Appiu stará appium.app. Nahrávání, ale i pouze procházení aplikace a hledaní elementů v rámci appium.app, není zrovna svižné. Pokud chceme například zkontrolovat nějakou konkrétní obrazovku, musíme zapnout Inspektora (appium.app), poté se v aplikaci doklikat na potřebnou obrazovku a následně je nutné Inspektora ručně obnovit, aby načetl aktuální obrazovku. Při této akci dělá Inspektor i screenshot, takže to chvilku trvá. U hybridní aplikace se mi dokonce občas stalo, že načtení obrazovky trvalo i kolem dvou minut.

Ne úplně rychlému běhu testů neprospívá ani velice zvláštní způsob spouštění testů. Konkrétně se mi vždy stalo, že při spuštění testu zapne Appium server simulátor. Po nabootování simulátoru server tento simulátor vždy zabil a zapnul simulátor znovu. Po opětovném nabootování teprve spustil samotný test. Vzhledem k tomu, že bootování simulátoru není příliš svižná záležitost, celkem citelně zvyšuje dobu běhu testů.

## 9.7 Možnosti testování

### 9.7.1 UI Testing in Xcode

Psaní testů v Xcode umožňuje vytvářet testy v zásadě všeho. S trochou kreativity lze v drtivé většině případů nasimulovat nejrůznější uživatelská gesta a akce. Toto všechno platí, pokud je aplikace dobře přístupná, tedy má kvalitně nastavené accessibility. Pokud tomu tak je, funguje velice pěkně UI Recording, takže i tvorba testů je velice svižná.

Problém při tvorbě testů nastává, jestliže aplikace nemá kvalitní accessibility. V tomto případě se chová Xcode občas podivně až velice nestandardně. V lepším případě jen vygeneruje nějaký velice dlouhý nesmyslný kód, který nic nedělá. V tom horším shodí celý Xcode. Často se potom stane, že pokud bylo nahrávání delší a byl již vygenerován nějaký kus kódu, smaže se, takže je potřeba začít nahrávat od začátku. V praxi se mi stávalo velice často, že Xcode padal po kliknutí na nepřístupný element.

Kde Xcode trochu pokulhává, je testování hybridních aplikací, kde přístup k elementům není tak dobrý jako u aplikací nativních. K některým elementům se mi vůbec nepodařilo dostat nějak obecně, ale pouze na základě konkrétního řetězce, což není vždy úplně ideální.

### 9.7.2 Appium

Velice zajímavou možností při psaní testů v Appium je možnost využívat Xpath pro přístup k jednotlivým elementům. Díky Xpath si lze uložit obecnou cestu k nějakému elementu a tu později využívat. Výhoda je, že tento postup lze využít i u hybridních aplikací pro přístup k elementům, ke kterým se například v Xcode dostat nějak jednoduše nelze.

Vzhledem k principu fungování Appia není nutné aplikaci při testování Appiem vůbec kompilovat, k testování mi stačí pouze hotová zkompilovaná aplikace, což může být v určitých případech výhoda.

Další výhoda, která nahrává použití Appia pro testování, je možnost napsání pouze jednoho testu, který může běžet jak na iOS, tak i na Androidu.

Dle mého názoru je při psaní testů nejužitečnější pomocník Property Inspector. Tedy aplikace, která je schopna ukázat informace o elementu, jeho accessibility ID nebo cestu k němu atd. S tímto souvisí samozřejmě i nahrávání

akcí, které tyto vlastnosti využívá. Proto je obrovská škoda, že appium.app není oficiálně vyvíjena týmem okolo Appia.

Appium.app je vyvíjena separátně, což má bohužel za následek, že je využívána obvykle starší verzi Appia oproti aktuálně vydané stabilní vývojové větvi. Důsledkem je, že například při využívání nahrávání se generuje zastaralý (deprecated) kód nebo často i kód, jenž v aktuální verzi Appia vůbec nefunguje.

Existence možnosti využívat více různých jazyku je podle mého názoru obrovská výhoda Appia, ale zároveň tak trochu nevýhoda. Občas se mi stalo, že funkční kód v jednom jazyce nemusí fungovat v jazyce jiném. Respektive například možnost otestovat existenci elementu po jeho vyhledávání funguje bez problému v Javě, ale v Pythonu již ne. Python totiž na rozdíl od Javy, jestliže nenajde element (například podle Xpath), automaticky vyhodí výjimku. Takže je nezbytné si občas ručně implementovat funkčnost, která je v jiném jazyce implementovaná, použití pak tedy není zcela konzistentní napříč různými jazyky.

Čas od času je také možné při použití Appia narazit na problém, kdy nějaká funkce z API přestane fungovat s novou verzi iOS. Příkladem budiž funkce ***swipe***, která od verze iOS 7 prostě nefunguje, takže je nutné ji řešit pomocí nějakých workaroundů, jejichž funkčnost je taktéž diskutabilní.

## 9.8 Property Inspector

Property Inspector (inspektor vlastností) patří dle mého mínění k jedné z nejužitečnější věci při tvorbě UI testů. Velmi usnadňuje samotnou tvorbu testů a hledání elementů a jejich ID.

### 9.8.1 UI Testing in Xcode

Xcode využívá Accessibility Inspectora pro zobrazení informací o přístupnosti jednotlivých elementů, a tedy i informací, jak je možné v rámci UI testu k danému elementu přistupovat. Po zapnutí Accessibility Inspectora se zobrazí okno, které je nad všemi ostatními okny a podle toho, kam ukazuje kurzor myši, zobrazuje informace o tom daném elementu. Mně osobně připadá tento způsob fungování celkem chaotický a ne moc přehledný. Často je potřeba si v Inspectoru rozbalit nějaké informace, takže je nutné pohnout kurzorem. Při pohybu se změní element, na který kurzor ukazuje a tak podobně.

### 9.8.2 Appium

Property Inspector využívaný v Appiu je součastí aplikace appium.app. Na rozdíl od Inspectora v Xcode však funguje jinak. Inspector zobrazuje vždy informace ke konkrétní obrazovce aplikace. Zobrazuje celý strom všech elementů, včetně konkrétních informací o aktuálně zvoleném elementu. Tento způsob je

podle mě mnohem přehlednější a navíc je krásně vidět celá hierarchie elementů dané obrazovky.

Drobnou nevýhodou je automatická tvorba snímku aktuálně zobrazované obrazovky, která zobrazení informací zpomaluje. Každopádně přehlednost zobrazených informací jednoznačně převyšuje nad tímto zpomalením.

## 9.9 Vyhodnocování výsledků

### 9.9.1 UI Testing in Xcode

Výhodou Xcode je automatické ukládání historie spouštěných testů, která umožňuje nejenom udržovat přehled o tom, kdy jaké testy běžely, ale také umožňuje udržovat přehled o jejich výsledcích.

Přehled testu také obsahuje časy strávené v jednotlivých krocích testů. Ve výchozím stavu jsou při testování průběžně dělány screenshoty aplikace, ke kterým se lze v přehledu testu jednoduše dostat, takže se lze snadno podívat, jak vypadala aplikace v konkrétním kroku testu.

### 9.9.2 Appium

S ohledem na to, že Appium testy se obvykle spouští z příkazové řádky, která v průběhu testu samozřejmě zobrazuje bohatý výpis toho, co se zrovna provádí, neobsahuje Appium žádný takový přehled, jaký poskytuje Xcode. Appium samozřejmě na konci testu zobrazí informaci o průběhu testu, ale nějaký historický přehled nelze očekávat. Stejně tak jako automatické screenshoty jednotlivých kroků testu a tak podobně.

## 9.10 Dokumentace

### 9.10.1 UI Testing in Xcode

Pokud zrovna pro tvorbu testů nepoužíváte UI Recording, dříve nebo později se dostanete do situace, kdy by se hodila dokumentace k implementaci nějaké konkrétní funkcionality. Co je ovšem velice nepříjemné je fakt, že oficiální dokumentace k UI Testing in Xcode v zásadě neexistuje.

Jediné oficiální dokumenty ze strany Applu jsou v podstatě přednáška z WWDC z roku 2015. Naštěstí existují alespoň komunitní dokumentace na základě hlaviček jednotlivých testovacích tříd a spousta různých tutoriálů.

### 9.10.2 Appium

Appium je na rozdíl od UI testování v Xcode velice pěkně zdokumentováno. Samozřejmě sem trochu zanáší chaos množství jazyků, které je možné s Appiem používat. Také je občas trochu matoucí, že některé funkce jsou zdokumentovány v dokumentaci Appia a některé věci jsou zdokumentovány přímo

v dokumentaci Driveru konkrétního jazyka, ale obvykle není zase takový problém po chvilce hledání dohledat vše, co je potřeba.

## 9.11 Zhodnocení

Z výše uvedený silných a slabších stránek Appia a testování v Xcode je patrně vidět, že jako nástroj s více kladnými vlastnostmi se jeví spíše použití UI Testing in Xcode. Všechny poznámky a připomínky k práci s oběma nástroji je samozřejmě nutné brát v kontextu, ve kterém chceme daný nástroj používat.

### 9.11.1 UI Testing in Xcode

Ze všech výše zmíněných argumentů já osobně preferuji pro testování iOS použít UI Testing in Xcode. Hlavním argumentem pro použití tohoto nástroje je, z mého pohledu, jednoznačně rychlost. Jak již bylo zmíněno, testování v Xcode je oproti Appiu opravdu citelně rychlejší, což dělá tvorbu i samotnou práci s testy mnohem příjemnější a zároveň odpadá nutnost dlouhého čekání při každém spouštění testu.

Dalším klíčovým argumentem je práce pouze v rámci jednoho editoru. Psaní testů přímo v Xcode je velice přímočaré a odpadá nutnost jakéhokoliv přepínání mezi různými okny s příkazovými řádkami a tak podobně.

Jednou z prezentovaných výhod Appia je možnost použít téměř libovolný jazyk pro tvorbu testů. Je pravda, že v některých případech je tato možnost volby příjemná. Nicméně vzhledem k tomu, že nativní aplikace jsou programovány v Objective-C a Swiftu, pro žádného programátora by pravděpodobně neměl být nějak zásadní problém v tomto jazyce psát i testy. Jestliže by se jednalo o aplikace hybridní, již bych se nad použitím Appia jednoznačně zamyslel.

### 9.11.2 Appium

I přes všechny argumenty hrající spíše ve prospěch testování v Xcode, má Appium určitě své místo a případy, kdy není úplně špatné se zamyslet, jestli pro testování nepoužít právě tento testovací nástroj.

Z mého pohledu tento nástroj slouží minimálně ke zvážení v případě, že potřebujeme testovat stejnou aplikaci jak na iOS, tak i na Androidu nebo pokud potřebujeme testovat nějakou hybridní aplikaci.

V případě multiplatformní aplikace výše zmíněné neduhy pro použití Appia může přebít fakt, že by bylo možné psát pouze jedny UI testy, které by byly použitelné jak pro iOS, tak i Android. Tento fakt může ušetřit spoustu času a potažmo i peněz při vývoji aplikace.

Použití Appia u hybridních aplikací zase dává smysl hlavně v tom, že díky možnosti v rámci Appia přepnout kontext přímo do WebView může progra-

mátor psát v zásadě Selenium testy, které jsou pro testování webových aplikací používány velice široce.

Kapitola 10

# Zhodnocení testovacích nástrojů

Při výběru testovacího nástroje vždy záleží hlavně na konkrétních požadavcích, které jsou na tento nástroj kladeny. Obecně by se testované testovací nástroje (viz kapitola 2.1) daly rozdělit do tří kategorií. Tyto kategorie jsou:

1. nativní testovací nástroje,
2. nástroje založené na frameworku v Objective-C,
3. nástroje založené na Selenium WebDriveru.

## 10.1 Nativní testovací nástroje

Mezi nativní testovací nástroje patří:

1. UI Automation (viz kapitola 3),
2. UI Testing in Xcode (viz kapitola 4).

Oba tyto nástroje jsou oficiálně od Applu. Jak již bylo zmíněno, do budoucna se počítá pouze s nástupcem UI Automation, tedy s UI Testing in Xcode. Jestliže budeme chtít zvolit nativní testovací nástroj, určitě je rozumnější zvolit takový, se kterým se do budoucna počítá, a zvolit UI Testing in Xcode.

### 10.1.1 Výhody

Výhody použití nativního testovacího nástroje jsou vcelku evidentní. Vzhledem k tomu, že nativní testovací nástroj je přímo od Applu, reflektuje jako první testovací nástroj nové vlastnosti nových verzí systému. Při představení nových vlastností systému tedy není potřeba čekat, až ji implementují testovací nástroje třetích stran.

Velkou výhodou použití nativních testovacích nástrojů je poměrně rychlý běh testů na rozdíl od ostatních testovacích nástrojů. Například test pro UI Testing in Xcode běží opravdu citelně rychleji než stejný test běžící v Appiu.

Další obrovskou výhodou oproti ostatním nástrojům je to, že s aktualizací systému se obvykle nestane, že by nějaké API pro psaní testů přestalo fungovat, pokud nebylo označeno jako zastaralé, což se bohužel u nástrojů třetích stran občas stává.

Nativní testovací nástroje mají také velmi dobrou integraci s Xcode. UI Automation skrze Instruments, ale i UI Testing in Xcode jsou v podstatě nový specifický target v projektu (viz kapitola 4), takže se s ním pracuje velice příjemně a veškerá práce je řešena pouze v rámci Xcode.

### 10.1.2 Nevýhody

Jako nevýhoda se pro někoho jeví nutnost použití pro UI Testing in Xcode jazyka Swift nebo Objective-C a v UI Automation použití JavaScriptu. Mně osobně nepřijde jako zásadní problém použít pro testy stejný programovací jazyk, který se používá pro vývoj celé aplikace, jak je tomu v případě UI Testing in Xcode. Nicméně je možné, že v rámci Continuous Integration může občas dávat větší smysl použití nějakého interpretovaného jazyka.

Testy napsané pro nativní testovací nástroje běží samozřejmě pouze na iOS, takže jakmile máme aplikaci napsanou pro více platforem, musíme pro ostatní platformy psát testy separátně.

Nativní testovací nástroje nejsou podle mého názoru ideální pro testování hybridních aplikací. Netvrdím, že to není možné, ale práce s hybridní aplikací není moc komfortní, a to hlavně kvůli tomu, že na rozdíl od jiných testovacích nástrojů neumožňuje například UI Testing in Xcode vyhledávání elementů podle CSS selektorů atd.

Jako nevýhoda může také někomu připadat nutnost kompilace aplikace před spuštěním testů.

### 10.1.3 Kdy použít

Jak již bylo zmíněno dříve, pokud zvolit nativní testovací nástroj, určitě bych vybral UI Testing in Xcode. Dle mého názoru je vhodné zvolit UI Testing in Xcode, jestliže se vaše požadavky kladené na testovací nástroj přibližují následujícím požadavkům:

1. testy pouze pro iOS,
2. svižný běh testů,
3. rychlá podpora nejnovějších vlastností systému,
4. integrace testování v Xcode.

## 10.2 Nástroje založené na frameworku v Objective-C

Mezi nástroje založené na frameworku napsaného v Objective-C se řadí:

1. Calabash (viz kapitola 7),
2. Frank (viz kapitola 8.1).

Jak bylo popsáno v příslušných kapitolách (viz kapitola 7 a 8.1), tyto nástroje fungují tak, že k testované aplikaci přilinkují HTTP server, se kterým poté komunikuje testovací skript.

Při volbě, zdali použít Calabash, nebo Frank, je podle mého názoru vhodnější použít Calabash. Je sice trošku těžší přidat Calabash do Xcode projektu aplikace než je tomu při použití Franka, nicméně Calabash je na rozdíl od Franka mnohem lépe zdokumentovaný a i samotná podpora je mnohem lepší. Další nevýhoda Franka oproti Calabashi je jeho zprovoznění spolupráce s CocoaPods (viz kapitola 8.1.3).

### 10.2.1 Výhody

Hlavní výhodou těchto nástrojů je použití Cucumberu (viz kapitola 7.1), tedy vcelku přirozenému zápisu testů, kterým rozumí i člověk, jenž není úplně technicky zdatný.

Další výhodou je, že po správném nastavení spouštění testované aplikace se serverem je testování vcelku jednoduché a například na rozdíl od nástrojů založených na Selenium WebDriveru není potřeba ručně startovat server pro běh testů, jelikož server se startuje automaticky po startu testované aplikace.

Tento typ nástrojů funguje také velice pěkně pro testování hybridních aplikací a konkrétně Calabash umožňuje multiplatformní testování, tedy testování jak iOS, tak i Android aplikací.

### 10.2.2 Nevýhody

Nevýhoda tohoto typu testovacích nástrojů je nutnost přilinkovat server k testované aplikaci, což není vždy zcela banální záležitost a občas to dá dost práce.

Při použití těchto nástrojů je také velice důležité dát pozor, aby se tento testovací nástroj nedostal do produkční aplikace, která se plánuje vydat v App Storu. Tyto nástroje používají privátní API, kvůli kterému aplikace bude v App Storu odmítnuta.

Hlavní výhoda těchto nástrojů, tedy použití Cucumberu, s sebou ovšem přináší i nevýhodu. Tato nevýhoda spočívá v nutnosti psát implementace testů v Ruby. Samozřejmě, že někomu to může vyhovovat, ale mně osobně se vynucení jednoho konkrétního jazyka, jenž je navíc odlišný od jazyka, ve kterém je psána aplikace, moc nezamlouvá.

Jelikož se nejedná o nativní nástroj od Applu, může se občas stát, že nějaké funkce přestanou fungovat po aktualizaci systému.

### 10.2.3 Kdy použít

Jak si můžeme všimnout z předchozího textu, jsou tyto nástroje poměrně značně odlišné od nativních nástrojů. Jak bylo zmíněno výše, jestliže bychom chtěli využít tento typ nástroje, je podle mě jednoznačně lepší použít Calabash. Já osobně bych tento nástroj použil v případě, pokud bych se blížil těmto požadavkům na testovací nástroj:

1. testy na iOS a Android,
2. psaní čitelných testů v Cucumberu,
3. zkušenost s Ruby,
4. komerční podpora testovacího nástroje.

## 10.3 Nástroje založené na Selenium WebDriveru

Testované nástroje, které jsou založeny na Selenium WebDriveru, jsou:

1. Appium (viz kapitola 6),
2. iOS-driver (viz kapitola 8.2).

Další kategorií testovacích nástrojů jsou nástroje, které jsou založeny na Selenium WebDriveru (viz kapitola 5.2). To znamená, že pro testování je nutné mít zapnutý server, se kterým komunikuje testovací skript. Na rozdíl od nástrojů založených na frameworku v Objective-C (viz kapitola 10.2) běží server mimo aplikaci, takže není nutné testovanou aplikací kompilovat, aby v sobě obsahovala server. Na druhou stranu je nutné server před testováním vždy manuálně zapnout.

Při volbě, zdali použít Appium, nebo ios-driver, má dle mého názoru jasně navrch Appium. Appium je mnohem lépe zdokumentované a také na rozdíl od WebDriveru je jeho spuštění velice jednoduché, jelikož jak již bylo zmíněno, ios-driver obsahuje závislosti, které v současné době nejsou moc udržovány. Appium je také multiplatformní, takže je možné psát stejné testy pro iOS i Android.

### 10.3.1 Výhody

Mezi hlavní výhody nástrojů založených na Selenium WebDriveru patří možnost testovat aplikaci, kterou není potřeba kvůli testu kompilovat. Takže stačí když má tester k dispozici již zkompilovanou aplikaci a může testovat.

Díky tomu, že jsou tyto nástroje založeny na Selenium WebDriveru, jsou plně kompatibilní s WebDriver API. Tento fakt velice ulehčuje testování hybridních aplikací, protože tyto aplikace je možné testovat v podstatě jako klasické webové aplikace pomocí Selenium WebDriveru.

Dalším zásadním plusem tohoto typu nástrojů je fakt, že testy se dají psát téměř v libovolném jazyce. Respektive oficiálně podporovaných je asi pět jazyků, ale díky implementacím třetích stran je těchto jazyků opravdu mnoho.

### 10.3.2 Nevýhody

Hlavní, a z mého pohledu poměrně zásadní nevýhoda těchto nástrojů, je jejich rychlost. Tyto nástroje jsou například oproti nativním testovacím nástrojům (viz kapitola 10.1) výrazně pomalejší.

Nevýhodou je samozřejmě také to, že na rozdíl od nativních aplikací se občas stane, že po vydání nové verze iOS přestanou nějaké funkce z API fungovat.

Další věcí, na kterou je nutné u tohoto typu nástrojů myslet, je spuštění serveru před zahájením testovacího skriptu. Vzhledem k tomu, že server není uvnitř testovací aplikace, je nezbytné ho vždy manuálně před testováním zapnout.

Možnost psát testy v různých jazycích je velká přednost těchto nástrojů, nicméně je to částečně i nevýhoda. Díky možnosti použití různých jazyků jsou dostupné tutoriály a fóra velmi roztříštěné, takže se často stává, že při hledání odpovědi, jak něco udělat, se odpověď sice najde, ale v jiném jazyce, než zrovna tester potřebuje.

### 10.3.3 Kdy použít

Z důvodů zmíněných výše bych jako testovací nástroj založený na Selenium WebDriveru zvolil jednoznačně Appium. Samotné Appium je z mého pohledu ideální využít, jakmile se požadavky na testovací nástroj blíží následujícím vlastnostem:

1. testy na iOS a Android,
2. testování bez nutnosti kompilovat aplikaci,
3. možnost volby jazyka pro psaní testů,
4. testování hybridních aplikací.

Kapitola **11**

# Integrační testování za pomocí mockovacích nástrojů

V předchozích kapitolách bylo popsáno mnoho příkladů využití integračního testovaní, včetně spousty konkrétních příkladů testů. I když je integrační testování specifický druh testování, má toho spoustu společného s ostatními druhy testování. Téměř ve všech druzích testování se dřív nebo později dostaneme do situace, kdy potřebujeme podmínky pro test trošku poupravit.

Při testování softwaru se obecně snažíme simulovat podmínky opravdového běhu aplikace, abychom měly zpětnou vazbu, že i po provedených změnách se aplikace chová tak, jak je žádoucí. V opravdovém světě ale ne všechno vždy funguje přesně takovým způsobem, jak předpokládáme. Typickým příkladem je připojení k internetu. Jakmile máme například aplikaci, která ke svému běhu vyžaduje připojení k internetu, můžeme se poměrně často dostat do situace, kdy připojení prostě nebude možné. V tomto případě aplikace samozřejmě nebude plnit svou funkci, ale s tím není možné nic dělat.

V reálném světě se počítá s tím, že aplikace, která vyžaduje připojení k internetu a nemá jej, nefunguje správně. Pokud ale chceme spustit test, který k úspěšnému provedení potřebuje připojení k internetu a v testu počítáme s tím, že je toto připojení dostupné, můžeme se dostat do situace, kdy připojení vypadne a test skončí chybou. To v podstatě znamená, že jestliže vypadne internet, nejsme schopni spouštět testy. Stejně tak se může stát, že připojení bude funkční, ale bude strašně pomalé, takže test může trvat velmi dlouho.

Dlouho běžící testy a také ty testy, které končí chybou jenom někdy a to kvůli vnějším vlivům, nejsou žádoucí. Takové chování může vést ke sporadickému spouštění testů místo pravidelného běhu. Výsledky testů také nebudou zcela důvěryhodné, jestliže nebudou fungovat vždy korektně. Tyto problémy je možné do značné míry eliminovat pomocí mocků a stubů.

## 11.1 Základní pojmy

Při testování za pomoci mocků a stubů se používá relativně mnoho pojmů, které se často pletou nebo se používají jako ekvivalentní. Z tohoto důvodu bych zde rád specifikoval, v jakém významu já chápu všechny zde používané výrazy. Budu se držet názvosloví definovaného podle Gerarda Meszarose [24, s. 524].

### 11.1.1 SUT

SUT je zkratka pro System Under Test. Jedná se tedy o aktuálně testovanou komponentu systému. Pro lepší představu si SUT lze představit například jako třídu (viz pseudokód ve zdrojovém kódu 11.1).

Zdrojový kód 11.1: Ukázka tvorby SUT bez závislosti na další komponentě

```
mySut = new MySut()
```

### 11.1.2 DOC

DOC je zkratka pro Depended-On Component. Jedná se tedy o komponentu, kterou využívá SUT. Zmíněná komponenta je nutná pro běh SUTu. Tato vazba je znázorněná v pseudokódu ve zdrojovém kódu 11.2. Vazba je znázorněná pomocí DI DOC třídy do SUT třídy.

Zdrojový kód 11.2: Ukázka závislosti SUT na DOC

```
myDoc = new MyDoc()
mySut = new MySut(myDoc)
```

### 11.1.3 Dummy

Dummy objekt se používá tam, kde je potřeba do funkce nebo metody předat nějaký parametr, který pro test není důležitý, ale pokud by chyběl, nebylo by možné funkci nebo metodu korektně zavolat. S dummy objektem by se v rámci testu nemělo vůbec pracovat [24, s. 526].

### 11.1.4 Fake

Fake objekt slouží k nahrazení nějaké komponenty testovaného systému, na kterém je testovaný systém závislý. Tento objekt by měl být oproti originál-

nímu objektu mnohem více odlehčený a jednodušší, takže by měl být i mnohem rychlejší [24, s. 525].

Používá se obvykle tam, kde není originální komponenta k dispozici nebo je příliš pomalá a nebo je prostě problém tuto komponentu pro test použít. Typickém příkladem využití fake objektu je použití In-memory databáze. Tato databáze pracuje se všemi daty jen a pouze v paměti, to znamená, že data nejsou ukládána perzistentně, což také implikuje mnohem vetší rychlost práce s takovou databází.

### 11.1.5 Stub

Stub slouží ke kontrole vstupu do SUT. Stub poskytuje určité zakonzervované odpovědi na různá volání během testu [24, s. 524].

Krásným příkladem využití stubu jsou HTTP požadavky. Pomocí stubu je možné definovat při požadavku na konkrétní URL konkrétní odpověď. Takže je možnost v jednom testu otestovat například úspěšný dotaz a v dalším testu zpracování chybového dotazu.

### 11.1.6 Mock

Mock se využívá k ověření výstupu ze SUT. Chování SUT se ověřuje pomocí specifikace očekávané interakce [24, s. 525]. Hlavní využití mockování je pro testování chování, které občas není jednoduché otestovat pomocí stavů v systému.

Příklad využití mockování je otestování používání cache. Pokud chceme otestovat, že aplikace před tím, než položí dotaz do databáze, položí nejdříve dotaz do cache.

### 11.1.7 Spy

Spy je v zásadě stejný jako mock (viz 11.1.6), ale s tím rozdílem, že oproti mocku pouze pozoruje výstup SUT, ale neověřuje ho. To znamená, že spy sleduje všechny jednotlivé operace prováděné v rámci testu. Na konci testu se poté můžeme na tyto operace dotazovat a tak zjistit, zdali se provedla operace, která nás zajímala [24, s. 525].

Vhledem k tomu, že se můžeme dotazovat jenom na to, co nás opravdu zajímá, nemusíme definovat všechny kroky testu, ale jen ty, které chceme. Z toho jasně plyne, že při použití spy jsou testy obvykle mnohem přehlednější.

Hlavní nevýhodou ve srovnání s mockem (viz 11.1.6) je to, že pokud operace skončí chybou, dozvíme se to až na konci testu, takže samotná chyba se pak velice špatně hledá na rozdíl od mocků, kde díky definici všech kroků aplikace skončí hned v problémovém místě.

## 11.2 Mock vs. Stub

V praxi se často stává, že pojmy mock a stub jsou brány jako totožné, přestože se jedná o vcelku rozdílné věci. Následující rozdíly jsou popsány na základě článku od Martina Fowlera [25].

I když mock a stub nejsou to samé, spousta testerů často používá v běžné řeči výraz mock, přestože se zcela přesně nejedná o mock. Příkladem může být mockovací framework Mockito, který se označuje jako mockovací framework, ale technicky správné označení je spy framework [26]. Pokud budu v dalším textu mluvit o mocku, je to myšleno obecně a nemusí se tedy jednat striktně o mock.

### 11.2.1 Stub

Jak již bylo popsáno v kapitole 11.1.5, stub poskytuje odpovědi na určitá volání. To tedy znamená, že pokud SUT dostane nějaké odpovědi od stubu, je následně nutné zkontrolovat, jestli se SUT pro danou odpověď zachoval korektně.

Při použití stubů se kontrola funkčnosti testovaného systému a úspěchu testu provádí za pomoci kontroly stavů systému. Musíme tedy vždy za pomoci assertů (porovnání) zkontrolovat, že po komunikaci se stubem je systém v předpokládaném stavu. Pokud tomu tak není, skončí daný test chybou.

### 11.2.2 Mock

Mockování na rozdíl od stubů neprovádí kontrolu funkčnosti aplikace za pomocí stavů. Mock využívá pro validaci funkčnosti kontrolu chování.

Kontrola chování znamená, že v rámci testu se definuje předpokládané chování systémů, tedy například volané metody atd. Test poté při běhu testu kontroluje, že se definované chování provádí a to v přesně definovaném pořadí. Pokud ne, test skončí chybou.

Je evidentní, že tento druh validace aplikace je odlišný od validace využívané při použití stubů (viz 11.2.1). Je zřejmé, že mock i stub mají svoje využití a případy, kdy je lepší využít jedno, či druhé.

## 11.3 Kdy použít mockování

V předchozích částech této kapitoly bylo popsáno, k čemu je vlastně mockování dobré a jak může pomoct při testování. Při použití mockování je samozřejmě potřeba dát pozor, aby se mockovalo s rozumem a celé testování se nakonec ve výsledku zbytečně nezkomplikovalo.

Situací, kdy se využívá mockování, je samozřejmě mnohem více, ale následující případy jsou dle mého názoru ty nejčastější důvody pro použití mocků.

### 11.3.1 Nedeterministické operace

Kandidátem na vytvoření mocku je komponenta, která provádí nějakou nedeterministickou operaci. Příkladem může být komponenta, která zajišťuje síťové požadavky. Operace prováděné takovou komponentou jsou závislé na připojení k internetu a samozřejmě také na dostupnosti serveru, který dotazuje. Nemůžeme si tedy být vždy jistý, že operace na totožný požadavek dopadne vždy stejně.

Při použití mocku můžeme v rámci testu otestovat chování úspěšně provedených požadavků, ale stejně tak můžeme celkem jednoduše nasimulovat požadavky chybové.

### 11.3.2 Pomalé operace

V posledních letech se těší stále větší popularitě při vývoji softwaru, využívá postup známý jako Continuous Integration (CI). Při použití CI je obvykle celá aplikace i několikrát denně testována, a sice vždy po provedení změn a jejich nahrání na CI server [27].

Při využití CI jsou tedy testy spouštěny relativně často. Z tohoto důvodu je vhodné, aby testy běžely přiměřeně dlouhou dobu a nestaly se zdrojem zdržování samotného vývoje.

Příkladem mohou být například síťové požadavky, které při velmi pomalém připojení značně prodlužují trvání testu. Dalším příkladem může být použití databáze, protože při větším množství přístupů do databáze samozřejmě stoupá i doba samotného testu.

### 11.3.3 Chybějící komponenta

Dalším typickým příkladem použití mocku je situace, kde potřebujeme pro otestování komponenty nějakou jinou komponentu, která ale ještě není implementovaná. V tomto případě si můžeme pomoci vytvořením mocku, který se bude tvářit jako chybějící komponenta.

## 11.4 Mockování ve Swiftu

Pro mockování se dříve na iOS daly využít mockovací frameworky jako například OCMock nebo Mockito. Situace se ale poměrně rapidně změnila s příchodem programovacího jazyka Swift. Hlavním důvodem je to, že Swift na rozdíl od Objective-C poskytuje pouze velmi limitovaný přístup do runtimu jazyka [28], který obvykle mockovací frameworky využívají.

Frameworky jako OCMock [28] a OCMockito [29] buď Swift nepodporují vůbec, nebo jen částečně. Například nástroj pro mockování síťových požadavků Nocilla [30] a podobné frameworky fungují ve Swiftu naprosto bez problémů.

### 11.4.1 Mockování bez frameworku

Tím, že většina mockovacích frameworků ve Swiftu není plně funkční, se stále víc testerů kloní k myšlence nevyužívat žádný framework, ale využívat ruční mockování za pomoci Dependancy Injection (DI). Mockování bez využití frameworků není nijak obtížné, jestliže se v rámci aplikace používá DI.

#### 11.4.1.1 Depedency Injection

Princip Dependency Injection je ve své podstatě velice jednoduchý. Pokud máme třídu, která využívá nějakou další třídu, daná třída její instanci nevytváří, ale je do ní injektována (vložena). DI využívá ke své funkčnosti protokoly. Vše začíná tvorbou protocolu. Příklad jednoduchého protokolu je vyobrazen ve zdrojovém kódu 11.3.

Zdrojový kód 11.3: Ukázka tvorby protokolu ve Swiftu

```
protocol Api {
  func getJson(path) -> JSON
}
```

Po vytvoření protokolu je nezbytné vytvořit třídu, která bude splňovat protokol 11.3. Tato třída je znázorněna ve zdrojovém kódu 11.4.

Zdrojový kód 11.4: Ukázka tvorby třídy splňující protokol ve Swiftu

```
class MyApi: Api {
  func getJson(path) -> JSON {
    // implementation goes here
  }
}
```

Jestliže máme třídu, která ke své funkci potřebuje využít námi definované ***Api*** (viz zdrojový kód 11.4), stačí toto ***Api*** do třídy injektovat. Existuje více možností jak to provést, ale ta nejpoužívanější je injektování požadované třídy v konstruktoru (initu) třídy. Injektování třídy ***Api*** (viz zdrojový kód 11.4) je znázorněno ve zdrojovém kódu 11.5.

Zdrojový kód 11.5: Ukázka DI ve Swiftu

```
class ApiLoader {
  let api: Api

  init(api: Api) {
    self.api = api
  }
}
```

Samotná tvorba instance třídy včetně injektování ***Api*** je znázorněna ve zdrojovém kódu 11.6.

Zdrojový kód 11.6: Ukázka tvorby instance a injektování třídy ve Swiftu

```
let myApi = MyApi()
let apiLoader = Apiloader(api: myApi)
```

#### 11.4.1.2 Mockování s použitím DI

Z kapitoly 11.4.1.1 je vidět, jakým způsobem funguje DI ve Swiftu. Po pročtení ukázek zdrojových kódu v této kapitole je myslím celkem jasné, jak využít DI k mockování.

Ze zdrojového kódu 11.5 je vidět, že třída ***ApiLoader*** přijímá jako parametr takové třídy, které splňují protokol znázorněný ve zdrojovém kódu 11.3. To znamená, že kdybychom si místo třídy ***MyApi*** udělali nějaký mock, který by splňoval stejný protokol, mohli bychom tuto třídu předat jako parametr. Protože obě třídy splňují stejný protokol, třída, do které je tento mock injektován, nepozná rozdíl, a s injektovanou třídou bude pracovat stejně, jako předtím. Ukázka mocku třídy pro API a následná inicializace je znázorněna ve zdrojovém kódu 11.7.

Zdrojový kód 11.7: Ukázka mocku třídy splňující protokol ve Swiftu

```
class MockApi: Api {
  func getJson(path) -> JSON {
    return JSON("mocked json")
  }
}

// create instance with mock injection
let mockApi = MockApi()
let apiLoader = Apiloader(api: mockApi)
```

## 11.5 Mockování UI testů v Xcode

Při UI testování v Xcode se také dostaneme do situace, kdy by se nám hodilo mockovat různé operace. Na rozdíl od Unit testů to však při UI testech není tak jednoduché. Důvodem je to, že při UI testování se testovaná aplikace spouští v samostatném procesu, takže nelze nějak jednoduše upravit injektované třídy (více viz kapitola 4).

### 11.5.1 Předání parametrů z UI testu do aplikace

Jestliže chceme při UI testech předat nějakou informaci do testované aplikace, možnosti jsou velmi omezené. UI testování v Xcode nám v zásadě umožňuje pouze při spouštění aplikace v testu předat do aplikace textové parametry a také dovoluje předat slovník s textovým klíčem a libovolným řetězcem. Ukázka předání dat z testu do testovací aplikace je ukázána ve zdrojovém kódu 11.8.

Zdrojový kód 11.8: Ukázka předání dat do aplikace při UI testu

```
let app = XCUIApplication()
app.launchArguments = ["mock"]
app.launchEnvironment = ["data": "{...JSON...}"]
```

### 11.5.2 Zpracování předáných parametrů z UI testu aplikací

V testované aplikaci lze tyto předané parametry zpracovat a nějak na ně reagovat. Díky tomu můžeme například v ***AppDelegate*** inicializovat aplikaci specifickým způsobem právě pro testovací účely. Ukázka zpracování předaných parametrů ze zdrojového kódu 11.8 je ve zdrojovém kódu 11.9. Tento kód z ukázky funguje například v metodě ***didFinishLaunchingWithOptions*** třídy ***AppDelegate***.

Zdrojový kód 11.9: Ukázka zpracování parametrů předaných do aplikace z UI testu

```
let appArgs = NSProcessInfo.processInfo().arguments

if (appArgs.count > 1 && appArgs[1] == "mock") {
  let processInfo = NSProcessInfo.processInfo()
  let jsonString = processInfo.environment["data"]
  // process jsonString here
}
```

Díky možnosti předat a detekovat předaný parametr jsme schopni v aplikaci reagovat, a tedy i teoreticky mockovat, co je potřeba. Můžeme také například pro testování inicializovat databázi (Core data) pro použití úložiště pouze v paměti, takže veškeré změny provedené během testování aplikace se v databázi po skončení testu ztratí.

Co je na tomto způsobu celkem nepříjemné, že tyto úpravy se dějí v aplikaci a ne v rámci UI testů, takže si aplikaci „špiníme“ testovacím kódem. Samozřejmě je možné si vytvořit nový target, kde by se tato logika řešila, ale jedná se o další práci navíc.

KAPITOLA 12

# Pomocný nástroj pro testování UI

Při programování nějaké trochu složitější aplikace se tester často může dostat do situace, kdy je nezbytné otestovat nějakou konkrétní obrazovku aplikace, která je v aplikaci schována někde hodně hluboko, takže je nutné udělat spoustu kroků, než se k dané obrazovce prokliká. Často je potřeba stejnou obrazovku otestovat pro různá data, jako je například různě dlouhý text nebo různé velikosti obrázků apod.

Jestliže tester provádí manuální testování nějaké obrazovky, která je v hierarchii aplikace hluboko a musí se pro každý test dané obrazovky proklikat přes mnoho jiných obrazovek, jedná se o poměrně časově náročnou záležitost.

Podobně je tomu i při psaní testů, které se mají spouštět automaticky. Automatizovaný test se samozřejmě zvládne proklikat aplikací mnohem rychleji než tester, ale kvůli testování konkrétní obrazovky je nezbytné v testu nadefinovat relativně dost kódu, aby se test dostal k potřebné obrazovce, takže se kvůli tomu test zbytečně komplikuje, místo toho, aby se soustředil na obrazovku, kterou je opravdu potřeba otestovat.

Z výše uvedených důvodů byl vytvořen tento pomocný nástroj, který se snaží maximálně usnadnit jak manuální testování testerům, tak také psaní automatických UI testů.

## 12.1 Pomocný nástroj

Pomocný nástroj nazývaný **ControllerMocker** se snaží řešit výše naznačený problém, tedy UI testování konkrétních obrazovek aplikace bez nutnosti proklikání celou hierarchií aplikace k těmto obrazovkám.

Principiálně funguje tento nástroj velice jednoduše. Pokud chceme otestovat například tři konkrétní obrazovky aplikace, musíme si manuálně vytvořit kontrolery těchto obrazovek a tyto kontrolery poté předat do pomocného ná-

stroje. Po spuštění aplikace se zavolá příslušná metoda pomocného nástroje, která zobrazí první z definovaných kontrolerů. Pomocný nástroj umožňuje mezi jednotlivými kontrolery plynule přepínat, takže je možné v klidu analyzovat, zdali daná obrazovka ukazuje vše, co má.

## 12.2 Instalace

Pomocný nástroj (dále jen **ControllerMocker**) je distribuovaný pomocí CocoaPods.

### 12.2.1 CocoaPods

CocoaPods je manažer závislostí napsaný v Ruby pro Cocoa projekty psané v Objective-C a Swiftu, který obsahuje tisíce různých knihoven [31]. CocoaPods je celkem standardní způsob distribuce nástrojů třetích stran.

Pro definování závislostí projektu využívá **Podfile**. V tomto souboru jsou definovány všechny závislosti, které projekt má. Ukázka **Podfile** pro instalaci **ControllerMockeru** je ve zdrojovém kódu 12.1.

Zdrojový kód 12.1: Ukázka Podfile pro instalaci ControllerMockeru

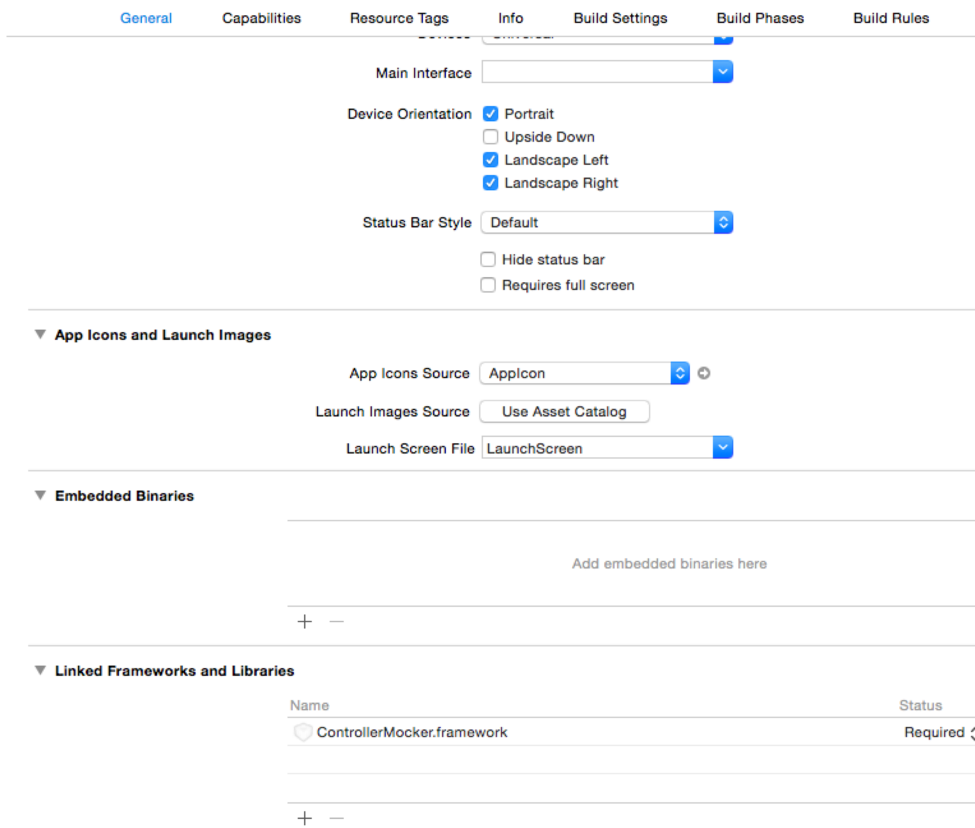
```
platform :ios, '8.0'
use_frameworks!

pod 'ControllerMocker', :git =>
'https://github.com/marysmech/ControllerMocker.git'
```

Po přidání **Podfile** ke svému projektu a připojení ControllerMockeru, jak je znázorněno ve zdrojovém kódu 12.1, stačí spustit instalaci Podů. Instalace se spustí pomocí příkazu demonstrovaného ve zdrojovém kódu 12.2.

Zdrojový kód 12.2: Ukázka spuštění instalace Podfile

```
pod install
```

Po úspěšné instalaci Podů by měl být ControllerMocker připraven k použití. V ojedinělých případech jsem narazil na problém, kdy Xcode ControllerMocker našel, ale z nějakého důvodu ignorovalo metody, které definuje. Pokud tato situace nastane, je nutné přilinkovat framework ControllerMocker. To se provede v nastavení targetu, kde chceme ControllerMocker použít. Přilinkování je naznačeno na obrázku 12.1.




Obrázek 12.1: Znázornění přilinkování ControllerMocker frameworku

## 12.3 Použití

Samotné volání ControllerMockeru z aplikace je řešeno pomocí rozšíření ***UIApplicationDelegate***. Toto rozšíření poskytuje dvě metody. První z nich je znázorněna ve zdrojovém kódu 12.3. Tato metoda slouží k zobrazení definovaných kontrolerů, které si můžeme podle libosti přepínat.

Zdrojový kód 12.3: Ukázka signatury metody pro mockování kontrolerů pomocí ControlerMockeru

```
public static func testViewControllers(
  window: UIWindow,
  controllers: [UIViewController]
)
```

Druhá metoda, která je znázorněna ve zdrojovém kódu 12.4, slouží jako

automatická slideshow definovaných kontrolerů.

Zdrojový kód 12.4: Ukázka signatury metody pro mockování kontrolerů pomocí ControlerMockeru s automatickou slideshow

```
public static func testViewControllersWithTimer(
  window: UIWindow,
  controllers: [UIViewController],
  delay: NSTimeInterval = 5
)
```

Jak lze vyčíst ze signatur metod (viz zdrojový kód 12.3 a 12.4), obě metody přijímají jako parametr okno aplikace a pole ***UIViewControllerů***. Druhá metoda poté obsahuje ještě volitelný parametr pro možnost volby času přepnutí ve slideshow.

Vzhledem k tomu, že se ControllerMocker volá přímo z ***AppDelegate***, je ideální pro jeho volání vytvořit nový Xcode target, aby zůstala aplikace nedotčena, jestliže plánuji dlouhodobé testování. Pokud chci samozřejmě pouze narychlo vyzkoušet nějaký kontroler, můžu použít ControllerMocker přímo v hlavním targetu nebo jakémkoliv jiném.

### 12.3.1 Ukázka inicializace ControllerMockeru

Jak již bylo zmíněno, metody pro použití ControllerMockeru jsou implementovány jako rozšíření ***UIApplicationDelegate***, takže samotné volání těchto metod probíhá právě v ***AppDelegate*** a to konkrétně v metodě ***didFinishLaunchingWithOptions***. Ukázka inicializace v metodě ***didFinishLaunchingWithOptions*** je vyobrazena ve zdrojovém kódu 12.5.

## 12.4 Možnosti pomocného nástroje

Možnosti a schopnosti pomocného nástroje (ControllerMockeru) budou demonstrovány na testovací aplikaci představené v kapitole 2.2. Konkrétně budeme chtít otestovat zobrazení detailu článku pro různě dlouhý obsah článku. To znamená, že v řeči ControllerMockeru budeme mockovat kontrolery pro detail článku.

### 12.4.1 Tvorba mocků

Abychom mohli mockovat detail článku, je nejprve zapotřebí vytvořit mock samotného článku, který se má zobrazit. Tvorba mocku článku v tomto konkrétním případě je vyobrazena ve zdrojovém kódu 12.6. Stejným způsobem si vytvoříme více mocků, které se budou lišit délkou obsahu a doménou.

Zdrojový kód 12.5: Ukázka inicializace ControllerMockeru

```
let myViewController1 = MyViewController(MyMock())
let myViewController2 = MyViewController(MyMock2())

let controllers = [
  myViewController1,
  myViewController2
]

AppDelegate.testViewControllers(
  window!,
  controllers: controllers
)
```

Zdrojový kód 12.6: Ukázka inicializace ControllerMockeru

```
let imgUrl = ControllerMockerDummy.getUrlToDummyImage()
let date = Date.getDateFromString("2016-03-06T18:11")

let mock = Article.MR_createEntity()
mock.nid = 123
mock.articleUrl = "http://www.example.com/article/123"
mock.author = "Marek Měchura"
mock.changed = date
mock.content = "Content of Article 123"
mock.created = date
mock.domainId = 1
mock.perex = "Article 123 - perex perex perex perex
perex perex perex perex perex perex perex perex perex"
mock.previewUrl = imgUrl
mock.thumbnailUrl = imgUrl
mock.title = "Article Mock 123"
```

### 12.4.2 Inicializace kontrolerů s mocky

Testovací mocky článků jsme vytvořili v kapitole 12.4.1, nyní je nutné tyto mocky injektovat do kontrolerů, které chceme testovat. Použije se zde tedy stejná technika, která je popsaná v kapitole 11.4.1.2. Ukázka injektování v tomto konkrétním případě je znázorněna ve zdrojovém kódu 12.7.

Zdrojový kód 12.7: Ukázka inicializace ControllerMockeru

```
let articleLoader = ArticleLoader()
let viewModel = CdrArticleViewModel(
  articleLoader: articleLoader
)

let articleVC = CdrArticleViewController(
  article: mock,
  viewModel: viewModel
)
let articleVC2 = DiitArticleViewController(...)
let articleVC3 = CdrArticleViewController(...)
let articleVC4 = DiitArticleViewController(...)
```

### 12.4.3 Inicializace ControllerMockeru

Po vytvoření všech kontrolerů způsobem demonstrovaným ve zdrojovém kódu 12.7 můžeme přejít k inicializaci ControllerMockeru. Ta probíhá stejným způsobem, jako je to znázorněno ve zdrojovém kódu 12.5.

Pro mockování kontrolerů ze zdrojového kódu 12.7 by vypadalo spuštění ControllerMockeru tak, jak je to ukázáno ve zdrojovém kódu 12.8.

Zdrojový kód 12.8: Ukázka mockování kontrolerů pomocí ControllerMockeru

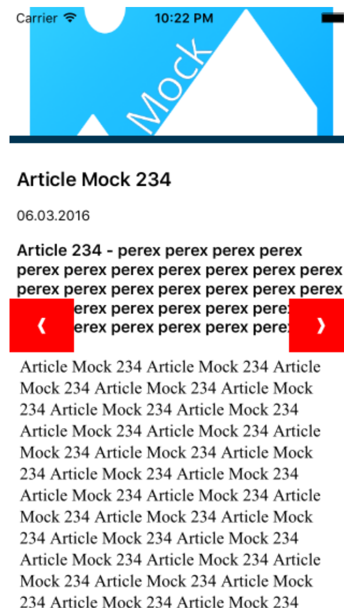
```
let controllers = [
  articleVC, articleVC2, articleVC3, articleVC4
]

AppDelegate.testViewControllers(
  window!,
  controllers: controllers
)
```

Pokud se aplikace zkompiluje a spustí přesně tak, jak je vyobrazeno ve zdrojovém kódu 12.8, výsledná spuštěná aplikace bude na obrázku 12.2.

### 12.4.4 Ovládací prvky

Po spuštění ControlleruMockeru, kterému jsou předány kontrolery, které se mají mockovat, je potřeba nějakým způsobem s těmito kontrolery pracovat a ovládat je. K tomu slouží přepínací tlačítka.

Obrázek 12.2: Ukázka mockovaného kontroleru pomocí ControllerMockeru




Obrázek 12.3: Ukázka druhého mockovaného kontroleru pomocí ControllerMockeru

#### 12.4.4.1 Přepínací tlačítka

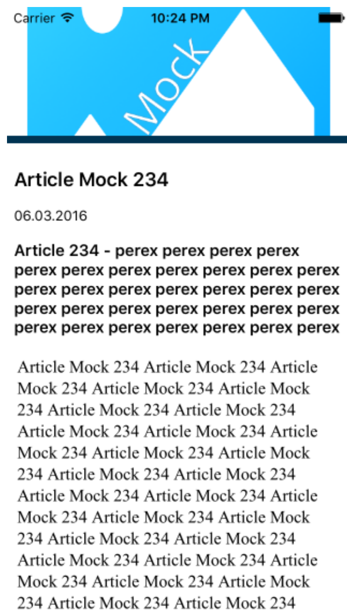
Z obrazovek na obrázcích 12.2 a 12.3 je vidět, že kromě mockovaného kontroleru se zobrazila i dvě ovládací tlačítka. Tato tlačítka slouží pro přepínaní mezi mockovanými kontrolery. Jakmile již jsme na prvním nebo posledním mockovaném kontroleru, příslušné tlačítko se zprůhlední a nejde na něj kliknout (viz obrázek 12.2).

#### 12.4.4.2 Skrytí tlačítek

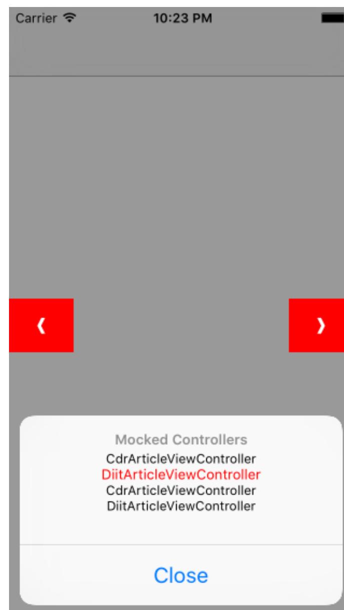
Mnohdy se může přihodit, že při testování kontroleru budeme chtít zkontrolovat nějaký zobrazený element, který je zrovna skrytý pod přepínacím tlačítkem ControllerMockeru. Pro tyto případy je možné dlouhým stiskem přepínacího tlačítka tato tlačítka skrýt. Tlačítka zůstanou skryta až do uvolnění stisklého tlačítka (viz obrázek 12.4).

#### 12.4.4.3 Zobrazení seznamu mockovaných kontrolerů

ControllerMocker také umožňuje zobrazení seznamu všech mockovaných kontrolerů. Tento seznam se zobrazí po dvojitém kliknutí na přepínací tlačítko. Výsledek je znázorněn na obrázku 12.5.




Obrázek 12.4: Ukázka skrytých tlačítek ControllerMockeru




Obrázek 12.5: Ukázka zobrazení seznamu všech mockovaných kontrolerů v ControllerMockeru

V zobrazeném seznamu mockovaných kontrolerů je barevně zvýrazněn kontroler, na kterém se v ControllerMockeru aktuálně nacházíme. Z obrázku 12.5 je vidět, že se aktuálně nacházíme na druhém kontroleru v pořadí. Po zavření přehledu se opět zobrazí aktuálně mockovaný kontroler.

### 12.4.5 Testovací obrázek

Při testování se nejednou dostaneme do situace, kdy potřebujeme otestovat nějakou obrazovku, kde je nutný obrázek nebo více obrázků pro otestování, že se vše zobrazuje, jak má. Také je nezbytné otestovat, že má obrázek korektní velikost, je správně ořezán, zmenšen, umístěn na správném místě atd.

ControllerMocker se snaží tuto situaci ulehčit a poskytuje testovací obrázek, který je možné využít při mockování kontrolerů, které potřebují pro otestování obrázek. Této vlastností bylo využito při mockování detailu článku ve zdrojovém kódu 12.6. Samotné získání obrázku je demonstrováno ve zdrojovém kódu 12.9.

Zdrojový kód 12.9: Ukázka využití testovacího obrázku ControllerMockeru

```
let imgUrl = ControllerMockerDummy.getUrlToDummyImage()
```

Z názvu metody je vidět, že vrací lokální URL na obrázek, který je možno poté využít podle potřeby.

## 12.5 Přínos

V úvodu této kapitoly byla zmíněna motivace a důvody pro tvorbu tohoto pomocného nástroje a co konkrétně by měl ulehčit. Dle mého názoru tento pomocný nástroj řeší vše, co si předsevzal za cíl.

### 12.5.1 Přínos pro testery

Při použití ControlleruMockeru odpadá pro testery a vývojáře nutnost proklikávat se celou hierarchií aplikace pro otestování nějaké konkrétní obrazovky. Konkrétní obrazovku je možné testovat velice jednoduše a rychle.

Další výhodou je to, že díky možnosti definovat seznam více kontrolerů, které chceme mockovat, lze velice jednoduše mockovat jeden kontroler pro všechny možné druhy dat a jednoduše mezi nimi přepínat. Na první pohled tedy lze vidět, jak se konkrétní obrazovka vykreslí pro různé druhy dat. Máme možnost tak například jednoduchým způsobem otestovat, jak se bude rozložení obrazovky chovat v závislosti na délce vstupních dat, jako je kupříkladu dlouhý text, krátký text nebo pokud se třeba nemá zobrazit text žádný.

Všechny aspekty a vlastnosti kontrolerů si tester nebo programátor vytváří ještě před mockováním, takže má vše plně pod svou kontrolou.

### 12.5.2 Přínos pro automatizované testy

Automatizované testy velice zefektivňují práci testera, ale z hlediska budoucího vývoje aplikace je nutné testy udržovat v rozumném stavu. V drtivé většině případů se každá aplikace s časem vyvíjí, mění a mnohdy se přidávají nové funkčnosti. Na tyto aspekty musí samozřejmě reagovat i testy, aby „držely krok“ s aplikací.

Díky použití ControllerMockeru odpadá při testování nutnost definice spousty kroků pro doklikání se na správnou obrazovku, kterou je potřeba testovat. Naopak, díky tomu, že máme k dispozici rovnou obrazovky, které chceme testovat, odpadá spousta testovacího kódu pro proklikání se k této obrazovce.

Další velká výhoda je v tom, že jestliže se časem změní struktura aplikace a testovaná obrazovka bude dostupná jinde v hierarchii aplikace, ale jinak se nezmění, není nutné měnit test, jelikož test bude stále testovat stejnou obrazovku.

Ovládací tlačítka ControllerMockeru jsou samozřejmě při UI testech také přístupná, takže je možné v rámci testu testovat všechny mockované kontrolery. To znamená, že v rámci jednoho testu můžeme definovat otestování libovolné obrazovky pro všechny možné velikosti dat atd.

# Závěr

V průběhu práce bylo otestováno několik testovacích nástrojů. Všechny nástroje byly analyzovány z hlediska funkčnosti, použitelnosti a také obecně podle složitosti a pohodlnosti práce s daným nástrojem. Pro nativní testovací nástroj a nástroj na bázi Selenia byly napsány testovací scénáře a také samotné testy k těmto scénářům. Testy byly implementovány jak pro nativní, tak i pro hybridní aplikaci. V rámci práce bylo také analyzováno mockování testů a byla realizována implementace pomocného nástroje pro testování konkrétních obrazovek aplikace.

Jedním z výsledků práce je analýza konkrétních testovacích nástrojů. U většiny testovacích nástrojů je i příklad použití a jsou zmíněny také problémy, se kterými jsem se při práci s daným nástrojem potýkal. Výsledkem analýzy je soupis slabých a silných stránek daného nástroje a to včetně doporučení, v jakém případě je vhodné použít příslušný testovací nástroj.

Dalším z výsledků práce je detailní srovnání testování pomocí nativního nástroje (UI Testing in Xcode) a nástroje na bázi Selenia (Appium). Nástroje jsou porovnávány z různých pohledů, například rychlost, pohodlnost, možnosti testování atd. Součástí tohoto srovnání je i ukázková sada testů, kde jsou vždy řešeny stejné úkoly v obou nástrojích, takže jsou hned vidět rozdíly při psaní testů v příslušných nástrojích.

Práce obsahuje také úvod a praktickou ukázku mockování ve Swiftu a to včetně praktické ukázky, jak lze do aplikace předat parametry při použití UI testování v Xcode.

Praktickým výstupem této práce je implementovaný pomocný nástroj pro testování konkrétních obrazovek aplikace. Tento nástroj umožňuje definovat seznam obrazovek, které se mají testovat. Poté umožňuje pomocí pomocného nástroje tyto obrazovky zobrazit a přepínat mezi jednotlivými obrazovkami. Nástroj také umí zobrazit seznam všech testovaných obrazovek atd.

V rámci budoucího rozvoje existuje určitě ještě velký prostor pro práci na implementovaném pomocném nástroji, který by do budoucna mohl například usnadnit tvorbu některých kontrolerů nebo přidat možnost zobrazení nějakých

ladících informací o konkrétní obrazovce.

Z mého pohledu byla tato práce velice přínosná, jelikož jsem se naučil pracovat se spoustou testovacích nástrojů, o kterých jsem si udělal celkem solidní obrázek, co se týče jejich silných a slabých stránek. Zároveň jsem si prakticky vyzkoušel práci s těmito nástroji, takže mám reálnou zkušenost, jak moc je, nebo není složité pracovat s těmito nástroji. Dále jsem si o něco více ujasnil terminologii ohledně používání mocků při testování a to včetně tvorby konkrétního pomocného mockovacího nástroje.

# Literatura

[1] Myers, G. J.: *The Art of Software Testing*. Hoboken, N.J.: John Wiley & Sons, druhé vydání, 2004, ISBN 0-471-46912-2.

[2] Martin Fowler: *Test Driven Development*. [cit. 2016-04-29]. Dostupné z: http://martinfowler.com/bliki/TestDrivenDevelopment.html

[3] Singh, Y.: *Software Testing*. New York: Cambridge University Press, první vydání, 2012, ISBN 978-1-107-01296-7.

[4] Apple Inc.: *Automate UI Testing in iOS*. [cit. 2016-04-08]. Dostupné z: https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UIAutomation.html

[5] Alex Vollmer: *Tuneup JS*. [cit. 2016-04-09]. Dostupné z: http://www.tuneupjs.org/

[6] Apple Inc.: *UI Testing in Xcode*. [cit. 2016-04-29]. Dostupné z: http://devstreaming.apple.com/videos/wwdc/2015/406o0doszwo8r15m/406/406_ui_testing_in_xcode.pdf

[7] *SeleniumHQ Browser Automation*. [cit. 2016-04-05]. Dostupné z: http://www.seleniumhq.org

[8] Avasarala, S.: *Selenium WebDriver practical guide*. Birmingham: Packt Publishing, první vydání, 2014, ISBN 978-1-78216-885-0.

[9] *SeleniumHQ Browser Automation*. [cit. 2016-04-05]. Dostupné z: http://www.seleniumhq.org/download/

[10] *Migrating From Selenium RC to Selenium WebDrive*. [cit. 2016-04-05]. Dostupné z: http://www.seleniumhq.org/docs/appendix_migrating_from_rc_to_webdriver.jsp

[11] Hans, M.: *Appium Essentials*. Birmingham: Packt Publishing, první vydání, 2015, ISBN 978-1-78439-248-2.

[12] a Aslak Hellesøy, M. W.: *The Cucumber Book: behaviour-driven development for testers and developers*. Dallas, Tex.: The Pragmatic Bookshelf, první vydání, 2012, ISBN 978-1-934356-80-7.

[13] Xamarin: *An Overview of Calabash iOS*. [cit. 2016-04-11]. Dostupné z: http://blog.lesspainful.com/2012/03/07/Calabash-iOS/

[14] Xamarin: *Tutorial: Link Calabash in Debug config*. [cit. 2016-04-11]. Dostupné z: https://github.com/calabash/calabash-ios/wiki/Tutorial:-Link-Calabash-in-Debug-config

[15] Xamarin: *Tutorial: Calabash config*. [cit. 2016-04-11]. Dostupné z: https://github.com/calabash/calabash-ios/wiki/Tutorial:-Calabash-config

[16] Xamarin: *Tutorial: Creating a cal Target*. [cit. 2016-04-11]. Dostupné z: https://github.com/calabash/calabash-ios/wiki/Tutorial:--Creating-a-cal-Target

[17] Xamarin: *Predefined steps*. [cit. 2016-04-11]. Dostupné z: https://github.com/calabash/calabash-ios/wiki/02-Predefined-steps

[18] Pete Hodgson: *Frank lightning talk*. [cit. 2016-04-29]. Dostupné z: http://moredip.github.io/frank_lightning_talk_slides.html

[19] *ios-driver documentation*. [cit. 2016-04-29]. Dostupné z: http://ios-driver.github.io/ios-driver/?page=home

[20] Facebook: *instruments-without-delay*. [cit. 2016-04-30]. Dostupné z: https://github.com/facebookarchive/instruments-without-delay

[21] Spiros Gerokostas: *Tinylog-iOS*. [cit. 2016-04-20]. Dostupné z: https://github.com/binarylevel/Tinylog-iOS

[22] Christophe Coenraets: *Belgian Beer Explorer App built with Ionic Framework*. [cit. 2016-04-27]. Dostupné z: https://github.com/ccoenraets/belgian-beer-explorer-ionic

[23] *Ionic Documentation Overview*. [cit. 2016-04-27]. Dostupné z: http://ionicframework.com/docs/overview/

[24] Meszaros, G.: *XUnit test patterns: refactoring test code*. Upper Saddle River: Addison-Wesley, první vydání, 2007, ISBN 978-0-13-149505-0.

[25] Martin Fowler: *Mocks Aren't Stubs*. [cit. 2016-04-23]. Dostupné z: http://martinfowler.com/articles/mocksArentStubs.html

[26] Szczepan Faber: *Mockito - FAQ*. [cit. 2016-04-23]. Dostupné z: https://github.com/mockito/mockito/wiki/FAQ

[27] Martin Fowler: *Continuous Integration*. [cit. 2016-04-23]. Dostupné z: http://martinfowler.com/articles/continuousIntegration.html

[28] Erik Dörnenburg: *OCMock and Swift*. [cit. 2016-04-23]. Dostupné z: http://ocmock.org/swift/

[29] Jon Reid: *OCMockito*. [cit. 2016-04-23]. Dostupné z: https://github.com/jonreid/OCMockito

[30] Luis Solano: *Nocilla*. [cit. 2016-04-23]. Dostupné z: https://github.com/luisobo/Nocilla

[31] *CocoaPods*. [cit. 2016-04-24]. Dostupné z: https://cocoapods.org/

Příloha **A**

# Scénáře pro testování nativní aplikace

## A.1 Testovací scénář 1

**TS 1** Přidání seznamu úkolů, přidání úkolu a označení úkolu jako hotový.

**TC 1** Přidání seznamu úkolů.

**S 1** Klikněte na tlačítko přidání nového úkolu (+).

***Ex 1*** *Zobrazení obrazovky pro přidání názvu seznamu úkolů a jeho barvy.*

**S 2** Zadejte název „TODO“, vyberte červenou barvu a klikněte na tlačítko **Save**.

***Ex 2*** *Zobrazení obrazovky pro přidání úkolů do právě vytvořeného seznamu úkolů s názvem TODO.*

**S 3** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 3*** *Zobrazení všech seznamů úkolů i s nově přidaným seznamem úkolů s název TODO.*

**TC 2** Přidání úkolu do seznamu úkolů.

**S 1** Klikněte na nově přidaný seznam úkolů s názvem TODO.

***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*

**S 2** Do názvu úkolu zadejte „Write some UI tests“ a potvrďte tlačítkem **Done**.

***Ex 2*** *Zobrazení stejné obrazovky, ale s přidaným úkolem „Write some UI tests“.*

**S 3** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 3*** *Zobrazení všech seznamů úkolů i s TODO, který má jeden přidaný úkol.*

**TC 3** Označení úkolu jako hotový.

**Krok 1** Klikněte seznam úkolů s názvem TODO.

***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*

**Krok 2** Označte jediný přidaný úkol jako hotový.

***Ex 2*** *Zobrazení stejné obrazovky se zaškrtnutým úkolem.*

**S 3** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 3*** *Zobrazení všech seznamů úkolů i s TODO, který nemá žádný přidaný úkol.*

## A.2 Testovací scénář 2

**TS 2** Práce s úkoly a archivace.

**TC 1** Přidání seznamu úkolů.

**S 1** Klikněte na tlačítko přidání nového úkolu (+).

***Ex 1*** *Zobrazení obrazovky pro přidání názvu seznamu úkolů a jeho barvy.*

**S 2** Zadejte název „Shopping list“, vyberte zelenou barvu a klikněte na tlačítko **Save**.

***Ex 2*** *Zobrazení obrazovky pro přidání úkolů do právě vytvořeného seznamu úkolů s názvem „Shopping list“.*

**S 3** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 3*** *Zobrazení všech seznamů úkolů i s nově přidaným seznamem úkolů s název „Shopping list“.*

**TC 2** Přidání tří úkolů do seznamu úkolů.

**S 1** Klikněte na nově přidaný seznam úkolů s názvem „Shopping list“.

***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*

**S 2** Do názvu úkolu zadejte „iPhone“ a potvrďte tlačítkem **Done**.

***Ex 2*** *Zobrazení stejné obrazovky, ale s přidaným úkolem „iPhone“.*

**Krok 3** Do názvu úkolu zadejte „iPad“ a potvrďte tlačítkem **Done**.

***Ex 3*** *Zobrazení stejné obrazovky, ale s přidaným úkolem „iPad“.*

**Krok 4** Do názvu úkolu zadejte „MacBook“ a potvrďte tlačítkem **Done**.

***Ex 4*** *Zobrazení stejné obrazovky, ale s přidaným úkolem „MacBook“.*

**S 5** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 5*** *Zobrazení všech seznamů úkolů i s „Shopping list“, který má tři přidané úkoly.*

**TC 3** Označení prostředního úkolu jako hotový a jeho přesun na konec.

**S 1** Klikněte seznam úkolů s názvem „Shopping list“.

***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*

**S 2** Označte prostřední úkol (iPad) jako hotový.

***Ex 2*** *Zobrazení stejné obrazovky se zaškrtnutým úkolem.*

**S 3** Klikněte na tlačítko **Edit**.

***Ex 3*** *Zobrazení nových tlačítek u jednotlivých úkolů.*

**S 4** Chytněte prostřední úkol (iPad) a přesuňte ho dolů.

***Ex 4*** *Úkoly jsou v pořadí MacBook, iPhone a iPad.*

**S 5** Klikněte na tlačítko **Done**.

***Ex 5*** *Zmizely možnosti úpravy u jednotlivých úkolů.*

**S 6** Klikněte na tlačítko **My Lists** pro návrat zpět.

***Ex 6*** *Zobrazení všech seznamů úkolů i s „Shopping list“, který má přidané dva úkoly.*

**TC 4** Archivace poslední položky (iPad).

**S 1** Klikněte na seznam úkolů s názvem „Shopping list“.

***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*

**S 2** Klikněte na tlačítko **Edit**.

***Ex 2*** *Zobrazení nových tlačítek u jednotlivých úkolů.*

**S 3** U poslední položky (iPad) klikněte na červené tlačítko (-).

***Ex 3*** *Zobrazení tlačítka **Archive** u položky iPad.*

**S 4** Klikněte na tlačítko **Archive**.

***Ex 4*** *Zmizela poslední položka v seznamu, takže je jich o jednu méně.*

**S 5** V tab baru klikni na ikonu archivace.

***Ex 5*** *Zobrazení archivace s položkou iPad.*

## A.3 Testovací scénář 3

**TS 3** Přejmenování, archivace a smazání listu úkolů.

**TC 1** Přidání seznamu úkolů.

**P** Jsou přidány dva listy úkolů. Jeden s názvem TODO a druhý libovolný.

**S 1** Na seznamu úkolů udělejte swipe vlevo.

***Ex 1*** *Zobrazí se tlačítka **Edit** a **Archive**.*

**S 2** Klikněte na tlačítko **Edit**.

***Ex 2*** *Zobrazí se obrazovka s možností úpravy jména seznamu úkolů a barvy.*

**S 3** Přepište „TODO“ na „todo to remove“ a klikněte na tlačítko **Save**.

***Ex 3*** *Zobrazí se seznam listů se seznamem „todo to remove“ a bez „TODO“.*

**TC 2** Archivace seznamů úkolů.

**S 1** Klikněte na tlačítko **Edit**.

***Ex 1*** *Zobrazí se možnosti úpravy položek.*

**S 2** Klikněte na červené tlačítko (-).

***Ex2*** *Zobrazí se tlačítka **Edit** a **Archive**.*

**S 3** Klikněte na tlačítko **Archive**.

***Ex 3*** *Zmizel seznam „todo to remove“.*

**TC 3** Smazaní seznamu úkolů.

**S 1** V tab baru klikněte na ikonu archivace.

***Ex 1*** *Zobrazení archivace seznamů včetně „todo to remove“ seznamu.*

**S 2** Na položce „todo to remove“ udělejte swipe vlevo.

***Ex 2*** *Zobrazí se tlačítka **Delete** a **Restore**.*

**S 3** Klikněte na tlačítko **Delete**.

***Ex 3*** *Zmizela položka „todo to remove“ ze seznamu archivovaných listů.*

**S 4** Klikněte na tlačítko **Close**.

***Ex 4*** *Zobrazení seznamů listů bez „todo to remove“.*

## A.4 Testovací scénář 4

**TS 4** Vyhledávání listů úkolů a přidávání/mazání většího množství úkolů.

**TC 1** Přidání seznamu úkolů.

**S 1** Klikněte na tlačítko přidání nového úkolu (+).

***Ex 1*** *Zobrazení obrazovky pro přidání názvu seznamu úkolů a jeho barvy.*

**S 2** Zadejte název „Numbers“ a klikněte na tlačítko **Save**.

***Ex 2*** *Zobrazení obrazovky pro přidání úkolů do právě vytvořeného seznamu úkolů s názvem „Numbers“.*

**S 3** Klikněte na tlačítko **My Lists** pro návrat zpět.

- ***Ex 3*** *Zobrazení všech seznamů úkolů i s nově přidaným seznamem úkolů s název „Numbers“.*
- **TC 2** Vyhledání, zobrazení a přidání úkolů do seznamu úkolu s názvem „Numbers“.
  - **S 1** Do vyhledávacího pole zadejte „Numbers“.
  - ***Ex 1*** *Mezi vyhledanými seznamy by měl být zobrazen i seznam s názvem „Numbers“.*
  - **S 2** Klikněte na list „Numbers“.
  - ***Ex 2*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*
  - **S 3** Do názvu úkolu zadejte „1“ a potvrďte tlačítkem **Done**.
  - ***Ex 3*** *Zobrazena stejná obrazovka s přidaným úkolem.*
  - **S 4** Předchozí krok opakujte pro čísla 1 - 10.
  - ***Ex 4*** *Přidaných 10 položek.*
  - **S 5** Klikněte na tlačítko **My Lists** pro návrat zpět.
  - ***Ex 5*** *Zobrazení všech seznamů úkolů i s „Numbers“, který má deset přidaných položek.*
- **TC 3** Smazání všech přidaných úkolů.
  - **S 1** Klikněte na list „Numbers“.
  - ***Ex 1*** *Zobrazení obrazovky pro přidání úkolů a seznamu všech přidaných úkolů.*
  - **S 2** Klikněte na první položku a udělejte swipe vlevo.
  - ***Ex 2*** *Zobrazí se tlačítko **Archive***
  - **S 3** Klikněte na tlačítko **Archive**.
  - ***Ex 3*** *Zmizení prvního tlačítka.*
  - **S 4** Kroky 2 a 3 opakujte pro všechny položky.
  - ***Ex 4*** *Zmizení všech úkolů.*
  - **S 5** V tab baru klikněte na ikonu archivace.
  - ***Ex 5*** *Zobrazení všech deseti archivovaných úkolů.*
  - **S 6** Klikněte na první položku a udělejte swipe vlevo.
  - ***Ex 6*** *Zobrazí se tlačítko **Delete** a **Restore**.*
  - **S 7** Klikněte na tlačítko **Delete**.
  - ***Ex 7*** *Odstranění zvolené položky.*
  - **S 8** Opakujte kroky 6 a 7 pro všechny položky.
  - ***Ex 8*** *Odstraněny všechny položky.*
  - **S 9** Klikněte na tlačítko **Close**.
  - ***Ex 9*** *Zobrazení prázdného seznamu úkolu „Numbers“.*
  - **S 10** Klikněte na tlačítko **My Lists** pro návrat zpět.
  - ***Ex 10*** *Zobrazení všech seznamů úkolů i s „Numbers“, který nemá žádnou přidanou položku.*

## A.5 Testovací scénář 5

**TS 5** Úprava použité velikosti fontu.

**TC 1** Zapnutí použití systémové velikosti fontu.

**S 1** Klikněte na ikonu ozubeného kola.

***Ex 1*** *Zobrazení obrazovky s nastavením aplikace.*

**S 2** Klikněte na položku **Text size**.

***Ex 2*** *Zobrazení nastavení velikosti fontu.*

**S 3** Přepněte přepínač pro zapnutí systémového fontu.

***Ex 3*** *Aktivní přepínač systémového fontu.*

**S 4** Klikněte na tlačítko **Settings** pro návrat zpět.

***Ex 4*** *Zobrazí se nastavení aplikace a u položky **Text size** bude napsáno „System font“.*

**S 5** Klikněte na tlačítko **Done** pro návrat a úvodní obrazovku.

***Ex 5*** *Úvodní obrazovka.*

**TC 1** Vypnutí použití systémové velikosti fontu.

**S 1** Klikněte na ikonu ozubeného kola.

***Ex 1*** *Zobrazení obrazovky s nastavením aplikace.*

**S 2** Klikněte na položku **Text size**.

***Ex 2*** *Zobrazení nastavení velikosti fontu.*

**S 3** Přepněte přepínač pro zapnutí systémového fontu.

***Ex 3*** *Neaktivní přepínač systémového fontu.*

**S 4** Klikněte na tlačítko **Settings** pro návrat zpět.

***Ex 4*** *Zobrazí se nastavení aplikace a u položky **Text size** bude napsáno „17“.*

**S 5** Klikněte na tlačítko **Done** pro návrat a úvodní obrazovku.

***Ex 5*** *Úvodní obrazovka.*

Příloha **B**

# Scénáře pro testování hybridní aplikace

## B.1 Testovací scénář 1

**TS 1** Vyhledávání piv podle názvu.

**TC 1** Vyhledání piva.

**S 1** Klikněte do pole pro vyhledávání.

***Ex 1*** *Aktivní pole pro vyhledávání.*

**S 2** Do vyhledávacího pole zadejte „400“.

***Ex 2*** *Ve výsledku zobrazení je pivo s názvem „400“.*

**S 3** Klikněte na pivo s názvem „400“.

***Ex 3*** *Zobrazení detailu piva.*

**S 4** Klikněte na tlačítko „Back“.

***Ex 4*** *Zobrazení výsledku vyhledávání s pivem „400“.*

**S 5** Klikněte na křížek pro zrušení vyhledávání.

***Ex 5*** *Zobrazení seznamu všech piv.*

## B.2 Testovací scénář 2

**TS 2** Vyhledávání podle pivovaru.

**TC 1** Vyhledávání piv z pivovaru.

**S 1** Klikněte do pole pro vyhledávání.

***Ex 1*** *Aktivní pole pro vyhledávání.*

**S 2** Do vyhledávacího pole zadejte název pivovaru „Brasserie De Bouillon“.

***Ex 2*** *Ve výsledku jsou zobrazeny piva z pivovaru „Brasserie De Bouillon“.*

**S 3** Klikněte na pivo s názvem „Airborne“.

***Ex 3*** *Zobrazený detail piva „Airborne“.*

**TC 2** Zobrazení piv pivovaru z detailu piva.

**S 1** V detailu piva „Airborne“ klikněte na tlačítko s názvem pivovaru „Brasserie De Bouillon“.

***Ex 1*** *Zobrazení seznamu všech piv z pivovaru „Brasserie De Bouillon“.*

**S 2** Smažte z vyhledávání text „Brasserie De Bouillon“.

***Ex 2*** *Zobrazení všech piv.*

## B.3 Testovací scénář 3

**TS 3** Vyhledávání podle vlastností.

**TC 1** Vyhledávání piv podle vlastností.

**S 1** Klikněte do pole pro vyhledávání.

***Ex 1*** *Aktivní pole pro vyhledávání.*

**S 2** Do vyhledávacího pole zadejte vlastnosti „blonde, high fermentation, triple“.

***Ex 2*** *Ve výsledku by mělo být jen pivo „Bersalis Tripel“ s vlastnostmi „blonde“, „high fermentation“ a „triple“.*

**S 3** Klikněte na pivo s názvem „Bersalis Tripel“.

***Ex 3*** *Zobrazený detail piva „Bersalis Tripel“.*

**TC 2** Zobrazení piv podle vlastnosti z detailu piva.

**S 1** V detailu piva „Bersalis Tripel“ zkontrolujte existenci vlastností „blonde“, „high fermentation“ a „triple“.

***Ex 1*** *V detailu piva „Bersalis Tripel“ jsou tlačítka s vlastnostmi „blonde“, „high fermentation“ a „triple“.*

**S 2** Klikněte na vlastnost „high fermentation“.

***Ex 2*** *Zobrazení všech piv s vlastností „high fermentation“.*

**S 3** Klikněte na křížek pro zrušení vyhledávání.

***Ex 3*** *Zobrazení seznamu všech piv.*

## B.4 Testovací scénář 4

**TS 4** Procenta alkoholu.

**TC 1** Kontrola procent alkoholu ve výpisu a v detailu piva.

**S 1** Klikněte do pole pro vyhledávání.

***Ex 1*** *Aktivní pole pro vyhledávání.*

**S 2** Do vyhledávacího pole zadejte název piva„Bersalis Tripel“.

***Ex 2*** *Ve výsledku je zobrazeno pivo „Bersalis Tripel“ s obsahem alkoholu 9.5%.*

**S 3** Klikněte na pivo s názvem „Bersalis Tripel“.

***Ex 3*** *Zobrazený detail piva „Bersalis Tripel“.*

**S 4** V detailu piva „Bersalis Tripel“ zkontrolujte procenta alkoholu.

***Ex 4*** *V detailu „Bersalis Tripel“ zobrazeno 9,5% alkoholu.*

**S 5** Klikněte na tlačítko „Back“.

***Ex 5*** *Zobrazení výsledku vyhledávání s pivem „Bersalis Tripel“.*

**S 6** Klikněte na křížek pro zrušení vyhledávání.

***Ex 6*** *Zobrazení seznamu všech piv.*

Příloha C

# Ukázky testů

## C.1 UI Testing in Xcode - nativní aplikace

Zdrojový kód C.1: Ukázka struktury testovacího scénáře A.1 pro **nativní** aplikaci v **Xcode**

```
// Test Scenario 1
class AddAndMarkTaskTestScenario: XCTestCase {
                                ⋮
  func testAddAndMarkTaskTestScenario() {
    self.addTaskListTestCase()
    self.addTaskToTaskListTestCase()
    ⋮
  }

  // Test Case 1
  func addTaskListTestCase() {
     implementation in source code C.2
  }

  // Test Case 2
  func addTaskToTaskListTestCase() {
    implementation in source code C.3
  }
                                  ⋮
}
```

Zdrojový kód C.2: Ukázka implementace TC 1 testovacího scénáře A.1 pro **nativní** aplikaci v **Xcode**

```
// Test Case 1
func addTaskListTestCase() {
  // ##### step 1 #####
  let app = XCUIApplication()
  let numberOfTaskLists = app.tables.cells.count

  // ##### step 2, 3 #####
  TaskHelper.addTaskList(app, name: "TODO",
    color: "red"
  )
  let taskListItems = app.tables.cells
  XCTAssertEqual(taskListItems.count,
    (numberOfTaskLists + 1)
  )
}
```

## C.2 Appium - nativní aplikace

## C.3 UI Testing in Xcode - hybridní aplikace

## C.4 Appium - hybridní aplikace

Zdrojový kód C.3: Ukázka implementace TC 2 testovacího scénáře A.1 pro **nativní** aplikaci v **Xcode**

```
// Test Case 2
func addTaskToTaskListTestCase() {
  // ##### step 1 #####
  let app = XCUIApplication()
  let tablesQuery = app.tables
  let taskListElement = tablesQuery
    .childrenMatchingType(.Cell)
    .elementBoundByIndex(0).staticTexts[" "]
  taskListElement.tap()

  // ##### step 2 #####
  tablesQuery.textFields["Add new task"].tap()
  let addTasktextField = tablesQuery
    .otherElements.childrenMatchingType(.TextField)
    .element

  // insert new task
  TaskHelper.pasteStringToElement(addTasktextField,
    stringToPaste: "Write some UI tests"
  )
  addTasktextField.typeText("\r")

  // wait if new task is really added
  let taskItems = tablesQuery
    .childrenMatchingType(.Cell)
  let taskCount = NSPredicate(format: "count == 1")
  expectationForPredicate(taskCount,
    evaluatedWithObject: taskItems,
    handler: nil
  )

  waitForExpectationsWithTimeout(10) { _ in
    // ##### step 3 #####
    app.navigationBars["TODO"]
    .buttons["My Lists"].tap()
  }
}
```

Zdrojový kód C.4: Ukázka struktury testovacího scénáře A.1 pro **nativní** aplikaci v **Appium**

```
# Test Scenario 1
class AddAndMarkTaskTestScenarion(unittest.TestCase):
                              ⋮
  # wrapper for all test cases
  def testAddAndMarkTaskTestScenarion(self):
    self.addTaskListTestCase()
    self.addTaskToTaskListTestCase()

  # Test Case 1
  def addTaskListTestCase(self):
     implementation in source code C.5

  # Test Case 2
  def addTaskToTaskListTestCase(self):
     implementation in source code C.6
                              ⋮
if __name__ == '__main__':
  unittest.main(verbosity=2)
```

Zdrojový kód C.5: Ukázka implementace TC 1 testovacího scénáře A.1 pro **nativní** aplikaci v **Appium**

```
# Test Case 1
def addTaskListTestCase(self):
  taskListingTableXpath = "//.../UIATableView[1]"

  cells = QueryHelper.getTableCells(self.driver,
    taskListingTableXpath
  )
  numberOfCells = len(cells)

  # ##### step 1, 2, 3 #####
  TaskHelper.addTaskList(self.driver, "TODO", "red")

  # check if number of task lists match
  cells = QueryHelper.getTableCells(self.driver,
    taskListingTableXpath
  )
  self.assertEqual(len(cells), (numberOfCells + 1))
```

Zdrojový kód C.6: Ukázka implementace TC 2 testovacího scénáře A.1 pro **nativní** aplikaci v **Appium**

```
# Test Case 2
def addTaskToTaskListTestCase(self):
  # ##### step 1 #####
  todoTaskList = self.driver.find_element_by_xpath(
    "//UIAApplication[1]/.../UIAStaticText[1]"
  )
  todoTaskList.click()

  taskListingTableXpath = "//.../UIATableView[1]"

  cells = QueryHelper.getTableCells(self.driver,
    taskListingTableXpath
  )
  numberOfCells = len(cells)

  # ##### step 2 #####
  addTaskTextField = self.driver.find_element_by_xpath(
    "//UIAApplication[1]/.../UIATextField[1]"
  )
  addTaskTextField.click()
  addTaskTextField.send_keys("Write some UI tests\\n")

  # check if number of tasks match
  cells = QueryHelper.getTableCells(self.driver,
    taskListingTableXpath
  )
  self.assertEqual(len(cells), (numberOfCells + 1))
```

Zdrojový kód C.7: Ukázka struktury testovacího scénáře B.3 pro **hybridní** aplikaci v **Xcode**

```
// Test Scenario 3
class SearchBeerByFeaturesTestScenario: XCTestCase {
                                ⋮
  func testSearchBeerByFeaturesTestScenario() {
    self.searchBeerByFeaturesTestCase()
    self.searchBeerByFeaturesFromBeerDetailTestCase()
  }

  // Test Case 1
  func searchBeerByFeaturesTestCase() {
     implementation in source code C.8
  }

  // Test Case 2
  func searchBeerByFeaturesFromBeerDetailTestCase() {
     implementation in source code C.9
  }
}
```

Zdrojový kód C.8: Ukázka implementace TC 1 testovacího scénáře B.3 pro **hybridní** aplikaci v **Xcode**

```
// Test Case 1
func searchBeerByFeaturesTestCase() {
  let app = XCUIApplication()

  // ##### step 1 #####
  let searchTextField = app.textFields["Search"]
  searchTextField.tap()

  // ##### step 2 #####
  searchTextField
    .typeText("blonde, high fermentation, triple")

  // ##### step 3 #####
  let searchedBeerButton = app
                .staticTexts["Bersalis Tripel"]

  let searchedBeerExists = NSPredicate(
                format: "exists == 1"
                )
  expectationForPredicate(searchedBeerExists,
    evaluatedWithObject: searchedBeerButton,
    handler: nil
  )

  waitForExpectationsWithTimeout(10) { _ in
    searchedBeerButton.tap()

    // check if title match
    let navigationBarTitle = app
                .staticTexts["Bersalis Tripel"]
    XCTAssertTrue(navigationBarTitle.exists)
  }
}
```

Zdrojový kód C.9: Ukázka implementace TC 2 testovacího scénáře B.3 pro **hybridní** aplikaci v **Xcode**

```
// Test Case 2
func searchBeerByFeaturesFromBeerDetailTestCase() {
  let app = XCUIApplication()

  // ##### step 1 #####
  let blondeButton = app.staticTexts["blonde"]
  let highFermentationButton = app
              .staticTexts["high fermentation"]
  let tripleButton = app.staticTexts["triple"]

  XCTAssertTrue(blondeButton.exists)
  XCTAssertTrue(highFermentationButton.exists)
  XCTAssertTrue(tripleButton.exists)

  // ##### step 2 #####
  highFermentationButton.tap()

  // check if title match
  let navigationBarTitle = app
              .staticTexts["Belgian Beer Explorer"]
  XCTAssertTrue(navigationBarTitle.exists)

  // ##### step 3 #####
  TaskHelper.cancelSearch(app)
}
```

Zdrojový kód C.10: Ukázka struktury testovacího scénáře B.3 pro **hybridní** aplikaci v **Appium**

```
# Test Scenario 3
class SearchBeerByFeaturesTS(unittest.TestCase):
                              ⋮
  # wrapper for all test cases
  def testSearchBeerByFeaturesTestScenarion(self):
    self.searchBeerByFeaturesTestCase()
    self.searchBeerByFeaturesFromBeerDetailTestCase()

  # Test Case 1
  def searchBeerByFeaturesTestCase(self):
     implementation in source code C.11

  # Test Case 2
  def searchBeerByFeaturesFromBeerDetailTestCase(self):
     implementation in source code C.12

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Zdrojový kód C.11: Ukázka implementace TC 1 testovacího scénáře B.3 pro **hybridní** aplikaci v **Appium**

```
# Test Case 1
def searchBeerByFeaturesTestCase(self):
  # ##### step 1 #####
  searchFieldXpath = "//.../UIATextField[1]"
  searchField = self.driver.find_element_by_xpath(
    searchFieldXpath
  )
  searchField.click()

  # ##### step 2 #####
  searchField.send_keys(
    "blonde, high fermentation, triple\\n"
  )

  # ##### step 3 #####
  searchedBeerXpath = "//.../UIAStaticText[1]"
  searchedBeer = self.driver.find_element_by_xpath(
    searchedBeerXpath
  )
  searchedBeer.click()

  # check title
  navBarTitle = TaskHelper.getCurrentScreenTitle(
    self.driver
  )
  self.assertEqual(navBarTitle,
    AppiumHelper.AppiumHelper.gnf("Bersalis Tripel")
  )
```

Zdrojový kód C.12: Ukázka implementace TC 2 testovacího scénáře B.3 pro **hybridní** aplikaci v **Appium**

```
# Test Case 2
def searchBeerByFeaturesFromBeerDetailTestCase(self):
  # ##### step 1 #####
  blondeButtonXpath = "//.../UIAStaticText[8]"
  highFermentationButtonXpath = "//.../UIAStaticText[10]"
  tripleButtonXpath = "//.../UIAStaticText[12]"

  blondeButton = self.driver.find_element_by_xpath(
    blondeButtonXpath
  )
  highFermentationButton = self.driver
    .find_element_by_xpath(highFermentationButtonXpath)
  tripleButtonButton = self.driver
    .find_element_by_xpath(tripleButtonXpath)

  self.assertEqual(blondeButton.text,
    AppiumHelper.AppiumHelper.gnf("blonde")
  )
  self.assertEqual(highFermentationButton.text,
    AppiumHelper.AppiumHelper.gnf("high fermentation")
  )
  self.assertEqual(tripleButtonButton.text,
    AppiumHelper.AppiumHelper.gnf("triple")
  )

  # ##### step 2 #####
  highFermentationButton.click()

  # check title
  navBarTitle = TaskHelper.getCurrentScreenTitle(
    self.driver
  )
  self.assertEqual(navBarTitle,
    AppiumHelper.AppiumHelper.gnf("Belgian Beer Explorer")
  )
```

Příloha D

# Seznam použitých zkratek

**WWDC** The Apple Worldwide Developers Conference

**HTTP** Hypertext Transfer Protocol

**CSS** Cascading Style Sheets

**UI** User Interface

**API** Application Programming Interface

**REST** Representational State Transfer

**SUT** System Under Test

**DOC** Depended-On Component

**DI** Dependency Injection

**CI** Continuous Integration

**SDK** Software Development Kit

Příloha E

# Obsah přiloženého CD

```
readme.txt ................................. stručný popis obsahu CD
├── exe ....................... adresář se zkompilovanou pomocnou aplikací
│   └── ControllerMocker.framework
├── src
│   ├── impl .......................... zdrojové kódy implementace a testy
│   │   ├── ControllerMocker ......... implementace pomocného nástroje
│   │   │   ├── ControllerMocker .................... zdrojové kódy aplikace
│   │   │   │   ├── AppDelegateExtensions.swift
│   │   │   │   ├── ControllerMocker.h
│   │   │   │   └── ⋮
│   │   │   ├── ControllerMocker.podspec ........ specifikace pro CocoaPods
│   │   │   └── ControllerMocker.xcodeproj ................ Xcode projekt
│   │   └── tests ..................................... implementace testů
│   │       ├── hybridApp .............. zdrojové kódy testů hybridní aplikace
│   │       │   ├── Appium .................................... Appium testy
│   │       │   │   ├── AppiumHelper.py
│   │       │   │   └── ⋮
│   │       │   └── UITestingInXcode .............. UI Testing in Xcode testy
│   │       │       ├── BeerAlcoholPercentage.swift
│   │       │       └── ⋮
│   │       └── nativeApp ............... zdrojové kódy testů nativní aplikace
│   │           ├── Appium .................................... Appium testy
│   │           │   ├── AddAndMarkTaskUITest.py
│   │           │   └── ⋮
│   │           └── UITestingInXcode .............. UI Testing in Xcode testy
│   │               ├── AddAndMarkTaskUITest.swift
│   │               └── ⋮
│   └── thesis ..................... zdrojová forma práce ve formátu LaTeX
└── text ................................................... text práce
    └── thesis.pdf ........................... text práce ve formátu PDF
```