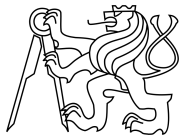
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Analýza nástrojů pro automatické testování mobilních aplikací a her
Student:	Aneta Steimarová
Vedoucí:	Ing. David Buchtela, Ph.D.
Studijní program:	Informatika
Studijní obor:	Informační systémy a management
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1. Sestavte přehled používaných vývojových prostředí pro tvorbu aplikací a her určených pro mobilní zařízení se systémem Android.
2. Proveďte analýzu dostupných nástrojů pro automatické testování mobilních aplikací a her. Rozeberte vlastnosti jednotlivých nástrojů a určete vhodnost jejich použití pro konkrétní vývojové prostředí.
3. Posuďte výhody a rizika nasazení nástrojů automatického testování pro mobilní aplikace a hry z ekonomického hlediska.
4. Po dohodě s vedoucím práce aplikujte získané poznatky na zvolené mobilní aplikaci nebo hře a proveďte zhodnocení.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 28. listopadu 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Analýza nástrojů pro automatické testování mobilních aplikací a her

Aneta Steimarová

Vedoucí práce: Ing. David Buchtela, Ph.D.

10. května 2017

Poděkování

Děkuji panu Ing. Davidu Buchtelovi, Ph.D. za nabídku vedení práce a pomoc s jejím zpracováním. Děkuji také své rodině a přátelům za podporu během studia.

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 10. května 2017 .....

Odkaz na tuto práci

Steimarová, Aneta. *Analýza nástrojů pro automatické testování mobilních aplikací a her*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce řeší problém výběru správného nástroje pro automatické testování mobilních aplikací a her určených pro zařízení s operačním systémem Android. Součástí bakalářské práce je sestavení přehledu aktuálně a nejčastěji používaných programovacích jazyků a vývojových prostředí pro vývoj mobilních aplikací a her určených pro zařízení s operačním systémem Android. Práce představuje i jednotlivé možnosti automatizovaného testování, respektivě jednotlivé nástroje pro automatické testování a to jak ty určené přímo pro vývoj v konkrétním jazyce, tak i ty multiplatformní. Tyto jednotlivé nástroje jsou zároveň posouzeny i z ekonomického hlediska. Bakalářská práce zároveň upozorňuje na nedostatky v oblasti nástrojů pro automatické testování aplikací a her na mobilních zařízení. Na základě zjištěných informací je možné porovnat jednotlivé projekty a určit, které nástroje pro automatické testování pro ně lze využít a které je nepodporují. Získané poznatky jsou následně aplikovány na vybrané aplikace a hry. Hlavním přínosem této práce je pomoci čtenářům zvolit správný nástroj pro automatické testování jejich aplikace nebo hry.

Klíčová slova analýza nástrojů pro automatické testování, automatizované testování, testování softwaru, mobilní aplikace, mobilní hry, vývojové prostředí, testování aplikací pro Android

Abstract

This bachelor thesis solves theproblem with choosing right automated testing tools for mobile applications and games for Andoid devices. The bachelor thesis contains an overwiev of sactual and mostoften used programming languages and environments for mobile game and application for Android deveopment. This thesis also represents options of automated testing - tools for automated testing designed for development in specific programming language and also multiplatform testing tools. These tools are also assessed from economic view. This bachelor thesis points out to shortcomingsin autmated testing tools for mobile applications and games for Android devices section. On the basis of the information foundis possible to compare projects and determine, which tools for automated testing are options to use for the project and which ones are not supported. Colected knowledges are applicated to selected applications and games. The main benefit of this thesis is help to readers to select right tool for automated testing for their applivations or games.

Keywords analysis of testing tools, automated testing, software testing, mobile applications, mobile games, framework, Android application testing

Obsah

Seznam obrázků

Úvod

V dnešní době si již jen málokdo dokáže představit moderní svět bez elektroniky a internetu. Téměř každý vlastní svůj počítač a mobilní telefon, případně i tablet. Většina z nás má v mobilním telefonu alespoň jednu aplikaci nebo hru. Téma mobilních aplikací a her se stalo jedním z nejaktuálnějších témat – téměř každá společnost má svou aplikaci, která je vytvořena přesně pro jejich potřeby a jejich zákazníci tak mají snadný přístup k jejich produktům. Kromě těchto aplikací existují aplikace pro každodenní potřeby – kalendáře, kalkulačky, různí pomocníci při sportu, hubnutí atd. . Aplikací je celá řada a ani hry v tomto odvětví rozhodně nezůstávají pozadu.

Existuje hned několik možností, jak takovou aplikaci nebo hru vyvíjet, ale lidé, kteří aplikaci nebo hru jen používají, si kolikrát ani nedokáží představit, co za jejím vývojem stálo a kolik práce bylo potřeba vynaložit, než se ta jednoduchá aplikace dostala do jejich chytrého telefonu.

Ten, kdo do vývoje aplikací či her vidí, ví, že samotný vývoj není jediné, co za dobrou aplikací nebo hrou stojí. Aplikaci je potřeba nejen vyvinout, ale také řádně otestovat. U jednodušších aplikací a her toto často řeší pouze skupinka testerů, kteří testují, zda je aplikace stabilní a zda vše funguje, jak by mělo. Jiné firmy využijí automatizovaných testů, které testy provádí za ně.

Význam tématu

Nástrojů pro automatizované testování existuje v současné době mnoho, ale vybrat si ten správný pro konkrétní případ již může být složitější. Například pro některé frameworky nástroje pro automatizované testování vůbec neexistují a musí si je programátor vytvořit sám, nebo využít nástojů, které jsou označovány jako multiplatformní. Nástrojů, které existují je celá řada a zatím není k dispozici mnoho zdrojů, které by tyto nástroje porovnávaly, a programátor si tak mohl zvolit takový nástroj, který bude právě pro jeho projekt vhodný. A právě tímto problémem se má bakalářská práce zabývá – analyzuje

jednotlivé nástroje pro automatické testování mobilních aplikací a her pro zařízení s operačním systémem Android, určuje jejich vhodnost pro jednotlivé projekty a zkoumá i přívětivost uživatelského rozhraní vybraných prostředí pro automatické testování.

Motivace

Své téma jsem si zvolila, abych vyřešila otázku problematiky automatického testování mobilních aplikací a her pro Android a pomohla tak vývojářům mobilních aplikací a her pro Android s volbou vhodného nástroje pro automatické testování pro daný projekt. Vzhledem k tomu, že v současné době je na trhu mnoho operačních systémů, rozhodla jsem se ve své práci zaměřit především na operační systém Android.

Kapitola 1

Zaměření

Má práce se zabývá analýzou jednotlivých nástrojů pro automatizované testování mobilních aplikací a her především na zařízeních s operačním systémem Android a jejich porovnáním. Některé poznatky vychází z jiných závěrečných prací. Vzhledem k tomu, že vývoj operačních systémů pro mobilní zařízení, stejně jako vývoj mobilních aplikací a her jde stále vpřed, jsou velké části těchto prací již neaktuální.

Ve své práci se zabývám analýzou těchto nástrojů, zkoumám, pro které projekty jsou vhodné a pro které se nehodí – ať už z pohledu zvoleného programovacího jazyka, vývojového prostředí, nebo využití aplikace či hry. V bakalářské práci lze najít i doporučení, kdy automatické testování pro daný projekt nasadit a kdy stačí pouze manuální testování skupinou testerů, včetně ekonomických dopadů.

1.1 Cíl rešeršní části práce

Získání přehledu o možných postupech při vývoji aplikací a her určených pro Android, a o již existujících nástrojích pro automatizované testování mobilních aplikací a her. Studium základních pojmů z oblasti (automatizovaného) testování softwaru. Analýza v současnosti dostupných vývojových prostředí a nástrojů pro automatické testování mobilních aplikací a her vytvořených v těchto prostředích.

1.2 Cíl praktické části práce

Prozkoumání stávajících metod pro vývoj aplikací a her a následná analýza vybraných nástrojů z pohledu uživatele. Zhodnocení ekonomické stránky jejich používání. Vytvoření studie o nasazení jednotlivých nástrojů pro automatické testování mobilních aplikací a her pro konkrétní případy aplikací a her.

1.3 Představení struktury práce

Celá práce začíná teoretickým přehledem z oblasti automatického testování softwaru. Vysvětluji zde také jednotlivé pojmy, které ve své práci používám.

Ve své práci nejprve představuji technologie (jednotlivé jazyky a vývojová prostředí), které se aktuálně používají pro vývoj aplikací a her pro zařízení s operačním systémem Android. U každé této technologie je k nalezení její stručný popis.

V následující části své práce pak představuji dostupné nástroje pro automatizované testování jednotlivých aplikací a her. Tato část vychází z prvotní analýzy jednotlivých technologií pro vývoj aplikací a her a u jednotlivých nástrojů pro automatické testování. Posuzuje vhodnost použití dané technologie.

Dále se pak věnuji projektům, pro které nástroje určené pouze pro ně neexistují a hledám vhodné řešení například použitím multiplatformního nástroje. Rozebírám zde vhodnost použití v návaznosti na typ projektu i zvolené technologii vývoje.

V další části zabrousím do ekonomického hlediska použití automatických testů. Na jednotlivých případech ukáži, zda je vhodné automatické testy používat a o kolik se sníží (resp. zvýší) náklady na testování v případě používání automatických testů. Analyzuji i vhodnost použití pro konkrétní projekty podle různých parametrů projektu.

V závěru své práce provádím zhodnocení celé studie.

Kapitola 2

Současný stav řešení dané problematiky

2.1 Programovací jazyky a vývojová prostředí

V současné době je na trhu k dispozici hned několik nástrojů pro vývoj aplikací a her pro mobilní zařízení s operačním systémem Android. Mezi nejpoužívanější patří vývoj v programovacím jazyce Java v prostředí Android Studio, nebo Eclipse.

Vývojář preferující programovací jazyk Lua zvolí za své vývojové prostředí Corona, nebo Gideros, vývojář zaměřený na HTML, CSS a JavaScript jistě uvítá prostředí, které s těmito jazyky pracuje – tedy Phonegap, nebo Titanium.

Pokud chceme aplikaci vyvýjet pod operační systémem Windows, lze využít i prostředí Microsoft Visual Studio. Tuto možnost ocení především milovníci C#.

Kromě vývojových prostředí, u kterých je nutné umět alespoň trochu programovat, existují i takové, kde vývojář nemusí umět žádný programovací jazyk, a i tak se může pustit do vývoje her. Takové prostředí je známé pod jménem Gamesalad.

Pro vývoj her lze využít herní engine Unity nebo Unreal Engine. Tyto možnosti jsou vhodné především v případě, jedná-li se o 3D hry.

2.2 Možnosti řešení funkčnosti softwaru

Při vývoji jakéhokoliv softwaru je možné výsledek programování buď otestovat, nebo doufat, že jsem dobrý programátor a chyby v kódu mít nebudu a vše poběží, jak má. Chybovat je lidské, a proto by otestování programu mělo být nedílnou součástí vývoje. Testovat lze buď ručně, nebo testy automatizovat. Ve své práci jsem si zvolila právě téma automatizace a jeho analýzu, protože

žádný kvalitní přehled nástrojů pro automatické testování mobilních aplikací a her zatím nebyl vytvořen.

2.3 Nástroje pro automatické testování

Nástrojů a jazyků, které se dají použít pro vývoj aplikací a her je tedy opravdu požehnaně, a to již napovídá, že stejně tak tomu bude i s nástroji testovacími. Jednotlivé testovací nástroje se dají aplikovat pouze na ty projekty, se kterými si rozumí, tedy jsou schopné spolupráce s daným programovacím jazykem, který vývojové prostředí, ve kterém je projekt napsán, používá.

Existují ale i nástroje, které se používají nad aplikací a nepotřebují tedy kód vůbec znát a dají se tak využít pro téměř všechno.

Nástroje pro automatické testování tedy existují, ale vzhledem k tomu, že jich je mnoho, je již složitější se vyznat v tom, který nástroj je vhodné použít pro konkrétní projekt. Může se dokonce stát, že pro konkrétní projekt a jeho zaměření vhodný nástroj pro automatické testování neexistuje.

Kapitola 3

Základní pojmy

3.1 Mobilní zařízení

V rámci této bakalářské práce pojem mobilní zařízení bude chápán jako bezdrátové přenosné zařízení s vlastním napájením. Jedná se tedy například o mobilní telefon nebo tablet.

3.2 Android

„Android je operační systém vyvíjený společností Google. Používá se dnes nejen v mobilech, kde začal, ale také v tabletech, chytrých hodinkách a dalších chytrých zařízeních. Díky tomu, že se jedná v základu o tzv. open-source projekt, může systém kdokoliv zdarma použít a vyrobit třeba svůj vlastní chytrý telefon.

Android má obrovskou základnu mobilních aplikací, které jsou ke stažení z obchodu Google Play.“ [1]

3.3 Mobilní aplikace

„Mobilní aplikace je typ aplikačního softwaru, který je určený ke spuštění na mobilním zařízení." [2] Aplikace jsou například hra, kalkulačka, různé užitečné nástroje apod. .

3.3.1 Nativní, webové a hybridní aplikace

„Nativní aplikace jsou vytvářené pro konkrétní platformu (např. Android, iOS) a fungují i v offline režimu. Jsou rychlé, spolehlivé a umí využívat hardwarových schopností telefonu (např. fotoaparát, kalendář, GPS).

Webové mobilní aplikace běží ve webovém prohlížeči mobilního zařízení a jsou psány v HTML5, CSS3 a JavaScript.

Pak tu máme ještě tzv. Hybridní aplikace, které jsou zajímavým kompromisem, dokážou využit hardwarových schopností telefonu a zároveň jsou použitelné na více platformách.“ [3]

3.4 Vývojové prostředí

„Vývojové prostředí (zkratka IDE = Integrated development environment) je sada nástrojů, která vám budou pomáhat s laděním programů, hledáním chyb, udržováním přehledu ve zdrojovém kódu, nápovědou k funkcím atd. .

Vývojová prostředí nejsou jen chytré textové editory s barevným zvýrazňováním zdrojového kódu, ale kompletní sada nástrojů, která vám bude pomáhat se psaním zdrojového kódu.“ [4]

3.5 Emulátor

„Emulátor je zjednodušeně řečeno program, který na vašem přístroji vytvoří prostředí pro běh programů či her určených pro diametrálně odlišný systém.“ [5]

3.6 API

„API (Application Programming Interface) se říká rozhraní, které programátor využívá pro komunikaci se softwarem. Jedná se o balík knihoven, funkcí či nějakých procedur, které může programátor využívat a ovládat či komunikovat se softwarem. API se využívá např. v bankovnictví (spárování příchozích plateb s účetním softwarem).“ [6]

3.7 Assert

V této práci se můžete setkat se slovem assert. V českém jazyce pro něj neexistuje vhodný překlad a mezi programátory se hojně používá. Za assert je považována nějaká funkce, která se spustí, pokud něco funguje špatně.

3.8 BDD (Behavior-Driven Development)

„BDD, neboli Bahavior-Driven Development, je proces vývoje softwaru, který vzniknul na základě řízených testů TDD. Spojuje jak obecné postupy TDD, tak také domény a zajišťuje nejen vývoj, ale také správu softwaru. V praxi se u BDD předpokládá využití různých speciálních nástrojů pro podporu vývojového procesu.

Tyto nástroje jsou určeny pro automatizaci do takzvaného všudypřítomného jazyka. BDD je usnadněno s pomocí jednoduchého doménového jazyka,

který se nazývá DSL. BDD se zaměřuje především na to, kde začít v procesu, co je potřeba otestovat, kolik je možné testovat jednotek najednou, nebo na pochopení problematiky selhání testu.“ [6]

3.8.1 TDD

„TDD, Test Driven Development, označující systém vývoje softwaru, je procesem, při kterém jsou praktikovány menší, postupné kroky, které vedou ke zkvalitnění práce softwarových vývojářů. Nejprve je potřeba definovat funkcionalitu, dále se napíše test pro její ověření. Po ověření se dostáváme k zapisování kódu a závěr spočívá v kódových úpravách.

Důležitou součástí testování TDD jsou unit testy, zaměřující se na tu nejmenší jednotku softwaru, jako je třeba metoda, třída nebo funkce. Automatické testování je možné spouštět stále dokola a testovat jedinou součást programu. Pro práci s TDD je třeba dobře znát kódování, ve většině případů se této činnosti věnují samotní vývojáři.“ [6]

3.8.2 Root telefonu

Root telefonu je přenastavení telefonu tak, že zařízení má přístup k administrátorským právům a umožní tak na zařízení provádět administrátorské operace, které jsou jinak na originálním zařízení zakázány.

3.8.3 Jailbreak

Jailbreak je stav, kdy v zařízení jsou odebrána všechna softwarová omezení.

3.8.4 Jenkins, Travis a CircleCI

Jenkins, Travis a CircleCI jsou webové služby (servery), na kterých lze provádět Continous integration, automatcké spouštěnítestů apod. .

Kapitola 4

Testování softwaru

Výsledky testování softwaru mohou pouze prokázat, že chyby existují, ale nemůžou prokázat, že software neobsahuje chyby. [7] Během testování se některé chyby objevit vůbec nemusí a přes to je program obsahuje.

4.1 Softwarová chyba

Softwarová chyba nastane, „pokud je splněna jedna nebo více z následujících podmínek (pravidel):

1. Software nedělá něco, co by podle specifikace produktu dělat měl.
2. Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
3. Software dělá něco, o čem se produktová specifikace nezmiňuje.
4. Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
5. Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný.

U softwarových chyb je příčinou číslo 1 specifikace. To, že hlavním "výrobcem"chyb je skutečně specifikace, má několik důvodů. V řadě případů specifikaci jednoduše vůbec nikdo nenapíše. Další příčinou může být specifikace, která není dostatečně podrobná, nebo se neustále mění, nebo ji dostatečně neprobrali všichni členové vývojového týmu. Plánování softwaru je tak opravdu životně důležité. Pokud se neprovádí správně, zanáší se do softwaru chyby.

Druhým největším zdrojem chyb je návrh. V něm programátoři popisují svůj plán softwaru.“ [8]

4.1.1 Náklady na chyby

„Chyby v softwaru můžeme najít hned od prvopočátku, od zahájení vývoje, přes plánování, programování, testování až po okamžik, kdy jej využívá veřejnost.

Výše nákladů má zde logaritmickou stupnici – to znamená, že s postupem času roste vždy na desetinásobek. Chyba, kterou jsme odhalili a opravili ještě v raných stádiích sestavování specifikace, nemusí stát téměř nic, nebo v našem příkladu 10 centů. Jestliže stejnou chybu objevíme až během kódování (programování) nebo testování softwaru, bude stát 1 až 10 dolarů. A jestliže ji najde až zákazník v hotovém produktu, vyskočí náklady klidně na 100 dolarů a více.“ [8]

4.2 Statické a dynamické techniky testování softwaru

„Kvalitu výstupu jednotlivých fází vývoje softwaru lze v zásadě hodnotit za použití dvou odlišných kategorií aktivit, které využívají buď techniky statické analýzy, nebo dynamické techniky – dynamickou analýzu a (dynamické) testování.

Jak je z názvu patrné, statická analýza má za cíl zkoumat produkt (respektive dílčí produkty, jako jsou např. návrhové dokumenty, databázové modely nebo zdrojový kód) manuálně, či pomocí různých nástrojů, ovšem aniž by došlo k jeho vlastnímu spuštění.

Vedle samotné aktivity testera zkoumajícího jednotlivé produkty a jejich specifikace můžeme mezi běžné techniky statické analýzy zařadit:

- Nástroje automatické statické analýzy mohou být samostatnými produkty, ale dnes jsou v určité podobě již standardní součástí integrovaných vývojových prostředí (IDE). Ve většině případů je automatická statická analýza zaměřena na zdrojový kód a podle kvality daného nástroje umožňuje např. kontrolu dodržování definovaných konvencí a standardů, kontrolu toku řízení a toku dat grafické znázornění.

- Neformální revize je rychlý a nejméně nákladný proces, kdy autor prochází a vysvětluje obsah dokumentu jednomu, či více kolegům, kteří jej komentují a hledají možné defekty či problémová místa. Většinou se o revizi ani nalezených defektech nevede žádný záznam a okamžitě se opravují. Přes svou jednoduchost jde o velmi efektivní způsob hledání defektů.

- Strukturované procházení dokumentu či kódu bývá o něco více formální (ale stále do určité míry flexibilní) skupinovou aktivitou, kterou většinou řídí sám autor procházeného dokumentu za přítomnosti dalších osob s dostatečnou znalostí dané problematiky, jehož role však nejsou přísně

vymezeny. Účelem není pouze odhalit defekty, odchylky od standardů, či zvážit možné jiné způsoby implementace, ale také sdílet znalosti o produktu v rámci týmu.

- Technickou revizí již lze označit za formální revizi. Jejím cílem je především zhodnotit, zda daný produkt splňuje požadavky pro zamýšlené použití, a identifikovat případné odchylky od specifikací a standardů či používaných konvencí. Výstupem není pouze seznam nalezených problémů, ale také doporučení, zda by neměl být daný produkt přepracován, zamítnut, nebo akceptován. Subjektem technické revize může být například specifikace požadavků, či testovací dokumentace.

- Inspekce je velmi formální a vychází z procesu, který vyvinul Michael Fagan v roce 1976 – odtud označení Fagantovská inspekce. Jde o plánovanou aktivitu s předem určenými rolemi jednotlivých členů (moderátor, průvodce, autor, oponenti, zapisovatel). Průvodce reprezentuje kód (či například specifikaci požadavků) s případnou pomocí autora, oponenti vznáší připomínky a hledají defekty či problémy, které zaznamenává zapisovatel. Moderátor usměrňuje a vede diskuzi. Tento postup je velmi efektivní, avšak náročná na čas a zkušenosti zúčastněných, proto se používá jen v nutných případech.“ [9]

Mezi nástroje pro statickou analýzu kódu patří PMD, který pracuje na úrovni zdrojového kódu a odhaluje např. porušení jmenných konvencí, FindBugs, který pracuje na úrovni byte kódu a CheckStyle, který kontroluje, zda je kód napsán podle jasně daných jmenných konvencí a pravidel. [7]

„Oproti tomu dynamická analýza zkoumá chování a vlastnosti produktu – systému či jeho komponentu – za běhu pomocí speciálních k tomu určených nástrojů, a navzdory častému omylu tak nejde o testování v pravém slova smyslu. Tato technika umožňuje například detailně analyzovat správu paměti a výkon systému a zjistit tak problémy, které by jinak byly velmi obtížně odhalitelné. Typickým příkladem využití dynamické analýzy je profilování aplikací v rámci zajišťování jejich optimalizace.“ [9]

4.3 Typy testů podle znalosti vnitřní struktury

4.3.1 Černá skříňka (Black box)

„Při testování černé skříňky ví tester, co má předložený software dělat – dovnitř skříňky se podívat nemůže a neví tedy, jak pracuje uvnitř. Jestliže napíše nějaký údaj na vstupu, dostane určitý odpovídající výsledek. Neví, proč a jak se zrovna tento výsledek objevil, pouze jej pozoruje.“ [8]

4.3.2 Bílá skříňka (White box)

„U testování bílé skříňky má oproti tomu softwarový tester přístup ke zdrojovému kódu programu a jeho zkoumání mu může pomoci při testování – vidí tedy "dovnitř"skříňky. Z pohledu dovnitř pak tester může odhadnout, jestli určitá čísla povedou častěji nebo méně často ke zkáze a podle těchto informací může lépe přizpůsobit další testování.“ [8]

4.3.3 Šedá skříňka (Grey box)

Šedá skříňka předpokládá kombinaci obou předchozích typů – víme, jaké algoritmy byly použity, ale neznáme implementaci. [7]

„U přístupu šedé skříňky je při návrhu testovacích případů k dispozici nejen specifikace požadavků na software, ale také dokumenty používané vývojáři a v určitých případech – v omezené míře – také zdrojový kód. To nám umožňuje navrhovat testy, které testují funkčnost systému a zároveň se zaměřují na potencionální slabá místa, oblasti a vyšší složitosti, nebo situace, které bychom s největší pravděpodobností minuli.“ [9]

4.4 Typy testů

4.4.1 Jednotkové testy

„Po ověření kódu programátorem přichází na řadu test jednotek. U objektově orientovaného programování se jedná o testování jednotlivých tříd a metod. Testovanou jednotkou v tomto případě rozumějme samostatně testovatelnou část aplikačního programu. Testy těchto jednotek se zapisují ve formě programového kódu. Proto jej z pravidla obsluhují vývojáři. Pro vytváření testů se využívá nástrojů na bázi frameworků. Testy jednotek se velmi špatně aplikují na již zaběhlých projektech. U již vytvořených aplikací se většinou musí provést kompletní refaktoring kódu či dokonce mnohem hlubší úpravy. Takováto časová investice se u menších projektů většinou nevyplatí, ale ani u velkých projektů takovýto zásah není příliš šťastný a často se nesetkává s podporou u vedoucího projektu. Proto je vhodné zabývat se těmito testy již v etapě návrhu aplikace a v té době se rozhodnout, zda tyto testy budeme využívat. Jelikož obecně platí, že čím dříve (v rámci životního cyklu software) chybu nalezneme a opravíme, tím méně času nad touto opravou strávíme. Proto bych doporučoval této úrovni věnovat maximální pozornost, a to již před samotným vývojem aplikace.“ [10]

Známými nástroji pro jednotkové testy jsou JUnit, NUnit, nebo TestNG. [7]

4.4.2 Testy komponent

Během těchto testů dochází k testování izolovaných komponent. Jsou podobné jednotkovým testům, ale testují větší části funkčností. Zaměřují se na komu-

nikaci mezi třídami, které reprezentují nějakou komponentu (část systému). Typicky se jedná o testování metodou bílé skříňky (white box).

Mezi typické nástroje pro testy komponent patří JUnit, NUnit, jMock a EasyMock. [7]

4.4.3 Integrační testy

„V době, kdy je vývojář hotov se svými testy, přichází na řadu testovací tým. Integrační testy tedy nepřipravuje programátor, ale především testovací tým. Někdy bývají označovány jako "testy vnitřní integrace". Musí být ověřena bezchybná komunikace mezi jednotlivými komponentami uvnitř aplikace. Integraci však lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. V této fázi se tak testuje integrace dosud jednotlivě ověřených částí. Postupně začínáme testovat integraci mezi dvěma komponentami a postupně přidáváme další. Integrační testy mohou být jak manuální, tak i automatizované.

Úroveň integračního testování je svým způsobem obsažena ve většině testovacích postupů softwaru. U menších projektů je však na tyto testy kladen velmi malý důraz. Má to své logické odůvodnění. Integrační testování lze v testovacím cyklu zcela vynechat. Na výslednou bezporuchovost softwaru to přitom nebude mít žádný vliv, tedy alespoň za předpokladu, že korektně provedeme následující úrovně testování. Chyba, kterou bychom odhalili během integračních testů, se zcela jistě projeví v průběhu dalších úrovní testování. Jak již bylo zmíněno dříve, během testování však platí: čím dříve chybu objevíme, tím méně úsilí nás stojí její oprava. Proto integrační testy mají svůj význam, nicméně nelze jejich použití nikterak přeceňovat.“ [10]

4.4.4 Smoke testy

„Občas se do češtiny nesprávně překládá jako "zahořovací test". Kategorie Smoke testů se využívá v okamžiku, kdy je dokončen vývoj aplikace a lze ji spustit. Tedy na konci úrovně integračního testování. Jedná se o krátký test, který slouží jako rychlé ověření, zda je vyvíjená aplikace připravena pro další fázi testování. Obvykle se provede jen jednoduché ověření, že všechny části aplikace jsou implementovány, nainstalovány a spuštěny. Smoke testy se zaměřují pouze na hlavní funkce programu, které nebývají příliš často upravovány. Často se také kombinují s kategorií testů splněním. Jelikož je rozsah smoke testů menší, než tomu bývá u ostatních kategorií a pokrývají pouze hlavní funkce, jsou velmi často automatizovány. Pokud nejsou automatizovány v plném rozsahu, zbytek testů je prováděn manuálně. Úspěšné provedení smoke testů je podmínkou pro vstup do systémové úrovně testování. Proto jsou velmi důležité. U menších projektů, kde je testování software opomíjeno nebo není příliš organizováno, se provádí pouze smoke testy.“ [10]

4.4.5 Systémové testy

„Spojení Integračních i Systémových testů je označována jako fáze SIT – System Integration Tests. Po ověření správné integrace nastává ten pravý čas na systémové testování. Během těchto testů je aplikace ověřována jako funkční celek. Tyto testy jsou používány v pozdějších fázích vývoje. Ověřují aplikaci z pohledu zákazníka. Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat. Obvykle probíhají v několika kolech. Nalezené chyby jsou opraveny a v dalších kolech jsou tyto opravy opět otestovány. Součástí této úrovně jsou jak funkční, tak nefunkční testy. Poslední úroveň testů, které se provádějí před předáním produktu zákazníkovi, jsou tedy systémové testy. Tato úroveň testů tak většinou slouží jako výstupní kontrola softwaru. Systémové testování je obsaženo prakticky v každém procesu testování. Bez této úrovně by celé testování softwaru nemělo žádný význam. Bezporuchovost výsledného produktu by byla významně ohrožena. Proto tuto úroveň testů považuji za stěžejní v celém postupu testování software. Na realizaci těchto testů by se mělo myslet již v raném stádiu návrhu postupu testování, tak aby bylo možné obsahu testů co možná nejvíce přizpůsobit očekávanému softwaru.“ [10]

Mezi nástroje pro systémové testy patří například Selenium, automaticky testující grafické uživatelské rozhranní (GUI), nebo SoapUI, automaticky testující služby. [7]

4.4.6 Progresní testy

„Progresní testy využíváme při kontrole nových funkcí nebo vlastností aplikace. K jejich správnému provedení je nutná dokumentace, která přesně popisuje nově implementované oblasti. Používáme je ve všech etapách testování.“ [10]

4.4.7 Progresní testy

„Regresní testy se využívají při opětovném testování funkcí a vlastností aplikace. Jejich smyslem je ověření, že provedené změny či implementace nových vlastností v aplikaci nemělo žádný vliv na stávající funkce a vlastnosti. Tedy především na oblasti, které zůstaly v programovém kódu nezměněny. Oblasti, které byly předmětem úprav, by správně již měly být otestovány funkčními testy. Takovéto situace z pravidla nastávají po opravení chyb či po novém release. Regresní testy je velmi vhodné automatizovat.

V praxi jsou regresní testy velmi rozšířené. V různých formách se používají prakticky u většiny projektů. Využívají se hlavně při retestech opravených chyb. Oproti tomu progresní testy nejsou příliš rozšířeny a často je tato kategorie testů rozložena do jiných kategorií.“ [10]

4.5 Manuální testování

„Manuální testování softwaru je to první, co se lidem obvykle vybaví pod pojmem testování softwaru. Hned na začátku bych rád upozornil, že manuální testování nelze zjednodušovat na často tolik oblíbené "klikání". Pojem "klikání"je naprosto neodborný a nic nevypovídající. Velmi často jej využívají lidé (zejména manažeři), kteří nemají pojem o kategoriích testů vhodné pro danou situaci.“ [11]

4.6 Automatizované testování

„Hlavním cílem automatizace testů softwaru je časová úspora při jejich spouštění. Obecně lze říci, že automatizace testů softwaru má za účel usnadnit a zefektivnit provádění postupu testování softwaru. Pokud jsou testy prováděny zcela automaticky, tj. bez možnosti zásahu lidského faktoru, výrazně se tím snižuje možnost chybného provedení postupu testování softwaru a tím také pravděpodobnost bezporuchového provozu softwaru.

Automatizovat je zejména vhodné především Smoke testy, Regresní testy, Komparační testy, ověřování změny dat a také Nefunkční testy. Obecně tedy testy, které spouštíme velké množství, jsou neměnné a pokrývají části aplikace, které se často nemění.

Pro realizaci automatizovaných testů softwaru, je nutné zajistit některé základní podmínky v postupu testování softwaru. Existující program, který je již v provozu, je obtížné začít automatizovaně testovat. Nejvhodnější etapa pro vytváření těchto zkoušek je tedy návrh a vývoj softwaru. Architektura aplikace musí umožňovat automatizaci zkoušek, proto je nutné aplikaci s tímto vědomím vytvářet.

Obecně se snažíme pokrýt testy co největší část aplikace. U aplikací s krátkou životností (v řádech měsíců, jedna verze) se nám nevyplatí vytvářet tyto testy, naopak u aplikací, kde předpokládáme delší životnost (řádově roky, více verzí), tak tam se nám zcela jistě realizace automatizovaných testů vyplatí. U automatizovaných testů se pokrývají především části, u kterých je reálná hrozba výskytu závažných chyb. Při realizaci testů se však musí přihlédnout k tomu, zda daná část aplikace nebude často upravována. To by totiž znamenalo častou aktualizaci automatizovaných testů.“ [12]

Kapitola 5

Vývojová prostředí pro tvorbu Android aplikací

Pro vývoj aplikací a her určených pro zařízení s operačním systémem Android neexistuje pouze jeden programovací jazyk a prostředí, ale je jich hned několik. V této části práce bych ráda jednotlivé možnosti vývoje představila.

„Na OS Android se naprostá většina aplikací a her vyvíjí v programovacím jazyce Java nebo C++. Pokud chcete vytvořit hru s 3D grafikou, sáhněte rovnou po C++ (většina nejlepších 3D enginů je napsána pro použití výhradně s C++). Jestliže budete tvořit 2D plošinovku, logickou hru, strategii, RPG apod., pak vystačíte s Javou.“ [13]

5.1 Android Studio

„Android Studio je oficiálním nástrojem pro tvorbu aplikací pro Android. Poskytuje nejrychlejší nástroje pro vytváření aplikací určených na všechny zařízení s OS Android.

Vývojové prostředí je světovou špičkou v editaci kódu, debuggování, výkonnostních nástrojích. Poskytuje flexibilní sestavování programu a okamžité sestavení/nasazení systému. Vše umožňuje zaměření se na vytvoření jedinečné a unikátní aplikace.“ [14]

Android Studio podporuje programování v jazyku Java, v němž je možné naplnit nejrůznější požadavky na funkčnost.

5.1.1 Osobní názor na Android Studio

Osobně při vývoji aplikací a her této možnosti dávám přednost. Vyzkoušela jsem si v Android Studiu i multiplatformní vývoj (Android, desktop a iOS) a práce v něm je příjemná a intuitivní a navíc na internetu se k němu dá dohledat téměř vše, když si člověk neví rady.

5.2 Eclipse

Eclipse je alternativou pro vývojáře, kteří z nějakého důvodu nechtějí využívat Android Studio. Výhodou Eclipse je, že se v něm dá programovat ve více jazycích – tedy je dostupné i pro C++, což ocení vývojáři pracující s 3D grafikou.

5.3 Corona SDK

„Pokud se nechcete učit Javu nebo návrh uživatelského rozhraní v XML, pak jsou tu i jiné možnosti. Jedna z nich je použít Corona SDK. Corona je SDK na vysoké úrovni založené na programovacím jazyku Lua. Lua je o mnoho jednodušší než Java a Corona SDK tak šetří programátora od utrpení spojeného s vývojem aplikací pro Android.

Corona přikládá sofistikovaný emulátor, který umožňuje programu, aby se okamžitě spustil bez nutnosti kompilace kódu. Když chcete vytvořit Android .apk soubor, build se spustí přes Coroniny online kompilátory a aplikace se uloží do vašeho počítače.

Corona je navržena především pro hry (ale ne pouze pro hry) a jako taková přikládá i knihovny pro sprite, zvuky, herní síťování a 2D engine pro fyziku. V Corona SDK je téměř vše zobrazování skrz OpenGL.“ [15]

5.3.1 Vlastní názor na Corona SDK

Instalace Corona SDK je jednoduchá, ale pro vývoj je potřeba nějaké prostředí pro psaní kódu (já si plně vystačila s nástrojem Gedit). Spuštění aplikace na emulátoru zabere jen pár sekund, jelikož se nemusí nic kompilovat. Jazyk Lua není nijak složitý a dá se v něm snadno orientovat.

5.4 Gideros

„Gideros je open source a zdarma a poskytuje multiplatformní technologie pro vytvoření úžasných her. Během několika hodin dokážete sestavit a rozběhnout vaší skvělou hru.“ [16]

Gideros je open source projekt, tedy vývoj v něm je zdarma a zcela bez omezení. Podporuje vývoj pro Android, iOS, Windows Phone, MacOSX, Windows a Windows RT. V Giderosu se vyvýjí v programovacím jazyku Lua, ale do Giderosu lze importovat existující kód v programovacím jazyku C, C++, Java, nebo objektové C. Gideros poskytuje svůj vlastní systém tříd založeným na objektově orientovaném programování a umožňuje tak vývojáři psát čistý a znovu použitelný kód pro jakýkoliv budoucí vývoj.

Vývoj v prostředí Gideros je snadné se naučit a během vývoje hry jde hru testovat skrz Wifi na reálném zařízení.

5.5 PhoneGap

„Jestliže již znáte HTML, CSS a především JavaScript, pak raději, než se učit programovací jazyk Java nebo Lua, můžete vytvořit aplikaci pro Android ze znalostí, které již máte. PhoneGap je založen na Apache's Cordova projektu. V základu vytváří webové zobrazení, se kterým můžete dále pracovat použitím jazyku JavaScript. Webová aplikace může interagovat s různými vlastnostmi zařízení, stejně jako nativní aplikace, odkazem na cordova.js soubor získávající vazby na API. Nativní funkce, které Phonegap podporuje jsou akcelerometr, kamera, poloha, lokální disk a podobně.“ [15]

5.5.1 Osobní názor na PhoneGap

Pro programátory věnující se webům je PhoneGap určitě velkým přínosem. Co se mi u PhoneGapu nelíbí, je žádný emulátor. Pokud programátor chce vyvíjet pro Android, musí si stáhnou aplikaci do mobilu a já, bohužel, zrovna neměla to štěstí, aby zvolená aplikace byla kompatibilní s mým telefonem. Zvolila jsem tedy webový emulátor Ripple, který se dá stáhnout jako rozšíření pro Google Chrom.

5.6 Titanium

„Otevřené, rozšiřitelné vývojové prostředí pro tvorbu pěkných nativních aplikací napříč různými mobilními zařízeními a operačnímy systémy přikládající iOS, Android a BlackBerry, stejně jako hybridní vývoj a HTML 5. Obsahuje open source SDK s více než 5000 zařízeními a API pro operační systémy, studio, verzatilní IDE založené na Eclipse, Alloy, MVC framework a cloudové služby pro snadno použitelné mobilní backend služby.“ [17]

Titanium je založené na jazyku JavaScript a CSS.

5.7 GameSalad

GameSalad Creator je nástroj určený především pro tvorbu mobilních her pro Android a iOS. Vývoj v tomto nástroji probíhá metodou drag&drop, takže pro tvorbu hry není třeba umět programovat - v programu se nekóduje. Díky tomu vývoj hry zabere mnohem méně času, než je tomu při běžném programování.

Vytvořenou aplikaci lze otestovat v prostředí mobilního zažízení pomocí GameSalad In-App náhledu, který je navržen tak aby co nejdůvěryhodněji napodoboval reálné zařízení. [18]

5.7.1 Osobní názor na GameSalad

Sama jsem si v GameSaladu zkusila vytvořit jenoduchou hru. Prostředí je navržené, aby bylo jednoduché, ale znalost programování je zde opravdu výhodou. Není třeba být skvělý programátor, stačí mít jen základy a rozumět základním funkcím jako podmínky, cykly, přiřazování proměnných a podobně. Hru pak lze vytvořit v krátkém časovém úseku a velmi jednoduše.

5.8 Microsoft Visual Studio

Ve Visual Studiu lze od verze Visual Studio 2015 vytvářet multiplatformní aplikace. V rámci vývoje si lze vybrat vývoj hybridní a nativní aplikace. Hybridní vývoj využívá Cordova Framework.

„Zjednodušeně se jedná o nativní obálku zapouzdřující HTML - Javascript aplikaci do balíku nativně instalovatelného na cílové platformy (BackBerry, Android, IOS, Windows, ...). Javascriptové Cordova knihovny dokážou unifikovaně volat "sesterské"nativními knihovny, které využívají HW senzory dané cílové platformy. Takto můžete sdílet jeden kód pro více platforem což je největší výhoda. Hlavní nevýhoda je, že výkonná aplikace je vlastně HTML/Javascript, která má prakticky vždy menší výkon než nativní aplikace. Jednotné web UI často neodpovídá zvyklostem konkrétní platformy.

Nativní aplikace využívající crossplatform Xamarin Framework. Xamarin je jeden z nejrozšířenějších crossplatform fremeworků, jenž z jednoho kódu dokážou kompilovat různé nativní aplikace. U Xamarinu je kód psán v C#, ten je pak postupně překládán nativními kompilátory Android, IOS nebo Windows. Rychlost je super, má-li však UI aplikace plně ctít zvyklosti cílové platformy musí se UI tvořit samostatně případně využít placené Xamarin Forms. Ve Visual Studiu najdete vše potřebné, jen je třeba pamatovat na to, že free licence Xamarinu ve Visual Studiu 2015 je omezena velikostí výsledné aplikace.

Abyste mohli spouštět a ladit Android aplikace, je vhodné přidat Visual Studio Android Emulator. Ten je volitelnou součástí instalace Visual Studia a také ho lze stáhnout samostatně zdarma, zcela nezávisle na instalaci Visual Studia. Visual Studio Android Emulator je založen na Hyper-V, pokud ho chcete používat, musí váš počítač podporovat a mít zapnutu Hyper-V virtualizaci. Pomocí Visual Studio Android Emulátoru pak můžete spouštět různé verze Androidu 4.x a .x s emulací různých zařízení.“ [19]

5.9 Unity

Unity je herním enginem, který lze použít pro vývoj pro mobilní zařízení. Pokud tedy chceme vytvořit 2D nebo 3D hru pro mobilní zařízení, může být Unity správnou volbou. V Unity je možný multiplatformní vývoj a dá se v něm vyvíjet pro Android, iOS, Windows Phone, Tizen a Fire OS.

5.10 Unreal Engine

Unreal Engine 4 podporuje vývoj 2D a 3D her pro operační systémy Windows, OS X, Linux, Android, iOS, Xbox One, PlayStation 4 a Ouya. Unreal Engine je verzatilní a není obtížné ho používat. Jeho velkou výhodou je, že nabízí bezkonkurenční grafické možnosti.

Kapitola 6

Nástroje určené pro specifické prostředí

Pokud to lze, je vhodné zvolit pro automatické testování nástroj, který je přímo určený pro konkrétní prostředí a programovací jazyk. Důvodem je, že testy jsou navrženy na principu bílé skříňky či šedé skříňky - nástroj "vidí"do kódu programu a je schopen s ním spolupracovat.

Použití těchto testovacích nástrojů je vhodné především pro testování logiky aplikace či hry. Důvod je zřejmý - nástroj dokáže reagovat i na náhodně generované případy (což ocení především vývojáři her). Výhodou je, že tyto nástroje mohou obsáhnout mnohem větší část programu, než nástroje multiplatformní.

Nevýhodou těchto nástrojů je složitější tvorba testovacích scénářů, což zabere více času než nástroje multiplatformní. Vývojář musí mít znalosti v oblasti programování a měl by znát kód programu, který testuje. Ideální je začít tyto testovací nástroje používat již v začátcích vývoje aplikace nebo hry.

6.1 Testování Android aplikací - jazyk Java

„Automatizace UI testů s použitím Android Studia probíhá tak, že se implementuje testovací kód v oddělené složce Android test (src/android/Test/java). Android Plug-in pro Gradle vytvoří testovací aplikace založené na testovacím kódu, pak načte testovací aplikaci na stejném zařízení jako cílové aplikace. V testovacím kódu můžete použít UI testovací framework pro simulaci interakce uživatele s cílovou aplikací s cílem provádět testovací úkony, které pokrývají konkrétní scénáře užití.“ [20]

Typy automatizovaných testů uživatelského rozhraní pro Android:

- **UI testy, které pokryjí aplikaci**: Tento typ testů slouží k ověření, že se aplikace chová podle očekávání při určité akci uživatele. Ověřujeme, zda se po akci vrací správný UI výstup. Pro tyto účely se využívá UI

testovací framework Espresso, které umožňuje simulaci akcí uživatele aověřování UI výstupů.

- **UI testy ověřující spolupráci více aplikací**: Tento typ testů zjišťuje, jak se aplikace chová při vzájemné interakci s jinými aplikacemi. Příkladem může být aplikace, která sdílí obrázky s třetí stranou jako jsou například sociální sítě, aplikace pro zobrazování fotek a podobně. Tyto testovací frameworky podporují multiplatformní interakci a mezi jejich představitele se řadí UI Automator. [20]

6.1.1 Android Testing Support Library

Knihovna Android Testing Support Library poskytuje sadu rozhraní API pro tvorbu testovacího kódu pro aplikace. Podporuje JUnit 4 a funkční testy uživatelského rozhraní. Testy pak lze spouštět z IDE Android Studio, nebo pomocí příkazové řádky. [21]

Knihovnu zprostředkovává Android SDK Manager a přikládá testovací nástroje AndroidJUnitRunner, Espresso a UI Automator.

6.1.1.1 AndroidJUnitRunner

Třída AndroidJUnitRunner spouští JUnit testy na zařízeních s operačním systémem Android a podává zprávu o výsledcích jednotlivých testů. Na výběr je ze dvou možností - JUnit 3 a JUnit 4 a jejich míchání ve stejném balíčku se nedoporučuje. V současnosti se více využívá JUnit 4, jelikož JUnit 3 je již zastaralé. Testy se mohou kombinovat s nástroji Espresso a UI Automator. [21]

Nástroj JUnitRunner jsem vyzkoušela na aplikaci kalkulačka. Implementovala jsem jednoduchý test pro operace sčítání, odčítání násobení a dělení.

Pro spuštění testů, je nutné mítmít v build.gradle nastaveno v sekci dependencies: testCompile 'junit:junit:4.12', jinak testy nebudou fungovat. Android Studio pak umí generovat kostry testu pro metody, které chceme otestovat. Logiku si pak programátor musí doplnit sám. Testy se ukládají obvykle do složky app \src\test, určené pro jednotkové testy, app\src\androidTest pak slouží pro ostatní testy (integrační, implementační, atd.). Unit testy jsou určené k otestování jednotlivých metod. Test vypadá obvykle tak, že se zadají vsupní hodnoty, které chceme do testu odeslat a očekávaná hodnota výsledku. Vstupní hodnoty se pak odešlou do metody a porovná se (přes assertEquals) tento výsledek s výsledkem očekávaným. Testy tedy slouží k otestování "vnitřní"stavby aplikace, nikoliv ke kontrole uživatelského rozhraní. [22]

6.1.1.2 Espresso

Espresso je testovací framework určený pro vytváření testů, který umožňuje psaní automatických testů uživatelského rozhraní. Framework je vhodný pro psaní

```java
package com.sample.foo.samplecalculator;

import android.support.test.runner.AndroidJUnit4;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;

@RunWith(AndroidJUnit4.class)
public class ExampleJUnit4Test {
    @Test
    public void testCounting(){
        assertEquals(0, 0 + 0);
        assertEquals(15, 18 - 3);
        assertEquals(12, 3 * 4);
        int a=0;
        while(a<10){
            assertEquals(0,  0 * a++);
        }
        assertEquals(10, 10 / 1);
    }
}
```

Obrázek 6.1: JUnitRunner - příklad kódu testu operací kalkulačky s použitím JUnit 4

testů ve stylu bílé skříňky - testovací kód využívá implementaci kódu z aplikace. [21]

Espresso se umí postarat o synchronizaci jakéhokoliv UI eventu - onStart, setContestView, onCreate a onResume, takže se není třeba obávat jakékoliv změny zobrazení, nebo implementačních detailů. Test má pracovat tak, že najde odpovídající zobrazení, na kterém se má test provést, provede nějakou akci a porovná zobrazený výsledek s výsledkem očekávaným. Pro spuštění testu je potřeba upravit dependencies v build.gradle. [23]

Mezi klíčové rysy testovacího frameworku Espresso patří:

- View matching - API, které umožňuje najít odpovídající view, které má být otestováno.
- Action APIs - rozsáhlý soubor API akcí pro automatizaci interakce s uživatelským rozhraním jako klikání do view, swipování, stisku tlačítka, psaní textu, otevření linku a tak dále.
- UI thread synchronization - synchronizace vlákna uživatelského rozhraní pro vylepšení testu spolehlivosti [21]

Pro simulaci interakce uživatele s UI komponentou se používají metody ViewInteraction.perform(), nebo DataInteraction.perform(). Třída ViewActions poskytuje metody pro kliknutí, psaní testu, scroll na obrazovce, stisk tlačítka, nebo smazání textu. [24]

Espresso obsahuje podsekci Espresso Intents umožňující otestovat aplikaci, aktivitu, nebo službu odděleně, zachycením odchozích záměrů a porovnáním výsledků. IntentsTestRule před každým testem inicializuje záměry Espresso Intents. Espresso Intents zaznamenávají všechny záměry, které se pokoušejí spustit aktivity z testované aplikace a tím se testuje, zda očekávaný záměr byl vidět. Na tyto aktivity lze poskytovat odpovědi. [25]

```
@Large
@RunWith(AndroidJUnit4.class)
public class SimpleIntentTest {

    private static final String MESSAGE = "This is a test";
    private static final String PACKAGE_NAME = "com.example.myfirstapp";

    /* Instantiate an IntentsTestRule object. */
    @Rule
    public IntentsTestRule<MainActivity> mIntentsRule =
      new IntentsTestRule<>(MainActivity.class);

    @Test
    public void verifyMessageSentToMessageActivity() {

        // Types a message into a EditText element.
        onView(withId(R.id.edit_message))
                .perform(typeText(MESSAGE), closeSoftKeyboard());

        // Clicks a button to send the message to another
        // activity through an explicit intent.
        onView(withId(R.id.send_message)).perform(click());

        // Verifies that the DisplayMessageActivity received an intent
        // with the correct package name and message.
        intended(allOf(
                hasComponent(hasShortClassName(".DisplayMessageActivity")),
                toPackage(PACKAGE_NAME),
                hasExtra(MainActivity.EXTRA_MESSAGE, MESSAGE)));

    }
}
```

Obrázek 6.2: Espresso - příklad kódu testu [24]

6.1.1.3 UI Automator

UI Automator je testovací framework pro sestavení testů uživatelského rozhraní, které provádí interakci uživatele s aplikací. Provádí operace jako otevření menu nastavení, nebo spouštění aplikace v testovacím zařízení. Framework je vhodný pro spsaní testů ve stylu černé skříňky - kód je nezávislý na implementačním provedení aplikace. [21]

Mezi klíčové rysy testovacího frameworku UI Automator patří:

- Automator Viewer – provádí kontrolu hierarchie rozložení
- Access to device state – API, které načítá informace o stavu a provádí operace na cílovém zařízení
- UI Automator APIs – podporuje miltiaplikační testování uživatelského rozhraní. [21]

Výhody nástroje UI Automator jsou, že může být použit na zařázeních s různým rozlišením, jelikož interakce s komponentami probíhají prostřednictvím obrazového rozlišování. Například tedy místo kliknutí na určité souřadnice nějakého tlačítka se použije kliknutí na obrázek komponenty, kterou si Automator Viewer dohledá. Testy mají vždy stejnou posloupnost, což se hodí především u kontroly aplikace na zařízeních s odlišným rozlišením.

Nevýhodou pak je, že je obtížné ho sloučit s OpenGL a HTML5 aplikacemi, protože tyto aplikace neobsahují Android UI komponenty.[26]

Nástroj je skvělou pomůckou pro testování mobilních aplikací, ale u her již může, kvůli logice, nastat s jeho použitím problém.

```
01 private void sendMessage(String toNumber, String text) throws UiObjectNotFoundException {
02         // Find and click New message button
03         UiObject newMessageButton = new UiObject(new UiSelector()
04         .className("android.widget.TextView").description("New message"));
05         newMessageButton.clickAndWaitForNewWindow();
06 
07         // Find to box and enter the number into it
08         UiObject toBox = new UiObject(new UiSelector()
09         .className("android.widget.MultiAutoCompleteTextView").instance(0));
10         toBox.setText(toNumber);
11         // Find text box and enter the message into it
12         UiObject textBox = new UiObject(new UiSelector()
13         .className("android.widget.EditText").instance(0));
14         textBox.setText(text);
15 
16         // Find send button and send message
17         UiObject sendButton = new UiObject(new UiSelector()
18         .className("android.widget.ImageButton").description("Send"));
19         sendButton.click();
20     }
```

Obrázek 6.3: UI Automator - příklad kódu testu - metoda, která stiskne tlačítko pro napsání nové zprávy, vloží telefonní číslo a stiskne tlačítko odeslat. [26]

6.1.2 monkeyrunner

Nástroj monkeyrunner poskytuje API pro aplikace napsané v jazyku Java, nebo emulátor pro aplikace používající jiný programovcí jazyk, například Python. V něm můžeme napsat program, který instaluje aplikaci pro Android a spustí ji. Umí pak do této aplikace zasílat vstupní akce jako klikání, mačkání různých kláves a další. V průběhu testu může pořizovat snímky obrazovky. Tento nástroj je primárně určen pro testování aplikací na funkční úrovni a spouštění jednotkových testů.

Nástroj monkeyrunner ovládá zařízení či emulátory z pracovní stanice zasíláním specifických příkazů.

Nástroj monkeyrunner poskytuje následující funkce pro Android testování:

- Vícenásobná kontrola zařízení: monkeyrunner API umžňuje spustit jeden, nebo více testů na různých zařízeních zároveň.
- Funkční testování: monkeyrunner umí spustit automatizovaný test a poskytuje vstupní hodnoty - simuluje stisk kláves, nebo dotyk obrazovky a zobrazuje výsledky jako snímky obrazovky.
- Regresní testování – monkeyrunner může otestovat stabilitu aplikace spuštěním aplikace a porovnáním jejích výstupů v podobě snímků obrazovky s očekávanými výsledky.
- Rozšiřitelná automatizace – vzhledem k tomu, že monkeyrunner je API sada nástrojů, je možné vytvořit celý systém modulů psaných v jazyku Python a programů ovládajících Android zařízení. Kromě využití samotného monkeyrunner API, je moné využít standardní Python a dílčího modulu pro volání Android nástrojů jako Android Debug Bridge.

Nástroj monkeyrunner používá Jython, který umožňuje API monkeyrunner snadno komunikovat s Andrid frameworkem. S Jythonem lze využít syntaxi Pythonu pro přístup ke konstantám, třídám a metodám API. [27]

6.2 Testovací nástroje vhodné pro prostředí používající jazyk Lua

6.2.1 Testování jednotek

Pro programovací jazyk Lua existuje velké množství jednotkových testovacích prostředí, které zde budu představovat. Vývojáři, kteří zvolí vývoj s využitím jazyku Lua některá z těchto prostředí jistě uvítají.

6.2.1.1 Lunit

Lunit je framework pro jednotkové testování. Poskytuje 27 funkcí typu assert, 8 typů kontrolních funkcí a nějaké další funkce. Požaduje verzi Lua5.1, nebo novější. [28]

```
require "lunit"

module( "my_testcase", lunit.testcase )

function test_success()
  assert_false( false, "This test never fails.")
end

function test_failure()
  fail( "This test always fails!" )
end
```

Obrázek 6.4: Příklad suboru v lunit [28]

```
# ./lunit my_testcase.lua
Loaded testsuite with 2 tests in 1 testcases.

    F.

2 Assertions checked.

  1) Failure (my_testcase.test_failure):
my_testcase.lua:10: failure
my_testcase.lua:10: This test always fails!

Testsuite finished (1 passed, 1 failed, 0 errors).
```

Obrázek 6.5: Spouštění shell scriptu v lunit a výstup [28]

6.2.1.2 Lunity

Lunity je dalším testovacím frameworkem pro jazyk Lua a obsahuje množství funkcí typu assert a chybových zpráv. [29]

```
require 'test/lunity'
module( 'TEST_RUNTIME', lunity )

function setup()
  -- code here will be run before each test
end

function teardown()
  -- code here will be run after each test
end

-- Tests to run must either start with or end with 'test'
function test1_foo()
  assertTrue( 42 == 40 + 2 )
  assertFalse( 42 == 40 )
  assertEqual( 42, 40 + 2 )
  assertNotEqual( 42, 40, "These better not be the same!" )
  assertTableEquals( { a=42 }, { ["a"]=6*7 } )
  -- See below for more assertions available
end

function test2_bar()
  -- Tests will be run in alphabetical order of the entire function name
end

function some_utility()
  -- You can define helper functions for your tests to call with impunity
end

runTests()
-- or runTests{ useANSI = false }
-- or runTests{ useHTML = true  }
```

Obrázek 6.6: Příklad kódu testu Lunity [29]

6.2.1.3 lunatest

Testovací framework Lunatest obsahuje přídavnou podporu pro náhodné testování a je psaný ve stylu xUnit. Nástroj podporuje funkce pro pořízení časové známky. [30]

```
module(..., package.seeall)
function test_saveFile()
require "com.jxl.core.services.ReadFileContentsService"
local saveFile = SaveFileService:new()
local data = "moo"
assert_true(saveFile:saveFile("test.txt", system.DocumentsDirectory, data), "Failed to save test.txt")
end
function test_readFile()
require "com.jxl.core.services.ReadFileContentsService"
require "com.jxl.core.services.SaveFileService"
local saveFile = SaveFileService:new()
local data = "moo"
assert_true(saveFile:saveFile("test.txt", system.DocumentsDirectory, data), "Failed to save test.txt for use in reading.")
local readFile = ReadFileContentsService:new()
local contents = readFile:readFileContents("test.txt", system.DocumentsDirectory)
assert_string(contents, "contents are not a string.")
assert_equal(contents, data, "Data written to file does not match what we just read out of it.")
end
```

Obrázek 6.7: Příklad kódu testu - lunatest [31]

6.2.1.4 LuaUnit

LuaUnit je testovací framework, který umožňuje psaní testovacích tříd a metod. Podporuje výstupy jako JUnit, nebo TAP, což umožňuje snadnou integraci do CI platforem (Jenkins, Maven,...). Testy lze vybírat podle jména, nebo vzorů, jde řídit výstupní formát, nastavovat podrobnosti a další.

LuaUnit spolupracuje s Lua 5.1, LuaJIT 2.0, LuaJIT 2.1 beta, Lua 5.2 a Lua 5.3. LuaUnit se dá používat na Windws 7, Windows Server 2012 R2 (x64), MacOs X 10.9.5 a Ubuntu 14.04, případně na dalších, které podporují jazyk Lua.

Tento framework se řadí mezi open-source frameworky a dá se tak jako balíček jednoduše přidat do projektu a používat. [**?**]

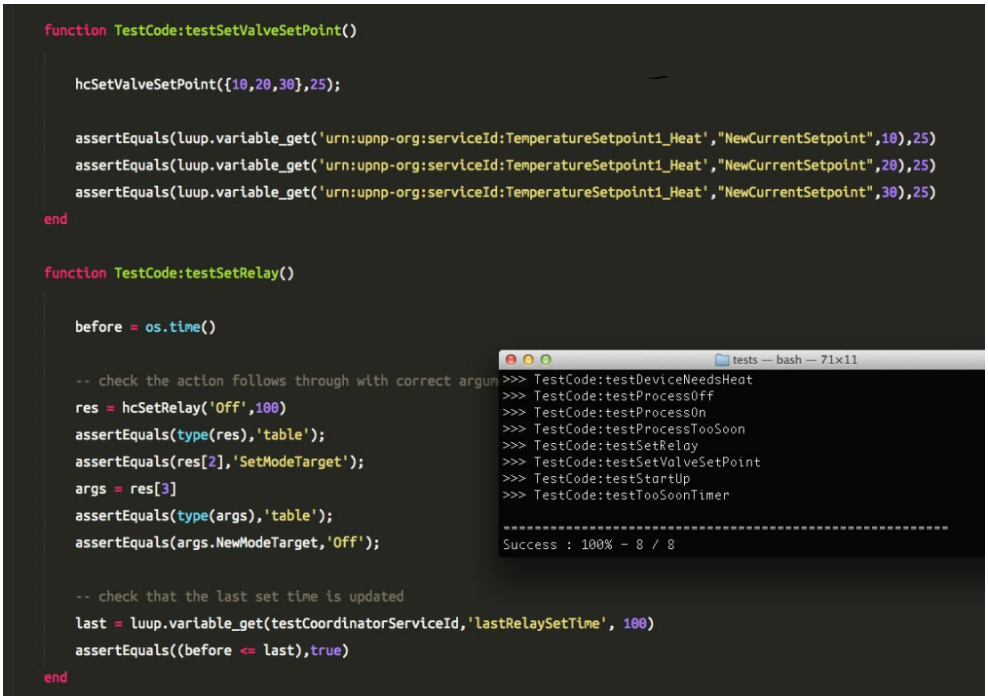
6.2.1.5 Shake

Shake je testovací engine pro Lua a předpokládá, že testy používají pouze standardní assert funkce. Dstribuuje se jako modul Lua a spouští se z příkazové řádky a CGILua aplikace.

Shake je open-source a používá stejnou licenci jako Lua 5.1. [33]

6.2.1.6 telescope

Telescope je knihovna pro Lua, která umožňuje deklarativní testy, kompatibilní s Lua 5.1 a 5.2. Další funkce typu assert se dají snadno přidávat. Testy



Obrázek 6.8: Příklad kódu testu - LuaUnit [32]

```
1      items = 10
2      -- checks the correct case
3      assert (items == 10, "this should not fail")
4
5      items = 20
6      -- checks an overflow case
7      assert (items == 10, "wrong number of items")
8
9      print("Verifying the total")
10     items = 10
11     total = 30
12     assert (items == total, "wrong total")
```

Obrázek 6.9: Příklad kódu testu - Shake [34]

```
~/workspace/luafilesystem/tests$ shake
->  test.lua OK!
________________

Tests: 27
Failures: 0
Errors: 0
```

Obrázek 6.10: Ukázka výstupu OK - Shake [34]

```
:~/workspace$ shake
----------------    test.lua failed!   ----------------

-- checks an overflow case
   #7 assert (items == 10, "wrong number of items")
   items -> 20

Verifying the total
   #12 assert (items == total, "wrong total")
   items -> 10
   total -> 30
________________

Tests: 3
Failures: 2
Errors: 0
```

Obrázek 6.11: Ukázka výstupu FAIL - Shake [34]

se spouští přes příkazový řádek a umožňuje vstup Lua snippet callbacks, díky čemuž lze využít například debugger při neúspěšných testech. [35]

```
context("A context", function()
  before(function() end)
  after(function() end)
  context("A nested context", function()
    test("A test", function()
      assert_not_equal("ham", "cheese")
    end)
    context("Another nested context", function()
      test("Another test", function()
        assert_greater_than(2, 1)
      end)
    end)
  end)
  test("A test in the top-level context", function()
    assert_equal(3, 1)
  end)
end)
```

Obrázek 6.12: Příklad kódu testu - telescope [35]

6.2.1.7 Lua-TestMore

Lua-TestMore je rozšiřitelný testovací framework pro jazyk Lua umožňující snadné psaní testů bez objektově orientovaného programování. Testy lze označit jako k dopracování, nebo mohou být přeskočeny. [36]

6.2.1.8 Busted

Busted je testovací framework, jehož hlavním cílem je snadné používání. Jeho požadavkem je Lua 5.1 nebo novější. Testy psané v tomto frameworku jsou

```
--------------------------------------------------------------------------------
A context:
A nested context:
  A test                                                                  [P]
  Another nested context:
    Another test                                                          [P]
A test in the top-level context                                           [F]
--------------------------------------------------------------------------------
A test with no context                                                    [U]
Another test with no context                                              [U]
--------------------------------------------------------------------------------
This is a context:
This is another context:
  this is a test                                                          [U]
  this is another test                                                    [U]
  this is another test                                                    [U]
--------------------------------------------------------------------------------
8 tests 2 passed 3 assertions 1 failed 0 errors 5 unassertive 0 pending

A test in the top-level context:
Assert failed: expected '3' to be equal to '1'
stack traceback:
  ...ib/luarocks/rocks//telescope/scm-1/lua/telescope.lua:139: in function 'assert_equal'
  example.lua:18: in function <example.lua:17>
  [C]: in function 'pcall'
  ...ib/luarocks/rocks//telescope/scm-1/lua/telescope.lua:330: in function 'invoke_test'
  ...ib/luarocks/rocks//telescope/scm-1/lua/telescope.lua:362: in function 'run'
  ...usr/local/lib/luarocks/rocks//telescope/scm-1/bin/ts:147: in main chunk
  [C]: ?
```

Obrázek 6.13: Ukázka výstupu - telescope [35]

```lua
-- 99example.t
#!/usr/bin/lua
require 'Test.More'

plan(9)

ok(true, "true")
ok(1, "1 is true")
nok(false, "false")
nok(nil, "nil is false")

is(1 + 1, 2, "addition")

like("with aaa", 'a', "pattern matches")
unlike("with aaa", 'b', "pattern doesn't match")

error_like([[error 'MSG']], '^[^:]+:%d+: MSG', "loadstring error")
error_is(error, { 'MSG' }, 'MSG', "function error with param")
```

Obrázek 6.14: Příklad kódu testu - Lua-TestMore [37]

```
$ lua 99example.t
1..9
ok 1 - true
ok 2 - 1 is true
ok 3 - false
ok 4 - nil is false
ok 5 - addition
ok 6 - pattern matches
ok 7 - pattern doesn't match
ok 8 - loadstring error
ok 9 - function error with param
```

Obrázek 6.15: Ukázka výstupu - Lua-TestMore [37]

```
$ prove 99example.t
99example.t .. ok
All tests successful.
Files=1, Tests=9,  0 wallclock secs ( 0.05 usr +  0.20 sys =  0.25 CPU)
Result: PASS
```

Obrázek 6.16: Ukázka výstupu s prove - Lua-TestMore [37]

snadno čitelné a ne příliš obsáhlé. Lze řetězit funkci assert a negace jako assert.not.equals. Do kódu testu lze přidávat popisy a tagy, takže lze spustit pouze konkrétní sadu testů. Knihovnu lze rozšířit o další assert funkce, případně zaregistrovat vlastní fráze.

Modulární výstup knihovny umožňuje přidání vlastního výstupního formátu. JSON umožnuje spouštět Busted na většině CI serverech. [38]

6.2.1.9 Gambiarra

„Gambiarra je Lua verze Kludjs a její ideou je minimální jednotkové testování.“ [39]

6.2.1.10 luaspec

„Specifický styl testovacího frameworku pro Lua.“ [40]

6.2.1.11 PenLight

„Knihovny PenLight obsahují malou knihovnu pro testování jednotek, převážně pro vnitřní testy.“ [41]

6.2.1.12 Build-test-deploy (BTD)

„Testování jednotek inspirované jUnit a založeno na LuaUnit.“ [41]

```lua
describe("Busted unit testing framework", function()
  describe("should be awesome", function()
    it("should be easy to use", function()
      assert.truthy("Yup.")
    end)

    it("should have lots of features", function()
      -- deep check comparisons!
      assert.are.same({ table = "great"}, { table = "great" })

      -- or check by reference!
      assert.are_not.equal({ table = "great"}, { table = "great"})

      assert.truthy("this is a string") -- truthy: not false or nil

      assert.True(1 == 1)
      assert.is_true(1 == 1)

      assert.falsy(nil)
      assert.has_error(function() error("Wat") end, "Wat")
    end)

    it("should provide some shortcuts to common functions", function()
      assert.are.unique({{ thing = 1 }, { thing = 2 }, { thing = 3 }})
    end)

    it("should have mocks and spies for functional tests", function()
      local thing = require("thing_module")
      spy.on(thing, "greet")
      thing.greet("Hi!")

      assert.spy(thing.greet).was.called()
      assert.spy(thing.greet).was.called_with("Hi!")
    end)
  end)
end)
```

Obrázek 6.17: Příklad kódu testu - Busted [38]

```
●●●●●●●●●●●●●●●●●
14 successes, 1 failure, and 1 pending in 2.000669 seconds.

Pending → ./spec/busted_core_spec.lua @ 118
I'll do it later.

Failure → ./spec/busted_core_spec.lua @ 122
explodes
bin/../lib/assert.lua:24: this should fail
```

Obrázek 6.18: Ukázka výstupu - Busted [38]

```
-- Simple synchronous test
test('Check dogma', function()
    ok(2+2 == 5, 'two plus two equals five')
end)

-- A more advanced asyncrhonous test
test('Do it later', function(done)
    someAsyncFn(function(result)
        ok(result == expected)
        done()     -- this starts the next async test
    end)
end, true)     -- 'true' means 'async test' here
```

Obrázek 6.19: Ukázka výstupu - Gambiarra [39]

```
require 'luaspec'
require 'stack'

describe["A Stack"] = function()
        before = function()
                s = Stack:new()
        end

        it["should be empty to start with"] = function()
                expect(s:top()).should_be(nil)
        end

        it["should allow items to be pushed"] = function()
                s:push(30)
                expect(s:top()).should_be(30)
        end

        it["should error when pop is called on an empty stack"] = function()
                expect(function() s:pop() end).should_error()
        end

        it["this is pending"] = pending

        describe["Calling pop on an empty stack"] = function()
                before = function()
                        err = track_error(function() s:pop() end)
                end

                it["errror should contain 'Nothing on Stack'"] = function()
                        expect(err).should_match(".*Nothing on the stack.*")
                end
        end

end

```

Obrázek 6.20: Příklad kódu testu - luaspec [40]

```lua
-- another SIP example, shows how an awkward log file format
-- can be parsed. It also prints out the actual Lua string
-- pattern generated:
-- SYNC%s*%[([+%-%d]%d*)%]%s*([+%-%d]%d*)%s*([+%-%d]%d*)

require 'pl'

s = [[
SYNC [1] 0 547 (14679 sec)
SYNC [2] 0 555 (14679 sec)
SYNC [3] 0 563 (14679 sec)
SYNC [4] 0 571 (14679 sec)
SYNC [5] -1 580 (14679 sec)
SYNC [6] 0 587 (14679 sec)
]]


local first = true
local start
local res = {}
local pat = 'SYNC [$i{seq}] $i{diff} $i{val}'
print(sip.create_pattern(pat))
local match = sip.compile(pat)
for line in stringx.lines(s) do
  if match(line,res) then
    if first then
      expected = res.val
      first = false
    end
    print(res.val,expected - res.val)
    expected = expected + 8
  end
end
```

Obrázek 6.21: Příklad kódu testu - PenLight [42]

```lua
local test = require('btd.lua.testapi)
...
TestClass = {}

TestClass:testMethod()
  a = {1,2,3}
  b = DeepCopy(a)
  test.equals(a,b)
  b[1] = 5
  test.differs(a,b) -- my DeepCopy method works
end
```

Obrázek 6.22: Příklad kódu testu - BTD [43]

6.2.1.13 Testy

Testy je další framework pro jednotkové testování pro jazyk Lua. Jeho požadavkem je Lua 5.1 a vyšší. Testovací funkce lze vložit do kódu modulu, aniž by se zasahovalo do veřejného rozhraní. Kód psaný v Testy vypadá jako psaný programovacím jazyku Lua. Nové moduly testy.extra testují očekávanou návratovou hodnotu, očekávané hodnoty iterací, očekávané chyby a tak dále. [44]

```lua
-- module1.lua
local M = {}

function M.func1()
  return 1
end

-- this is a test function for the module function `M.func1()`
local function test_func1()
  assert( M.func1() == 1, "func1() should always return 1" )
  assert( M.func1() ~= 2, "func1() should never return 2" )
  assert( type( M.func1() ) == "number" )
end

function M.func2()
  return 2
end

-- this is a test function for the module function `M.func2()`
local function test_func2()
  assert( M.func2() == 2 )
  assert( M.func2() ~= M.func1() )
end

return M
```

Obrázek 6.23: Příklad kódu testu - Testy [44]

```
$ testy.lua module1.lua
func1 ('module1.lua')   ...
func2 ('module1.lua')   ..
5 tests (5 ok, 0 failed, 0 errors)
```

Obrázek 6.24: Ukázka výstupu - Testy [44]

6.2.1.14 Minctest

Minctest je dalším nástrojem pro testování jednotek v jazyku Lua, originálně napsán v ANSI C. Je vhodný převážně pro malé projekty. Implementuje hlavně assert aporovnávací funkce. U testů, které selhaly, se v reportu objeví na kte-

rých řádcích testovacího kódu došlo k selhání. Pokud test selhal, pokračuje testování dál.

Licence Minctest je volná téměř pro každé použití. [45]

```
require "minctest"

lrun("test1", function()
    lok('a' == 'a');          --assert true
end)

lrun("test2", function()
    lequal(5, 5);             --compare integers
    lfequal(5.5, 5.5);        --compare floats
end)

return lresults();            --show results
```

Obrázek 6.25: Příklad kódu testu - Minctest [45]

```
        test1         pass: 1   fail: 0      0ms
        test2         pass: 2   fail: 0      1ms
ALL TESTS PASSED (3/3)
```

Obrázek 6.26: Ukázka výstupu - Minctest [45]

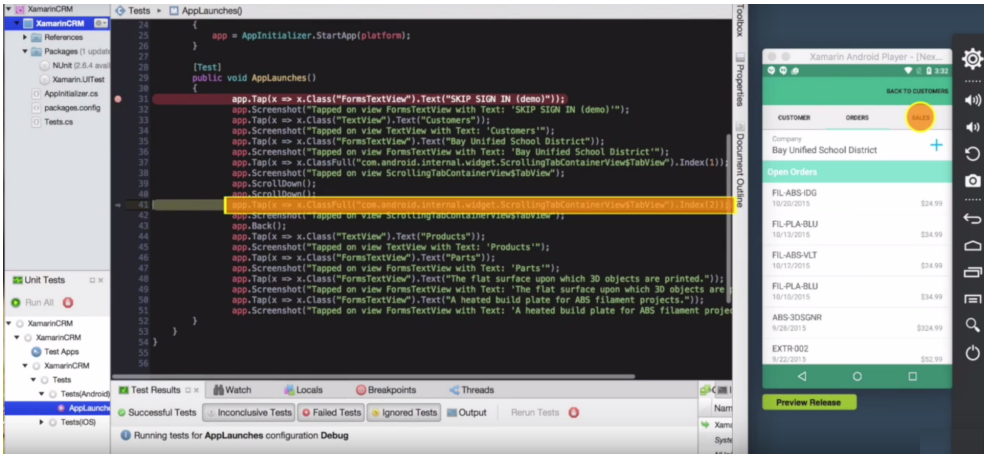
6.3 Testovací nástroje pro Microsoft Visual Studio

6.3.1 Xamarin Test Recorder

Xamarin Test Recorder je nástroj určený k automatizaci testů aplikací pro mobilní zařízení (iOS a Android). Jeho funkce je založena na nahrávání interakce s aplikací, které je pak schopen znovu reprodukovat. Umí zaznamenat dotyk obrazovky, přetažení přes obrazovku, změnšení/zvětšení pomocí dvou prstů a rotaci zařízení.

Zaznamenané testy uživatel nahrává na Xamarin test cloud - službu pro testování mobilních aplikací, kde může test spustit na více než 2000 zařízeních s různými operačními systémy. Pro výsledky testů Xamarin test cloud poskytuje uživatelsky příjemné rozhraní s množstvím funkcí.

Zaznamenané testy jsou napsány jako UI testy pro C#. [46]



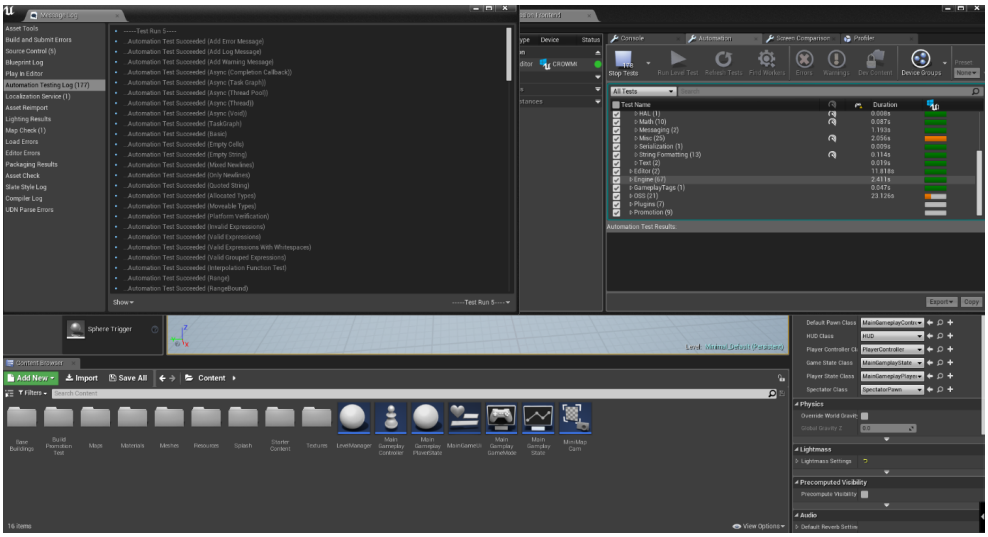
Obrázek 6.27: Xamarin ukázka testu [46]

6.4 Testovací nástroje pro Unreal Engine

Unreal Engine má svůj vlastní nástroj pro automatické testování.

„Automatický systém umožňuje provádět testování jednotek, funkční testování a Stres testy obsahu pomocí Unreal Message s cílem zvýšení stability. Systém automatizace funguje provedením jednoho, nebo více automatických testů. Testy mohou být rozděleny do jednotlivých kategorií v závislosti na jejich účelu, nebo funkci:

- Jednotkové testy – Testy ověřující úroveň API.
- Funkční testy – Testy na systémové úrovni ověřující statistiky ve hře, změny rozlišení.
- Smoke testy – Testy určené k tomu, aby běžely co nejrychleji, aby se daly spouštět před samotným spuštěním projektu v editoru apod. .
- Obsahové stress testy – Hlubší testování jednotlivých systémů pro eliminaci pádů aplikace jako např. načítání map a kompilování všech blueprintů (= skript vytvořený systémem grafického programování v Unreal Engine 4).
- Porovnávání snímků obrazovky – Nástroj, umožňující porovnávání snímků obrazovky a vyhledávání jejich rozdílů. Tím umožňuje rychlé vyhledávání např. renderovacích chyb.“ [47]



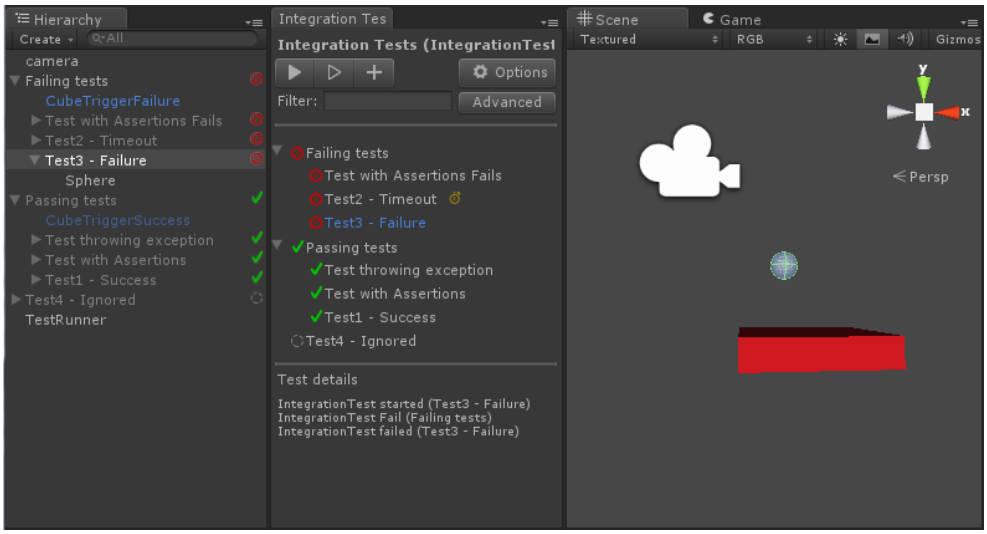
Obrázek 6.28: Příklad testu - Unreal Engine

6.5 Testovací nástroje pro Unity

Unity má svůj vlastní nástroj pro psaní testů.

„Unity Test Tools je testovací framework pro hry a interaktivní obsah vytvořený v Unity. Pomocí těchto nástrojů je možné vytvářet od základu kvalitní produkty. Unity Test Tools je nastaven tak, aby to vypadalo jako přirozené rozšíření editoru. Testy jsou dostupné zdarma v Asset Store.“ [48]

Testovací nástroje pro Unity umožňují integrační testování pomocí Integration Test Framework. Testy se navrhují tak, aby běžely na samostatné scéně, která může obsahovat více testů, ale v jednu chvíli může běžet pouze jeden test. Na scénu se umisťují testovací objekty, které mohou být ve skupinách podle testů. Test lze poté spustit na zvolené platformě [49]



Obrázek 6.29: Unity ukázka scény integračního testu - 7 testovacích objektů a 2 skupiny, kde první prochází, druhá selhává [49]

Kapitola 7

Multiplatformní nástroje pro automatizované testování

Pokud pro automatické testování zvolíme multiplatformní nástroje, musíme počítat s tím, že ve většině případů pracují na principu černé skříňky. Testovací nástroj tedy "nevidí"do kódu programu a takové testy najdou využití především pro jednodušší aplikace, případně hry. Jsou skvělé na otestování uživatelského rozhraní - například zda položky v menu odkazují, kam mají.

Multiplatformní nástroje fungují na princpu automatizace akcí - do kódu testu se tedy nějakým způsobem nahraje, jaké akce má nástroj po sobě dělat. Toto nahrávání se dělá několika způsoby. Buď přes vyfocení prvnku na obrazovce (Sikuli, TestQuest), se kterým bude test interagovat a vložení snímku do kódu testu, nebo jiné nástroje přímo pomocí rekordéru test nahrávají (Robotium, T-Plan Robot, TestQuest, Ranorex, MonkeyTalk). Poslední možností je samotné psaní kódu (Appium, SilkMobile, Selendroid) .

Některé multiplatformní nástroje lze dokonce kombinovat a vznikne tak ještě lepší prostředí pro automatizované testování.

7.1 Sikuli

Sukuli je framework určený pro automatizaci. Pracuje na principu rozpoznávání obrazu. Dokáže rozeznat a identifikovat jednotlivé komponenty grafického rozhraní a tak s nimi interagovat. Sikuli byl vyvynut pro automatizaci čehokoliv, tedy nejen pro automatizované testy.

Sikuli se řadí mezi open-source projekty a jeho softwarové balíky najdete pod MIT licencí. [50]

Jedná se o platformně nezávislý nástroj - jde s ním tedy pracovat na jakémkoliv operačním systému ať už se jedná o Windows, Linux, nebo MAC. Podporuje virtuální zařízení a mobilní simulátory - Android a iOS, vzdálenou plochu a web s Flash, HTML a JavaScript. [51]

Nástroj Sikuli je vhodný především pro testování uživatelského rozhraní. Výborně umí nahradit práci testerů, kteří kontrolují, zda se na vybrané obrazovce nachází přesně to, co podle popisu má. Na základě porovnávání obrázků - vzorového (jak to má vypadat) a výsledného (co vyšlo z aplikace) umí najít rozdíly na obrázcích obrazovky a určit jejich odlišnosti.

Tento nástroj umí interagovat s určitou komponentou, aniž by znal název, nebo ID této komponenty. Díky grafickému rozpoznávání tak stačí danou komponentu vyfotit a nahrát tento obrázek přímo do kódu programu. Mezi funkce Sikuli patří nalezení komponenty podle obrázku, kliknutí na komponentu, nalezení konkrétní komponenty na obrazovce (pokud je jich několik stejných) podle grafického parametru a další. [52]

Pro testování aplikací na Android zařízení postačí emulátor a apk aplikace, kterou chceme testovat.

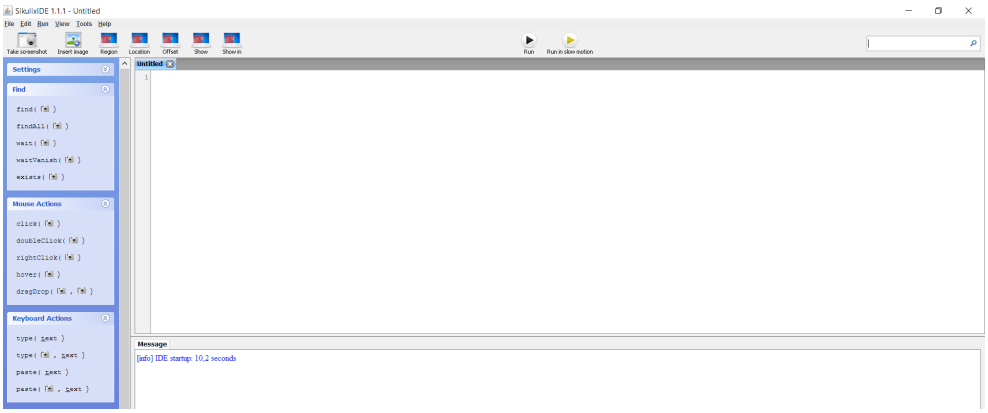
7.1.1 Vlastní vyzkoušení nástroje Sikuli

Sikuli je velmi intuitivní a jednoduché prostředí. Tvorba automatizovaných kroků je velice snadná. Vytvoření testu, kde je třeba aplikaci proklikat a zjistit, zda se zobrazilo, co mělo, zabere pár minut.

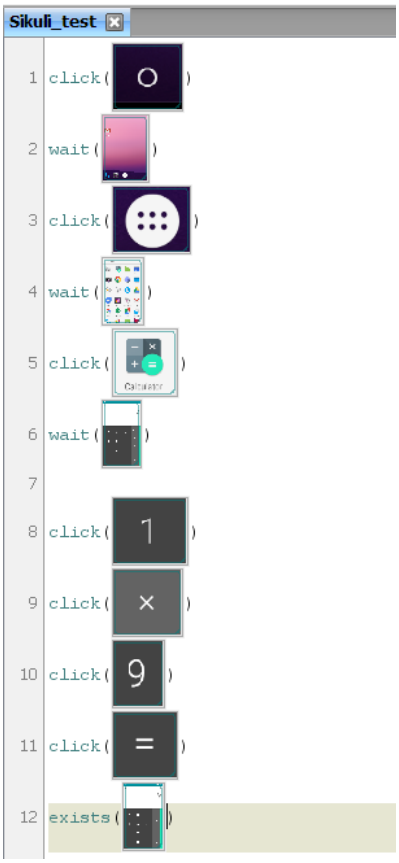
Pro tvorbu složitějších testů lze použít podmínky, for cykly a další funkce.

Pro spuštění automatizace pak stačí kliknout na "Run"a test se spustí. Sikuli přímo používá myš uživatele. V případě problému s provedením zadané akce Sikuli přímo označí řádek, kde nastala chyba, skript se přeruší a výpis z programu je červený. Pokud žádná chyba nenastala, výpis je zelený.

Pro vyzkoušení nástroje Sikuli jsem zvolila aplikaci Calculator, kterou jsem testovala na emulátoru. Vytvořila jsem jednoduchý test pro ukázku tvorby jednotlivých kroků testu.



Obrázek 7.1: Sikuli ukázka prostředí pro psaní testů



Obrázek 7.2: Sikuli ukázka ukázka testu kalkulačky

7.2 eggPlant Mobile

eggPlant provádí testování s použitím iOS Gateway a Android Gateway. V mobilním testování se používá pro testování mobilních aplikací na iOS, Android, Windows Phone, BlackBerry a jiné. eggPlant je zaměřený na obrazové testování uživatelského rozhraní a skrze uživatelské rozhraní probíhá také interakce s testovanou aplikací. [53]

Nástroj eggPlant se řadí mezi placené, ale za to poskytuje množství možností. Pro přesné ceny za jednotlivé licence je třeba společnost, vydávající tento software, kontaktovat. Společnost testPlant mi pro účely bakalářské práce poskytla 4 týdenní klíč pro trial verzi.

Testy se musí spouštět z počítače (Windows, Linux, nebo Mac) s prostředím eggPlant Functional, ale mohou běžet na libovoném zařízení (desktop, virtuální počítač, mobilní zařízení), kterému se říká SUT. Pro přenos příkazů myší nebo klávesnicí se používá Virtual Network Computing (VNC) nebo Remote Desktop Computing (RDP). Příkazy se vztahují k obrázkům objektů

Obrázek 7.3: Sikuli ukázka bezchybného i chybového výstupu

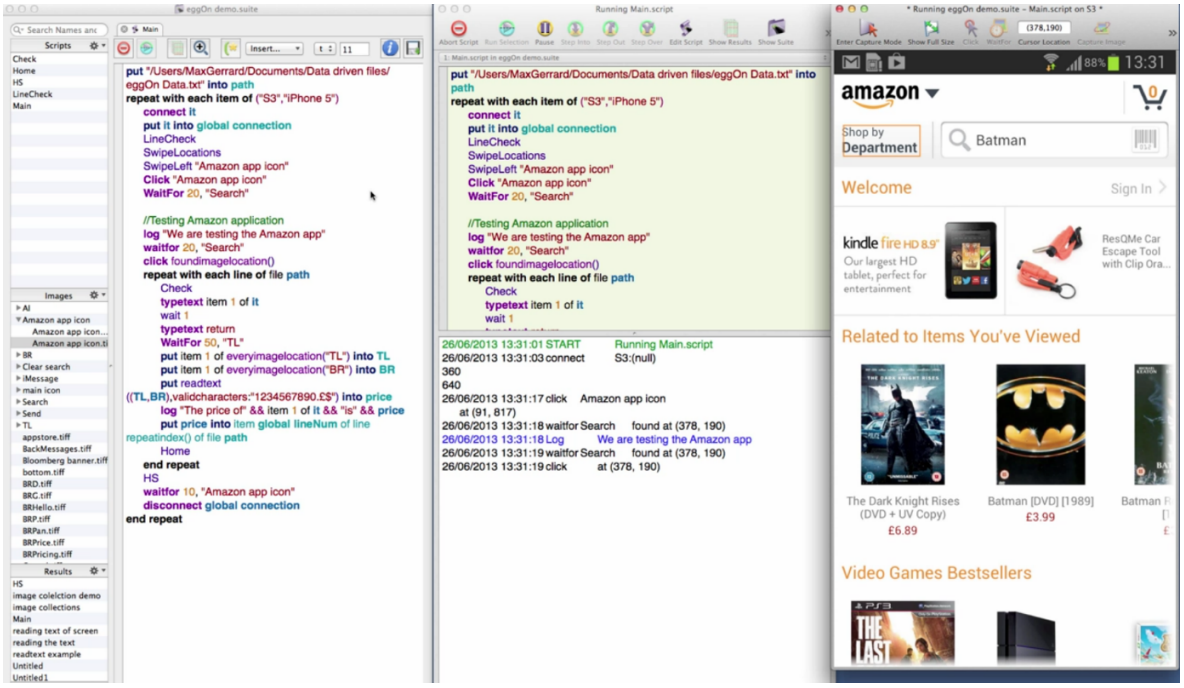
(např. kliknout na ikonu aplikace), takže příkaz se může vztahovat k čemukoliv, co je "vidět"na obrazovce. V průběhu testu lze pořizovat snímky obrazovky.

Po proběhnutí testu jsou k dispozici výsledky testů, statistiky a podrobné informace o běhu testu. [54]

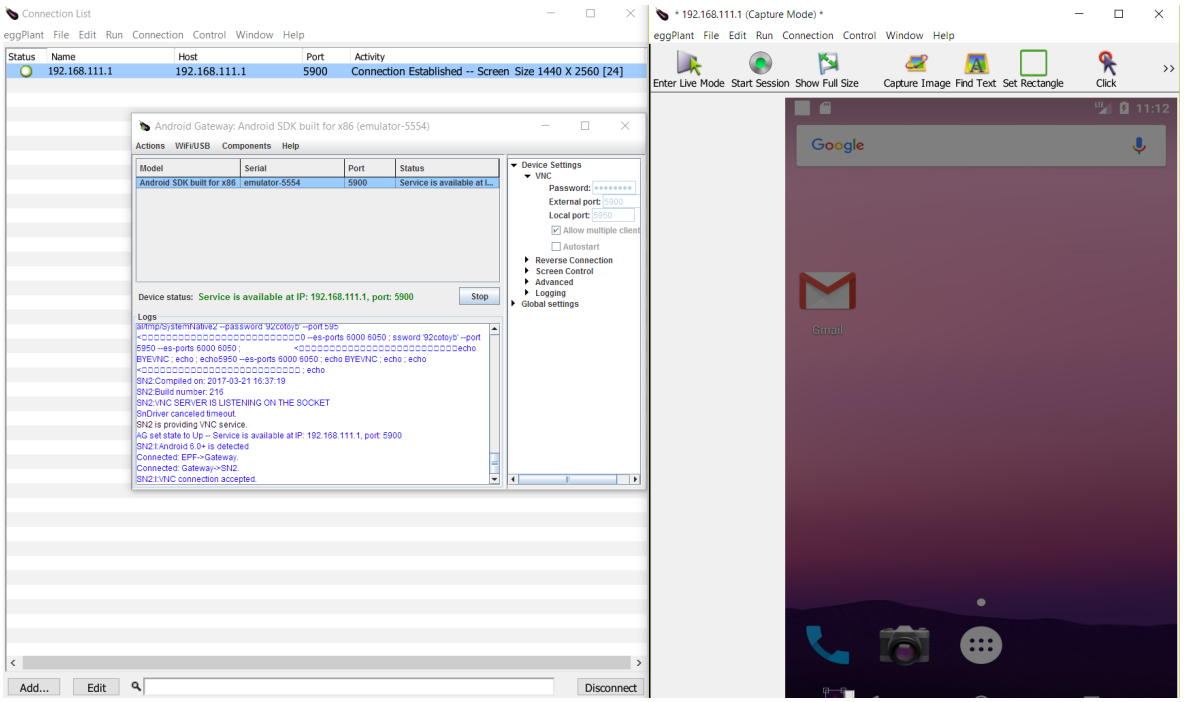
Instalalace tohoto nástroje je složitější, ale obsáhlá dokumentace k softwaru, která obsahuje i instrukce k instalaci, mi s tímto problémem velmi pomohla. Pro vytváření testů pro Android zařízení je nutné nainstalovat balíček eggPlant Functional, dále pak Android Gateway a VNC server - já využila RealVNC.

Vytváření jednotlivých scénářů probíhá tak, že se jednotlivé kroky natočí, upraví se jednotlivé zachycené obrázky během natáčení a následně se ze zachycených kroků vygeneruje skript, který je možné dále upravovat. Vytváření testů je tedy poměrně jednoduché.

Musím podotknout, že tento nástroj je náročnější na výkon počítače a konkrétně můj notebook s ním měl velké problémy.



Obrázek 7.4: Příklad testu - eggPlant [53]



Obrázek 7.5: Propojení nástroje eggPlant s emulátorem

7.3 Appium

Hlavním cílem nástroje Appium je, aby testování aplikací nevyžadovalo SDK nebo rekompilaci aplikace. Jedná se o open-source projekt, zaměřující se na automatizaci jakékoliv mobilní aplikace vytvořené v jakémkoliv programovacím jazyku, či prostředí s přístupem do back-end API a databáze z testovaného kódu.

Appium podporuje tvorbu testů v následujících jazycích: Python, Node, Java, Ruby, PHP, .NET, Perl a dalších. [55] Cílem Appium je poskytnout nástroj pro jakýkoliv programovací jazyk a jakýkoliv framework bez nutnosti změny aplikace, nebo zdrojového kódu, či přístupu k němu.

Tento framework je vyvíjen pod open-source Apache 2.0 Licencí a lze s ním vytvářet testy pro Android pod platformou OS X, Windows i Lunux. Podporovány jsou nativní, hybridní i webové mobilní aplikace. [56]

Samotný test pro Android aplikaci lze psát v programovacím jazyku Java, případně Ruby a provádí se na emulátoru.

```
1  package com.saucelabs.appium;
2
3  import org.junit.Test;
4  import org.openqa.selenium.By;
5  import org.openqa.selenium.WebElement;
6
7  import java.util.List;
8
9  import static org.junit.Assert.assertEquals;
10
11 public class AndroidTest extends BaseDriver{
12
13
14     @Test
15     public void apiDemo(){
16         WebElement el = driver.findElement(By.xpath(".//*[@text='Animation']"));
17         assertEquals("Animation", el.getText());
18         el = driver.findElementByClassName("android.widget.TextView");
19         assertEquals("API Demos", el.getText());
20         el = driver.findElement(By.xpath(".//*[@text='App']"));
21         el.click();
22         List<WebElement> els = driver.findElementsByClassName("android.widget.TextView");
23         assertEquals("Activity", els.get(2).getText());
24     }
25
26 }
```

Obrázek 7.6: Appium příklad multiplatformního kódu testu [57]

```
module Screens
  module Android

    class MyobLogin < Screens::MyobLogin

      def title
        "MYOB"
      end

      def set_email(name)
        email_field = find_element(:id, "login_view_username")
        email_field.type name
      end

      def set_password(password)
        password_field = find_element(:id, "id/login_view_password")
        password_field.type password
      end

    end
  end
end
```

Obrázek 7.7: Appium příklad kódu jednotkového testu [58]

7.4 Robotium

Robotium je další z frameworků, které jsou určené k automatickému testování pro Android a podporuje jak nativní, tak i hybridní aplikace. Jeho funkčnost se opírá o černou skříňku. V tomto nástroji se snadno píšou funkční testovací scénáře, které prochází napříč několika Android aktivitami.

Pro psaní testů v Robotium je třeba minimální znalost testované aplikace. Ve srovnání se standardními testovacími nástroji je čitelnost testovacích scénářů snadnější. [59]

7.4.1 Robotium Recorder

Robotium Recorder je mocný nástroj, který umožňuje automatizovat testování mobilních aplikací s využitím prostředí, ve kterém byla aplikace vytvořena - Eclipse, nebo Android Studio. Robotium v tomto případě funguje jako plugin. K testování lze použít fyzické zařízení, nebo emulátor.

Pro nahrávání testu je třeba vytvořit samostatný soubor - Robotium Test a vybrat apk soubor testovaného projektu. Po spuštění aplikace Robotium Recorder nahrává všechny aktivity, které se spustí. Robotium Recorder postupně nahrává všechny kroky, které v aplikaci provedeme a ukládá si je. Zároveň umí pomocí funkce assert kontrolovat, zda se zobrazilo, co mělo. Dále umožňuje jednotlivé kroky mazat a přidávat pořízení snímku obrazovky. Zaznamenaný test se uloží jako třída Java a může být jednoduše znovu spuštěn.

Počet testovacích scénářů není nijak omezen. [60]

```
@MediumTest
public void testStartSecondActivity() throws Exception {
  final String fieldValue = "Testing Text";

  // Set a value into the text field
  solo.enterText((EditText) solo.getView(R.id.etResult), fieldValue);

  // find button and click it
  solo.clickOnButton("Launch");
  // or solo.clickOnView(solo.getView(R.id.btnLaunch));

  // Wait 2 seconds for the start of the activity
  solo.waitForActivity(SecondActivity.class, 2000);
  solo.assertCurrentActivity("Should be second activity", SecondActivity.class)
  // Search for the textView
  TextView textView = (TextView) solo.getView(R.id.tvResult);

  // Validate the text on the TextView
  assertEquals("Text should be the field value", fieldValue,
      textView.getText().toString());

  // Return to the original activity
  // We have to manage the initial position within the emulator
  solo.goBack();
}
```

Obrázek 7.8: Robotium příklad kódu testu [61]

7.5 T-Plan Robot

T-Plan Robot umí simulovat uživatelskou interakci s aplikací a to hned na několika zařízeních ve stejnou chvíli. Podmínkou je, že používaný operační systém podporuje Javu, pak lze automatizovat iOS a Android testy.

T-Plan robot obsahuje vlastní rekordér, pomocí kterého se testy nahrávají a vytvářejí se tak znovu použitelné scripty. Výhodou použití tohoto nástroje je, že není třeba znát testovaný kód, ani mít zkušenosti s programováním. Pomocí tohoto nástroje lze simulovat reálné uživatelské akce jako klepnutí, přetažení, roztahování a přiblížení pomocí dvou prstů, psaní textu a tak dále. [62]

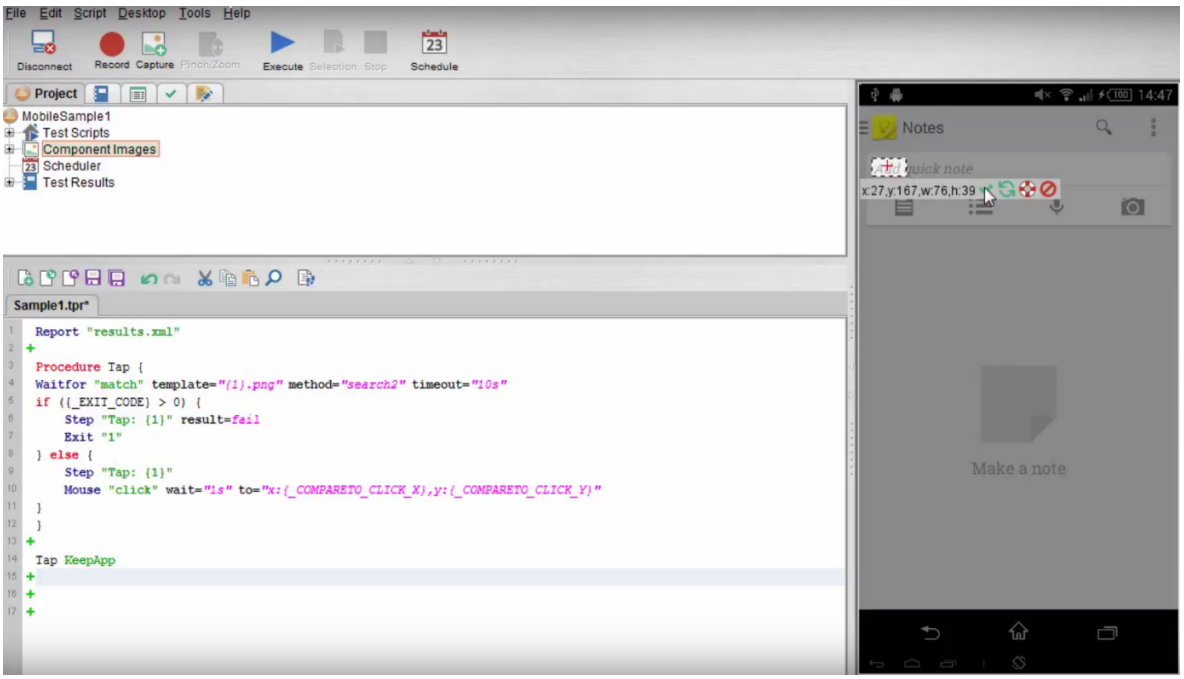
"T-Plan Robot umí identifikovat a ověřit položky na obrazovce pomocí metod pro rozpoznávání obrázků a/nebo textu. Pokročilé algoritmy vyhledávání poskytují rychlý a spolehlivý způsob ověřování přítomnosti položky na obrazovce, často v milisekundách!

Není třeba ani rozsáhlých kódů, ani mapování na architekturu objektu.“ [62]

Celé vytváření testovacích scénářů je navrženo na principu, kdy uživatel dělá zvolené akce, které program ukládá do scriptu. K jednotlivým akcím je možné přidat nějaké podmínky a co se stane při jejich nedodržení. Například čekáme, než se zobrazí nějaká komponenta. Lze určit, jak dlouho se na to bude

čekat, co se stane, pokud se zobrazí/nezobrazí, pokud se jich zobrazí více a tak dále. V rámci testu lze určit, kdy se mají vytvářet snímky obrazovky.

Navržené testovacíé scénáře lze pak jednoduše spustit v rámci tohoto nástroje. [63]



Obrázek 7.9: T-Plan Robot ukázka použití [64]

7.6 TestQuest

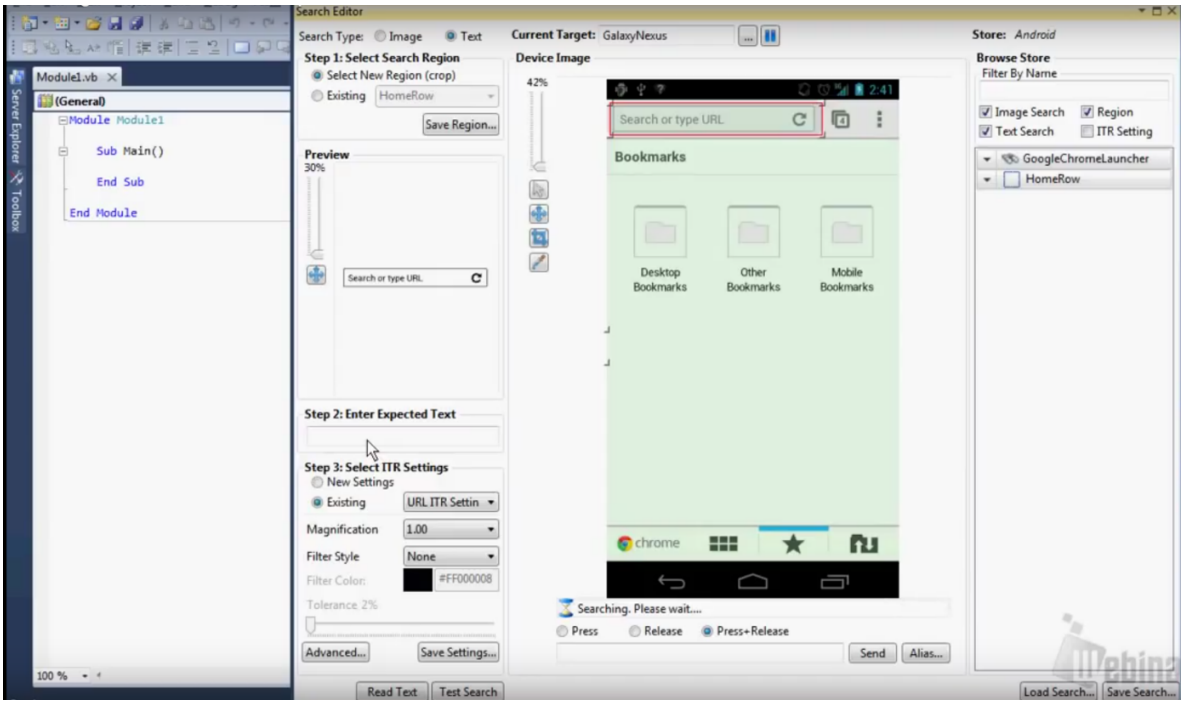
TestQuest 10 je jedno z řešení automatizace testů navržené pro různé softwary. Umožňuje rychle vytvářet testy, které mohou být prováděny v distribuovaném, spolupracujícím prostředí.

TestQuest mohou používat vývojáři napříč celým světem a tak umožňuje snadnou spolupráci rozptýlených týmů. Lze ho integrovat do Microsoft Visual Studio a jiných integrovaných vývojových prostředí (IDE) třetích stran a umožňuje vývoj testů použitím příkazové řádky, C#, Visual Basic 2010, IronPython a IronRuby. Testovací prostředí je kompatibilní s 32 a 64bitovou verzí Windows 7.

Pro vytváření testů slouží TestDesigner a Test Runner urychluje provádění testovacích cyklů automatických testů. Připojení k více testovacím zařízení poskytuje Device Server. [65]

TestQuest 10 funguje jako plugin k Visual Studio. Sérii kroků testů lze nahrávat pomocí rekordéru, nebo kód psát ručně. Psaní kódu ručně se dopo-

ručuje pro tvorbu složitějších testovacích scénářů, které vyžadují podmínky, cykly a podobně. Nahranou sérii kroků lze vygenerovat do kódu a dále kód upravovat. Testy spolupracují s Asset Editorem, který zaznamenává jednotlivé UI prvky (obrázky, text), které lze na obrazovce vidět, například ikonu kalendáře. Tyto prvky lze pak při testování použít třeba pro kliknutí - nástroj je na obrazovce schopný UI prvek rozeznat. [66]



Obrázek 7.10: TestQuest příklad - tvorba UI prvku testovacího scénáře [67]

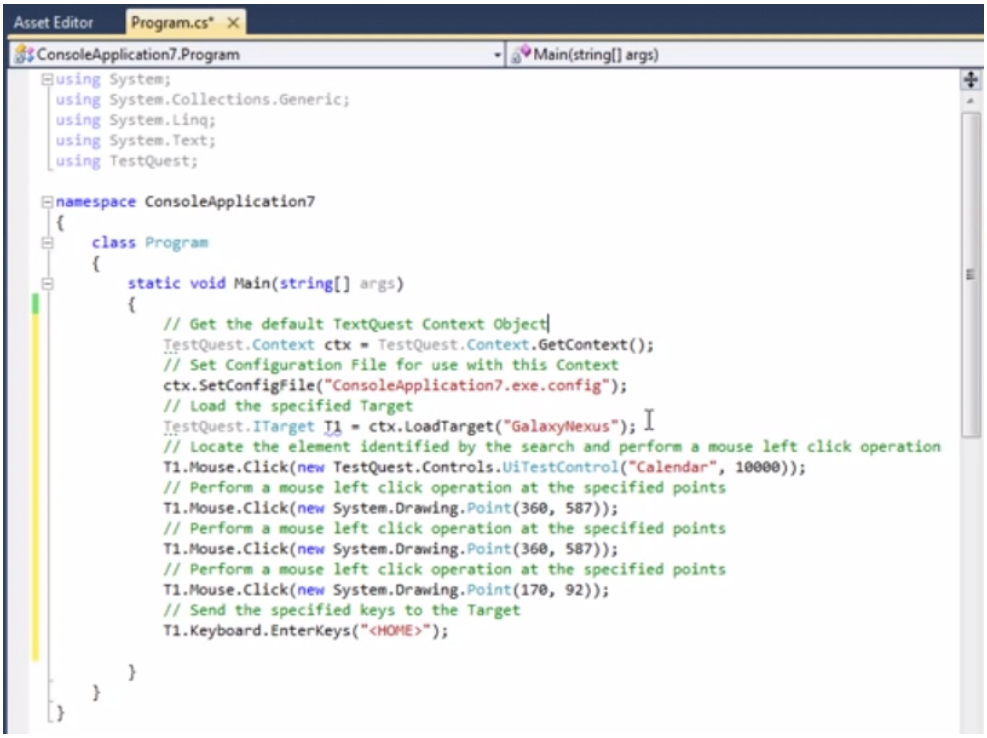
7.7 Silk Mobile

Micro Focus® Silk Mobile ™ poskytuje výkonné mobilní testovací platformy. Poskytuje prostředí pro tvorbu a správu funkčních a zátěžových testů, čímž přispívá k urychlení dodání aplikace na trh. Díky testům dokáže zajistit, že aplikace bude pozitivně přijímána zákazníky.

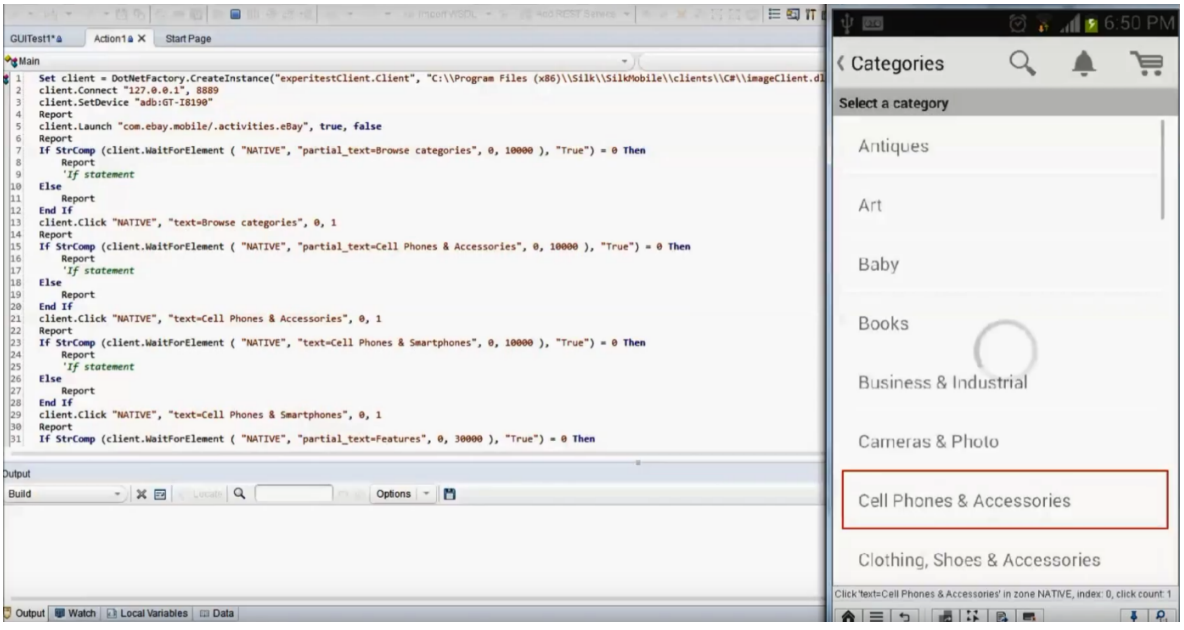
Silk Mobile podporuje mobilní platformy jako iOS a Android. Testování probíhá na reálných zařízeních i na simulátorech. [68]

7.8 Ranorex

Ranorex je řešením automatizace testů, podporující .NET, WFP, Windows Forms, Qt, Java, SAP, Delphi, HTML5, Flash, Flex, Silverlight, iOS, An-



Obrázek 7.11: TestQuest příklad - kód testovacího scénáře [66]

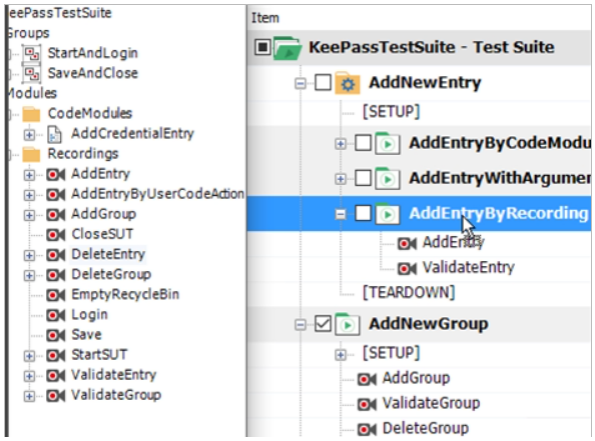


Obrázek 7.12: Silk Mobile příklad použití [67]

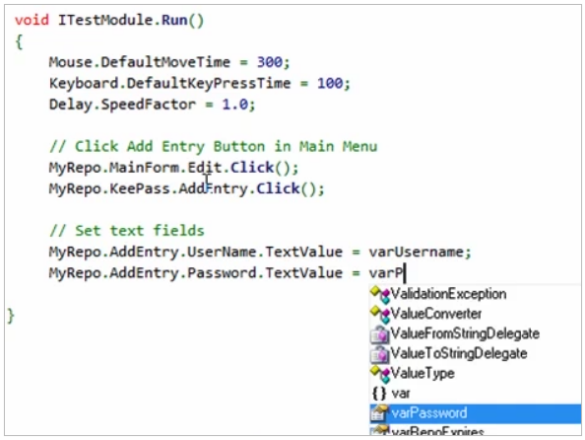
droid, Windows Apps (nativní/hybridní/mobilní webové aplikace) a mnoho dalších. [69]

V prostředí Ranorex je možné psát testovací scénáře s využitím intuitivního drag & drop prostředí nebo testy přímo programovat. Pro programování testů lze použít API pro C# a VB.NET Ranorex umí automatizovat desktopové, mobilní i webové aplikace. [70]

Ranorex Test Recorder umožňuje jednoduché vytváření testovacích kroků. Funguje na principu nahrání akcí rekordérem a jejich uložení jako funkci. Nástroj je pak schopen tyto nahrané akce opakovat.



Obrázek 7.13: Ranorex ukázka vytváření testovacích scénářů pomocí drag & drop [67]

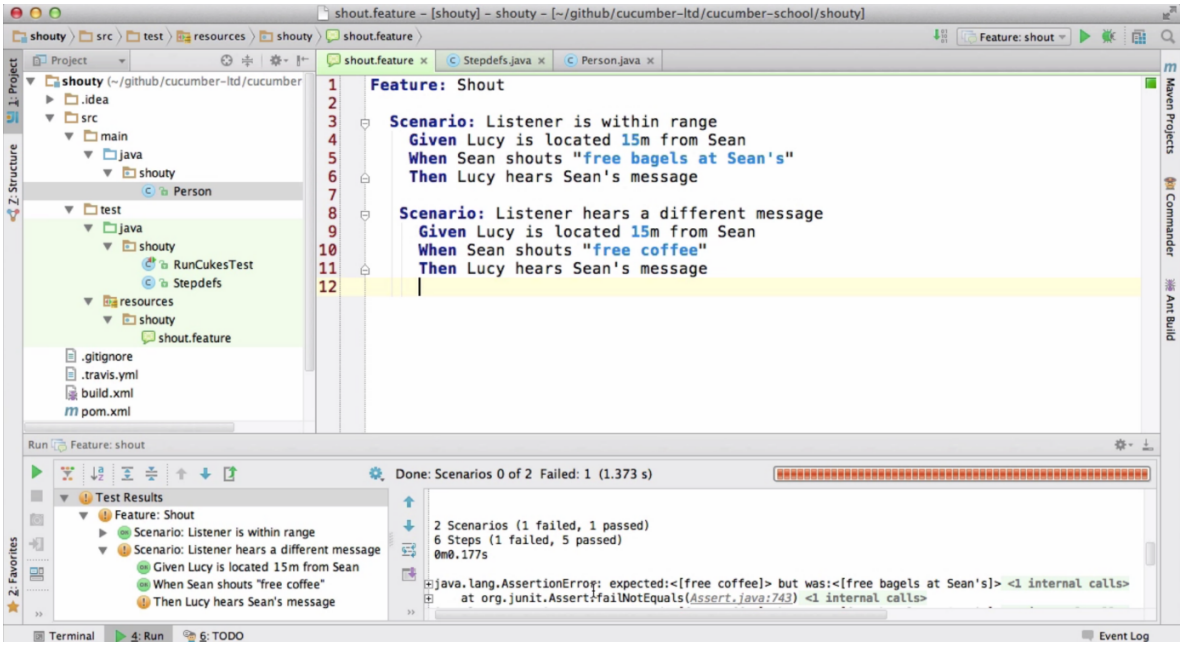


Obrázek 7.14: Ranorex ukázka vytváření testovacích scénářů kódováním [67]

7.9 Cucumber

Cucumber je další testovací nástroj, který definuje chování aplikace s využitím anglického textu - jazykem zvaným Gherkin. Cucumber byl nejprve implementován v Ruby a později rozšířen o Java framework - oba podporují JUnit. [71]

Cucumber je dostupný v implemetacích pro Ruby/JRuby, Java, Groovy, JavaScript, Clojure, Gosu, Lua, .NET, PHP, Jython, C++ a Tcl.



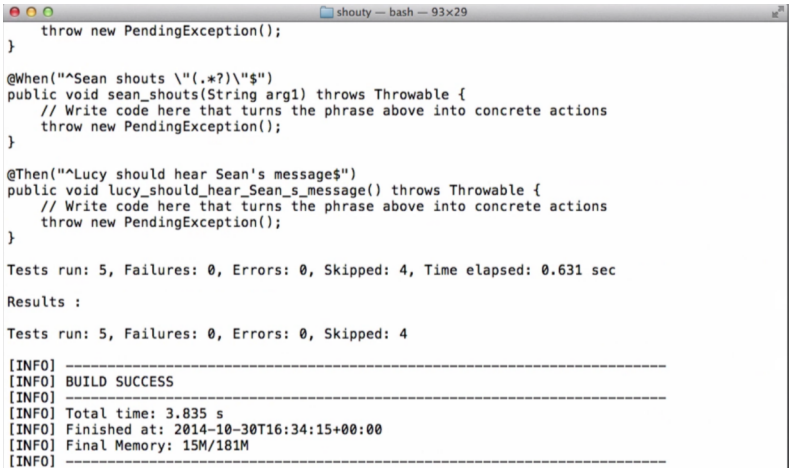
Obrázek 7.15: Příklad kódu testu - Cucumber [72]

7.10 MonkeyTalk

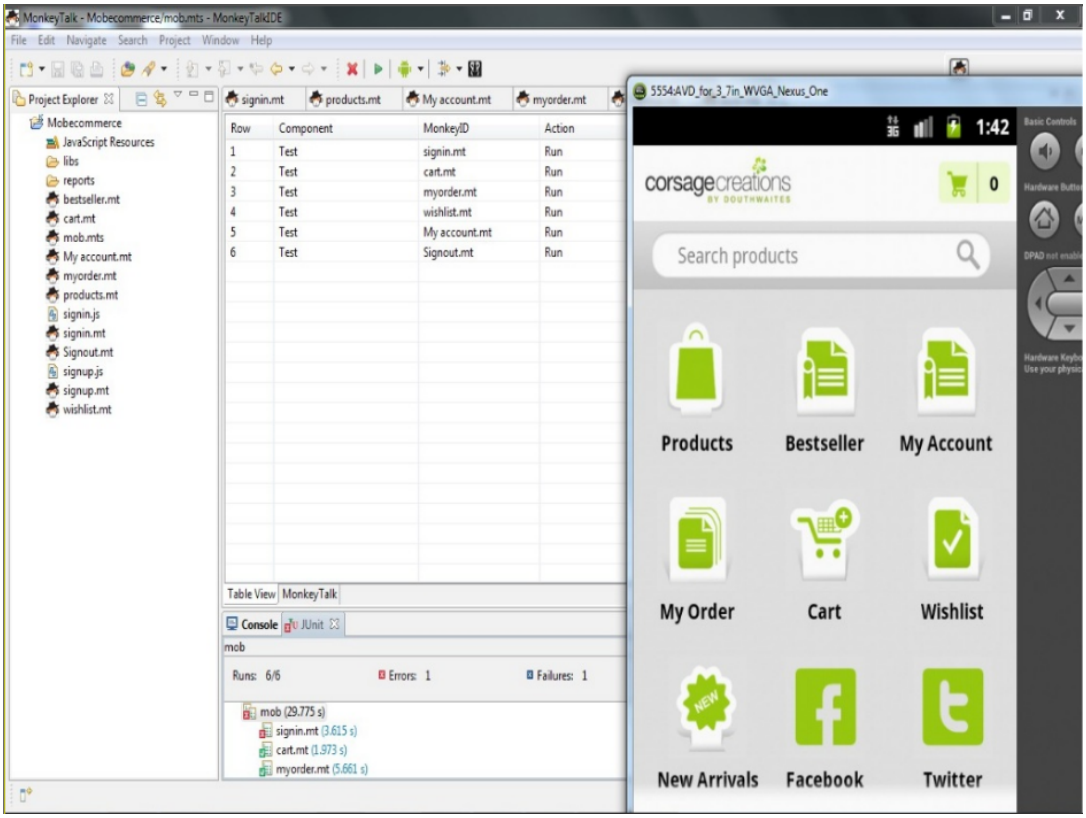
MonkeyTalk se řadí mezi open-source testovací nástroje. Je vyvíjen společností Gorilla Logica. Využívá se pro funkční testování řízené daty a přehrávání funkních testů určených pro iOS a Android. MonkeyTalk umí nahrát interakci s uživatelským rozhraním a pak znovu přehrávat. Aktuální výsledky pak porovnává s očekávanými. Skripty, které MonkeyTalk vytváří jsou čitelné a snadno se vytváří a udržují, automaticky se převádí na JavaScript. Skripty lze libovolně rozšiřovat. [73]

MonkeyTalk lze přidat jako plugin do Eclipse.

MonkeyTalk podporuje operační systémy Windows, Mac a Linux. Spolupracuje s emulátory a hardwarový zařízeními – telefony a tablety. Pro jeho používání není třeba obsáhlých skriptovacích či programovacích znalostí. Výsledky testů jsou generovány ve formátu HTML a XML. [74]



Obrázek 7.16: Příklad výstupu testu - Cucumber [72]



Obrázek 7.17: MonkeyTalk ukázka programu [75]

7.11 Calabash

Calabash je open-source testovacím nástrojem společnosti Xamarin, umožňující psaní a spouštění automatizovaných akceptačních testů mobilních aplikací. Podporuje nativní a hybridní aplikace pro Android a iOS. Umožňuje simulovat interakci uživatele s aplikací využitím funkcí kliknutí, přetahu či rotace zařízení. V případě chyb volá Assert funkce. Dále poskytuje funkce pro pořizování snímků obrazovky, díky kterým lze pak porovnávat, zda se očekávaný výsledek shoduje s výsledkem aktuálním. [76]

Calabash lze použít současně s Xamarin Test Cloud a umožnit tak běhu několika automatickým testům na více než 1000 zařízeních na cloudu.

```
Feature: Initial experience
  As a user I want a helpful and simple initial
  experience with the app. I should be able to get help
  and login to an existing WordPress site.

  @reinstall
  Scenario: Obtaining more information         # features/login.feature:7
    Given I am on the first experience screen  # features/step_definitions/login_steps.rb:39
    And I choose to get more information       # features/step_definitions/login_steps.rb:43
    Then I am taken to the information screen  # features/step_definitions/login_steps.rb:47

  Scenario: Create new account          # features/login.feature:13
    Given I am about to login           # features/step_definitions/login_steps.rb:1
    Then I am able to create an account # features/step_definitions/login_steps.rb:55

  Scenario: Add site - Invalid login                                    # features/login.feature:17
    Given I am about to login                                           # features/step_definitions
    When I enter invalid credentials                                    # features/step_definitions
    Then I am presented with an error message to correct credentials # features/step_definitions
```

Obrázek 7.18: Calabash ukázka kódu [76]

7.12 Selendroid

Selendroid je nástroj pro automatizaci testů pro nativní, nebo hybridní Android aplikace a mobilní web. Testy využívají API Selenium 2. [77]

7.12.1 Vlastnosti nástroje Selendroid

Při použití frameworku Selendroid není třeba provádět jakékoliv úpravy v kódu aplikace. Framework podporuje gesta (cliknutí, swipe, a tak dále), umí hledat UI elementy podle různých typů lokace. Pro testování lze využívat jak reálná zařízení, tak emulátory, přičemž emulátory se spouštějí automaticky. Testy lze spouštět na více zařízeních zároveň. [77]

Následující ukázku kódu jsem převzala z oficiálních stránek tohoto nástroje. Ukázka má otestovat registraci uživatele.

```
/**
 * Base Test to demonstrate how to test native android apps with Selendroid. App under test is:
 * src/main/resources/selendroid-test-app-0.9.0.apk
 *
 * @author ddary
 */
public class UserRegistrationTest {
  private WebDriver driver = null;

  @Before
  public void setup() throws Exception {
    driver = new SelendroidDriver(new SelendroidCapabilities("io.selendroid.testapp:0.9.0"));
  }
  @Test
  public void assertUserAccountCanRegistered() throws Exception {
    // Initialize test data
    UserDO user = new UserDO("u$erNAme", "me@myserver.com", "mySecret", "John Doe", "Python");

    registerUser(user);
    verifyUser(user);
  }
  private void registerUser(UserDO user) throws Exception {
    driver.get("and-activity://io.selendroid.testapp.RegisterUserActivity");

    WebElement username = driver.findElement(By.id("inputUsername"));
    username.sendKeys(user.username);

    driver.findElement(By.name("email of the customer")).sendKeys(user.email);
    driver.findElement(By.id("inputPassword")).sendKeys(user.password);

    WebElement nameInput = driver.findElement(By.xpath("//EditText[@id='inputName']"));
    Assert.assertEquals(nameInput.getText(), "Mr. Burns");
    nameInput.clear();
    nameInput.sendKeys(user.name);

    driver.findElement(By.tagName("Spinner")).click();
    driver.findElement(By.linkText(user.programmingLanguage)).click();

    driver.findElement(By.className("android.widget.CheckBox")).click();

    driver.findElement(By.linkText("Register User (verify)")).click();
    Assert.assertEquals(driver.getCurrentUrl(), "and-activity://VerifyUserActivity");
  }
  private void verifyUser(UserDO user) throws Exception {
    Assert.assertEquals(driver.findElement(By.id("label_username_data")).getText(), user.username);
    Assert.assertEquals(driver.findElement(By.id("label_email_data")).getText(), user.email);
    Assert.assertEquals(driver.findElement(By.id("label_password_data")).getText(), user.password);
    Assert.assertEquals(driver.findElement(By.id("label_name_data")).getText(), user.name);
    Assert.assertEquals(driver.findElement(By.id("label_preferedProgrammingLanguage_data"))
        .getText(), user.programmingLanguage);
    Assert.assertEquals(driver.findElement(By.id("label_acceptAdds_data")).getText(), "true");
  }
  @After
  public void teardown() {
    driver.quit();
  }
}
```

Obrázek 7.19: Selendroid ukázka kódu [78]

Kapitola 8

Multiplatformní testování aplikace na více zařízeních

8.1 Testmunk

Testmunk byl vyvinut pro pomoc s testováním mobilních aplikací a zavedl jeho automatizaci a podporuje jak nativní, tak i hybridní aplikace. Testmunk pomáhá s vytvořením vhodných testovacích scénářů a rychlým provedením testů na různých mobilních zařízeních s operačními systémy Android a iOS. Testmunk především poskytuje aktuálně nejpoužívanější zařízení, která jsou v originálním nastavení - není u nich root, ani jailbreak.

Testmunk se připojí k aplikaci a sestavuje sespolu s aplikací. Tyto soubory se nahrávají na Jenkins, Travis nebo CircleCI a automaticky se spouští. Pokud dojde k chybě, je o ní podáno hlášení.

Testmunk používá pro tvorbu testovacích scénářů intuitivní, pro lidi snadno čitelný jazyk a poskytuje rozsáhlou knihovnu s více než 50 testovacími kroky, které je možné rozšířit. [79]

Testmunk je možné využít v několika verzích – Basic, Starter, Professional a Expert. Basic je jediná z verzí, která je zdarma. Umožňuje měsíčně testovat 3 hodiny na 1 zařízení firmy testmunk. Basic verze je určená pouze pro 1 uživatele a 1 projekt. Ostatní verze jsou placené a cena se odvíjí od počtu hodin, kdy se využívají testovací zařízení formy Testmunk. [80]

8.2 Testdroid

Testdroid umožňuje spouštění testů na téměř každém Android zařázení, které existuje. Nabízí tak stovky zařízení. Poskytuje sadu nástrojů pro vývoj mobilních softwarů a testovacích produktů. Umí zaznamenávat modelové případy na reálném zařízení a vytvářet tak testovací scénáře. Tyto testovací scénáře lze pak spouštět na všech Testdroid zařízeních v cloudu, a to současně. Díky této

funkci velmi šetří čas vývojářům, kteří by jinak potřebovali množství testerů, kteří by takové testy prováděli.

Při použití Testdroid se mohou hodit znalosti frameworků, jako je Robotium, Appium, UI Automator. Výstup z Testdroid je opatřen detailními zprávami o proběhlém testování a to z každého Android zařízení zvlášť. Během testů vytváří snímky obrazovky, které pak můžete porovnat nástrojem pro grafické porovnávání a odhalit případné rozdíly. [81]

8.3 NeoLoad

NeoLoad poskytuje řešení pro testování zátěže a výkonu. Umí simulovat uživatelské akce a monitoruje chování infrastruktury. Tím přispívá k eliminaci chyb ve webových a mobilních aplikacích.

NeoLoad nabízí prostředí umožňující testovat s aktuálně používanými technologiemi a je plnohodnotným testovacím nástrojem pro webové a mobilní aplikace. Podporuje WebSocket, HTTP/2, GWT, HTML 5, adaptivní datový přenos a další technologie. [82]

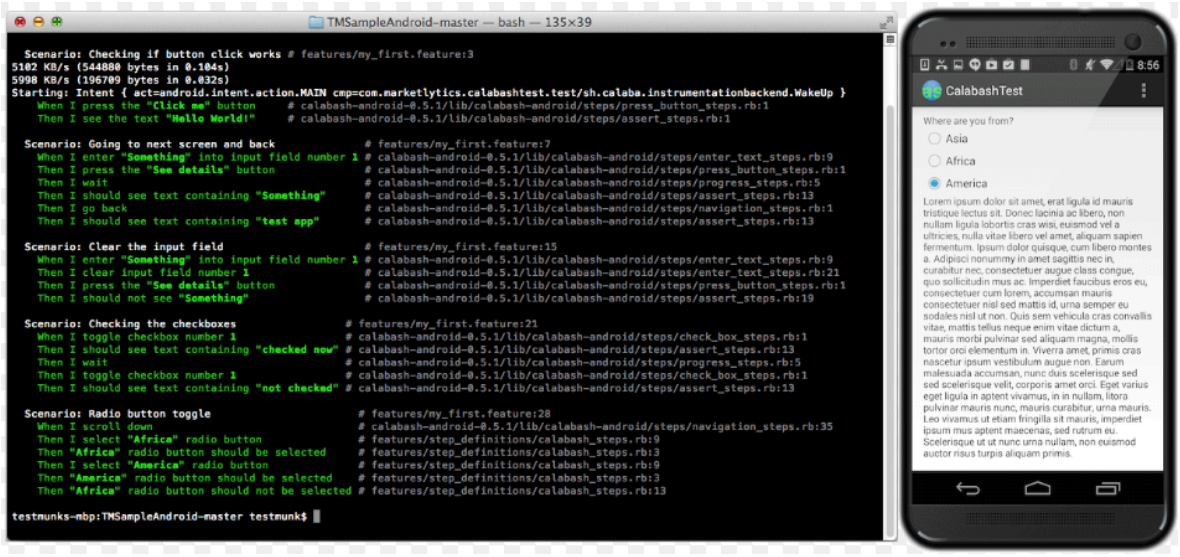
NeoLoad se řadí mezi placené testovací nástroje a jeho cena se odvíjí podle předpokládaného užití, počtu virtuálních uživatelů, používaných protokolů, a dalšího.

8.4 Perfecto Mobile

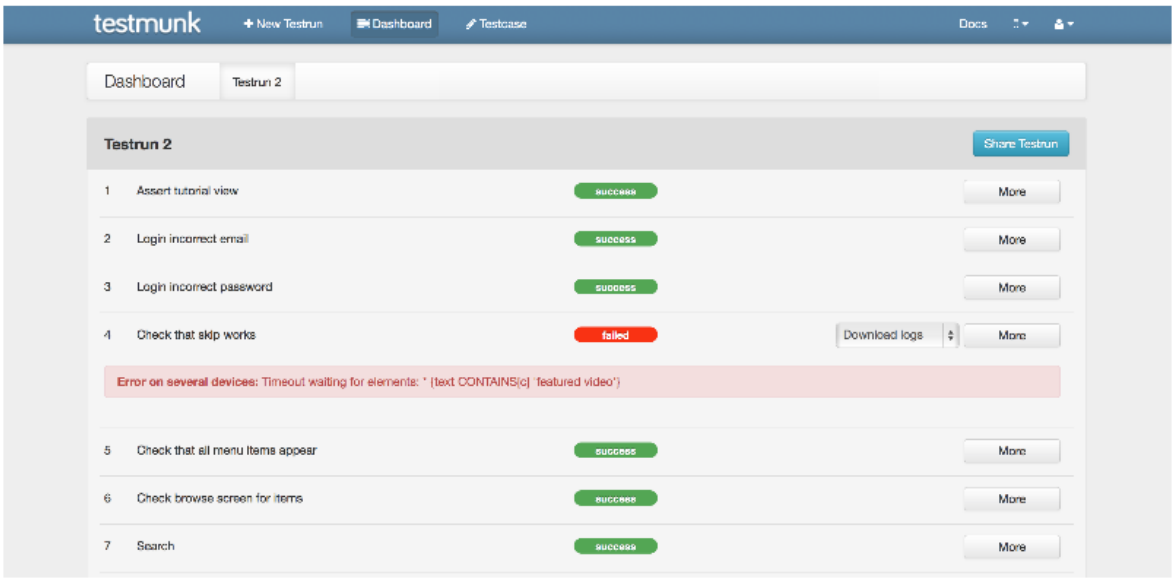
Perfecto Mobile je framework určený pro spouštění testů psaných v Espresso, XCTest, Selenium nebo jiných testovacích frameworcích. Poskytuje různá zařízení a různé platformy pro běh testů. [83]

Perfecto lze stáhnout přímo do Android Studia jako plugin. Pak je možné aplikaci spustit využitím nástroje Espresso a vybrat zařízení, na kterých se má daná aplikace spustit.

Testy lze spouštět na nejnovějších verzích vícero zařízení, kde je možné nastavit různé podmínky sítě a předvolby prostředí. [84]



Obrázek 8.1: Příklad probíhajícího testu - Testmunk [85]



Obrázek 8.2: Ukázka zobrazení výsledků na účtu tesmunk [85]

Kapitola 9

Ekonomický pohled na automatické testování

Využití automatického testování může být ve většině případů pro projekt přínosem. Automatické testy mohou běžet kdykoliv a třeba i na více zařízeních současně. Pro spouštění testů na více zařízeních existují firmy, které poskytují množství zařízení právě pro automatické testování. Tyto služby bývají ale placené. Jejich velikou výhodou je, že umožňují aplikaci řádně otestovat, zda funguje na všech zařízeních, pro které je určena.

9.1 Testovat nebo netestovat a automatizovat nebo neautomatizovat

U malých, jednoduchých projektů (např. zápisník), kde se předpokládá použití jen na konkrétních zařízeních, ze kterých má programátor většinu k dispozici, nebo je může vyzkoušet pomocí emulátoru bych doporučila zvolit testování manuální. Vytvářet automatizované testy by byla práce navíc.

Jakmile projekt obsahuje nějakou logiku, po vhodném testovacím nástroji je určitě vhodné se poohlédnout. Například u aplikace jako je kalkulačka automatické testy mohou během krátkého času vyzkoušet nespočet možností a rozhodně dokáží projít více případů, než běžný tester.

Důležité je zvolení vhodného testovacího nástroje. Strávit nad psaním testovacích scénářů delší čas, než nad vývojem samotné aplikace se opravdu nevyplatí. Pokud testovací scénáře jdou vytvořit snadno a rychle, ušetří to programátorům spoustu práce. Po jakékoliv změně v programu pak stačí testy spustit a ty se samy postarají o to, zda vše fuguje, jak má. Programátorovi pak stačí zkontrolovat, zda testy proběhly v pořádku a případně projít výstupní soubory a snadno tak odhalit chyby, které změna přinesla.

9.2 Finanční přínosy testování

Již v úvodní části své práce, kde popisuji testování softwaru je zmínka o tom, na kolik vyjde taková chyba v závislosti na tom, kdy je objevena. Automatické testování může pomoci chybu objevit včas již během vývoje. Testy ale musí být dobře napsány a pokrývat co možná největší oblast dané aplikace nebo hry.

Vzhledem k tomu, že automatické testy mohou probíhat samy - stačí je pouze spustit, jejich používání vede ke snížení nákladů na oblast kontroly kvality, protože není třeba zaměstnávat několik testerů, ale pouze někoho, kdo testy spustí. Kontrolu výsledků testů pak můžou provádět samy programátoři, kteří mohou rovnou reagovat na nalezené chyby, případně test s nějakými parametry, které pomohou odhalení konkrétního problému spustit znovu.

9.3 Nevýhody a rizika automatizovaného testování

I když se napíší sebelepší testovcí scénáře, nikdy nepokryjí celou aplikaci nebo hru. Vždy je na to tedy potřeba myslet. Návrh testu je tedy vždy potřeba promyslet a pokrýt co možná nejvíce různých variant, které jde v dané aplikaci dělat.

Aplikace se testují snadněji, než hry. Většina aplikací má nějaké menu a pak další stránky, které se zobrazí většinou po kliknutí na položku v menu. Testování je tedy relativně jednoduché. Stačí simulovat klikání a kontrolovat, zda se zobrazilo, co mělo. Pro takové případy si vývojář zcela vystačí s nástroji, které umí zaznamenávat akce.

Testování her, jedná-li se pak o složitější 3D hry je někdy až nemožné. Pokud se ve hře generuje mnoho věcí náhodně, je nutné pro tvorbu testu využít nástroj, který "vidí"přímo do kódu testu. V mnoha případech ani to nestačí a je potřeba si některé akce naprogramovat sám. Pak už je jen na uvážení vývojáře, zda zvolí programování svého testovacího frameworku, nebo bude testovat manuálně.

Hlavním rizikem automatizovaného testování je tedy volba nevhodného nástroje, případně programování vlastního prostředí, jehož doba vývoje testů přesáhne únosnou hranici. Únosná hranice záleží na velikosti projektu. Například vývoj testovacího prostředí trvající třičtvrtě roku u projektu, jehož vývoj trvá rok smysl nemá. Nejspíš se v průběhu bude stejně testovat alespoň manuálně a navíc se bude muset zaměstnat tým, který se bude starat o vývoj testovacího prostředí pro psaní automatických testů.

Závěr

Bez řádného testování by nebyl cyklus vývoje softwaru úplný a zůstavalo by v něm mnoho neodhalených chyb, které by se nejspíš dostaly až ke koncovému zákazníkovy. Včasné odhalení chyby během testování mobilních aplikací znamená veliký přínos pro vývoj samotný a ušetření nákladů na opravu dané chyby. S tím souvisí zvolení správných metod a nástrojů pro testování.

Testovat aplikaci může tester, skupina testerů, nebo může probíhat automatizovaně. U většiny projektů je provedení automatizace testování fnančně tou nejméně zatěžující možností. S testováním je vhodné začít hned v počáteční fázi vývoje mobilní aplikace nebo hry. Testy se mohou spouštět při sememenší změně a to na různých zařízeních a několikrát po sobě a zajistit tak, že se tak chování aplikace nezměnilo a vrací vždy očekávané výsledky. A přitom v průběhu testování není třeba přítomnosti nikoho, stačí jen vývojář, který testy spustí a následně zkontroluje výsledky.

Pro automatické testování mobilních aplikací existují různé nástroje podle prostředí a jazyku, ve kterém byly napsané, nebo jsou na trhu i multiplatformní, které zajišťují především testování uživatelského rozhraní. Jestliže pro dané prostředí existuje nástroj přímo určený právě pro dané prostředí, je vhodné ho použít, jelikož může zajistit tesování i složitějších operací a logiky. V případě aplikací je využití multiplatformních nástrojů, testujících ve stylu černé skříňky vhodnější než u her. Hry většinou obsahují složitější logiku, kterou takový nástroj není shopen popsat a simulovat tak hraní hry.

Tvorba vlastního testovacího frameworku může být řešením v případě, že nástroje pro daný projekt neexistují, nebo nejsou vhodné. Osobně bych tvorbu vlastních nástrojů zvolila jen, pokud jeho tvorba zabere jen malý časový úsek v poměru s vývojem aplikace nebo hry. Jakmile bude vývoj takového prostředí trvat příliš dlouho a vhodný nástroj neexistuje, vyplatí se pro daný úkon zaměstnat testery - především u menších projektů.

Vždy je dobré předem odhadnout, jak dlouho by se testovací prostředí vyvíjelo a jak dlouho se bude používat. Pak si spočítat, kolik zaměstnaných testerů by bylo pro daný projekt pro úplné testování potřeba a tato čísla si po-

rovnat. Pokud náklady za zaměstnané testery výrazně přesahují náklady na vývoj frameworku pro automatizované testování, vývoj frameworku samotného se určitě vyplatí - a to i s případným zaměstnáním testerů v průběhu vývoje testovacího frameworku. Po vytvoření testovacích scénářů bude třeba minimum času pro jejich správu a kontrolu výsledků testů.

Práce pro mě byla velkým přínosem, jelikož jsem si v této oblasti udělala pěkný přehled. Do začátku psaní své práce jsem nevěděla, že existuje tolik možností pro vývoj mobilní aplikace, nebo hry. Nevěděla jsem ani, že existují nástroje, které jsou založené na principu rozeznávání snímků obrazovky a jejich následné porovnávání, což mě opravdu zaujalo. Věřím, že má analýza nástrojů pro automatické testování mobilních aplikací a her bude přínosem pro vývojáře, kteří řeší otázku týkající se testování mobilních aplikací určených pro zařízení s operačním systémem Android.

Literatura

[1] Mobilizujeme.cz: *Android [online]*. ©2017, [cit. 2017-02-26]. Dostupné z: https://mobilizujeme.cz/stitky/android

[2] Techopedia: *Mobile Application (Mobile App) [online]*. ©2017, [cit. 2017-03-01], (překlad vlastní). Dostupné z: http://www.techopedia.com/definition/2953/mobile-application-mobile-app

[3] Pixelfield: *Co musíte vědět o testování mobilních aplikací [online]*. ©2017, [cit. 2017-03-01]. Dostupné z: https://pixelfield.cz/blog/co-musite-vedet-o-testovani-mobilnich-aplikaci/

[4] Sallyx.org: *Vý vojová prostředí a překladače [online]*. ©2017, [cit. 2017-03-01]. Dostupné z: http://www.sallyx.org/sally/c/vyvojova-prostredi.php

[5] Olejník, J.: Emulátory: Když hry z Google Play nestačí, díl 1[online]. *Svět Androida*, [vyd. 2013-01-28], [cit. 2017-03-01]. Dostupné z: https://www.svetandroida.cz/emulatory-kdyz-hry-z-google-play-nestaci-dil-1-201301

[6] IT slovník.cz [online]. [cit. 2017-04-14]. Dostupné z: https://it-slovnik.cz/

[7] Mlejnek, J.: Testování aplikací [online]. *edux.fit.cvut.cz*, [cit. 2017-03-16]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-SI1/_media/lectures/09/09.prednaskacb.pdf

[8] Patton, R.: *Testování softwaru*. Praha: Computer Press, 2002, ISBN ISBN 80-7226-636-5.

[9] ROUDENSKÝ, P. a. H. A.: *Řízení kvality softwaru: průvodce testováním*. Praha: Computer Press, 2002, ISBN ISBN 80-7226-636-5.

[10] Hlava, T.: Fáze a úrovně provádění testů [online]. *Testování softwaru*, 2011, [cit. 2017-03-16]. Dostupné z: http://testovanisoftwaru.cz/category/druhy-typy-a-kategorie-testu

[11] Testování softwaru: *Manuální testování [online]*. [cit. 2017-03-10]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani

[12] Testování softwaru: *Automatizované testování [online]*. [cit. 2017-03-10]. Dostupné z: http://testovanisoftwaru.cz/automatizovane-testovani/

[13] Android Tip: *(NÁVOD) JAK VYTVOŘIT HRU NA ANDROID? KOMPLETNÍ PRŮVODCE TVORBOU ANDROID HER [online]*. 2015, [cit. 2017-03-08]. Dostupné z: http://www.androidtip.cz/navod-jak-vytvorit-hru-na-android-kompletni-pruvodce-tvorbou-android-her/

[14] Android Studio [online]. [cit. 2017-03-18]. Dostupné z: http://www.androidtip.cz/navod-jak-vytvorit-hru-na-android-kompletni-pruvodce-tvorbou-android-her/

[15] Android Authority: *I want to develop Android Apps – What languages should I learn? [online]*. [cit. 2017-03-21], (překlad vlastní). Dostupné z: http://www.androidauthority.com/want-develop-android-apps-languages-learn-391008/

[16] Gideros Mobile [online]. [cit. 2017-03-22], (překlad vlastní). Dostupné z: http://giderosmobile.com/

[17] Titanium [online]. [cit. 2017-03-22], (překlad vlastní). Dostupné z: http://www.appcelerator.com/titanium/7/

[18] Makawiel: GameSalad Creator [online]. *Tvorba her*, [vyd. 2014-09-27], [cit. 2017-03-22]. Dostupné z: http://tvorbaher.cz/gamesalad-creator/

[19] Visual Studio zdarma a pro tvorbu Android aplikací [online]. [cit. 2017-03-22]. Dostupné z: http://www.zive.cz/clanky/visual-studio-zdarma-a-pro-tvorbu-android-aplikaci/sc-3-a-179454/default.aspx

[20] Android Developers: *Automating User Interface Tests [online]*. [cit. 2017-04-08], (překlad vlastní). Dostupné z: https://developer.android.com/training/testing/ui-testing/index.html

[21] Android Developers: *Testing Support Library [online]*. [cit. 2017-04-08], (překlad vlastní). Dostupné z: https://developer.android.com/topic/libraries/testing-support-library/index.html

[22] Unit Testing with Android Studio - JUnit 4 [online]. *YouTube*, [vyd. 2016-10-19], [cit. 2017-05-08], (překlad vlastní). Dostupné z: https://www.youtube.com/watch?v=dlWjn4QrVK4

[23] UI testing with Espresso - Android Testing Patterns #2 [online]. *YouTube*, [vyd. 2016-05-25], [cit. 2017-05-08], (překlad vlastní). Dostupné z: https://www.youtube.com/watch?v=kL3MCQV2M2s

[24] Android Developers: *Testing UI for a Single App [online]*. [cit. 2017-04-08], (překlad vlastní). Dostupné z: https://developer.android.com/training/testing/ui-testing/espresso-testing.html

[25]

[26] Automatic Android* Testing with UiAutomator [online]. *Developer Zone*, [vyd. 2014-01-23], [cit. 2017-05-08], (překlad vlastní). Dostupné z: https://software.intel.com/en-us/articles/automatic-android-testing-with-uiautomator

[27] Android Developers: *monkeyrunner [online]*. [cit. 2017-04-14], (překlad vlastní). Dostupné z: https://developer.android.com/studio/test/monkeyrunner/index.html

[28] Michael Roth: *lunit - Unit Testing Framework for Lua [online]*. 2009, [cit. 2017-04-15], (překlad vlastní). Dostupné z: https://www.mroth.net/lunit/

[29] Kistner, G.: Lunity [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: https://github.com/Phrogz/Lunity/blob/master/README.md

[30] Kistner, G.: lunatest [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: https://github.com/silentbicycle/lunatest/blob/master/README.md

[31] Warden, J.: Unit Testing in Corona SDK Using Lunatest [online]. *Software, fitness, and gaming - Jesse Warden*, 2012, [cit. 2017-04-15], (překlad vlastní). Dostupné z: http://jessewarden.com/2012/07/unit-testing-in-corona-sdk-using-lunatest.html

[32] Developing Z-Wave plugins for Vera with Lua and Luup [online]. *dmlogic*, [vyd. 2014-04-15], [cit. 2017-04-15], (překlad vlastní). Dostupné z: https://github.com/bluebird75/luaunit/blob/master/README.md

[33] Kepler Project: *Shake [online]*. 2007, [cit. 2017-04-15], (překlad vlastní). Dostupné z: http://shake.luaforge.net/

[34] Kepler Project: *Shake [online]*. [vyd. 2007-12-21], [cit. 2017-04-15], (překlad vlastní). Dostupné z: `http://shake.luaforge.net/examples.html`

[35] Clarke, N.: telescope [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://github.com/norman/telescope/blob/master/README.md`

[36] Perrad, F.: *Lua-TestMore [online]*. ©2009-2016, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `http://fperrad.github.io/lua-TestMore/#lua-testmore`

[37] Perrad, F.: *Test.More [online]*. ©2009-2016, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://fperrad.github.io/lua-TestMore/testmore/`

[38] San Francisco: Olivine Labs: *Busted [online]*. ©2013, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `http://olivinelabs.com/busted/#overview`

[39] Zaitsev, S.: Gambiarra [online]. *Bitbucket*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://bitbucket.org/zserge/gambiarra`

[40] Irven, M.: luaspec [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://github.com/mirven/luaspec/blob/master/README`

[41] Lua-users-wiki: *Unit Testing [online]*. [cit. 2017-04-15], (překlad vlastní). Dostupné z: `http://lua-users.org/wiki/UnitTesting`

[42] Donovan, S. J.: sipscan.lua [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://github.com/stevedonovan/Penlight/blob/master/examples/sipscan.lua`

[43] TestApi.lua [online]. [cit. 2017-04-15], (překlad vlastní). Dostupné z: `http://users.skynet.be/adrias/Lua/ut/testapi.lua.html`

[44] Janda, P.: Testy [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://github.com/siffiejoe/lua-testy/blob/master/README.md`

[45] Winkle, L. V.: Minctest [online]. *GitHub*, ©2017, [cit. 2017-04-15], (překlad vlastní). Dostupné z: `https://github.com/codeplea/minctest-lua/blob/master/README.md`

[46] Xamarin Recorder [online]. *Xamarin Inc.*, ©2017, [cit. 2017-04-28], (překlad vlastní). Dostupné z: `https://www.slideshare.net/ContusQA/monkey-talk-automation`

[47] Automation System Overview [online]. *Unreal Engine*, ©2004-2017, [cit. 2017-04-23], (překlad vlastní). Dostupné z: https://docs.unrealengine.com/latest/INT/Programming/Automation/index.html

[48] Test tools [online]. *Unity*, ©2017, [cit. 2017-04-28], (překlad vlastní). Dostupné z: https://unity3d.com/unity/qa/test-tools

[49] How to use the Integration Test Framework [online]. *Bitbucket*, ©2017, [cit. 2017-05-04], (překlad vlastní). Dostupné z: https://bitbucket.org/Unity-Technologies/unitytesttools/wiki/IntegrationTestsRunner

[50] Sikuli Script [online]. 2014, [cit. 2017-04-16], (překlad vlastní). Dostupné z: http://www.sikuli.org/

[51] Practical Sikuli: using screenshots for GUI automation and testing [online]. *SlideShare*, ©2017, [cit. 2017-04-16], (překlad vlastní). Dostupné z: https://www.slideshare.net/vgod/practical-sikuli-using-screenshots-for-gui-automation-and-testing

[52] Sikuli for Mobile Testing [online]. *YouTube*, [vyd. 2012-11-14], [cit. 2017-04-29], (překlad vlastní). Dostupné z: https://www.youtube.com/watch?v=01jFl8KrEMY

[53] TestPlant, L.: EggPlant Mobile [online]. ©2017, [cit. 2017-04-16], (překlad vlastní). Dostupné z: https://www.testplant.com/eggplant/testing-tools/eggplant-mobile-eggon/

[54] TestPlant, L.: Getting Started with eggPlant Functional [online]. ©2017, [cit. 2017-05-05], (překlad vlastní). Dostupné z: http://docs.testplant.com/?q=content/getting-started-1

[55] Foundation, J.: Introducing Appium [online]. ©2017, [cit. 2017-04-18], (překlad vlastní). Dostupné z: http://appium.io/

[56] Getting started with appium [online]. [cit. 2017-05-08], (překlad vlastní). Dostupné z: http://appium.io/slate/en/tutorial/android.html?ruby#getting-started-with-appium

[57] Krishna, S.: AndroidTest.java [online]. *GitHub*, ©2017, [cit. 2017-04-18], (překlad vlastní). Dostupné z: https://github.com/appium/sample-code/blob/master/sample-code/examples/java/junit/src/test/java/com/saucelabs/appium/AndroidTest.java

[58] Bhasin, J.: Cross platform test automation using Appium [online]. *SlideShare*, ©2017, [cit. 2017-04-18], (překlad vlastní). Dostupné z: https://www.slideshare.net/thinkexist/cross-platform-test-automation-using-appium

[59] Reda, R.: Robotium [online]. *GitHub*, ©2017, [cit. 2017-04-18], (překlad vlastní). Dostupné z: `https://github.com/RobotiumTech/robotium`

[60] Robotium Recorder User Guide [online]. *YouTube*, [vyd. 2014-02-19], [cit. 2017-04-29], (překlad vlastní). Dostupné z: `https://www.youtube.com/watch?v=bmZJ0UpYrL0`

[61] UI Testing with Robotium [online]. *Codepath*, [cit. 2017-04-18], (překlad vlastní). Dostupné z: `http://guides.codepath.com/android/ui-testing-with-robotium`

[62] Mobile Test Automation [online]. *T-Plan*, © 1989, [cit. 2017-04-20], (překlad vlastní). Dostupné z: `http://www.t-plan.com/mobile-test-automation/`

[63] T-Plan Robot - v4.1 Screen Recorder Preview for UI Test Automation [online]. *YouTube*, [vyd. 2014-12-19], [cit. 2017-04-29], (překlad vlastní). Dostupné z: `https://www.youtube.com/watch?v=8yhbBy9kyls`

[64] T-Plan Robot - Mobile Sample [online]. *YouTube*, [vyd. 2013-03-13], [cit. 2017-04-18], (překlad vlastní). Dostupné z: `https://www.youtube.com/watch?v=6itcB1Xk8vg`

[65] Corporation, B.: TestQuest 10 [online]. Dostupné z: `http://www.bsquare.com/professional-services/automated-testing/testquest-10/`

[66] TestQuest 10 Overview Demo Video [online]. *YouTube*.

[67] 05 Creating a Text Search flv59temp44 [online]. *YouTube*.

[68] Silk Mobile [online]. *Micro Focus*, ©2017, [cit. 2017-04-21], (překlad vlastní). Dostupné z: `https://www.microfocus.com/media/data-sheet/silk_mobile_testing_ds.pdf`

[69] Supported Technologies [online]. [cit. 2017-04-22], (překlad vlastní). Dostupné z: `http://www.ranorex.com/supported-technologies.html`

[70] How Test Automation Works [online]. [cit. 2017-04-22], (překlad vlastní). Dostupné z: `http://www.ranorex.com/how-test-automation-works.html`

[71] Cucumber Tutorial [online]. Dostupné z: `https://www.tutorialspoint.com/cucumber/`

[72] Cucumber School [online]. [cit. 2017-04-22], (překlad vlastní). Dostupné z: `https://cucumber.io/school/`

[73] MonkeyTalk [online]. [cit. 2017-04-23], (překlad vlastní). Dostupné z: http://www.testertools.com/blog/gorilla-logic-releases-monkeytalk-for-cross-platform-ios-android-functional-testing/

[74] Schneider, P.: 56. MonkeyTalk – Features & Benefits [online]. *Automation Testing - Blog*, 2013, [cit. 2017-04-23], (překlad vlastní). Dostupné z: https://automationprofessional.wordpress.com/2013/06/02/108/

[75] ContusQA: MonkeyTalk Automation Testing For Android Application [online]. *SlideShare*, ©2017, [cit. 2017-04-23], (překlad vlastní). Dostupné z: https://www.slideshare.net/ContusQA/monkey-talk-automation

[76] Calabash [online]. [cit. 2017-04-24], (překlad vlastní). Dostupné z: http://calaba.sh/

[77] Selendroid [online]. © 2012 - 2015, [cit. 2017-05-08], (překlad vlastní). Dostupné z: http://selendroid.io/

[78] Selendroid [online]. © 2012 - 2015, [cit. 2017-05-08], (překlad vlastní). Dostupné z: http://selendroid.io/quickStart.html

[79] Testmunk [online]. *testmunk*, ©2016, [cit. 2017-04-17], (překlad vlastní). Dostupné z: https://www.testmunk.com

[80] Pricing [online]. *testmunk*, ©2016, [cit. 2017-04-17], (překlad vlastní). Dostupné z: https://www.testmunk.com

[81] Schneider, P.: Testdroid aneb testování droidů! [online]. *Ackee*, [vyd. 2014-05-29], [cit. 2017-04-23]. Dostupné z: https://www.ackee.cz/blog/testing-droids-on-testdroid/

[82] NeoLoad [online]. [cit. 2017-04-23], (překlad vlastní). Dostupné z: https://www.neotys.com/neoload/overview

[83] Ltd, P. M.: 4 Ways Your Development World Can Be Better (and Faster) [online]. Dostupné z: https://www.perfectomobile.com/

[84] Ltd, P. M.: Launch on Multiple Cloud Devices [online]. Dostupné z: https://www.perfectomobile.com/solutions/devtunnel/launch-on-multiple-cloud-devices

[85] Poschenrieder, M.: A Beginner's Guide to Automated Mobile App Testing [online]. *testmunk*, [vyd. 2014-08-29], [cit. 2017-04-17], (překlad vlastní). Dostupné z: https://blog.testmunk.com/tutorial-for-automated-mobile-app-testing-calabash/

Příloha A

Seznam použitých zkratek

OS Operating System

API Application Programming Interface

HTML HyperText Markup Language

XML eXtensible Markup Language

CSS Cascading Style Sheets

3D 3 Dimensions

2D 2 Dimensions

GPS Global Positioning System

IDE Integrated Development Environment

BDD Behavior-Driven Development

TDD Test-Driven Development

SIT System Integration Tests

RPG Role-Playing Game

GUI Graphical User Interface

SDK Software Development Kit

PHP Hypertext Preprocessor

SDK Software Development Kit

MVC Model-View-Controller

CI Continous Integration

HW HardWare

UI User Interface

VNC Virtual Network Computing

TAP Test Anything Protocol

MIT Massachusetts Institute of Technology

ID IDentification

VNC Virtual Network Computing

RDP Remote Desktop Protocol

WFP World Food Programme

SAP Service Access Point

VB.NET Visual Basic .NET

HTTP Hypertext Transfer Protocol

GWT Google Web Toolkit

Příloha **B**

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
thesis ..... dokumenty k závěrečné práci
    thesis.tex ..... zdrojová forma práce ve formátu LaTeX
    thesis.pdf ..... text práce ve formátu PDF
    assignment.pdf ..... zadání práce ve formátu PDF
```