Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Automatické testování mobilních aplikací

Bakalářská práce

Autor: Tomáš Novák

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Odborný konzultant: Ing. David Petrík,

Systemart s. r. o.

Hradec Králové duben 2019

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 15.4.2019 Tomáš Novák

Poděkování:

Děkuji vedoucímu bakalářské práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení práce.

Dále bych rád poděkoval Ing. Davidu Petríkovi a celé společnosti Systemart s.r.o. za předané zkušenosti a možnost se na tomto projektu podílet.

Poslední poděkování patří mým rodičům a přítelkyni, za motivaci a podporu při psaní této bakalářské práce i během celého bakalářského studia.

Anotace:

Bakalářská práce se zabývá tematikou automatického testování mobilních aplikací. Teoretická část objasňuje jeho význam a seznamuje se základními pojmy s testováním spojené. Věnuje se také popisu nástrojů, vývojových modelů a automatických frameworků, přičemž se nejvíce zaměřuje na framework Appium, který byl použit pro projekt popsaný v praktické části. Praktická část představuje konkrétní řešení automatizace testů pro mobilní aplikace na projektu „Výkaz práce“. Kromě popisů jejích jednotlivých komponent jsou zde také ukázky kódu.

Annotation:

Title: Automatic testing of mobile applications

My bachelor thesis deals with the topic of automatic mobile aplication testing. The theoretical part clarifies its meaning and introduces the basic concepts related to the testing itself. It also describes tools, development models and automatic frameworks. My work is mostly focused on the framework Appium which was used for the project described in the practical part. Practical part intorduces a specific solution of automatic mobile appliation testing on the „Logeto“ project. Apart from describing individual components, there are also some code samples.

Obsah

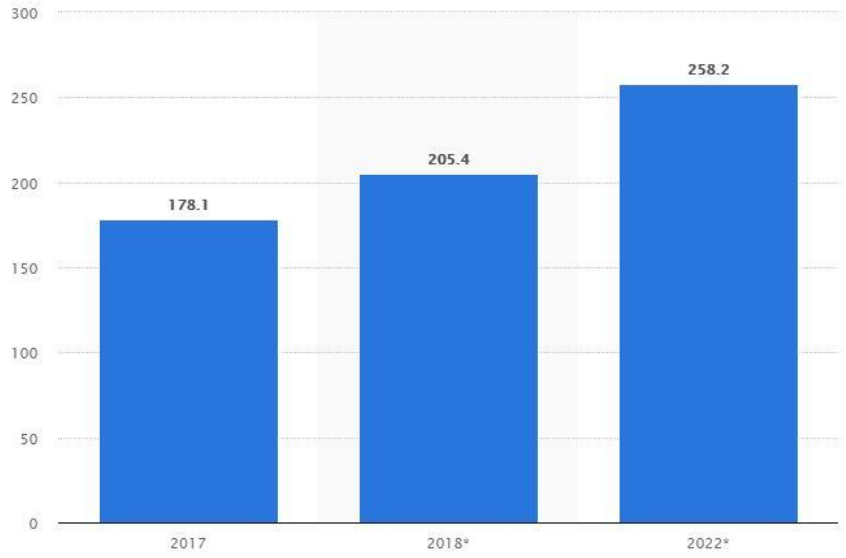
Seznam obrázků

Seznam grafů

Seznam tabulek

1 Úvod

Během posledních let se chytré mobilní telefony a jejich aplikace staly nedílnou součástí života většiny populace. V průběhu roku 2017 došlo napříč dvěma hlavními platformami pro distribuci digitálního obsahu na mobilních telefonech (Google Play Store a Apple App Store) k 178 miliardám jedinečných stažení aplikací. [1] Dle předpokladu se toto číslo do roku 2022 zvýší až na 258 miliard unikátních stažení viz Obr. 1.



Obr. 1 Předpověď unikátních stažení do roku 2022 [1]

Vzhledem k tomuto tempu začíná být nemyslitelné, aby veškeré zveřejněné aplikace odpovídaly kvalitě, kterou uživatel očekává. Již nyní se lze často setkat se situací, kdy je mobilní aplikace odinstalována ihned po několika použitích, v některých případech dokonce i pár sekund po jejím prvním spuštění. Velké množství aplikací, které potká stejný osud, jsou odinstalovány z důvodu nepřehledného GUI (grafické uživatelské rozhraní), pomalé odezvy, nestability a jiných technických problémů. Jak Joshua Pramis uvádí ve svém článku:[1] *„studie rovněž ukázala, že 80-90 % všech stažených aplikací* e použita *pouze jednou a poté jsou uživateli smazány“, (přeloženo z* [2]). Počet nespokojených uživatelů je tedy velmi vysoký a dle předpokladů lze říci, že bude i nadále stoupat.

[1] „the study also showed that anywhere from 80 to 90 percent of all downloaded apps are used once and then eventually deleted by users“ [2]

Nabízí se tak otázka jak tento trend zvrátit. První, nejjednodušší variantou, je zavedení přísnějších kritérií pro publikování aplikací na jednotlivých distribučních platformách po vzoru společnosti Apple [3]. Vývojáři jsou nuceni dodržovat určité standardy na chování, vzhled a funkčnost aplikací, čímž je zajištěna jejich určitá minimální úroveň. Další možností – z personálního hlediska náročnější, je zvýšení kvality vývoje zapojením testování do jeho celého procesu. Díky zapojení testovaní do vývoje, je možné odhalit téměř veškeré závažné chyby, většinu drobných nedostatků a také zvýšit kvalitu ergonomie celé aplikace.

1.1 Vymezení práce

1.1.1 Důvod výběru tématu

Autorem této práce je student oboru Aplikovaná informatika na Univerzitě Hradec Králové. Díky zkušenostem nabytých během studia získal pozici v lokální softwarové firmě Systemart s.r.o., která se zabývá vývojem vlastního docházkového systému Výkaz práce. Autor se zde zabývá testováním aplikace pro všechny platformy a vývojem automatických testů pro mobilní operační systém Android a Univerzální platformu Windows. Bakalářská práce tak reflektuje autorovy vlastní zkušenosti nabyté v praxi. Automatizované testování může při vývoji ušetřit nemalé náklady a do určité míry vyřešit problémy s rychlým odchodem uživatelů z aplikace, tento text je tak určen jako zajímavý informační zdroj pro ty, kteří se chtějí dozvědět víc o tematice automatického testování, případně ho využít na reálných projektech.

1.1.2 Cíl práce

Cílem této práce je představit doménu testování softwaru, popsat a porovnat některé vlastnosti nástrojů pro automatizované testování mobilních aplikací a následně nastínit postup řešení na konkrétním projektu pomocí frameworku Appium s ukázkou odlišného přístupu použití.

1.1.3 Struktura práce

Tato práce je rozdělena do tří částí. První z nich se věnuje popisu testování jako obecné součásti vývoje softwaru. Pojednává o důvodech využívání testování a popisuje metodiky, které je možné během celého procesu využívat. Druhá část se zaměřuje na popis automatizovaného testování a jeho pozitivních vlastností. Dále práce zmiňuje několik nástrojů určených pro automatizované testování a vzájemně porovnává jejich vlastnosti na základě autorových zkušeností získaných při práci s nimi a informací které lze získat z dokumentací těchto nástrojů a článků na webu.

Poslední část práce se zabývá popisem vlastností frameworku Appium a praktickými ukázkami jeho použití při testování mobilních aplikací. K tomuto účelu jsou zde uvedeny části zdrojového kódu z projektu automatizovaného testování docházkové aplikace Výkaz práce. Uvedené zdrojové kódy byly částečně modifikovány tak, aby žádná z částí neporušovala NDA (dohoda o sdílení důvěrných materiálů [4]). Zároveň však zůstala zachována jejich struktura tak, aby jejich vypovídající a naučná hodnota zůstala co možná největší.

2 Literární rešerše

Doméně testování softwaru již bylo věnováno několik odborných publikací, výukových textů i akademických prací. Tato kapitola obsahuje soupis několika hlavních zdrojů, ze kterých bylo čerpáno při tvorbě této práce. K vyhledání zdrojů bylo využito katalogu Google Scholar, katalogu Studijní a vědecké knihovny v Hradci Králové a jiných volně dostupných zdrojů, jako jsou blogy zabývající se tematikou testování softwaru.

2.1 Knižní zdroje

Vědomostní doména a dovednost testování softwaru je mezi zaměstnavateli stále diskutovanější a žádanější problematikou. Není tedy nic zvláštního na tom, že se této problematice věnovalo již několik desítek odborných publikací, které nepokrývají jen samotné testování, ale také byznys strategie pro zavedení testování do již existujících projektů, popis přístupů k testování nejen desktopových a mobilních aplikací, ale i aplikací pro nositelnou elektroniku a chytrou domácnost.

2.1.1 Seznam hlavních knižních zdrojů

- **Testování softwaru, RON PATTON (2000) [7]**

  Jedná se o jednu z nejobsáhlejších publikací zabývajících se testováním, která byla přeložena do češtiny. Publikace pokrývá doménu testování od úplné historie až po budoucí trendy (ty už z hlediska stáří publikace nejsou příliš aktuální. pozn. autora), publikace se také zabývá dokumentováním testování, vytvářením testovacích případů a kariérními otázkami softwarového testera.

- **Řízení kvality softwaru, P. ROUDENSKÝ, A. HAVLÍČKOVÁ (2013) [8]**

  Tato publikace pojednává o celkové problematice kontroly kvality. Její obsah popisuje technické normy, metodiky vývoje i různé typy testů. Zaměřuje se také na management testování, motivování testerů a metodiky pro určování metrik pro měření kvality.

- **Hands-On Mobile App Testing, DANIEL KNOTT (2015) [9]**

  Sám autor publikaci označuje, jako kompletní příručku pro kohokoliv, kdo se chce dozvědět veškerá tajemství testování mobilních aplikací. Tato publikace je věnována také popisu a porovnání rozdílů mobilních platforem, typů aplikací a funkcí senzorů, které lze u mobilních zařízení nalézt a co vše je nutné znát pro jejich důkladné testování.

2.2 Akademické práce

Tematikou testování mobilních aplikací se již také zabývalo několik akademických prací. Inspirací pro vypracování této bakalářské práce byly ty obsahově nejzajímavější a tématu nejbližší z nich.

2.2.1 Seznam použitých akademických prací

- **Automatizované testování softwaru, TOMÁŠ BURDA (2017) [10]**

  Jedná se o akademickou práci firemního kolegy a nadřízeného autora tohoto textu. Tomáš Burda zde srozumitelnou formou pojednává o způsobech testování. V praktické části se věnuje automatizaci testování webového prostředí aplikace Výkaz práce pomocí frameworku Selenium.

- **Testování mobilních aplikací, VIKTOR MAČURA (2017) [11]**

  Tato akademická práce se zabývá testováním mobilních aplikací z pohledu monitorování a evidence chyb, která umožnuje projektům plánovat budoucí strategie a přístupy testování a vývoje. Ve své praktické části se autor zabývá využitím těchto nástrojů pro mobilní aplikaci společnosti Alza.cz a.s.

- **Přehled nástrojů pro automatické testování aplikací, EDUARD VESELOVSKÝ (2014) [12]**

  Autor se v práci zaměřuje opět na způsoby automatizace testování, avšak s tím rozdílem, že pojednává o automatizaci pomocí placených služeb fungujících pomocí způsobu nahrávání jednotlivých úkonů testovacích případů. V praktické části se věnuje porovnání cenové i technologické výhodnosti jednotlivých řešení.

3 Testování softwaru

Přesto, že lze intuitivně vytušit, co se pod pojmem *Testování softwaru* skrývá, není úplně snadné najít jeho správnou a ucelenou definici. Téměř každá publikace a autor si definici testování upraví dle svých představ a zkušeností. Stejně tak je tenká hranice mezi tím, co ještě do testování patří a co nikoliv. Například Petr Roudenský a Anna Havlíčková ve své knize Řízení kvality softwaru ([8], s. 45) definují testování jako „*spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů*“.

Ron Patton se ve své knize Software testing [7] vyhýbá přesné definici testování. Svou pozornost soustředí zejména na popis a definici chyby, stejně jako na důvody, proč je nutné si vyjasnit co to vlastně chyba je (více v kapitole *3.2*.). Přestože zde testování není přesně definováno, je z kontextu knihy patrné, že by se jeho interpretace lišila od definice P. Roudenského a A. Havlíčkové. Na rozdíl od spoluautorů přihlíží také k možnosti testování dokumentací a požadavkům uživatele, což jsou principy, na které Roudenský a Havlíčková vůbec nebrali zřetel, viz ([8], s. 45) „*spouštění softwarového produktu“*.

Další definice dle ISTQB (International Software Testing Qualifications Board) zní:

„Testování softwaru je proces spouštění programu nebo aplikace s úmyslem nalezení softwarové chyby.

- *Může být také definováno jako proces validace a verifikace, že program, aplikace nebo produkt:*
    - *splňuje obchodní a technické požadavky, které stanovily jeho design a vývoj,*
    - *pracuje podle očekávání,*
    - *je naimplementován podle dané specifikace.“* (přeloženo z [13])

3.1 Význam testování

I přesto, že se lze často setkat s neporozuměním potřeby testovat, ze strany zadavatele či investora projektů, je testování důležitou součástí vývoje softwaru. Testování je zmiňováno ve většině publikací zabývajících se návrhem a tvorbou softwaru. Často se zde o testování mluví jako o jedné z nejdůležitějších součástí vývoje, během tohoto procesu je odhaleno až 50 % všech chyb viz Obr. 2, což výrazně šetří další náklady za opravy u finálního projektu. Společnost Capgemini před několika lety provedla rozsáhlý průzkum na několika stovkách projektů z celé Evropy [14], který měl přesně určit míru nákladů na opravení chyby v určité fázi vývoje a také míru počtu detekovaných chyb v dané fázi. Z tohoto výzkumu vyplynulo, že náklady na odstranění chyby až v produkční fázi je pro projekt 4krát nákladnější než její odhalení a odstranění již během fáze testování.

	Analýza	Design	Vývoj	Vývojářské testy	Testování	Produkce
Relativní počet zanesených chyb v dané fázi	10%	40%	50%			
Relativní počet detekovaných chyb v dané fázi	3%	5%	7%	25%	50%	10%
Normalizované náklady na opravu chyb (EUR)	250	250	250	1 000	3 000	12 500

Obr. 2 Náklady na opravu chyby v závislosti na fázi vývoje ([15], s. 16)

Z jejich dat také vyplývá, že projekty, které měly do fáze testování zahrnutou i revizi analytických dokumentů mohly během testování zlevnit opravu části nalezených chyb až 50krát ([15], s. 17) viz Obr. 2.

3.2 Chyba

Chyba je označení pro chování produktu nebo aplikace, které neodpovídá požadovanému záměru a znepříjemňuje, či dokonce úplně znemožňuje používání produktu. Stejně jako u pojmu testování je obtížné sestavit přesnou definici chyby. Důvodem toho je množství názvů, kterými lze chybu označit např.: vada, závada, selhání, problém, omyl, odchylka, chyba, anomálie ([7], s. 13). Tyto názvy nemají stejný význam a může tak docházet k jejich různým interpretacím a tím k neporozumění.

Pojmy jako je vada, závada a selhání se nejběžněji používají pro druh chyby, který je opravdu závažný, v některých případech i životu nebezpečný. Jiné pojmy jako je anomálie nebo odchylka značí chybu, která vzniká buďto nedopatřením, zanesením ze starších verzí, nebo špatnou interpretací zadání, v žádném případě se však nejedná o chybu, která by svým chování zabraňovala v používání produktu. Zřejmě nejčastěji používanými pojmy jsou poté problém, omyl a chyba. Při vyslovení jednoho z těchto pojmů se nelze setkat s nepochopením ani mezi laiky.

Přesnějšího popisu významu těchto pojmů lze dosáhnout v případě, že je produkt konkretizován na software a ten dále podrobován porovnání vůči specifikacím produktu. Specifikace produktu je dohoda mezi členy vývojového týmu. Tato dohoda detailně popisuje nejen to, jak by se měl navrhovaný produkt chovat, jak by měl vypadat, ale také chování, kterého by se měl produkt vyvarovat.

Pokud bude popisovaný produkt, nyní specifikovaný na software, porovnáván vůči specifikaci, tak lze použít definici softwarové chyby, kterou sestavil R. Patton ([7], s. 14):

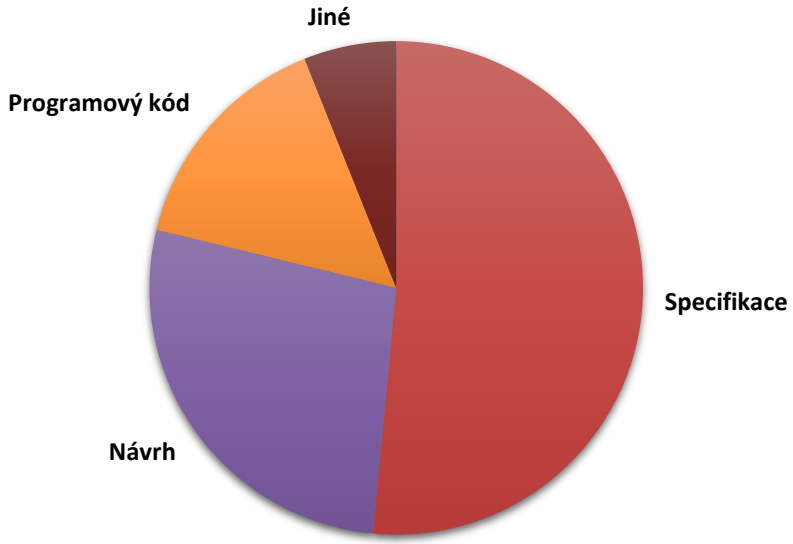
„O softwarové chybě hovoříme právě tehdy, pokud je splněna jedna nebo více z následujících podmínek:

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

O chybě tak lze mluvit také v případě, že produkt nesplňuje požadavky zadavatele, či uživatele, nebo některým chováním odporuje specifikaci produktu.

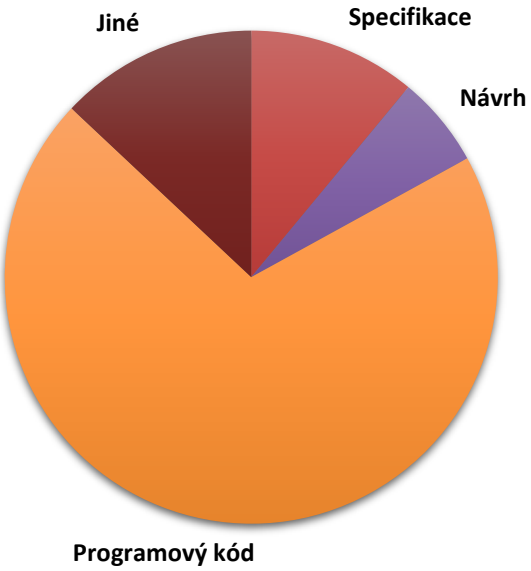
3.2.1 Důvody vzniku chyb

Vznik chyby může mít hned několik různých důvodů. V předešlých letech bylo nejvíce zastávaným názorem, že většina chyb nevzniká při samotném programování, či návrhu ale již v průběhu specifikace. Důvodem těchto chyb ve specifikaci bylo často opomenutí některých požadavků klienta, špatný popis požadavků a v hlavním případě absence či nedodržení některých norem, které specifikují postup tvorby produktu. Tento názor vycházel z několika studií, které zkoumaly projekty různých velikostí. Přesné rozpoložení důvodů dle těchto studií je zobrazeno na Graf 1.



Graf 1 Rozložení příčiny softwarových chyb do roku 2001 ([7], s. 15)

Ovšem i přes to, jak často je tento graf citovaný se dnešní situace výrazně změnila. Díky zavedení velkého množství státních regulací, moderním softwarům pro tvorbu specifikace a všeobecně přijímaným normám, které udávají směr modernímu návrhu softwaru a správného postupu při tvorbě specifikace, se největší chybovost přesunula od specifikace a návrhu k samotným programátorům a do dalších kategorií, jako je například špatné pochopení zákazníka. Díky této změně je nyní procento chyb *„vzniklých ve zdrojovém kódu okolo 70-80 %“* ([8], s. 39). Nynější rozložení je vidět na Graf 2.



Graf 2 Rozložení příčiny softwarových chyb od roku 2001 ([8], s. 40)

3.2.2 Náklady na chybu

Vývoj jakéhokoliv produktu, a především softwaru je často vysoce komplexní záležitostí, tím pádem je jisté, že obtížnost odhalit nějakou chybu i náklady na její odstranění rostou s každým dalším úkonem směrem k produkčnímu nasazení, což dotvrzuje několik studií, viz poslední část tabulky na Obr. 2.

3.3 Testování jako součást vývoje

Vytváření softwaru je komplikovaný a velice komplexní proces, proto se téměř nelze setkat s případem, kdy by na vývoji pracovala jen jediná osoba, která by zastávala několik souběžných rolí. Ačkoliv se sestava vývojářského týmu může případ od případu lišit, vyskytují se téměř vždy následující role: manažer, návrhář, programátor, tester a uživatelská podpora. Jedná se jen o krátký a obecný výčet, počet rolí i osob, které je zastupují se může snižovat, či naopak zvyšovat v závislosti na složitosti projektu a dalších specifických požadavků. [7]

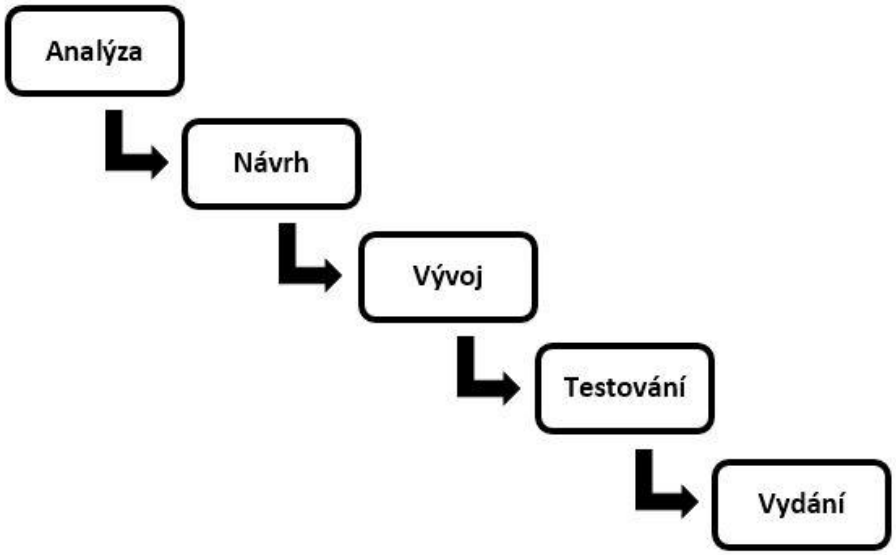
Pointou této kapitoly je poukázat na jeden z možných způsobů jakým lze dosáhnout vzájemné kooperace mezi všemi rolemi. Přesně k tomuto účelu slouží modely životního cyklu softwaru.

Modely životního cyklu softwaru jsou důležité pro plánování testů, jejich průběh a také pro hlášení nalezených chyb. Zároveň je díky těmto modelům možné určit, kdy je nutné testování zapojit do vývoje. [7] Znalost těchto modelů je tak pro testera nezbytnou nutností.

V dalších kapitolách jsou popsány tři nejzákladnější a nejpoužívanější modely životního cyklu, ze kterých poté vycházejí jejich další modifikace, přičemž není možné říct, který z nich je ten nejlepší, každý používá své vlastní metodiky, má své výhody i nevýhody. Jejich použití při vývoji aplikace je závislé na konkrétním zadání a situaci.

3.3.1 Vodopádový model

Vodopádový model je nejjednodušším modelem pro vývoj softwaru s vysokou mírou pochopení, díky čemuž se s ním lze často setkat již při výuce na školách. Vývoj pomocí tohoto modelu postupuje vždy směrem kupředu (směrem dolů ve všech jeho vizualizacích viz Obr. 3), využívá sekvenční přístup u jednotlivých kroků a je charakteristický tím, že lze přestoupit z jedné fáze do druhé až po tom, co jsou splněny všechny požadavky výstupní fáze, bez nároku na krok zpět. Kvůli této podmínce je nutné, aby první fáze, byla zpracována co nejlépe a je na ni kladen velký důraz. [16]



Obr. 3 Vodopádový model [Vlastní zpracování]

Velkou výhodou tohoto modelu je přesná specifikace a předurčenost. Všechny kroky jsou přesně naplánované samostatné části, u kterých stačí mít dostupnou jen určitou skupinu lidí a zdrojů, což minimalizuje veškeré prostředky na vývoj. Z tohoto důvodu se model používá především u menších projektů, kde je zmíněné předurčení vhodnou strategií. Hlavní nevýhodou tohoto modelu je pak nutnost kvalitního zpracování analýzy a návrhu, pokud nejsou návrh a analýza správně provedeny, může nastat situace, kdy je chyba odhalena až ke konci vývoje, právě ve fázi testování. V této fázi už není možné se vrátit a změnit zadání projektu. Především proto není tento model vhodný pro velké projekty a projekty u kterých se očekává pozdější rozšiřování a udržování. [7]

3.3.2 Spirálový model

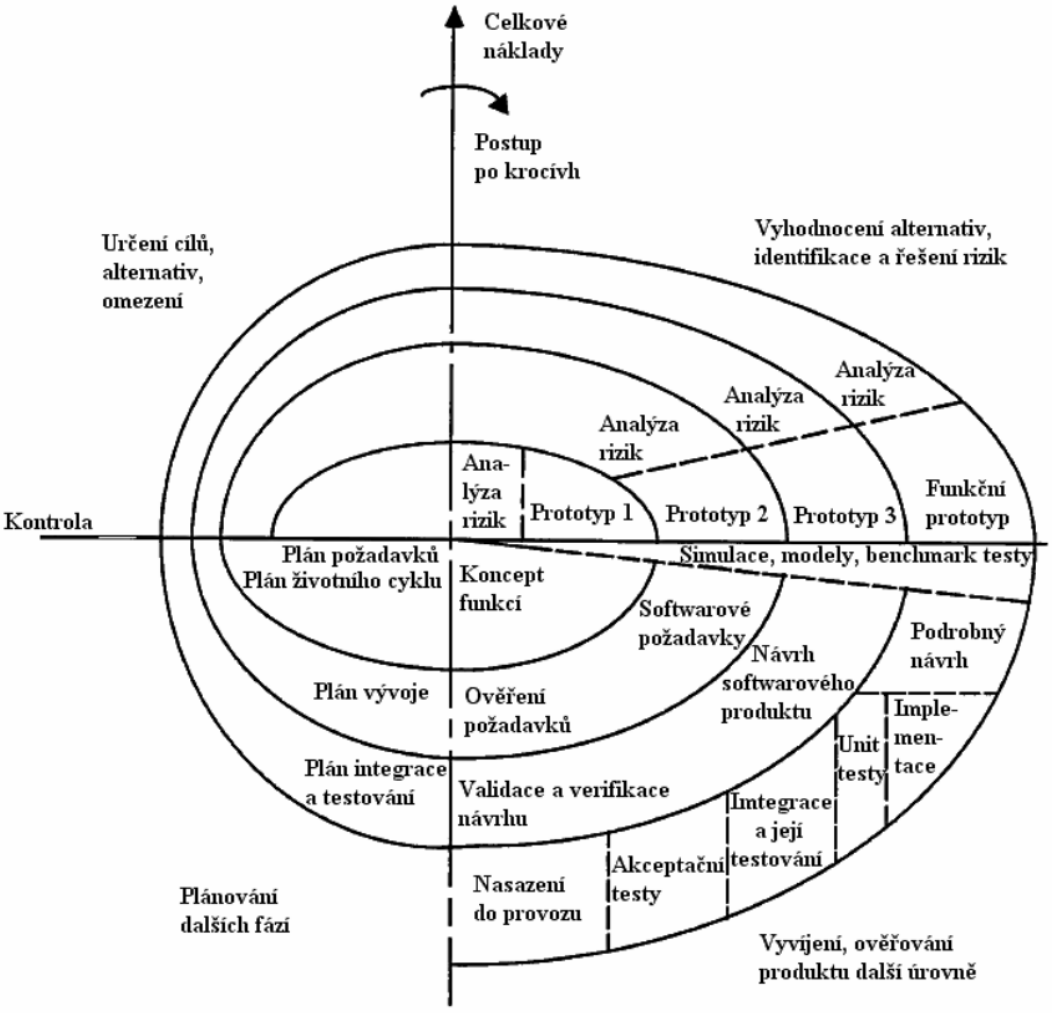
Spirálový model řeší většinu nedostatků vodopádového modelu a přináší několik vlastních inovací. U tohoto modelu má každá fáze několik vlastních kroků, ty dle R. Pattona [7] jsou:

- určení cílů, alternativ a omezení,
- rozpoznání a řešení rizik,
- vyhodnocení alternativ,
- vývoj a testování aktuální fáze,
- plánování další fáze,
- rozhodnutí o postupu na další fázi.

Přechod z jedné fáze modelu do další je závislý na splnění zmíněných šesti kroků, na vyhodnocení alternativ, identifikaci a řešení rizik. Proto je spirálový model řazen do skupiny přístupů řízených riziky.

Implementace spirálového modelu spočívá v definování hrubých požadavků klienta hned na začátku vývoje. Následně je k nim sepsána dokumentace a celá část je podrobena testování. Nakonec se tato hotová část předá klientovi na připomínkování, na základě těchto připomínek je projekt posunut do další fáze, ve které mohou být přidány nové funkce. Tento princip se opakuje, dokud není klient s projektem spokojen a jsou pokryty veškeré požadované funkce, viz Obr. 4.

Díky tomu, že se celý proces vývoje neustále opakuje, přidávají se nové funkce, případně se upravují ty stávající, je tento model velice vhodný pro větší projekty, ve kterých se mnohem snáze vyskytnou chyby, jenž se dají díky spirálovému modelu rychleji objevit. Stejně tak je jeho užití výhodné pro testera, který má díky pravidelnému testování po každé fázi vývoje projekt neustále pod dohledem. Jelikož testování může procházet několika iteracemi jen s drobnými změnami je výhodné pro tento model zavést také automatizované testování. [17]



Obr. 4 Spirálový model [17]

3.3.3 Modely agilního vývoje

Modely vycházející z vývoje pomocí agilního přístupu se od předchozích dvou jmenovaných liší především v tom, že jsou schopny rychle reagovat na změnu požadavků v průběhu jednoho cyklu, a také tím, že se u nich upozaďuje dodržování doporučených procesů a používání daných nástrojů. [8]

Agilní přístup se skládá z cyklů vývoje stejně jako spirálový model, avšak tyto cykly nemají přesně stanovený postup řešení konkrétních částí vývoje. Jednotlivé požadavky se v těchto cyklech implementují vždy na základě jejich priority, čemuž odpovídá i pořadí jednotlivých částí vývoje v daném cyklu. Délka cyklů bývá obvykle stanovena na 2 až 4 týdny. [8] Mezi agilní přístupy vývoje patří například Scrum (přírůstkový model vývoje) a Test-Driven Development (vývoj řízený na základě testů). [8]

4 Automatické testování

Jak již bylo několikrát zmíněno, vývoj větších projektů je finančně, personálně i technicky náročný. Zdroje, které může automatické testování ušetřit, rozepsal R. Patton ([7], s. 186) do čtyř následujících vlastností vypovídajících ve prospěch automatizace testů:

„Vlastnosti testovacích nástrojů a automatizace testů jsou:

- ***Rychlost.*** *Uvažujte, jak dlouho by vám trvalo, než byste manuálně vyzkoušeli několik tisíc testových případů v Kalkulačce Windows. Jeden testový případ vám v průměru zabere nějakých 5 sekund. Automatické nástroje mohou testové případy provádět 10krát, 100krát, nebo i 1000krát rychleji.*

- ***Efektivita.*** *Během práce na provádění testových případů již nemůžete dělat nic jiného. Jestliže dostanete do rukou testovací nástroj, který zkrátí čas nezbytný k provádění testů, pak vám zbyde více času na plánování tesů a promýšlení nových testových případů.*

- ***Správnost a přesnost.*** *Po ručním vyzkoušení několika stovek testových případů se zákonitě oslabí vaše pozornost a začnete ve své práci dělat chyby. Testovací nástroj udělá stejnou práci, vykoná stejné testy a zkontroluje výsledky pokaždé naprosto dokonale.*

- ***Neúnavnost.*** *Automatické testovací nástroje neznají únavu a nikdy svou práci jen tak nevzdají. Fungují asi jako králíčci s bateriemi Duracell ze známé televizní reklamy – pořád běží a běží a bubnují a bubnují…“*

Automatické testování přináší možnost snížit dva ze tří zdrojů popsaných v prvním odstavci ještě o něco víc než testování samotné. Někteří manažeři, kteří se rozhodnou zavést automatické testování do svých projektů se mylně domnívají, že se jedná o všemohoucí nástroj, který jim umožní rapidně snížit počet testerů, tím snížit náklady na testování a zároveň tím dosáhnout téměř 100% odhalení chyb.

Názor o snížení počtu testerů je jedním z nejčastějších omylů, kterého se manažeři dopouštějí. Pro vyvinutí kvalitního řešení automatizace je však nutné nejdříve počet dobře kvalifikovaných testerů zvýšit, po skončení vývoje je možné počet testerů vrátit

na původní úroveň, ale není možné jejich počet snížit ([8], s. 176-177). Je to především z důvodu, dalšího udržení automatických testů v podobě reflektující testovaný software a také proto, že automatizace nikdy nemůže zcela nahradit manuální testování.

4.1 Automatické versus manuální testování

Automatické testy tedy prozatím nejsou schopny plnohodnotně nahradit testování manuální. Oba přístupy jsou často vnímány jako dva oddělené celky což nemusí být nutně pravdou, správné užití automatických testů může sloužit jako podpora a ulehčení některých procesů manuálního testování. Nejčastěji jsou užívány pro účely regresivního testování[2]. Z toho vyplývá, že se automatické testy nejvíce hodí pro úkony, které se často opakují a mají velké množství vstupů, zároveň by se však tato data neměla často měnit, aby nebylo nutné neustále předělávat skripty automatických testů. Výhoda poté spočívá v tom, že stroj provádějící úkony nepociťuje únavu a nudu, která by se při vykonávání repetitivních úkonů dostavila u testera, který by v důsledku toho mohl opomenout některý důležitý detail či chybu ([7], s. 186).

Předchozí odstavec popisuje nejčastější využití automatizaci testů a jeho největší výhodu. Nyní je třeba představit také její hlavní nevýhodu, kterou je právě výše zmíněná nenahraditelnost manuálního otestování. Stroj nebude nikdy moci testovat úlohy, které závisí na intuici, kreativitě a porovnání proměnných či nepředvídatelných okamžiků. Příkladem může být okamžik, kdy dojde ke drobné změně v textu tlačítka, s čímž si již stroj nedokáže poradit. ([19], s. 45)

Následující kapitoly porovnávají některé další výhody a nevýhody manuálního i automatizovaného testování. Avšak tento soupis není ani zdaleka konečný a výhod i nevýhod by se dalo najít nespočet.

[2] *„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.“ [18]*

4.1.1 Další výhody a nevýhody automatického testování

Výhody:

- úspora času při zjišťování místa závady[3] ([20], s. 35),
- vysoká spolehlivost, je eliminován lidský faktor ([7], s. 186),
- lze provádět kdykoliv[4].

Nevýhody:

- *„Komplikace při rozpoznání a detekci objektů“* ([21], s. 23),
- nelze s nimi otestovat náročné a mezní situace vyžadující kreativitu ([7], s. 201),
- vyšší náročnost na znalosti při tvorbě testů.

4.1.2 Další výhody a nevýhody manuálního testování

Výhody:

- tester se snadno přizpůsobí změnám v testovaném produktu ([20], s. 36),
- nižší vstupní náklady na nástroje a proškolení ([20], s. 36),
- lze odhalit neobvyklé kreativní kombinace, na které by stroj nikdy nepřišel.

Nevýhody:

- malé množství osob, nedokáže kvalitně otestovat všechny situace ([20], s. 36),
- méně kvalitní z důvodu lidského faktoru a únavy ([20], s. 36),
- nevhodné pro testování velkého objemu dat.

[3] Podmínkou této úspory je dobré logování průběhu testů. – pozn. aut.
[4] Je to především z důvodu, že stroje nenají omezenou pracovní dobu. - pozn. aut.

4.2 Nástroje pro automatické testování

Na poli automatizace existuje velké množství nástrojů pro tvorbu automatizovaných testů. Tyto nástroje lze dělit do dvou typů:

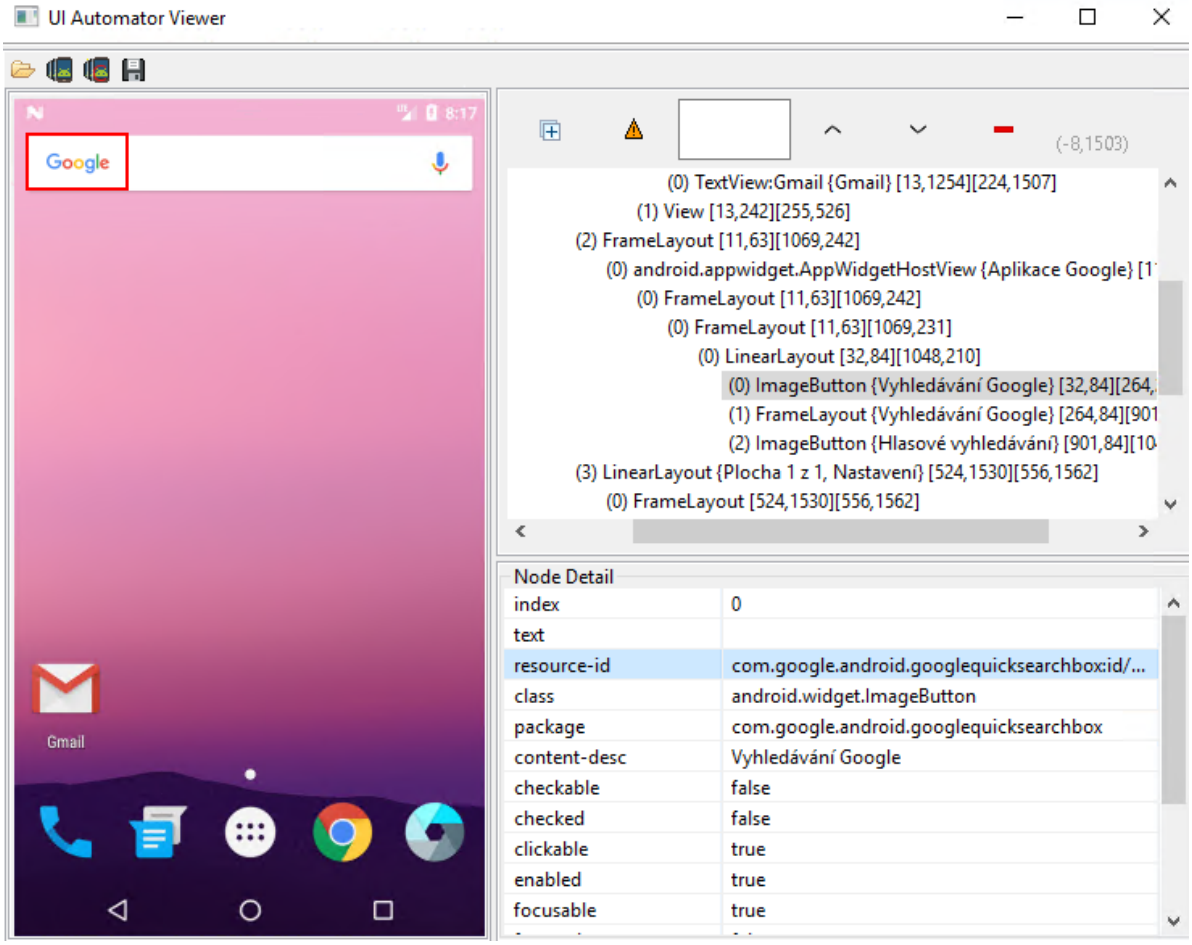
- **Pro testování černé skříňky –** Nástroje pro testování černé skřínky slouží k vizuálnímu otestování funkcí, pro zjištění správné funkcionality není nutné znát procesy skryté v kódu systému. Stačí ověřit, že se na výstupu objeví výsledky odpovídající zadaným vstupům. ([7], s. 48)

- **Pro testování bílé skříňky –** Na rozdíl od testování černé skříňky je u tohoto typu testování nutné znát také vnitřní funkcionalitu, díky tomu je možné nahlížet také do zdrojového kódu a ověřovat zda funkcionalita vykonává svou funkci korektně a ne pouze dohlížet na správnost výsledků. Tomu jsou také přizpůsobeny nástroje, například při testování síťové komunikace se hodí nástroje, které umí rozeznat komunikaci daných protokolů a sledovat komunikaci i na nižších síťových vrstvách. ([7], s. 49, 175)

Nástroje pro automatizované testování lze dále dělit i do kategorií, dle toho, v jaké části testování a pro jaký účel mají být využity. Toto rozdělení použil také R. Patton ve své knize, nástroje rozdělil do následujících skupin: *Prohlížeče a monitory, Ovladače, Makety, Zátěžové nástroje, Generátory šumu a Analytické nástroje,* ([7], s. 186) všechny tyto kategorie budou v následujících kapitolách vysvětleny. Toto rozdělení a užití jednotlivých kategorií jsou z pohledu testování více než jasné, avšak pro dnešní dobu možná již částečně zastaralé – dnes je většina těchto nástrojů integrována do jednoho komplexního a uceleného balíku nástrojů.

4.2.1 Prohlížeče a monitory

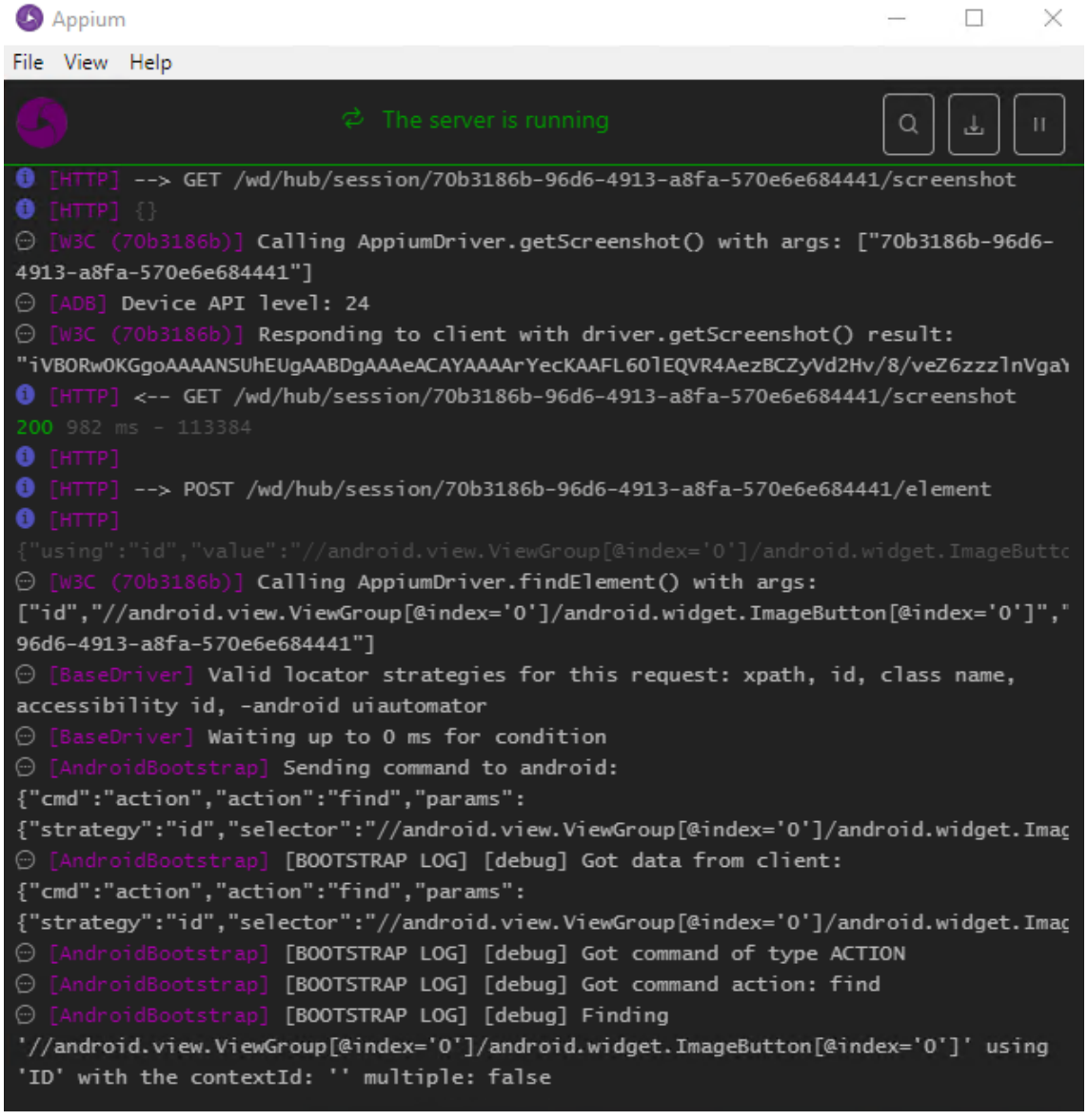
Nástroje v této kategorii slouží především pro zobrazení vnitřní struktury GUI (nejčastěji zobrazené v podobě XML stromu), mezi takové patří například Windows Inspect, pro UWP (univerzální platforma Windows) aplikace či UIAutomatorViewer pro mobilní platformu Android. V těchto dvou příkladech je sice struktura GUI zpět sestavena do uživatelsky přívětivé podoby, avšak zaznamenaná obrazovka daného programu se dá z těchto prohlížečů exportovat do formátu XML.



Obr. 5 Ukázka prohlížeče používaného pro emulátor Android

4.2.2 Ovladače

Ovladače neboli drivery slouží pro samotné spouštění testovacích scriptů a vykonávání jiných ovládacích funkcí jako je například swipe u mobilních zařízení, nastavení simulované GPS (globální systém pozicování) lokality nebo mačkání jednotlivých kláves. Pro platformu Android jsou to například nativní drivery UIAutomator, UIAutomator2 a Espresso. Pro platformu Windows je to WinAppDriver.



Obr. 6 Ukázka komunikace frameworku Appium s konkrétním driverem

4.2.3 Makety

Makety jsou na rozdíl od ovladačů výstupním nástrojem, v dnešní době nezřídka obsaženým v samotných ovladačích. Slouží například pro zisk textové hodnoty z daného prvku nebo snímku obrazovky celého zařízení či v případě UWP pouze okna aplikace. Lze sem však obecně řadit i nástroje umožňující emulaci výstupních zařízení, jako je tiskárna.

4.2.4 Zátěžové nástroje

Tyto nástroje, jak už jejich název napovídá, simulují zátěž testovaného zařízení či softwaru. Může se jednat o simulaci výkonové zátěže, ale také o zátěž v podobě generování dlouhých řetězců či velkého objemu dat, které jsou poté do testované aplikace vkládány.

4.2.5 Generátory šumu

Generátory šumu jsou vysoce specifické nástroje, které se používají především jako podpora zátěžových nástrojů. Simulují náhodné zátěžové stavy, například nahodilé stavy nedostatečné paměti či zátěž procesoru, které se v průběhu používání mohou vyskytnout. ([7], s. 191)

4.2.6 Analytické nástroje

Analytické nástroje jsou podpůrným nástrojem pro testera, může do nich spadat jak jednoduché ukládání výsledků testů do tabulkového procesoru, tak stopky a ostatní typy měřidel pro záznam času testů. Z pohledu komplexnějších nástrojů sem spadají různé typy softwaru pro nahrávání a snímání obrazu ([7], s. 191) či velice komplexní nástroje jako jsou systémy pro vývoj a správu projektů (např. JIRA) nebo systémy pro sestavení, nasazení a průběžnou integraci (např. TeamCity), které mohou být ještě dále rozšiřovány pomocí pluginů pro ně vytvořených.

4.3 Dělení nástrojů dle složitosti

V dnešní době se lze čím dál víc setkat s trendem integrace a propojováním různých systémů, z toho důvodu lze mezi testovacími nástroji nalézt různé typy nástrojů které v sobě integrují některé funkce z výše popsaných kategorií. Poté již lze mluvit o komplexních nástrojích pro automatické testování, které se dále dělí podle složitosti jejich použití pro samotného testera.

Zadavatel projektu si pak může vybrat, jestli raději zvolí méně složité nástroje a najme na jejich obsluhu méně akreditovaný personál i za cenu složitější udržitelnosti testů, nebo raději najme dostatečně kompetentní personál pro obsluhu automatizačních nástrojů, ve kterých lze projekty bez větších problémů udržovat a vylepšovat bez nutnosti velkých změn.

4.3.1 Záznam maker

Nástroje pro záznam a přehrávání maker jsou jedny z nejjednodušších automatizačních nástrojů, přesto jsou stále společnostmi nabízeny jako komerční řešení, u kterého není třeba žádných odborných znalostí. Přináší to s sebou však také obtíže v podobě kompatibility, kdy musí být v případě změny některé ze základních funkcionalit celá sada testů nahrána znovu. Zaznamenávání a přehrávání maker nenabízí možnost verifikace výsledků a tester se tak musí spokojit pouze s tím, že nemusí neustále dokola manuálně provádět určité scénáře. Kvůli nemožnosti verifikace není možné do maker zavést implicitní čekání na zobrazení některých prvků či webových stránek. ([7], s. 192)

4.3.2 Skriptování maker

Nástroje této komplexity již vyžadují určitou znalost programování, která s tím také přináší odstranění nevýhody v podobě kompatibility zmiňované v předešlém odstavci. Stále se však jedná jen o záznam maker, tentokrát v podobě skriptů. Existují zde přesto některé nevýhody, v podobě absence proměnných, rozhodovacích příkazů, maket (nemožnost verifikace), zátěžových nástrojů a některých analytických nástrojů. Tyto problémy lze vyřešit použitím komplexních automatizačních frameworků. ([7], s. 194)

4.3.3 Plně programovatelné automatizované nástroje

Automatizační frameworky jsou tím nejkomplexnějším nástrojem pro automatizované testování, obsahují v sobě ovladače, makety i některé analytické nástroje. Nevýhodou těchto frameworků je jejich složitost, testy jsou zde totiž skriptovány za pomocí vysokoúrovňových programovacích jazyků a je zde tak opět nutná určitá znalost testované platformy. I přes tyto nevýhody jsou však automatizační frameworky tím nejmocnějším nástrojem testera. Možnosti testování jsou zde omezeny téměř jen schopnostmi toho, co tester zvládne naprogramovat. Velké množství automatizačních frameworků zastává open-source řešení a je zde tedy prostor pro nové inovace, pokud se programátor rozhodne zapojit do komunity vývojářů daného frameworku. Konkrétnímu popisu jednotlivých frameworků a jejich porovnání budou věnovány další kapitoly.

4.4 Typy testovaných mobilních aplikací

Tester by měl také znát typy aplikací, které lze na mobilních zařízeních testovat. Aplikace pro mobilní zařízení se dělí do tří kategorií - nativní, hybridní a webové. Podrobnému popisu toho, jak fungují nativní, hybridní a webové aplikace je věnováno několik kapitol v knize Mobile Application Testing: A Tutorial, kterou sepsal Jerry Gao a spol. [23] V následujících podkapitolách budou tyto tři typy aplikací stručně představeny.

4.4.1 Nativní aplikace

Nativní aplikace jsou vytvořeny pouze pro konkrétní operační systém. To znamená, že aplikaci vyvinutou pro operační systém iOS nelze nainstalovat a použít na operačním systému Android a obráceně. Pokud je potřeba, aby aplikace byla multiplatformní, je nutné vytvořit několik separátních klientských aplikací pro každou platformu. Existují zde ale metody, jak určité části zdrojového kódu využít u obou platforem a metody pro portování aplikací na odnože pro nositelnou elektroniku nebo chytrou domácnost postavenou na dané platformě. [23] Tyto nativní aplikace jsou běžně distribuovány skrze nativní distribuční sítě, jako jsou Google Play Store a Apple App Store.

4.4.2 Webové aplikace

Mobilní webové aplikace jsou vždy uživateli zprostředkovány skrze webový prohlížeč zabudovaný v mobilním zařízení. Jelikož tyto aplikace cílí na webové prohlížeče, lze je využít na tabletech, mobilních telefonech i osobních počítačích nezávisle na operačním systému. [23]

4.4.3 Hybridní aplikace

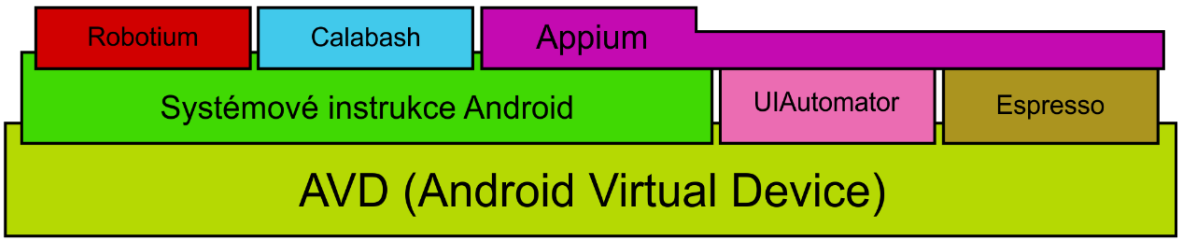
Hybridní aplikace jsou kombinací nativních a mobilních webových aplikací. Tento typ aplikace zajišťuje kompatibilitu mezi platformami, ale stále může přistupovat k hardwaru telefonu. [23] Častěji se k implementaci takovýchto aplikací využívá WebView komponenta.

5 Automatizační frameworky

Tato kapitola se zaměřuje na popis automatizačních frameworků pro platformu Android. Dle několika nezávislých zdrojů citovaných v této kapitole bylo vybráno pět nejpoužívanějších automatizačních frameworků, kterým budou věnovány následující odstavce. Jedná se o open source frameworky Appium a Robotium, ale také frameworky spadající pod licence velkých společností, Espresso, UIAutomator a Calabash. Tyto frameworky figurují téměř ve všech článcích, které se k testování pomocí automatizačních frameworků dají nalézt.

Automatizační frameworky pro platformu Android je možné rozdělit do dvou hlavních větví, které se mohou v některých bodech překrývat. Tyto dvě větve se dělí na frameworky používající drivery, které komunikují s testovaným zařízením přímo pomocí systémových instrukcí, a na frameworky, které používají pro tuto komunikaci Android API (aplikační programové rozhraní). [22] Zmíněné překrývání se týká především frameworků UIAutomator, Espresso a Appium. UIAutomator a Espresso spadají do kategorie, která pro komunikaci používá přímo systémové instrukce, jsou obsaženy již v systému Android a lze je tedy využít jako API pro komunikaci se systémem. Toho využívá framework Appium ve kterém je možné jako driver využít dva předchozí zmíněné frameworky a využít tak pro komunikaci se systémem jejich API. [22]

Pro vizuální pochopení může posloužit Obr. 7 Rozdělení frameworků dle způsobu komunikace s AVD (virtuální zařízení Android).



Obr. 7 Rozdělení frameworků dle způsobu komunikace s AVD [22]

5.1 Popis automatizačních frameworků

5.1.1 Espresso

Espresso je nativní testovací framework pro platformu Android. Lze jej použít především pro testování nativních aplikací, ale některé z jeho funkcí však mohou najít využití i u hybridních aplikací (WebView aplikace). Popis toho, jak fungují nativní, hybridní a webové aplikace je vysvětlen v kapitole 4.4.

Espresso byl vyvinut společností Google. Od verze 3.0.0 již Espresso podporuje jen Android API 15 a vyšší. V některých pracích i článcích se stále lze dočíst, že Espresso podporuje i nižší API 8, 9 a 10, tuto skutečnost lze však vyvrátit na základě release note Espresso 3.0.0 [24]. Framework Espresso se zaměřuje především na testování GUI, řadí se tudíž mezi nástroje pro testování černé skříňky. Avšak při jeho použití je nutné k testované aplikaci přistupovat jako k šedé skříňce, jelikož pro lokalizaci prvků je nezbytná znalost vnitřní struktury GUI, viz kapitola 4.2.1. Psaní testů v tomto frameworku je možné pouze pomocí programovacího jazyka Java a Kotlin [25].

5.1.2 UIAutomator

UIAutomator je taktéž nativní framework pro automatizované testování operačního systému Android vyvinutý společností Google, jedná se v podstatě o předchůdce frameworku Espresso. UIAutomator následně dostal svou verzi 2.0 a nyní jsou Espresso a UIAutomator 2.0 separátní větve automatizačních frameworků, které vycházejí ze stejných základů. Aktuální verze UIAutomator podporuje pouze Android API 18 a vyšší. [26]

Na rozdíl od frameworku Espresso je možné pomocí UIAutomator testovat také GUI webových aplikací v mobilním zařízení, avšak pouze klikáním na souřadnice určené horizontální a vertikální osou [27]. UIAutomator podporuje psaní testovacích scénářů v programovacím jazyce Java a Kotlin [26].

5.1.3 Calabash

Tento framework se od prvních dvou jmenovaných v mnohém liší, v tomto případě se již nejedná o nativní testovací framework pouze pro operační systém Android. Existuje ve dvou separátních provedeních, přičemž jedna větev je určena pro testování na platformě iOS a druhá pro testování na platformě Android. Calabash je vyvinut společností Xamarin spadající do portfolia společnosti Microsoft. [28]

Na platformě Android je možné vytvářet a testovat v Calabash jak nativní aplikace, tak i ty hybridní a webové. [9] Testovací scénáře v tomto frameworku je možné psát v programovacím jazyku Ruby. Výhodou pro laické uživatele a zájemce o automatizované testování může být to, že testy lze v Calabash psát také za pomocí Cucumber Gherkin, což je nástroj umožňující psát testovací scénáře v „lidsky“ čitelné podobě. ([22], [28], [29])

Další výhodou, díky které se lze rozhodnout pro použití Calabash při vytváření automatizovaných GUI testů, je přítomnost Xamarin Test Cloud, kde společnost Xamarin, jako vývojář frameworku Calabash, nabízí možnost psát a spouštět testovací scénáře ve webovém prostředí. Zároveň zde lze nalézt velkou nabídku reálných zařízení připojených do cloudu. Tato zařízení fungují na operačních systémech iOS i Android v různých kombinacích hardwarového i softwarového nastavení. Dle Xamarin je zde přes 1000 zařízení, přičemž je seznam každý měsíc navyšován o stovky dalších. Nevýhodou tohoto řešení je cena. Zájemci si však celý systém mohou vyzkoušet na 30 dní zdarma. [28]

5.1.4 Robotium

Robotium byl jedním z nejpoužívanějších testovacích frameworků v počátcích vývoje operačního systému Android [30]. Následně se objevilo několik konkurentů, kteří se mu funkcionalitou více či méně vyrovnávali. Robotium však nabízí funkci, jež umožňuje sledování momentálně využívané android aktivity, kdy se v případě nutnosti na tuto aktivitu automaticky přepne, což je obrovskou výhodou [31]. U ostatních popisovaných frameworků tato funkčnost chybí a je nutné si ostatní aktivity obsluhovat ručně, případně pracovat jen s jednou předem určenou aktivitou.

Robotium je framework určený pouze pro platformu Android s podporou všech verzí API. Vychází z frameworku Selenium, který je určen pro testování webových aplikací a díky tomu také podporuje testování všech typů aplikací, nativních, hybridních i webových [9]. Pro psaní testovacích scénářů je možné v Robotium použít pouze programovací jazyk Java. ([22], [32])

5.1.5 Appium

Appium je framework s největší komunitou uživatelů i přispěvatelů mezi frameworky pro automatizované testování mobilních aplikací. Díky tomu Appium nabízí velké množství nástrojů, které lze použít při psaní testů pro různé platformy. Appium umožnuje testování nativních, hybridních i webových aplikací na operačním systému Android i iOS [9]. Pro Android je zde nejnižší podporované API verze 16 a pro iOS zde žádná omezení nejsou. Dále je možné Appium použít také při testování desktopových Windows UWP i Win32 aplikací a OS X desktopových aplikací. Vše závisí jen na použití vhodného Appium Driveru. ([22], [33])

Při použití frameworku Appium má vývojář na výběr hned z několika jazyků, ve kterých lze psát testovací scénáře. Mezi tyto jazyky patří Java, Python, JavaScript, Ruby, C# a také PHP. Takový seznam podporovaných jazyků lze jen těžko hledat u jiných frameworků, což bohužel nese i svá úskalí. Každý z těchto jazyků má vlastní vývojovou větev frameworku Appium, tudíž se u některých jazyků lze setkat s funkcemi, které do jejich vývojové větve ještě nebyly implementovány, přesto že je takováto funkce popsána v dokumentaci Appium. ([30], [31], [33])

O vývoj tohoto frameworku se stará komunita zajímající se o automatizované testování jakéhokoliv softwaru, prozatím nespadá pod žádnou velkou společnost, tak jako je tomu například u Calabash. [33]

Pro využívání Appium je nutné si z jejich webových stránek stáhnout Appium Server v podobě desktopové aplikace, nebo tento server stáhnout v podobě JavaScript modulu skrze správu balíčků Npm. Poté stačí Appium balíček importovat do IDE (vývojové prostředí) dle zvoleného jazyku. Následně je možné začít psát testy, tak jak je popsáno v dokumentaci. [33]

5.2 Porovnání automatizačních frameworků

	Espresso	*UIAutomator*	*Calabash*	*Robotium*	*Appium*
Andorid	Ano	Ano	Ano	Ano	Ano
Android API	15 a výše	18 a výše	Vše	Vše	Vše
iOS	Ne	Ne	Ano	Ne	Ano
Desktop	Ne	Ne	Ne	Ne	Ano
Typ testované aplikace	Nativní	Nativní + omezeně i ostatní	Nativní, ostatní jen na Android	Nativní, hybridní i webové	Nativní, hybridní i webové
Scriptovací jazyk	Java	Java, Kotlin	Ruby, Cucumber Gherkin	Java	Java, Python, Javascript, Ruby, C#, PHP
Vývojáři	Google	Google	Xamarin (Microsoft)	Komunita	Komunita
Testovací zařízení	Android Studio	Android Studio	Android Studio, Xamarin Test Cloud	Android Studio	Android Studio, Genymotion
Reálná zařízení	Ano	Ano	Ano	Ano	Ano

Tabulka 1 Souhrnné porovnání popsaných frameworků [Vlastní zpracování]

Dle soupisu informací popsaných v předchozích kapitolách je možné dojít k závěru, že jsou na trhu dva frameworky, které nabízí nejvíce variací k použití. Těmito frameworky jsou Calabash a Appium.

Calabash má výhodu především díky široké nabídce zařízení, běžících v cloudu, které lze k testování využít. Dalším rozdílem oproti Appium je zde možnost psát skripty testovacích scénářů za pomocí Cucamber Gherkin. Appium oproti tomu nabízí možnost testovat také desktopové aplikace a webové aplikace na iOS, také je zde velký výběr programovacích jazyků určených pro psaní testů samotných. Tato skutečnost byla také důvodem, proč byl pro poslední část této práce vybrán právě framework Appium, jelikož společnost Systemart s.r.o, pro kterou bylo automatické testování jejich aplikace programováno, požadovala, aby testy byly psány v jazyce C#, jenž společnost používá napříč celým svým portfoliem.

Dle vlastních zkušeností s frameworkem Appium a na základě komunikace s podporou Calabash i Appium při získávání informací pro tuto práci, je na místě domněnka, že framework Calabash je využíván i určen spíš pro uživatele, kteří upřednostňují hotová a funkční řešení. Zato framework Appium je více používán uživateli, kteří si rádi problémy řeší svépomocí, či za pomoci komunity a jsou ochotni se připojit do komunity vývojářů.

6 Aplikace Výkaz práce

Aplikace Výkaz práce je docházkový systém pro jednotlivce i firmy. V aplikaci je možné evidovat docházku, provedenou práci a pohyby firemních vozidel. Tyto funkce je možné využívat na pracovišti, na cestách nebo z domova. Aplikace je postavena na platformách ASP.NET, Xamarin a Microsoft SQL server. Vyvíjí ji společnost Systemart s.r.o., která byla založena v roce 2003, se sídlem v Hradci Králové a další pobočkou v Brně.

Celá aplikace se skládá z několika vzájemně propojených agend, které umožnují komplexní vyhodnocení zadaných údajů. Správce účtu má možnost si systém nastavit dle potřeb vlastních i potřeb případných pracovníků. Tato bakalářská práce pojednává o ve verzi 7.5.x. Nyní probíhají přípravy nové verze 7.6.x, ve které budou některé funkcionality kompletně přepracovány, a počet agend zredukován.

6.1 Struktura aplikace

6.1.1 Docházka

Docházka umožnuje pracovníkům zaznamenávat příchody, odchody, absence a přestávky na pracovišti. Ke každému záznamu lze přiřadit lokalitu, která může být zaznamenána ze senzorů klientů automaticky, což umožnuje vytvářet věrohodné podklady pro mzdy a vytvářet přehled přítomnosti pracovníků na pracovišti.

6.1.2 Rozpis práce

Rozpis práce nabízí možnost rozepisovat intervaly záznamů z docházky dle jednotlivých úkonů a zakládat záznamy samostatných úkonů vykonaných během pracovní doby. K záznamům lze přiřadit jednotlivé zakázky, určit jejich fakturovatelnost, přiřadit nákladovou i fakturační sazbu a zadávat ostatní uživatelsky konfigurovatelné informace v podobě vlastních polí. V jedné z novějších minoritních verzí verze 7.5.x byla správcům přidána možnost schvalovat takto vytvořené záznamy, aby se zvýšila jejich věrohodnost a zamezilo se podvodným záznamům ze strany zaměstnanců.

6.1.3 Kniha jízd

V knize jízd lze evidovat firemní jízdy vozidel. Jednotlivým jízdám lze přiřazovat také zakázky a nastavovat sazby, s jejichž pomocí lze u vozidla sledovat i příslušné náklady. U každé jízdy lze také evidovat prostoje s rozdílnou sazbou oproti zbytku jízdy. I v agendě Kniha jízd je zavedena určitá ochrana proti zneužití ze strany zaměstnanců, která spočívá v omezení zadávání záznamů za jednotlivá vozidla a právo zakládat a upravovat všechny jízdy přiřazené k danému vozidlu.

6.1.4 Zakázky

Zakázky lze přiřadit ke každé zadané práci, jízdě s vozidlem, nebo k výdaji. Zakázky jsou dále členěny na podzakázky, a ke každé zakázce lze také přidat plánovaný, nákladový a časový rozpočet. Díky tomu lze získat detailní přehled o profitu z jednotlivých zakázek a k nim vykázaných záznamů.

6.1.5 Výdaje

Pomocí výdajů lze evidovat ostatní jednorázové náklady a výnosy, které nejsou vykázány za žádnou pracovní činnost. Lze k nim také přiřazovat zakázky pro komplexní přehled ziskovosti dané zakázky.

6.1.6 Fakturace

Fakturace je určena pro vytváření podkladů pro fakturaci z vykázané práce, nebo jízd. Vytvořené podklady pro fakturaci lze poté exportovat a využít je tvorbu faktur v některém z ekonomických systémů.

6.2 Přístup k aplikaci

Hlavní část aplikace Výkaz práce je nyní přístupná jako webová aplikace, kterou podporuje velké množství moderních webových prohlížečů. Konkrétně je aplikace testována na Internet Explorer 11, Microsoft Edge, Google Chrome, Firefox a Safari. Tato webová aplikace je koncipována jako klient-server, provádí se zde veškeré vyhodnocení záznamů, vytváření statistik a vyhodnocování práv. Klientská strana se stará

o vytváření záznamů zasílaných do webové aplikace a udržování své lokální databáze záznamů a číselníků dle práv přihlášeného uživatele, dále se klientské aplikace starají například o přiřazení GPS lokality k záznamům.

K webové aplikaci se dá přistupovat pomocí klientů pro mobilní telefony se systémem Android a iOS, tyto klientské aplikace jsou určeny především pro užití konkrétní osobou, z toho důvodu je již na serveru vyhodnoceno, ke kterým informacím má uživatel přístup. Uživateli se tak následně synchronizují pouze informace a push notifikace, na které má nastavena práva.

Dále jsou vyvíjeny i klientské aplikace pro osobní počítače, přesněji se jedná o aplikace univerzální platformy Windows. Hlavní klientská UWP aplikace se liší od mobilních aplikací tím, že v ní lze přepínat i na jiné uživatele. Hlavní přihlášená osoba je pouze jedna, jakmile jsou však této osobě přiřazena práva zobrazovat nebo zakládat záznamy v některé z agend za jinou osobu, tak se jí zpřístupní možnost přepnout se na tuto novou osobu. Aplikace si následně stahuje a ukládá jen informace o osobách a číselnících, na které má přihlášená osoba práva se přepnout.

Poslední klientská aplikace funguje také na platformě UWP. Jedná se o aplikace pro terminálové kiosk zařízení určené pro provoz ve Windows Assigned access, do aplikace je možné se přihlásit pouze pomocí názvů účtu a bezpečnostního klíče generovaného ve správcovské sekci webové aplikace. Aplikace umožnuje přihlášení pracovníků pomocí NFC (bezdrátová komunikace na krátkou vzdálenost) čipů a modulu na čtení otisků prstů. Firma Systemart s.r.o k této aplikaci vyrábí a dodává také vlastní hardwarové řešení.

6.3 Vývoj a testování

Aplikace je vyvíjena ve dvou pobočkách, první z nich se nachází v sídle společnosti v Hradci králové, na této pobočce působí dva vývojáři, oddělení testerů (to nyní čítá pouze dva zaměstnance), analytik a zákaznická podpora, které se stará mimo jiné o vytváření dokumentace k aplikaci. Druhá pobočka se nachází v Brně, pracují zde pouze vývojáři, kteří jsou cekem tři. Celkově se tak na vývoji celé aplikace od návrhu až po kompletaci dokumentace podílí devět osob.

Celý vývoj aplikace je řízen dle metodiky agilního vývoje. V každé iteraci jsou tak požadavky a nalezené chyby z předchozí iterace zaneseny do nástroje JIRA, a následně rozděleny podle své priority mezi vývojáře. Po vyřešení všech úkolů je daná iterace uzavřena a iterace nasazena k testování na testovací webový server. Při nasazování probíhá také automatické jednotkové testování, které spravují samotní vývojáři.

Úkolem testerů je spustit automatické testy pro UWP platformu, Webovou aplikaci a také pro platformu Android. Tyto testy mají za úkol otestovat základní funkčnost, aby se ověřilo, že při implementaci nových funkcí nebyla chyba zanesena také do funkcionalit, které měly zůstat nezměněny. Testovací scénáře jsou dále rozděleny mezi testery k manuálnímu otestování. V testovacích scénářích se nachází identifikátor prostředí, kterého se daný scénář týká, podrobný postup, jak otestovat požadovanou funkčnost a vlastnosti aplikace a také připojený úkol z fáze vývoje pro případnou kontrolu částí zdrojového kódu pomocí programu FishEye, nebo zadání samotného úkolu. Každý takto vytvořený scénář by měli otestovat všichni momentálně dostupní testeři, aby se co nejvíce omezila možnost, že některá chyba zůstane neodhalena. V případě výskytu chyb se tato iterace vrátí zpět do vývoje a celou situaci popsanou výše absolvuje, dokud nejsou vyřešeny veškeré nalezené chyby, nebo neúplnost nově zaváděných funkcionalit.

Finální fází celého vývoje aplikace je zveřejnění konkrétní iterace na produkční server a následné spuštění automatických testů pro opětovné otestování základních funkcí aplikace.

7 Testování aplikace s využitím frameworku Appium

Jak již bylo popsáno v kapitole 1.1, hlavní pracovní náplní autora této práce ve firmě Systemart s.r.o. je návrh, implementace a udržování automatických testů pro platformu Android a UWP. Na základě autorových zkušeností získaných v tomto zaměstnání se následující kapitoly pokusí nastínit, jak takovéto automatické testy vypadají, co je jejich součástí a jak využít PageObject přístup při psaní těchto testů.

K rozhodnutí o použití automatických UI testů (testy uživatelského rozhraní) během vývoje aplikace Výkaz práce, dospěla firma Systemart s.r.o. a její testovací oddělení v době, kdy aplikace začala být natolik rozsáhlá, že počet testovacích případů základních funkcí aplikace začal být pro počet testerů, kterými tou dobou firma disponovala neúnosný. Testovací případy pro základní funkce aplikace jsou velice repetitivní, jedná se především o zakládání záznamů, kontrolování součtů zadaných hodnot a UI, což při velkém množství takovýchto úkonů ovlivňuje pozornost testerů. Začalo tak mnohem častěji docházet k situacím, kdy testeři ztráceli po velkém množství testů koncentraci a v aplikaci se poté zvyšoval počet chyb, které se vyskytli i ve finální verzi.

Pro vytvoření automatických testů byl vybrán automatizační framework Appium, který je více popsán v kapitole 5.1, a pro testovaná zařízení byl vybrán emulátor reálných zařízení Android.

7.1 Android emulátor

Během výběru zařízení, na kterém se budou testy vykonávat, je možné volit mezi dvěma variantami, které jsou frameworkem Appium podporovány. První z nich jsou reálná zařízení, tedy mobilní telefony s operačním systémem Android. Hlavní výhodou reálných mobilních telefonů je fakt, že jejich chování je přímo takové, jaké bude u konkrétního modelu uživatele. Tímto chováním je myšlena především správa paměti a módy pro úsporu baterie, které mohou u některých funkcionalit aplikace, především push notifikace na pozadí, mít neblahý vliv na funkčnost. Nevýhodou tohoto řešení je především pořizovací cena, nestačí totiž pořídit pouze jedno zařízení, aby se otestovalo co nejvíce možných konfigurací, je nutné vlastnit několik modelů zařízení od různých

výrobců. Nevýhodou může být také to, že reálné zařízení má svůj omezený výkon a rychlost, která již nejde dále navyšovat, zato u emulovaného zařízení je výkon limitován jen zařízením, na kterém spuštěný emulátor běží.

Další nevýhodou reálných zařízení je absence možnosti použití Appium serveru pro nastavení některých parametrů zařízení, jsou to například Language a Locale (oba tyto parametry ovlivňují nastavení jazyku a regionální nastavení zařízení), dalšími parametry jsou oprávnění udělovaná pro aplikace k přístupu k úložišti, poloze a další. Celkový výčet všech parametrů, které lze takto nastavovat lze nalézt na [34]. Na základě těchto zjištění bylo rozhodnuto o použití emulátoru operačního systému Android.

7.1.1 Capabilities

Parametry, nebo také schopnosti (z anglického Capabilities), které jsou popsány v předchozí kapitole, umožňují nastavit několik dalších vlastností zařízení, a ovlivňují také to, které zařízení Appium server spustí a ke kterému zařízení se připojí. Mohou zde být nastavena také implicitní čekání na výskyt některého elementu na obrazovce, nebo timeout příkazů zasílaných na Appium serveru.

Některé parametry se dají nazvat i povinnými, jsou to především parametry zobrazené v horní části Obr. 8, jedná se o Avd, PlatformName, DeviceName a PlatformVersion slouží pro identifikaci zařízení, které má být spuštěno a ke kterému se má Appium driver připojit.

```
1.  DesiredCapabilities capabilities = new DesiredCapabilities();
2.  capabilities.SetCapability(AndroidMobileCapabilityType.Avd, "AND_8.0.0_1080x1920_5.0in_CS");
3.  capabilities.SetCapability(MobileCapabilityType.PlatformName, "Android");
4.  capabilities.SetCapability(MobileCapabilityType.DeviceName, "AND_8.0.0_1080x1920_5.0in_CS");
5.  capabilities.SetCapability(MobileCapabilityType.PlatformVersion, "8.0.0");
6.  capabilities.SetCapability(MobileCapabilityType.App, "C:\\apk\\cz.vykazprace-Signed.apk");
7.  capabilities.SetCapability(AndroidMobileCapabilityType.AppWaitPackage, "cz.vykazprace");
8.  capabilities.SetCapability(AndroidMobileCapabilityType.AppWaitActivity, "*.ManualLoginActivity");
9.
10. capabilities.SetCapability(MobileCapabilityType.NewCommandTimeout, "60");
11. capabilities.SetCapability("autoGrantPermissions", true);
12. capabilities.SetCapability(MobileCapabilityType.Language, "cs");
13. capabilities.SetCapability(MobileCapabilityType.Locale, "CZ");
```

Obr. 8 Capabilities používané pro jeden z emulátorů

Následující tři parametry App, AppWaitPackage a AppWaitActivity slouží k identifikaci aplikace, kterou má Appium server nainstalovat, a na kterou první aktivitu po spuštění má čekat. V případě testování aplikace Výkaz práce dochází k automatickému

rozbalení instalačního souboru ve formátu .apk ihned po sestavení aplikace. A jeho uložení na server, na kterém se tyto testy vykonávají, podle zvolené cesty. Aplikace má poté pojmenovaný hlavní balíček jako *„cz.vykazprace“*. První aktivitou, která se zde vyskytne a je detekovatelná je *„*.ManualLoginActivity“*.

Parametr *„autoGrantPermissions“* který se nachází na 11. řádku obsluhuje nastavení automatického přijetí všech oprávnění, které může testovaná aplikace požadovat. Ostatní parametry zobrazené na Obr. 8 slouží k nastavení prostředí samotného operačního systému Android, dojde tak k přepnutí jazyku a lokálního nastavení na *„cs_CZ“*. To je skrz Appium nastavováno zvlášť jako dva parametry Language a Locale.

Na Obr. 8 je ukázána pouze jedna z několika, v projektu používaných, konfigurací. Vzhledem k tomu, že cílem je otestovat co největší množství různorodých zařízení, je zde vystavěna vlastní rozsáhlá třída, ve které jsou pomocí několika parametrů přepínány takovéto podobné konfigurace. Momentálně jsou tyto konfigurace sestaveny pro operační systém Android ve verzích 6 až 9, pro dvě v dnešní době nejpoužívanější rozlišení používané na systému Android, a to xhdpi a xxhdpi [35], nakonec je v plánu otestovat aplikace ve všech podporovaných jazycích, kterými jsou čeština, slovenština a angličtina. To dává značný počet konfigurací, přesněji 4 verze operačního systému, 2 rozlišení a 3 jazyky, tedy celkově 24 konfigurací. Tyto konfigurace jsou poté z projektu předávány na Appium server při připojování k němu samotnému, což bude popsáno v následující kapitole.

7.2 Připojení k Appium serveru

Pro připojení k Appium serveru je potřeba pouze jednoho řádku ve zdrojovém kódu (viz Obr. 9). O připojení se stará některý z podporovaných driverů (viz 4.2.2), v tomto případě se jedná o driver pro operační systém Android, je tedy nutné vytvořit instanci třídy AndroidDriver které se jako parametry předává URI adresa serveru, capabilities popsané v předchozí kapitole a čas v sekundách určující dobu maximálního čekání na připojení driveru k Android emulátoru. Tato doba zahrnuje zapnutí emulátoru, jeho nastavení, instalaci a spuštění aplikace, až po detekci první aktivity aplikace. URI (jednotný identifikátor zdroje) adresa serveru je zobrazena přímo při zapnutí GUI rozhraní Appium serveru, avšak ve své localhost formě, pokud je tedy testování prováděné na vzdáleném zařízení, je nutné upravit IP adresu v tomto URI. Zasílání příkazů poté probíhá přímo pomocí instance této třídy.

```
1.  driver = new AndroidDriver<AppiumWebElement>
2.       (
3.            new Uri("http://192.168.5.220:4723/wd/hub"),

4.            caps.GetCapabilities
5.                 (
6.                      "8.0.0",
7.                      "1080x1920",
8.                      "C:\\apk\\cz.vykazprace-Signed.apk"
9.                 ),
10.           TimeSpan.FromSeconds(60)
11.      );
```

Obr. 9 Vytvoření instance Android driveru

Třída AndroidDriver neurčuje konkrétní driver pro ovládání Android emulátoru. Appium nabízí hned několik možností a to nejen pro platformu Android, kompletní seznam všech podporovaných driverů lze nalézt na [36]. To, který driver bude reálně použit lze zvolit prostřednictvím capabilities, přesněji se jedná o parametr *„automationName“*, pokud je tento parametr opomenut, nebo nenastaven tak dojde k automatickému použití přednastaveného UiAutomator driveru i přesto, že Appium již před nedávnou dobou označila tento driver jako deprecated, tím se podpora toho driveru oficiálně stala zastaralou a již dále neudržovanou.

```
if (testing)
{
     Environment.SetEnvironmentVariable
          (
               AppiumServiceConstants.NodeBinaryPath,
               @"C:\nodejs\node.exe"
          );

     Environment.SetEnvironmentVariable
          (
               AppiumServiceConstants.AppiumBinaryPath,
               @"C:\appium-desktop\resources\app\node_modules\appium\build\lib\main.js"
          );

     // Setup appium service
     service = new AppiumServiceBuilder()
         .WithIPAddress("127.0.0.1")
         .WithLogFile(new FileInfo(Path.Combine("C:\\Systemart\\UITests", "log.txt")))
         .UsingPort(4728)
         .Build();

     // Start appium service
     service.Start();

     // Testing
     driver = new AndroidDriver<AppiumWebElement>
          (
               Constants.Service.ServiceUrl,
               caps.GetCapabilities
                    (
                         "7.0.0",
                         "768x1280",
                         "C:\\apk\\cz.vykazprace-Signed.apk"
                    ),
               TimeSpan.FromSeconds(60)
          );
}
else
{
     // Developing
     driver = new AndroidDriver<AppiumWebElement>
          (
               new Uri("http://192.168.5.220:4723/wd/hub"),
               caps.GetCapabilities
                    (
                         "8.0.0",
                         "1080x1920",
                         "C:\\apk\\cz.vykazprace-Signed.apk"
                    ),
               TimeSpan.FromSeconds(60)
          );
}
```

Obr. 10 Vzdálené spuštění Appium service a připojení k ní.

Jelikož v případě práce na projektu Výkaz práce bylo požadováno dosáhnout téměř úplné automatizace, nebyl zmíněný jediný řádek pro konfiguraci dostačující. Pro jednodušší proces vývoje těchto testů a samotné testování vznikla část kódu, kterou lze vidět na Obr. 10. V tomto kódu je vidět využití třídy AppiumServiceBuilder, která se stará o spuštění Appium serveru přímo ze zdrojového kódu. Avšak nejdříve je nutné nastavit cestu k programu Node.js a Appium modulu pro Node.js mezi proměnné prostředí Windows, aby AppiumServiceBuilder věděl, kde tyto součásti nalézt. Poté již

stačí vytvořit instanci této třídy a nastavit jí některé z volitelných parametrů, zde se jedná o IP (internet protokol) adresu Appium serveru, port, na kterém naslouchá a cestu k souboru do kterého se ukládá logování Appium serveru.

7.3 PageObject přístup

I přes to, že se zatím může zdát, že je automatické testování aplikací jednoduché, stále může nastat situace, kdy se celý projekt vymkne kontrole a stane se nepřehledným. Jedná se především o testování aplikací, které tak jako v uvedeném případě vznikají metodikou agilního vývoje. Běžný způsob, jak automatické testy vznikají, spočívá ve psaní každého příkazu pro interakci se zobrazeným elementem zvlášť viz Obr. 11.

```
1.  driver.FindElementById("OpenFormButton").Click();
2.  driver.FindElementById("TextBoxOnForm").SendKeys("Hello, World!");
3.  driver.FindElementById("SaveFormButton");
```

Obr. 11 Standartní scriptování automatických testů

Pokud by bylo v některém z dalších testů potřeba opět vepsat nějaká data do textového pole z Obr. 11, tak by bylo nutné tyto tři příkazy napsat znovu.

V aplikaci, která vzniká agilním vývojem, jenž se neustále rozšiřuje a upravuje funkcionalitu může tento způsob psaní testů brzo způsobit problém. Změna jednoho prvku v aplikaci, na který se spoléhá hned několik testů, bude vést k nutnosti tyto testy také upravit. To přináší velkou ztrátu času a frustraci z neustálé nutnosti přepisovat již vytvořené testy. To je důvod pro užití PageObject přístupu, který každou obrazovku aplikace definuje jako vlastní třídu, kterou je možné využít na několika místech současně a zároveň ji spravovat z jednoho místa.

7.3.1 PageFactory

Třída PageFactory je rozšířením pro PageObject přístup. Předtím, než mohou být použity elementy definované v jednotlivých Page třídách, musí být každá Page třída inicializovaná skrze třídu PageFactory.

PageFactory je možné použít několika způsoby, tím nejjednodušším pro pochopení je využít inicializace požadované Page třídy vždy přímo v samotném testu v místě, kde je třeba využít elementy obsažené v této třídě viz Obr. 12.

```
1.  LoginPage loginPage = PageFactory.intElements(driver, LoginPage);
2.  loginPage.WriteNameToTextBox();
3.  loginPage.WritePasswordToTextBox();
4.  loginPage.ClickOnButton();
5.
6.  WelcomePage welcomePage = PageFactory.intElements(driver, WelcomePage);
7.  welcomePage.ClickToLogout();
8.
9.  loginPage.WriteNameToTextBox();
```

Obr. 12 Jednoduché použití PageFactory

Druhým způsobem, který lze použít, a také tím, pro který se rozhodl vývojářský tým „Výkazu práce“, je inicializace Page třídy pomocí generické metody na vytváření instancí (viz Obr. 13). Tato metoda je výhodnější především v úspoře řádků kódu a v jeho přehlednosti. Při jejím využití může kus kódu zobrazený na Obr. 11 vypadat tak, jako na Obr. 14.

```
1.  public static class OurPageFactory
2.  {
3.      private static T GetPage<T>() where T : new()
4.      {
5.          T page = new T();
6.
7.          PageFactory.InitElements(page, new RetryingElementLocator
8.                                                  (
9.                                                      driver,
10.                                                     TimeSpan.FromSeconds(60)
11.                                                 ));
12.         return page;
13.     }
14.
15.     public static Login Login => GetPage<Login>();
16.     public static SettingPage SettingPage => GetPage<SettingPage>();
17. }
```

Obr. 13 Generická inicializace jednotlivých Page tříd

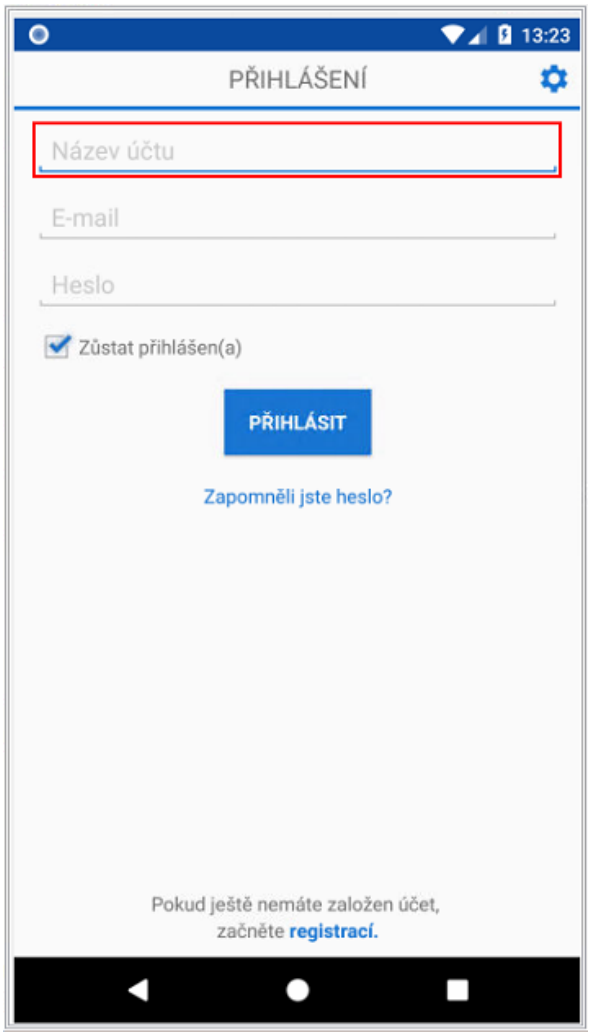
Již při pohledu na Obr. 14 je patrné, že kus kódu je mnohem kompaktnější, přehlednější, a je z něj ihned odlišitelné, ke které Page třídě daný řádek kódu patří.

```
1.  PageFactory.LoginPage.WriteNameToTextBox();
2.  PageFactory.LoginPage.WritePasswordToTextBox();
3.  PageFactory.LoginPage.ClickOnButton();
4.  PageFactory.WelcomePage.ClickToLogout();
5.  PageFactory.LoginPage.WriteNameToTextBox();
```

Obr. 14 Použití PageFactory bez nutnosti inicializace v kódu

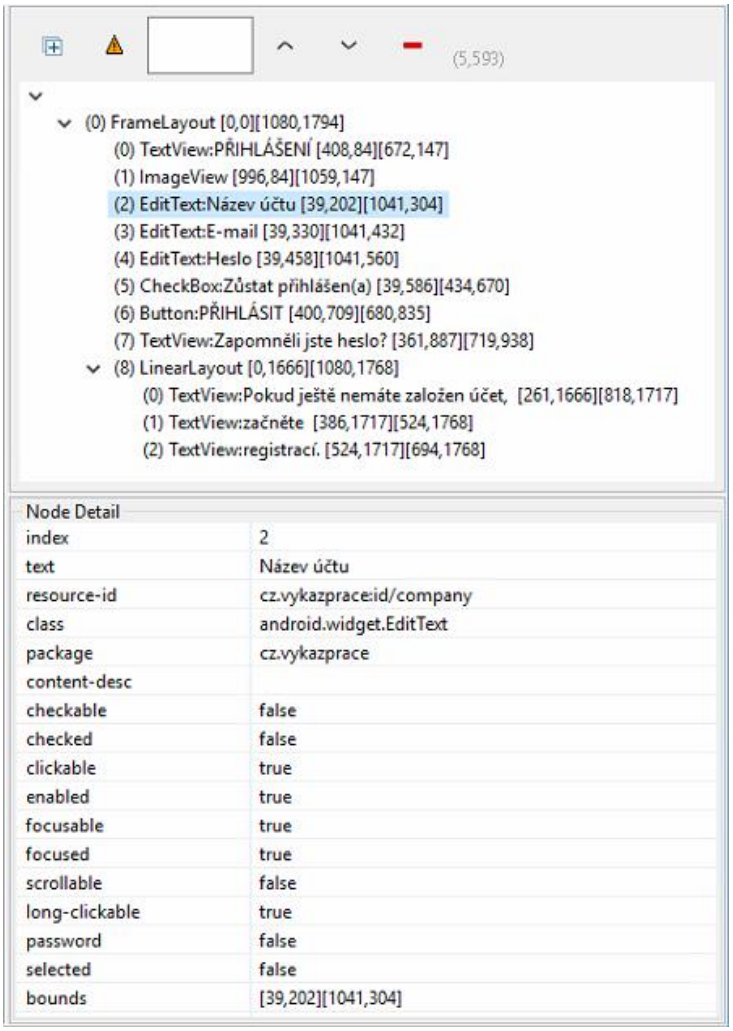
7.3.2 Jednotlivé Page třídy

Způsob, jakým je možné vytvářet jednotlivé Page třídy, je nejlepší představit přímo na konkrétním případu. K tomuto účelu je vybrána přihlašovací obrazovka aplikace Výkaz práce, a to z důvodu dostatečné rozsáhlosti a reprezentativnosti.



Obr. 15 Přihlašovací obrazovka aplikace Výkaz práce

To, jak vypadá přihlašovací obrazovka aplikace Výkaz práce pro Android, je patrné na Obr. 15, její DOM (objektový model dokumentu) struktura je zobrazena na Obr. 16, zároveň je zde vidět, jak vypadají parametry červeně označeného elementu pro vkládání názvu účtu z předchozího snímku. Ve většině případů je důležitý parametr „resource-id“ který je často nejunikátnějším identifikátorem elementů obsažených na obrazovce. Ostatní zobrazené parametry často obsahují stejné či podobné hodnoty, a nedá se podle nich jednoznačně určit element, na který je třeba reagovat.



Obr. 16 DOM struktura přihlašovací obrazovky aplikace Výkaz práce

Pro každou takovouto obrazovku je poté nutné vytvořit vlastní Page třídu v tomto případě třídu LoginPage, ve které bude každý z elementů v DOM reprezentován vlastní proměnnou. U takovéto proměnné je třeba použít anotaci @FindBy, která umožní nalézt daný element na obrazovce. K tomuto účelu ještě existují výčtové typy How, které určují,

dle jaké strategie bude anotace elementy vyhledávat. Anotace je schopna přijímat například strategie Id[5] (v DOM *„resource-id“*), Name (v DOM *„text“*), ClassName (v DOM *„class“*), nebo XML cestu, která umožňuje najít daný element na základě jeho rodičů a potomků.

Aby přístup PageObject mohl fungovat tak, jak byl zamýšlen, je dobré mít všechny proměnné elementů nastavené přístupnost na private, a přístup k nim obsluhovat pouze pomocí metod třídy LoginPage. V těchto metodách je poté třeba interagovat přímo s elementy. V případě, že by došlo ke změně některého z elementů, stačí upravit pouze anotaci, nebo metodu, která s ním pracuje. Poté testovací případ vypadá jako na Obr. 14. Může tak být vytvořená jedinou metodu, která se postará o celé přihlášení, vyplní všechny textové pole a zmáčkne tlačítko pro přihlášení.

[5] Díky využití parametru AppWaitPackage v Capabilities viz kapitola 7.1.1 nemusíme u Id uvádět jeho plnou podobu, ale Appium si vystačí s částí za lomítkem, pokud element spadá do stejného balíčku.

Všechny výše popsané postupy, jak implementovat ilustrační třídu LoginPage je možné vidět na Obr. 17.

```
1.  public class Login
2.  {
3.      [FindsBy(How = How.Id, Using = "company")]
4.      private IWebElement Account { set; get; }
5.
6.      [FindsBy(How = How.Id, Using = "email")]
7.      private IWebElement Email { set; get; }
8.
9.      [FindsBy(How = How.Id, Using = "password")]
10.     private IWebElement Password { set; get; }
11.
12.     [FindsBy(How = How.Id, Using = "connect")]
13.     private IWebElement Connect { set; get; }
14.
15.     [FindsBy(How = How.Id, Using = "settings")]
16.     private IWebElement Settings { set; get; }
17.
18.     public void LogInToApp(string account, string email, string password)
19.     {
20.         Account.SendKeys(account);
21.         Email.SendKeys(email);
22.         Password.SendKeys(password);
23.
24.         if (IsConnectButtonVisible())
25.         {
26.             ClickToConnect();
27.         }
28.     }
29.
30.     public void ClickToSettings()
31.     {
32.         Settings.Click();
33.     }
34.
35.     public void ClickToConnect()
36.     {
37.         Connect.Click();
38.     }
39.
40.     public bool IsConnectButtonVisible()
41.     {
42.         return Connect.Displayed;
43.     }
44. }
```

Obr. 17 Implementace třídy LoginPage

7.4 Implicit a Explicit wait

Framework Appium umožňuje dva hlavní principy čekání. Tato čekání lze využít například při čekání na kompletní načtení obrazovky aplikace, vyčkání, než určitý element stránky nabyde požadované hodnoty, nebo jakoukoliv další změnu na obrazovce aplikace, kterou je třeba zachytit. Je to důležité především z důvodu, že pokud není užito žádné z těchto čekání, tak může často docházet k situacím, které Appium vyhodnotí jako

chybový stav, a ukončí tak průběh testů s chybou. Taková chyba nastává v momentě, kdy je driver požádán o nalezení některého elementu, který ještě nebyl plně načten, na obrazovce se nenachází, nebo již byl z nějakého důvodu změněn. Tyto situace mohou být někdy dokonce vyžadovány kvůli zjištění informace o tom, jestli již byl některý element změněn. Z toho důvodu výběr principu (nebo jejich kombinace) záleží na jednotlivých preferencích.

```
1.  driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(30);
```

Obr. 18 Nastavení implicitního čekání

Implicitní čekání se nastavuje přímo pro instanci driveru, popsaného v kapitole 7.2, pomocí příkazu zobrazeného na Obr. 18. Tento příkaz lze využít a přenastavit jím čekání před místem, kde je očekáváno jeho využití. Pokud tento příkaz není využit v žádné části testů, tak doba čekání zůstane nastavena na 0 milisekund. V podstatě tedy čekání nebude aktivní, a při jakémkoliv zpoždění načtení obrazovky aplikace dojde k chybovému stavu, který byl popsán výše.

Fungování implicitního čekání spočívá v opakovaném procházení DOM struktury testované aplikace, dokud nedojde k vypršení časovače nebo nalezení požadovaného elementu obrazovky. Obecně se ale v komunitě uživatelů a vývojářů frameworku Appium nedoporučuje používání implicitního čekání. Využití implicitního čekání je doporučeno využívat spíše jako pojistky pro místa, kde nejsou očekávány žádné potíže. V tomto případě je vhodné nastavit hodnotu implicitního čekání ihned po vytvoření instance driveru. Dále lze v celém projektu použít pouze explicitní čekání.

```
1.  WebDriverWait wait = new WebDriverWait
2.                               (
3.                                    driver,
4.                                    TimeSpan.FromSeconds(30)
5.                               );
6.
7.  wait.Until(ExpectedConditions.ElementIsVisible(By.Id("Button1")));
```

Obr. 19 Použití explicitního čekání

Explicitní čekání lze využít tím způsobem, že se vytvoří instance třídy WebDriverWait, které se při inicializaci předá instance námi používaného driveru a určí se doba maximálního čekání. Následně je možné čekání využít na požadovaných místech

pomocí příkazu na Obr. 19. Princip tohoto čekání je lehce odlišný od toho, jakým způsobem funguje implicitní čekání. Explicitní čekání funguje tím způsobem, že instance třídy WebDriverWait volá každých 500 milisekund příkaz ExpectedConditions. Toto se opakuje až do doby, kdy je příkaz na vyhledání některého elementu na obrazovce úspěšně dokončen, nebo dokud není dosaženo maximálního čekání. [37]

Nevýhodou explicitního čekání je fakt, že pro jeho použití je nutné využít framework Selenium, který slouží pro automatizované testování webových stránek, přesněji stačí použít jen jeho část – Selenium support, tuto utilitu je nutné použít i u PageObject přístupu, jak již bylo popsáno v předchozí kapitole.

7.5 Mock location

Na emulátoru operačního systému Android je možné pomocí frameworku Appium nastavit také simulovanou GPS polohu. Tato funkce je nazývána mock location v češtině falešná poloha. Správného lokalizování polohy by mělo být testováno alespoň na jednom místě v každé aplikaci, která tuto možnost nabízí. To, jak přesně lokalizační senzory v mobilních zařízeních fungují, podrobně popsal například Jerry Gao a spol. [23] ve své knize věnované automatizovanému testování mobilních aplikací.

V případě aplikace Výkaz práce se funkce hodí především pro otestování docházky, kde poloha slouží pro určení, zda se pracovník právě nachází na předem definovaném pracovišti či nikoliv. Dále je tato funkce uplatněna při testování knihy jízd, kde jak už název agendy napovídá, slouží poloha k určení počáteční a koncové lokality jízdy, případně pro její průjezdní body.

```
1.  driver.Location.Altitude  = 94.23;
2.  driver.Location.Latitude  = 12.21;
3.  driver.Location.Longitude = 11.56;
```

Obr. 20 Nastavení falešné lokality

Dle dokumentace Appium [38] probíhá nastavení pouze pomocí příkazů na Obr. 20, tyto příkazy nastaví zeměpisnou šířku a délku doplněnou o nadmořskou výšku. Pro určení polohy ovšem stačí zadat pouze zeměpisnou šířku a délku, po nastavení těchto dvou hodnot bude Android emulátor ihned zobrazovat svou lokalitu právě dle zvolených bodů. Kontrola může být provedena například s využitím aplikace Google Maps, ve které je možné zobrazit vlastní polohu.

Dostupné informace však nevyhovovaly všem požadavkům aplikace. „Výkaz práce“ uživatelům zobrazuje pouze „lidsky“ čitelnou adresu, pokud je tedy v zadaném místě adresa dostupná. Za využitím těchto adres aplikace pomocí Google Maps API určí trasu mezi počátečním a koncovým bodem, na základě čehož aplikace zjistí ujetou vzdálenost.

```
1.  internal class Location
2.  {
3.      public double latitude { get; set; }
4.      public double longitude { get; set; }
5.      public string city { get; set; }
6.
7.      public Location(double latitude, double longitude, string city)
8.      {
9.          this.latitude = latitude;
10.         this.longitude = longitude;
11.         this.city = city;
12.     }
13.
14.     public void SetLocation()
15.     {
16.         driver.Location.Latitude = latitude;
17.         driver.Location.Longitude = longitude;
18.     }
19. }
```

Obr. 21 Třída obstarávající nastavení a uložení nastavené lokality

Z tohoto důvodu bylo rozhodnuto vytvořit vlastní třídu Location viz Obr. 21, která uchovává poslední nastavenou lokalitu a přidává k ní i proměnou, která obsahuje město, pro které je tato lokalita určena, při testování jsou poté nastavovány lokality sousedících měst, čímž se testuje výpočet vzdálenosti, ale především návaznost jízd, pokud obsahují průjezdní body.

7.6 Touch actions

Framework Appium má také implementované funkce pro práci s dotykovým displejem, jako je například dlouhé podržení, potažení do některé ze stran, nebo přitažení a odtažení (Pinch-to-Zoom).

```
1.  public class TouchAction
2.  {
3.      private OpenQA.Selenium.Appium.MultiTouch.TouchAction action { get; }
4.
5.      public TouchAction(AppiumDriver<AndroidElement> driver)
6.      {
7.          action = new OpenQA.Selenium.Appium.MultiTouch.TouchAction(driver);
8.      }
9.
10.     public void Swipe(int startX, int startY, int endX, int endY)
11.     {
12.         action.Press(startX, startY).MoveTo(endX, endY).Release().Perform();
13.     }
14.
15.     public void LongPress(AndroidElement element, int endX, int endY)
16.     {
17.         action.LongPress(element, endX, endY).Perform();
18.     }
19. }
```

Obr. 22 Použití dotykového ovládání

Výčet všech dostupných funkcí lze nalézt v dokumentaci [39]. V této dokumentaci jsou popsány i funkce, které nejsou implementovány ve verzi Appium pro .NET, lze je však nahradit jinými funkcemi, případně jejich řetězením. Z tohoto důvodu byla stejně jako u simulace lokality vytvořena speciální třída. Ta obsahuje předdefinované metody, ve které jich je zřetězených hned několik tak, aby bylo dosaženo potřebných funkcí (viz metoda Swipe na Obr. 22).

7.7 Testy

Testovací případy jsou rozděleny do tříd dle agend, číselníků a oblastí do kterých spadají podle svého výskytu v aplikaci Výkaz práce. V těchto třídách se nachází metody určené pro řízení konkrétních testovacích případů. Názornou ukázkou je zde agenda Rozpis práce, v této agendě je nutné testovat manuální zakládání záznamů, potvrzování generovaných záznamů, kontrolovat návaznost různých typů záznamů a výpočet souhrnů pod záznamy. Všechny tyto testovací případy jsou obsaženy ve třídě TimeTrackingTests.

Zmíněná třída tak obsahuje další metody, kde každá reprezentuje jeden testovací případ. Tyto testovací metody vykonávají sekvenci kroků, která vede k úspěšnému otestování veškerých funkcionalit. Vyplňování všech formulářů je ověřeno ihned po jejich odeslání a následně i ve formě řádku v přehledu všech záznamů. Díky tomuto dvojitému ověření je minimalizována možnost opomenutí některé chyby.

V SetUp části testovacích případů vždy probíhá zapnutí aplikace, následné přihlášení a navigace do místa, kde má konkrétní testovací případ probíhat. Další část – TestCase obsahuje metody testovacích případů popsaných výše, které vedou k úspěšnému splnění testovacích případů. Nakonec ve finální části TearDown probíhá vypnutí aplikace. V případě, že testovací příklad končí chybou, je vyvolána metoda obstarávající vytvoření a uložení snímku obrazovky pro případ pozdějšího nalezení místa chyby.

7.7.1 FormData třídy

Ke každé testovací třídě náleží třída FormData, ve které jsou definována data, která jsou vkládána do aplikace pro účel testování. Jednotlivé atributy vždy odpovídají datovým typům polí, do kterých jsou hodnoty obsažené v těchto atributech vkládány. Data jsou poté předána testovacím metodám pomocí atributu TestCaseSource. Tato metoda oddělení testovacích dat přispívá k větší přehlednosti a další znovupoužitelnosti.

Jako nejlepší ukázka třídy FormData může posloužit třída AttendanceFormData, která se stará o uložení dat pro testování agendy Docházka viz Obr. 23. V této ukázce je patrné využití další FormData třídy, která se stará o uložení dat pracovních aktivit použitých v aplikaci, ale také použití speciálně vytvořené pomocné třídy Time a pomocné třídy Location, která již byla představena v kapitole 7.5.

```
internal class AttendanceFormData
{
    public class Activities
    {
        public static ActivitiesFormData.FormData Work =
             ActivitiesFormData.FormData.ActivitiesGlobalTestSets.ActivityWork;
        public static ActivitiesFormData.FormData Absence3 =
             ActivitiesFormData.FormData.ActivitiesGlobalTestSets.ActivityAbsence3;
    }

    public class FormData : FormDataBase
    {
        public enum AttendanceTestSet { New, Edit }

        public bool EnterMultipleDays { get; set; }
        public DateTime Date { get; set; }
        public ActivitiesFormData.FormData Activity { get; set; }
        public bool? EnterNumberOfHours { get; set; }
        public Time Time { get; set; }
        public Location Location { get; set; }
        public string Description { get; set; }

        public FormData(AttendanceTestSet attendanceTestSet)
        {
            switch (attendanceTestSet)
            {
                case AttendanceTestSet.New:
                    EnterMultipleDays = true;

                    Date = DateTime.Now;
                    Activity = Activities.Work;
                    EnterNumberOfHours = Activities.Work.PreferToEnterHours;
                    Time = new Time("17:00");
                    Location = new Location(50.077215, 14.436779, "Vinohrady");
                    Description = "Testing description - New";
                    break;
                case AttendanceTestSet.Edit:
                    EnterMultipleDays = false;
                    Date = DateTime.Now;
                    Activity = Activities.Absence3;
                    EnterNumberOfHours = Activities.Absence3.PreferToEnterHours;
                    Time = new Time("15:30");
                    Location = new Location(50.207786, 15.832510, "Hradec Králové");
                    Description = "Testing description - Edit";
                    break;
            }
        }

        public class AttendanceTestSets
        {
            public static FormData New => new FormData(AttendanceTestSet.New);
            public static FormData Edit => new FormData(AttendanceTestSet.Edit);
        }
    }
}
```

Obr. 23 Ukázka uložených dat ve třídě AttendanceFormData

7.7.2 Pomocné třídy

V celém projektu existuje několik pomocných tříd, které usnadňují porozumění kódu a omezují redundanci. Mezi tyto třídy patří například již zmíněné třídy Location, TouchAction nebo FormData. Jdou zde však také implementované třídy, které se starají o výpočty nebo ověření některých vkládaných dat, jedná se například o výpočet ujeté vzdálenosti v Knize jízd, výpočet odpracované doby nebo převody jednotek využívané mezi různými jazykovými mutacemi aplikace.

Většina těchto tříd již nemá příliš společného se samotným frameworkem Appium, ale využíváme zde obecné principy programování v jazyce C#. Domnívám se tedy, že zde není nutná ukázka těchto tříd, jelikož jejich rozsah, použití i metody budou vždy vytvořeny přesně na míru aplikace, kterou dotyčný čtenář testoval.

7.7.3 Vyhodnocení automatizovaných testů

Na trhu existuje řada nástrojů, která se dá pro automatické testování použít. V případě práce na projektu Výkaz práce bylo využito rozšíření NUnit do vývojového prostředí Visual Studio. Toto rozšíření v okně Text Explorer přináší přehled o průběhu jednotlivých testů a jaký je jejich status (například v případě chyby se zobrazí její popis a výskyt).

Výsledky testů jsou rozděleny do několika skupin. Níže jsou zmíněné tři z nich.

- **Passed** – Testy proběhly v pořádku.
- **Failed** – Testy obsahují chybu. Speciálně vytvořená metoda se postará o vytvoření a uložení snímku obrazovky, pro případ nutnosti ověřit v jakém stavu se aplikace během chyby nacházela.
- **Skipped** – Testy, které byly přeskočeny. Testy na přeskočení jsou takto většinou označeny z důvodu nefunkčního testu, který čeká na opravu.

8 Závěr

Cílem této práce bylo představit doménu testování softwaru a její automatizace, zaměřit se na popis vybraných nástrojů pro automatizované testování mobilních aplikací spolu se srovnáním některých jejich vlastností a následné popsání implementace funkcí frameworku Appium nad konkrétní aplikací Výkaz práce.

V samotném úvodu této práce je nastíněno stále se zvyšující tempo oblíbenosti mobilních aplikací jako takových i přes velkou míru okamžité odinstalace. Tento fakt tlačí vývojáře ke stále většímu důrazu na kvalitu aplikací a tudíž i k nutnosti zavést testování softwaru během vývoje. To je jeden z důvodů, proč oblast testování softwaru v oboru informačních technologií stále nabírá na své důležitosti. Dále je zde věnováno několik podkapitol vymezení záměru této práce.

Druhá kapitola je věnována vysvětlení samotného pojmu testování softwaru. Tato kapitola nastiňuje význam testování a ukazuje co nejčastěji vede k výskytu chyb v průběhu celého vývoje aplikace, jak mohou tyto chyby celý vývoj softwaru prodražit pokud nejsou včas odstraněny. Následně je zde popsáno zasazení testovacího procesu do celého vývoje a představeno několik modelů jeho životního cyklu, na kterých je ukázáno, kdy dochází k testování v konkrétním modelu.

Další část textu pojednává o automatizaci testování jako důležité oblasti testování softwaru, její výhody a nevýhody. Dále jsou zde popsány typy nástrojů pro automatizované testování a jejich možné dělení dle složitosti použití pro testera. Následující kapitola se zabývá popisem konkrétních nástrojů pro automatizované testování. Popisované nástroje spadají do kategorie plně programovatelných nástrojů pro automatizované testování. Na konci kapitoly jsou popsané vlastnosti souhrnně sepsány do tabulky k porovnání. Z tohoto porovnání je patrné, že nejvariabilnějšími nástroji pro automatizované testování jsou frameworky Calabash a Appium.

Poslední část této práce je věnována použití frameworku Appium s použitím objektového přístupu PageObject nad reálnou aplikací Výkaz práce. Z důvodu nízkého počtu testerů a vysoké komplexnosti celé aplikace se společnost Systemart s.r.o. již delší

dobu pokoušela zefektivnit proces testování. To i z důvodu, že oba z testerů zaměstnaných v této společnosti obstarávají i jiné, pro tuto společnost důležité činnosti, a tak nezbývá dostatek času na kvalitní otestování obecných funkcionalit aplikace.

Před zavedením automatických testů bylo s každou verzí aplikace nutné otestovat přibližně 200 obecných testů, ověřujících obecné zakládání záznamů, správné zobrazení GUI, synchronizaci a oprávnění. Nyní po zavedení automatických testů pro webovou aplikaci [10] a operační systém android se počet obecných testů snížil na číslo pohybující se kolem 90 testů. To odpovídá eliminování 55 % obecných testů. V rámci této práce došlo k eliminování takřka 40 obecných testů z původních 200. Zásluhu na tom má i T. Burda, [10] který obstaral v rámci své bakalářské práce automatické testování webové aplikace, které eliminovalo zbylých 70 obecných testů z celkově 110 eliminovaných v rámci celého projektu automatizovaného testování aplikace Výkaz práce. Avšak tyto testy jsou pouze přibližné, jelikož počty obecných testů se s každou verzí mohou lišit až s rozdílem 10 testů.

V průběhu testování aplikace Výkaz práce zde stále zůstává již zmiňovaných 90 obecných testů. Tyto testy se z velké části zaměřují na funkce, které nyní možné otestovat na jednotlivých agendách, ale týkají se společné funkčnosti, jako jsou části synchronizace, nebo oprávnění. Zbytek testů, které do této kategorie nespadá, jsou obecné testy jedinečné pro ostatní platformy, které zatím nebyly automatickým testováním pokryty. To vytváří potenciální prostor pro další vylepšení.

Jedno z potenciálních vylepšení této práce by mohlo spočívat v propojení automatických testů pro operační systém android a webovou aplikaci. Díky vzájemné komunikaci by tak mohlo docházet k synchronnímu zadávání záznamů, jejich synchronizaci napříč zařízeními a následné verifikaci. Případně by poté bylo možné otestovat také oprávnění, jelikož změna některého oprávnění ve webové aplikaci může naprosto změnit hlavní funkce ve všech klientských aplikacích. Avšak propojení těchto dvou velice rozdílných projektů bude složité a také závislé na zbytku vývojářského týmu aplikace Výkaz práce. Propojit tyto dva projekty by bylo možné například díky vznikajícímu API.

Další rozšíření této práce by se také mohlo zaměřit na eliminování zbylých obecných testů, které jsou specifické pro některé z dalších klientských aplikací Výkazu práce. Toto rozšíření práce by spočívalo ve vytvoření dalších projektů, které by byly zaměřeny na automatizované testování iOS, nebo UWP aplikace.

I přes předchozí vyjmenované úspěchy, kterých automatické testování dosáhlo je zde situace, která vedla k tomu, že je projekt nyní přepracováván. Přepracování probíhá z důvodu, aby projekt odpovídal tomu, jak bude aplikace Výkaz práce vypadat ve své nové verzi 7.6.x, jelikož v této verzi došlo k redesignu a změně téměř všech prvků GUI.

Do budoucna bude projekt dále udržován a směrován k úplné automatizaci a propojení s dalšími projekty automatického testování, jak bylo popsáno výše. Při celkovém zhodnocení lze s jistotou říct, že se projekt automatického testování aplikace Výkaz práce vyplatil, i přes nutnost nynějšího přepracování. Oddělení testerů má nyní více času věnovat se svým dalším pracovním povinnostem ve společnosti Systemart s.r.o. a také důkladnějšímu testování zbylých obecných testů a testů nových funkcí. Avšak ve spojitosti s touto prací a projektem je důležité si uvědomit, že zvolené postupy a nástroje byly vybrány pro konkrétní společnost a aplikaci. V případě jiné společnosti, nebo aplikace, může dojít k výběru jiných, vhodnějších nástrojů a přístupů.

9 Seznam použité literatury

[1] *Number of mobile app downloads worldwide in 2017, 2018 and 2022 (in billions)* [online]. New York: Statista, 2018 [cit. 2019-01-16]. Dostupné z: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/

[2] **PRAMIS, Joshua**. *Are you a rarity? Only 16 percent of people will try out an app more than twice*, 2013 [online]. [cit. 2019-01-16]. Dostupné z: http://www.digitaltrends.com/mobile/16-percent-ofmobile-userstry-out-a-buggy-app-more-than-twice/

[3] *App Store Review Guidelines* [online]. Apple, 2018 [cit. 2019-01-16]. Dostupné z: https://developer.apple.com/app-store/review/guidelines/

[4] Non-disclosure agreement. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-01-16]. Dostupné z: https://cs.wikipedia.org/wiki/Non-disclosure_agreement

[5] *Google Scholar* [online]. California: Google, 2004 [cit. 2019-01-16]. Dostupné z: https://scholar.google.cz/

[6] *Studijní a vědecká knihovna v Hradci Králové* [online]. Hradec Králové: Studijní a vědecká knihovna v Hradci Králové, 2009 [cit. 2019-01-16]. Dostupné z: https://primo.svkhk.cz/

[7] **PATTON, Ron**. *Testování softwaru*. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.

[8] **ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ**. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.

[9] **KNOTT, Daniel**. *Hands-on mobile app testing: a guide for mobile testers and anyone involved in the mobile app business.* New York: Addison-Wesley, [2015]. ISBN 978-0134191713.

[10] **BURDA, Tomáš**. Testování softwaru [online]. Hradec Králové, 2017 [cit. 2019-03-17]. Dostupné z: https://theses.cz/id/dsbtq2/. Bakalářská práce. Univerzita Hradec Králové, Fakulta informatiky a managementu. Vedoucí práce doc. RNDr. Petra Poulová, Ph.D..

[11] **MAČURA, Viktor**. Testování mobilních aplikací [online]. Praha, 2016 [cit. 2019-03-17]. Dostupné z: https://theses.cz/id/hjlhhx/. Bakalářská práce. Vysoká škola ekonomická v Praze. Vedoucí práce Jan Ženíšek.

[12] **VESELOVSKÝ, Eduard**. Přehled nástrojů pro automatické testování aplikací [online]. Plzeň, 2014 [cit. 2019-03-17]. Dostupné z:

https://theses.cz/id/b3myrv/. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí práce Ing. Richard Lipka, Ph.D..

[13] ISTQB EXAM CERTIFICATION, *What is Software Testing?* [online]. [cit. 2019-01-16] Dostupné z: http://istqbexamcertification.com/what-is-a-software-testing/

[14] *Capgemini: The cost of repairing defects according to the phase in which they are found* [online]. Paris: Capgemini, 2010 [cit. 2019-01-16]. Dostupné z: https://www.capgemini.com/

[15] **BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ**. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

[16] **HLAVA, Tomáš**. Vodopádový model. *Testování softwaru* [online]. [cit. 2019-01-16]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/vodopadovy-model/

[17] **HLAVA, Tomáš**. Spirálový model. *Testování softwaru* [online]. [cit. 2019-01-16]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/spiralovy-model/

[18] **HLAVA, Tomáš**. Progresní a regresní testy. *Testování softwaru* [online]. 20.8.2011 [cit. 2019-01-16]. Dostupné z: http://testovanisoftwaru.cz/tag/regresni-testy/

[19] **HAVLÍČKOVÁ, Anna**. *Testování Softwaru*. [Online] 5. srpen 2008. [cit.: 2019-01-16] http://testovanisoftwaru.blogspot.cz/p/dptestovanisoftwarupdf.html

[20] **ČECHÁKOVÁ, Lucie**. *Automatizace regresního testování*. Praha, 2016. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Alena Buchalcevová.

[21] **MALANÍK, Martin**. *Testování aplikací pro mobilní telefony*. Zlín, 2011. Diplomová práce. Univerzita Tomáše Bati ve Zlíně. Vedoucí práce František Gazdoš.

[22] **MATEEN, Abdul**. *ANDROID AUTOMATION: OPEN SOURCE FRAMEWORKS* [online]. [cit. 2019-03-12]. Dostupné z: https://www.globallogic.com/blogs/android-automation-open-source-frameworks/

[23] **GAO Jerry, Xiaoying BAI, Wei-Tek TSAI a Tadahiro UEHARA**. *Mobile Application Testing: A Tutorial.* Computer [online]. 2014, 47(2), 46-55 [cit. 2018-06-17]. DOI: 10.1109/MC.2013.445. ISSN 0018-9162. Dostupné z: http://ieeexplore.ieee.org/document/6693676/

[24] *Test Release Note Archive* [online]. Google [cit. 2019-03-12]. Dostupné z: https://developer.android.com/jetpack/androidx/releases/archive/test#rel notes-20140108

[25] *Espresso* [online]. Google [cit. 2019-03-12]. Dostupné z: https://developer.android.com/training/testing/espresso

[26] *UIAutomator* [online]. Google [cit. 2019-03-12]. Dostupné z: https://developer.android.com/training/testing/ui-automator

[27] *UiObject* [online]. Google [cit. 2019-03-12]. Dostupné z: https://developer.android.com/reference/androidx/test/uiautomator/UiObj ect.html#click()

[28] *Calabash* [online]. Xamarin [cit. 2019-03-12]. Dostupné z: https://calaba.sh/

[29] Cucumber (software). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-03-12]. Dostupné z: https://en.wikipedia.org/wiki/Cucumber_(software)

[30] Top 5 Android Testing Frameworks With Code Examples. *BitBar*. [online]. [cit. 2019-03-12]. Dostupné z: https://bitbar.com/top-5-android-testing-frameworks-with-examples/

[31] Top 5 Open Source Automation Tools for iOS and Android. *AFourTechnologies* [online]. [cit. 2019-03-12]. Dostupné z: https://afourtech.com/automation-tools-for-ios-and-android-apps/

[32] Robotium. *GitHub* [online]. [cit. 2019-03-12]. Dostupné z: https://github.com/RobotiumTech/robotium

[33] Introduction to Appium. *Appium.io* [online]. [cit. 2019-03-12]. Dostupné z: http://appium.io/docs/en/about-appium/intro/

[34] Appium Desired Capabilities. *Appium.io* [online]. [cit. 2019-01-16]. Dostupné z: http://appium.io/docs/en/writing-running-appium/caps/

[35] Distribution dashboard. Developers Android [online]. [cit. 2019-01-16]. Dostupné z: https://developer.android.com/about/dashboards/

[36] Appium Platform Support. *Appium.io* [online]. [cit. 2019-01-16]. Dostupné z: http://appium.io/docs/en/about-appium/platform-support/

[37] WebDriver Implicit, Explicit and Fluent Wait Examples. *Testing Excellence* [online]. [cit. 2019-01-16]. Dostupné z: https://www.testingexcellence.com/webdriver-explicit-implicit-fluent-wait/

[38] Set Geolocation. *Appium.io* [online]. [cit. 2019-01-16]. Dostupné z: http://appium.io/docs/en/commands/session/geolocation/set-geolocation/

[39] Touch. *Appium.io* [online]. [cit. 2019-01-16]. Dostupné z: http://appium.io/docs/en/commands/interactions/touch/

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Tomáš Novák**
Studium:	I1500402
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Automatické testování mobilních aplikací**
Název bakalářské práce AJ:	Automatic testing of mobile applications

Cíl, metody, literatura, předpoklady:

Cíl práce: Cílem práce je představit doménu testování softwaru, popsat a porovnat nástroje pro automatizované testování mobilních aplikací a následně nastínit postup řešení na konkrétním projektu pomocí frameworku Appium s ukázkou odlišných přístupů použití. Struktura práce: 1) Úvod 2) Literární rešerše 3) Obecný popis problematiky. 4) Popis a porovnání nástrojů 5) Příklady testování za pomocí frameworku Appium. 6) Výsledky a závěr

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5. KNOTT, Daniel. Hands-on mobile app testing: a guide for mobile testers and anyone involved in the mobile app business. New York: Addison-Wesley, [2015]. ISBN 978-0134191713. GAO, Jerry, Xiaoying BAI, Wei-Tek TSAI a Tadahiro UEHARA. Mobile Application Testing: A Tutorial. Computer [online]. 2014, 47(2), 46-55 [cit. 2018-06-17]. DOI: 10.1109/MC.2013.445. ISSN 0018-9162. Dostupné z: http://ieeexplore.ieee.org/document/6693676/ ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 9788025138168.

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.1.2015