TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikita Putyatin 194150IADB

React Native mobiililrakenduse automaattestimine Zipstall'i näitel - analüüs ning juurutamine

Bakalaureusetöö

Juhendaja: Maili Markvardt
MSc

Tallinn 2022

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Nikita Putyatin

16.05.2022

Annotatsioon

Üks ilmsemaid viise rakenduste arendamise kiirendamiseks ja sujuvamaks muutmiseks on automatiseeritud testide loomine. Käesoleva bakalaureusetöö eesmärk on luua React Native'is loodud mobiilirakendusele Zipstall automaattestid ning hinnata nende toimivust idufirma jaoks.

Testide loomiseks analüüsiti kõiki mobiilirakenduste testimise tüüpe ning valiti välja kaks parimat testimisraamistikku. Uuriti ka mobiilitestimise põhitõdesid ning loodi ainulaadne testimispüramiid.

Praktilises osas koostati iga testimisliigi jaoks mitu testi. Samuti kirjeldati üksikasjalikult, milleks iga test mõeldud on ja kuidas see mobiilirakenduse arendamisel abiks on.

Kokkuvõttes viidi läbi kogu lõputöö praktilise osa analüüs. Esimeseks järelduseks on, et automaattestide koostamise eelduseks peab testitava rakenduse koodi testitavus olema piisaval tasemel. Põhiküsimusele anti vastus - testid osutusid väga tõhusaks ka väikesele ettevõttele. Töö tulemusena muutis saadud lahendus kõigi meeskonnaliikmete töö mugavamaks ja lihtsamaks ning automatiseeritud testid muutsid arenduse stabiilsemaks.

Lõputöö on kirjutatud eesti keeles keeles ning sisaldab teksti 22 leheküljel, 5 peatükki, 13 joonist.

Abstract

Automated Testing for React Native Mobile Application on the Example of Zipstall - Analysis and Implementation

One of the most obvious ways to speed up and streamline application development is to create automated tests. In this thesis, all development was done for the Zipstall mobile application, which uses React Native. The aim of this bachelor's thesis is to create tests for a mobile application and evaluate their performance for a start-up.

To create tests, all types of mobile application testing were analysed, and the two best testing frameworks were selected. The basics of mobile testing were also investigated, and a unique testing pyramid was created.

In the practical part, several tests were created for each type of testing. Also, was described in detail what each test was designed for and how it helps in the development of a mobile application.

In conclusion, analysis of the entire practical part of the thesis was carried out. As a result, the main conclusions were obtained - need to initially write testable code. The answer to the main question was given - the tests turned out to be very effective even for a small company. As a result of the work, the resulting solution made the work of all team members more comfortable and easier, and automated tests made development more stable.

The thesis is in Estonian and contains 22 pages of text, 5 chapters, 13 figures.

Lühendite ja mõistete sõnastik

API	*Rakendusliides, komplekt rakendustarkvara ehitamiseks (ingl Application Program Interface*
Appium	*On avatud lähtekoodiga automatiseerimise mobiilitestimise vahend*
End-to-end testimine	*Automaattestimise tehnika, mille puhul kontrollitakse rakenduse tervet teatud töövoogu*
Gray box testiminet	*Halli kasti testimine on tarkvara debugimise strateegia, kus testijal on testitava rakenduse kohta piiratud teadmine*
Mock	*Mitte päris, vaid näib või teeskleb, et on täpselt millegi moodi*
Onboarding	*Õpetus uutele Zipstall'i kasutajatele*
React Native	*on JavaScript raamistik natiivselt mobiilirakenduste arendamiseks (iOS-i ja Androidi jaoks).*
Snapshot	*Killuke infot*
UI	*Kasutajaliides (ingl User Interface)*

Sisukord

1 Sissejuhatus 8
2 Tehnoloogiate ülevaade 9
2.1 Testimise tüübid 9
2.1.1 Ühiktestimine 9
2.1.2 Integratsioonitestimine 9
2.1.3 Komponendi testimine 10
2.1.4 End-to-End testimine 11
2.2 Mobiilitestimise eripärad 11
2.3 Automaattestimise reeglid 13
2.4 Testimisvahendite tutvustus 14
2.4.1 Jest 14
2.4.2 Detox 15
3 Realisatsioon 16
3.1 Süsteemi ülevaade 16
3.2 Näidistestid 16
3.2.1 Snapshot näidistest 16
3.2.2 Ühiktestimine näidistest 18
3.2.3 Integratsioonitestimine näidistest 21
3.2.4 End-to-End näidistest 22
3.3 Testide kasutamise juhendi koostamine 24
4 Testide analüüs 25
4.1 Koodi testitavus 25
4.2 Automaattestimisest saadud tulemused 27
4.3 Edasised tegevused 28
5 Kokkuvõte 29
Kasutatud kirjandus 30
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 31

Jooniste loetelu

Joonis 1 Autori loodud automaattestimine püramiid. 13
Joonis 2 Snapshot testi viga. 17
Joonis 3 Ekraan enne ja pärast muudatusi. 18
Joonis 4 Funktsioon, mis loeb parkimise alguse ja lõpu vahelist aega. 19
Joonis 5 Ühikutestid. 20
Joonis 6 Ühikutest mis peaks vea viskama. 20
Joonis 7 Erinevus oodatud ja tulemuse vahel. 20
Joonis 8 Integratsioonitestimine näidistest. 21
Joonis 9 Üksuse- ja integratsioonitestide edukas läbimine. 22
Joonis 10 E2E näidistest, kus kasutaja lisab uue sõiduki. 23
Joonis 11 Testide kasutamise juhend, vasakul lähtekood, paremal juhend ise. 24
Joonis 12 Funktsioon, mida pole võimalik testida. 26
Joonis 13 Funktsioon lisatud argumendiga. 27

1 Sissejuhatus

Zipstall on mobiilirakendus, mis on välja töötatud React Native raamistikus. Pool aastat Zipstalli mobiilirakenduse kallal idufirmas töötades selgus, et kolleegid üldse ei kasuta automaattestimist. Rakenduse arendusprotsessis on probleem, et pidevalt rakendusse uusi võimalusi lisades varem või hiljem rakenduse vana osa läks katki ja viga on õigel ajal raske leidal. Pealegi läks rakendus rikki peamiselt vaid teatud kohtades, näiteks animatsioonis ja navigatsioonis. Selle probleemi lahendamiseks otsustasin luua mobiilirakenduse jaoks testid, mille loomist ja analüüsi on käesolevas lõputöös kirjeldatud.

Testid lahendavad veel ühe olulise probleemi, millest vähesed teavad. Autor avas selle rakenduse kallal tööle asudes ja selgus, et algusest peale on rakenduse äriloogikast raske aru saada. Kuid seda probleemi saab lahendada automattestidega, kuna need näitavad, kuidas rakendus peaks töötama. Seega saab uus töötaja loogikast paremini aru ega karda, et tema muudatused lõhuvad midagi muud, mis on töö algfaasis oluline.

Käesoleva töö eesmärk on luua automaatteste ning analüüsida nende efektiivsust idufirma jaoks.

Selleks, et koostada mõistlikult hallatavad ja kvaliteetsed automaattestid, kõigepealt on vaja tutvuda mobiilirakenduste testimise ja automaattestide loomise parimaid praktikaid. Selleks valitakse välja erinevad automatiseeritud testimise raamistikud, mis võivad sobida React Native mobiilirakenduste testimiseks.

Peatükis 2 esitatakse tehnoloogiate ülevaade, kus on kirjeldatud kõiki võimalikke React Native'i mobiilirakenduse testimise tüüpe ja parimad raamistikud nende rakendamiseks. Teises peatükis räägitakse ka mobiilitestimise funktsioonidest ja automaattestimise reeglitest üldiselt. 3 peatükis on näidistestid kirjutatud iga valitud testitüübi jaoks. Peatükis 4 on testide analüüs ja nende tõhususe hindamine idufirma jaoks.

2 Tehnoloogiate ülevaade

Käesolevas osas lugeja tutvub mobiilirakenduse testimise võimalike tüüpidega React Native'is. Autor selgitab, millised on mobiilitestimise omadused ning selgitab ka automaattestimise reegleid. Lugejale tutvustatakse ka testimisvahendeid.

2.1 Testimise tüübid

2.1.1 Ühiktestimine

Ühiktestimine hõlmab koodi väikseimaid osi, näiteks üksikuid funktsioone või klasse. Sellel on peamiselt üks või paar sisendit ja see annab üht väljundit. Seda tüüpi testimine on aega ja kulude säästev. Antud testimine peaks vältima sõltuvust teiste testide tulemustest ning olema lühike ja lihtne [1]. Mõnikord on sellise testimise jaoks nõue luua näidisobjekte (*mock objects*), et rahuldada või täita meetodi või funktsiooni sõltuvust. Kui funktsioon sõltub teisest funktsioonist ja muud funktsiooni pole veel loodud, siis kasutatakse näidisfunktsiooni objekti.

2.1.2 Integratsioonitestimine

Integratsioonitestimine on määratletud kui testimise tüüp, kus tarkvaramoodulid loogiliselt integreeritakse ja testitakse rühmana [2]. Tüüpiline tarkvaraprojekt koosneb mitmest tarkvaramoodulist, mis on loodud erinevate programmeerijate poolt. Selle taseme testimise eesmärgiks on välja selgitada defektid nende tarkvaramoodulite vahelises koostoimes, kui need on integreeritud.

Veel üks oluline aspekt, mida tuleks arvesse võtta, on ühik- ja integratsioonitestimise erinevus. Tarkvaratehnika ühiktestimise eesmärk on kontrollida suhteliselt väikese tarkvaraosa käitumist sõltumatult teistest osadest. Teisest küljest näitavad integratsioonitestid, et süsteemi erinevad osad töötavad koos reaalses keskkonnas. Need kinnitavad keerulisi stsenaariume (integratsiooniteste võib pidada kasutajaks, kes teeb meie süsteemis mõnda kõrgetasemelist toimingut) ja nõuavad tavaliselt väliressursside (nt andmebaaside või veebiserverite) olemasolu.

2.1.3 Komponendi testimine

React'i komponendid vastutavad teie rakenduse esitusviisi eest ja kasutajad kasutavad otse neid väljundeid. Isegi kui rakenduste äriloogika on suure testimise ulatusega ja on õige, võite ilma komponentide testimiseta siiski oma kasutajatele edastada katkist kasutajaliidest [3]. Komponentide testimine võib kuuluda nii ühik- kui ka integratsioonitestimisesse, kuid kuna need on React Native'i keskne osa, neid käsitletakse eraldi.

2.1.3.1 Kasutaja interaktsioonide testimine

Kasutaja interaktsioonide testimine peab tagama, et komponent käitub õigesti, kui kasutaja seda kasutab. Näiteks kui nupul on onPress'i kuulaja, soovite testida, kas nupp kuvatakse õigesti ja kas komponent käsitleb nupu puudutamist õigesti [3].

Antud töös ei kasuta kasutaja interaktsioonide testimist, kuna Zipstall'i rakenduse arendamisel harva esineb väheolulisi vigu. Kuid vajadusel saab kasutaja interaktsioone testida ka teise tüübi testimise abil.

2.1.3.2 Snapshot testimine

Snapshot testimine on täiustatud testimise tüüp võimaldatud Jest'i poolt [3]. Snapshot testid on väga kasulik tööriist, kui on soov veenduda, et rakenduse kasutajaliides ootamatult ei muutu.

Tüüpiline shapshot testjuhtum salvestab kasutajaliidese komponendi, teeb snapshot'i ja seejärel võrdleb seda snapshot'i võrdlusfailiga, mis oli salvestatud testi kõrval. Test ebaõnnestub siis, kui kaks hetktõmmist ei ühti: muudatus on ootamatu või tuleb viite hetktõmmist värskendada kasutajaliidese komponendi uue versioonini [4].

Snapshot testid rakenduse testimisel on enamasti ebaefektiivsed, kuna need ei taga komponendi renderdusloogika õigsust. Snapshots on vaevu head selleks, et kontrollida seda, kas testitava React'i puu komponendid saavad oodatuid omadusi.

Zipstalli rakenduse snapshot testimist kasutatakse ainult Onboarding'u testimiseks (õpetus uutele Zipstall'i kasutajatele). Vastasel juhul seda tüüpi testimine ei ole tõhus. Autor isiklikult töötas Onboarding'u välja ja kasutas selle rakendamiseks valmisekraane. Nii et kui põhiekraani kasutajaliides muutub, siis koos sellega on muutunud ka

Onboarding'us kasutatav ekraan. See aitab vältida dubleerimist ja samade muudatuste rakendamist kaks korda.

Aeg näitas seda, et mõnikord muudatused katkestavad Onboarding põhiekraanil ja seda avastatakse liiga hilja. Nende kriitiliste vigade vältimiseks Onboarding ekraane testitakse snapshot testimise abil.

2.1.4 End-to-End testimine

Erinevalt ühiktestimisest end-to-end testimise eesmärk on katta nii palju rakenduse funktsioone kui võimalik, simuleerides kasutaja tegelikke toiminguid. Mida rohkem alasid test hõlmab, seda usaldusväärsem see on.

E2E testimise teegid võimaldavad leida ja juhtida elemente rakenduse ekraanil: näiteks saavad E2E testid tegelikult vajutada nuppe või sisestada teksti tekstisisestustesse samamoodi nagu tegelik kasutaja seda teeks. Seejärel saab väita, kas teatud element on rakenduse ekraanil olemas või mitte, kas see on nähtav või mitte, millist teksti see sisaldab ja nii edasi [3].

2.2 Mobiilitestimise eripärad

„Mobiiltestimine nõuab palju käsitestimist ja seda ei saa veel asendada testimise automatiseerimise ega muude vahenditega.” [5]

Mitte iga mobiilseadme erifunktsiooni (nt kaamera) saab automatiseerida ning muid keskkonnast sõltuvaid andmeid on väga raske testida. Testija kohustus on tagada seda, et rakendatud andureid ja liideseid kasutatakse õigesti. Samuti on oluline kontrollida, et katki läinud andurid ei mõjutaks rakendust negatiivselt.

Zipstall'i mobiilirakenduse arendamise käigus autor pidevalt puutus kokku seadme funktsioonidega, mida ei ole võimalik mobiilsimulaatoril automatiseerida ega korrektselt esitada. Näiteks Zipstall'i automaatse parkimisfunktsiooni testimiseks peab olema võimalik kasutada Bluetooth'i. Seda saab teha ainult reaalse mobiilseadmega. Samuti kaamera kasutamine QR-koodi skaneerimiseks ja erineva Interneti kiirusega testimine mobiilisimulaatorites on väga piiratud ja automaattestimise puhul on vaeva teostatav.

Selguse huvides toon välja piirangu, et Zipstall'i arendamine eeldab tingimata õiget geograafilist asukohta: see on testimise ajal üks sagedamini kasutatavatest mobiilseadme erifunktsioonidest. Aga geograafilist asukohta on lihtne simuleerida isegi automaattestide jaoks, kuna kasutaja asukoht on sisuliselt kaks numbrit (laius- ja pikkuskraad).

„Ainult käsitestimine võib toimida, kuid see samuti ei ole tõhus.” [5] Käsitestimist tuleks läbi viia ainult siis, kui see vastab järgmistele kriteeriumidele:

- Rakendus on väga lihtne ja elementaarne.
- Rakendusel on väga piiratud funktsionaalsus.
- Rakendus on saadaval vaid piiratud aja jooksul.

Zipstall ei vasta ühelegi neist kriteeriumidest. Nii et antud juhtumi puhul ma otsustasin ühendada käsi- ja automaattestimist. Enne testimise automatiseerimist peaks alati läbima käsitestimise. Iga uut funktsiooni tuleb erinevates seadmetes süstemaatiliselt käsitsi testida. Pärast käsitestimise lõpetamist saab määrata rakenduse osi, mis vajavad testimise automatiseerimist.

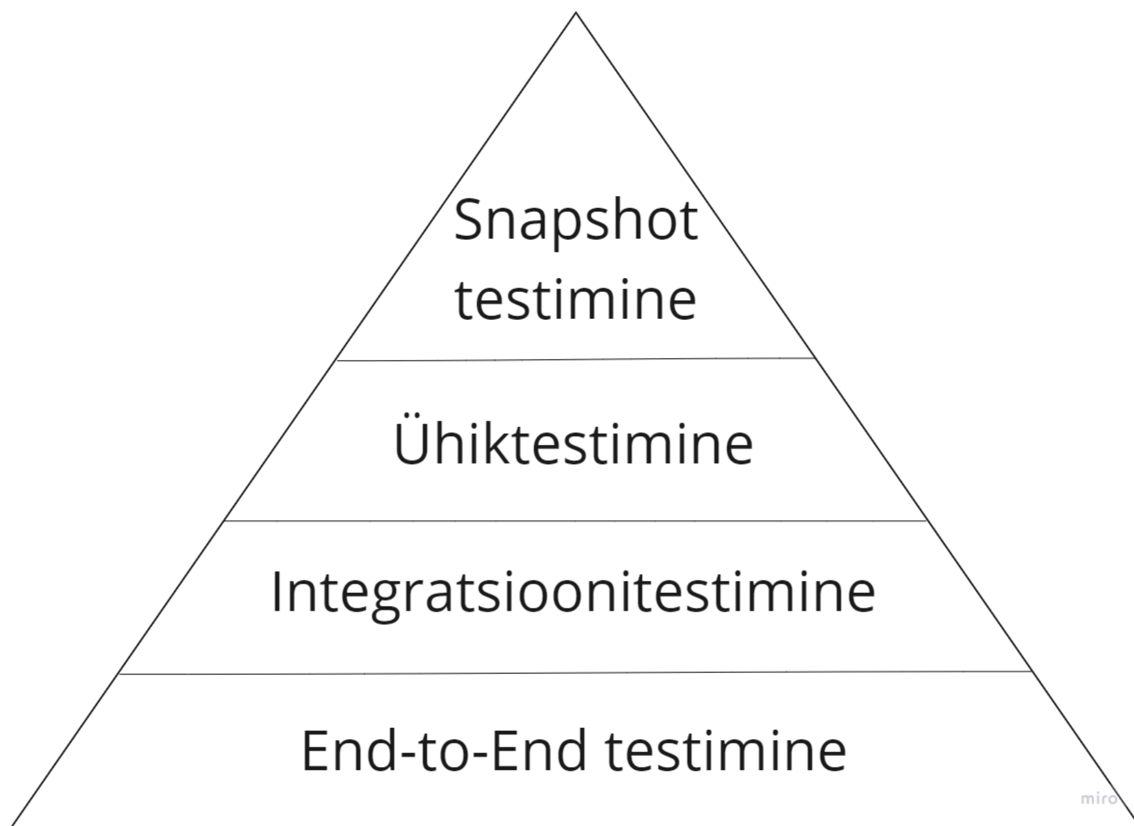
Mobiilsel automaattestimisel kasutatakse püramiidmeetodit, nagu pakkus Mike Cohn [6]. Kõigi rakenduste jaoks sama lähenemisviisi kasutamine ei ole realistlik. Seetõttu kasutab autor selle lõputöö jaoks oma poolt loodud püramiidi (vt Joonis 1). Antud püramiid ei pruugi mõnes teises mobiilirakenduses olla tõhus, kuna see on mõeldud ainult Zipstall'i rakenduse jaoks.

Selles püramiidi versioonis on Snapshot testid püramiidi tipus, kuna neid kasutatakse harva (Onboarding).

Järgmine on ühiktestimise kiht, mis on teine suuruse järgi. See kehtib, sest mitte iga mobiilirakenduse ühikut või meetodit ei saa eraldi testida. Mõnel juhul erinevaid API-sid, kihte või süsteeme võib võltsida või maketeerida selleks, et panna väikest ühikut tööle. See kehtib ka kõigi teiste tarkvararakenduste puhul, kuid mõnel juhul on mobiilirakenduste muude süsteemide maketeerimine või võltsimine palju keerulisem. See ei ole tõhus tehnilisest ega majanduslikust seisukohast. Kuid see ei ole vabandus mobiilseadmete ühiktestide mitte kirjutamiseks. Rakenduse äriloogikat tuleb testida ühiku tasemel.

Integratsioonitestimise kiht on tehtud integreeritud moodulite/komponentide testimiseks, et kontrollida, kas need töötavad ootuspäraselt, et testida mooduleid, mis töötavad eraldi ning ei teki integreerimisel probleeme.

Viimane etapp on end-to-end testimise automatiseerimise kiht. Selle kihi sees testitakse rakendust kasutaja vaatenurgast, et veenduda, kas kogu süsteem töötab korralikult.



Joonis 1 Autori loodud automaattestimine püramiid.

2.3 Automaattestimise reeglid

Pärast mõne käsitestimise kaitsete läbiviimist ning tulemuste mustrite ja testi läbiviimise sageduse tuvastamist peab kaaluma kõige tavapärasemate või prognoositavamate katsete automatiseerimist.

Mobiilirakenduse arendamisega muutub ka tootevoog, samuti muutuvad ka kasutajaliidese nõuded ja erifunktsioonid. Selle tulemusena nõuab iga muudatus automaattesti skripti värskendamist. Kui töötatakse väikse projektiga, siis mobiilirakenduste automatiseeritud testimiseks vajalik hooldus on üksluine ja kulukas.

Automatiseerimise eesmärk on tõsta nii mobiilirakenduse testimise tõhusust (aega ja kulu) kui ka kvaliteeti.

Automaattestimine sobib kõige paremini suurte projektide jaoks, mis nõuavad eelnevalt kirjutatud skriptide pidevat või korduvat testimist, mille eeliseks on see, et mitut samaaegset testimist saab korraldada erinevate komponentide vahel [7].

Esimene punkt on see, et turul pole saadaval "ühtki kõigile sobivat" testimise automatiseerimise vahendit. Igal vahendil on oma eelised ja puudused ning mitte iga vahend sobib igasse arenduskeskkonda ja torustikku. vahend A võiks hästi töötada projektiga A, kuid mitte projektiga B, mis tähendab, et hindamist tuleb korrata iga projekti puhul.

2.4 Testimisvahendite tutvustus

Hetkel on saadaval 5 suurt testimisvahendit, mis sobivad mobiilirakenduste testimiseks React Native'is. Zipstall'i testimiseks olid valitud kaks JavaScript'i mobiilitestimise raamistikku, Jest ja Detox. Selgus, et nendest kahest testimisvahendist piisab igat tüüpi testide läbiviimiseks. Lisaks neid testimisvahendeid saab omavahel siduda, mis hõlbustab testimisprotsessi ja muudab projekti "puhtamaks". Seetõttu autor valis testimisvahendi andmed, mis on üksikasjalikumalt käsitletud allpool.

2.4.1 Jest

Jest'i tarnitakse NPM-paketina, seda saab installida mistahes JavaScripti projektis. Tänapäeval Jest on üks populaarsemaid testijaid ja React'i projektide vaikimisi valik.

Jest testid jooksevad paralleelselt, mis omakorda oluliselt vähendab testi täitmise aega. Iga Jest'i test töötab oma liivakastis, mis tagab, et kaks testi ei sega ega mõjuta üksteist. Jest testid toetavad igat tüüpi maketeerimist: olgu see siis funktsionaalne maketeerimine, taimeriga maketeerimine või üksikute API-kutsete maketeerimine.

Snapshot testimine on React'i vaatenurgast asjakohane. Jest toetab testitava reaktsioonikomponendi snapshot'i jäädvustamist. Seda saab kinnitada komponendi tegeliku väljundiga. See oluliselt aitab kontrollida komponendi käitumist [4].

2.4.2 Detox

Detox on mobiilirakenduste end-to-end testimise ja automatiseerimise raamistik. Detox'i raamistik testib mobiilirakendust, kui see töötab reaalses seadmes/simulaatoris, käitudes nagu päris kasutaja.

Mobiilse kasutajaliidese testimise peamised probleemid on aeglus ja ebaühtlus. Sellised tööriistad nagu Appium on täiesti musta kasti meetod ja nad kasutavad kliendi-serveri arhitektuuri, mis põhjustab ebaühtlust kasutajaliidese testides. Detox on mõeldud mobiilse kasutajaliidese testimise aegluse ja ebaühtluse probleemi lahendamiseks.

Detox on halli kasti testimismeetod tööriist, mis võimaldab teil pääseda juurde teie mobiilirakenduste koodile ja andmetele [8]. Detox pakub suurepäraseid abstraktsioone elementide valimiseks ja käivitamiseks. Detox jälgib rakenduses asünkroonseid toiminguid, et vähendada rakenduse kasutajaliidese asünkroonsete elementide leidmise ebakindlust.

Peamiseks põhjuseks, miks Detox oli valitud E2E testide loomiseks Appium'i asemel, ei olnud isegi mitte selle testi täitmise kiirus, vaid asjaolu, et Detox'i saab kasutada iga testijaga, selle projekti jaoks oli valitud loomulikult Jest.

3 Realisatsioon

Peatükis kirjeldatakse üksikasjalikult testide loomist iga valitud testimistüübi jaoks kui ka juhendi loomine.

Selles lõputöös ei ole juhiseid mobiilitestimise raamistiku installimiseks rakendusse, kuna autor järgis testide installimisel ja rakendamisel selgelt ametlikke dokumente. Pole mõtet korrata juhiseid, mida saab hõlpsasti leida ja üksikasjalikult lugeda iga raamistiku dokumentatsioonist.

3.1 Süsteemi ülevaade

Zipstall see on mobiilirakendus, mis lahendab parkimisprobleemi ja samal ajal aitab kohalikke ettevõtteid. Nagu enne öeldi, Zipstalli arendamisel kasutatakse React Native'i ja see on välja töötatud nii iOS-i kui ka Androidi jaoks. Samuti kasutatakse Zipstalli arendamiseks paljusid avatud lähtekoodiga pakette, mis raskendab oluliselt testide koostamist. Praegune rakendus on alles arendusjärgus ning seega on võimalik testida piiratud osa. Uute testide loomise prioriteet on kiirus ja lihtsus.

3.2 Näidistestid

Selles jaotises on näidatud näidistestid iga testimise tüübi jaoks mis on automaattestimise püramiidis. Ärisaladuse tõttu ei ole võimalik selles töös kõiki teste näidata. Seetõttu otsustati demonstreerida iga testimise tüübi jaoks ühte näidistesti. Üks näide on täiesti piisav, et näidata igat tüüpi testimise põhimõtet, kuna selgus, et kõik teatud tüüpi testid on üksteisega võimalikult sarnased ja pole vaja arvestada paljude identsete näidetega.

3.2.1 Snapshot näidistest

Nagu varem mainitud, on selles projektis Snapshot teste vaja peamiselt Onboarding'u testimiseks. See tähendab, et kui keegi muutis arenduse käigus mõnda Onboardinguga seotud ekraani või komponenti, võimaldavad testid seda kiiresti märgata ja parandada.

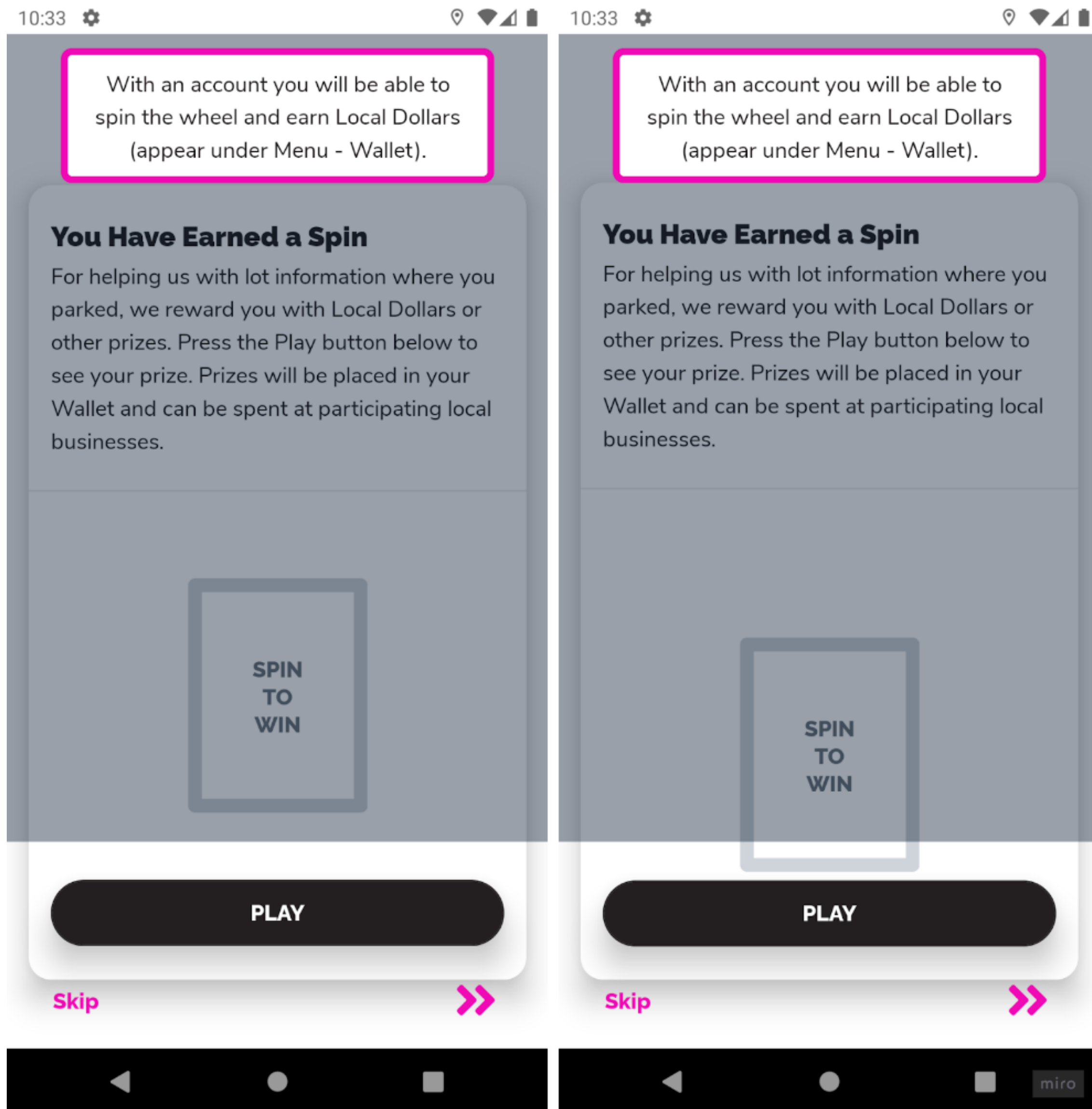
Näiteks võtsin ekraani, mis näitab, et enne parkimist peab kasutaja ratast keerutama ja saama kohalikku raha. Oletame, et arenduse käigus läks ratas alla ja nüüd peate tegema muudatusi ka Onboarding'us (vt Joonis 3).

Kogemus näitab, et mida suuremaks projekt läheb, seda keerulisem on arendajal pärast uute muudatuste lisamist viga ennustada. Siin aitavad snapshot testid, nüüd on lihtne kontrollida, kas Onboarding on muutunud, tuleb lihtsalt käivitada käsk „*yarn test*". Selles näites andsid testid järgmise vea (vt Joonis 2).

```
 FAIL  __tests__/SnapshotTests.tsx
  ● N16_LocalDollarsWheel

    expect(received).toMatchSnapshot()

    Snapshot name: `N16_LocalDollarsWheel 1`

    - Snapshot  - 1
    + Received  + 4

    @@ -232,11 +232,14 @@
                            Array [
                              Object {
                                "flex": 1,
                              },
                              Object {
    -                           "justifyContent": "center",
    +                           "marginBottom": 4,
    +                         },
    +                         Object {
    +                           "justifyContent": "flex-end",
                              },
                            ]
                          }
                        >
                          <View

      23 | test('N16_LocalDollarsWheel', () => {
      24 |   const tree = renderer.create(<N16_LocalDollarsWheel route={undefined} />).toJSON();
    > 25 |   expect(tree).toMatchSnapshot();
         |                ^
      26 | });
      27 |

      at Object.<anonymous> (__tests__/SnapshotTests.tsx:25:16)

 › 1 snapshot failed.
```

Joonis 2 Snapshot testi viga.

See viga näitab, et mõni objekt on alla liikunud. See tähendab, et arendaja peab seda konkreetset ekraani vigade suhtes kontrollima. Nüüd on näha paremal ekraanil (vt Joonis 3), et tumenemine on valesti näidatud, lõigates osa rattast ära ja see tuleb parandada.



Joonis 3 Ekraan enne ja pärast muudatusi.

3.2.2 Ühiktestimine näidistest

Ühiku testi näite jaoks valiti funktsioon, mis loeb parkimise alguse ja lõpu vahelist aega (vt Joonis 4). Funktsioon „*countSessionDuration*“. tagastab väärtuse stringina, mis tavaliselt näeb välja selline: „1 h 30 min“.

```
export function countSessionDuration(start: string, end: string) {

        const startTime = moment.tz(start, 'UTC');

        const endTime = moment.tz(end, 'UTC');

        const duration = moment.duration(endTime.diff(startTime));
```

```
        const diff = moment.tz(endTime.diff(startTime), 'UTC');

        const diffHour = diff.format('H');

        const diffDay = Math.floor(duration.asDays());

        const days = diffDay > 0 ? diffDay + ' d ' : '';

        const hours = diffHour === '0' ? '' : diffHour + ' hr ';

        const minutes = diff.format('m');

        return days + hours + minutes + ' min';

}
```

Joonis 4 Funktsioon, mis loeb parkimise alguse ja lõpu vahelist aega.

Selle funktsiooni testimiseks loodi eraldi testplokk sama nimega „*countSessionDuration*“. See plokk sisaldab 3 erandi testi ja 3 standardjuhtumi testi.

```
describe('countSessionDuration', () => {

        test('same dates', () => {

          expect(countSessionDuration('2022-03-30T11:00:00.000Z', '2022-
        03-30T11:00:00.000Z')).toBe('0 min');

         });

         test('minutes', () => {

          expect(countSessionDuration('2022-03-31T07:13:29.000Z', '2022-
        03-31T07:44:25.000Z')).toBe('30 min');

         });

         test('hours + minutes', () => {

          expect(countSessionDuration('2022-03-31T04:59:31.000Z', '2022-
        03-31T07:04:08.000Z')).toBe('2 hr 4 min');

         });

         test('days + minutes', () => {

          expect(countSessionDuration('2022-03-30T11:15:20.000Z', '2022-
        04-01T11:34:59.000Z')).toBe('2 d 19 min');

         });

         test('days + 0 minutes', () => {

          expect(countSessionDuration('2022-03-28T11:15:20.000Z', '2022-
        03-30T11:15:22.000Z')).toBe('2 d 0 min');

         });
```

```
      test('days + hours + minutes', () => {

        expect(countSessionDuration('2022-03-30T11:15:20.000Z', '2022-04-01T08:26:59.000Z')).toBe('1 d 21 hr 11 min');

      });

});
```

Joonis 5 Ühikutestid.

Ootuspäraselt lõppesid kõik testid edukalt ja kontrollimiseks kulus vähem kui sekund. See kinnitab teooriat, et ühikutestid on väga kiired. Seetõttu on võimalik neid teha palju, testides rakenduse kõiki aspekte.

Näiteks tegin viimases testis vea ja nüüd vaatame, kuidas Jest viga näitab.

```
test('days + hours + minutes', () => {

   expect(countSessionDuration('2022-03-30T11:15:20.000Z', '2022-04-01T08:26:59.000Z')).toBe('2 d 31 hr 41 min');

 });
```

Joonis 6 Ühikutest mis peaks vea viskama.

Jest näitas selgelt, kus täpselt on erinevus oodatud ja tulemuse vahel (vt Joonis 7). See kiirendab vigade leidmist ja üldiselt testimist.

```
yarn run v1.22.18
$ jest App-test
 FAIL  __tests__/App-test.tsx
  countSessionDuration
    ✓ same dates (3 ms)
    ✓ minutes
    ✓ hours + minutes
    ✓ days + minutes (1 ms)
    ✓ days + 0 minutes
    × days + hours + minutes (3 ms)

  ● countSessionDuration › days + hours + minutes

    expect(received).toBe(expected) // Object.is equality

    Expected: "2 d 31 hr 41 min"
    Received: "1 d 21 hr 11 min"

      24 |
      25 |   test('days + hours + minutes', () => {
    > 26 |     expect(countSessionDuration('2022-03-30T11:15:20.000Z', '2022-04-01T08:26:59.000Z')).toBe('2 d 31 hr 41 min');
         |                                                                                          ^
      27 |   });
      28 | });
      29 |

      at Object.<anonymous> (__tests__/App-test.tsx:26:90)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 5 passed, 6 total
Snapshots:   0 total
Time:        0.761 s, estimated 1 s
Ran all test suites matching /App-test/i.
error Command failed with exit code 1.
```

Joonis 7 Erinevus oodatud ja tulemuse vahel.

3.2.3 Integratsioonitestimine näidistest

Integratsioonitestide demonstreerimiseks kontrollin kahe asünkroonse funktsiooni ühistööd. Samal ajal kontrollin API-teenuse kaudu päringute tõhusust.

Selles näites testitakse äriloogikat, kus ainult tsooni ID olemasolu korral peaks tagastama Google Mapsis määratud aadressi (vt Joonis 8). Otseselt seda teha ei saa, vaid tuleb esmalt välja selgitada parkimistsooni (lot) koordinaadid ja seejärel saata päring google Maps API-le. Selle tulemusena saadi 2 funktsiooni aadressi õigsuse kontrollimiseks. Esimene kontrollib, kas koordinaadid (laius- ja pikkuskraad) on õiged. Teises funktsioonis, juba selle punkti põhjal kaardil, annab google Maps meile täpse aadressi.

```
describe('parkingLotAPI', () => {

    test('get lot address where id is 1', async () => {

      const data = await parkingLotAPI.getLotLocations(1);

      const lotLocationData = data[0];

      expect(lotLocationData.lat).toBe(53.5402083307);

      expect(lotLocationData.lng).toBe(-113.4976062163);

      const endingPoint = await parkingLotAPI.getGeocodeAddress(
   lotLocationData.lat, lotLocationData.lng);

      expect(endingPoint).toBe('10046 103 St NW, Edmonton, AB T5J
   0X2, Canada');

    });

});
```

Joonis 8 Integratsioonitestimine näidistest.

Selle funktsionaalsuse jaoks piisab ühest testist ja selle loogika testimiseks pole ajaliselt efektiivne kirjutada täiendavaid teste. Nagu tulemusest näha (vt Joonis 9), kulub üks API test rohkem kui 100 ms, samas kui eelmised ühikutestid võtavad igaüks umbes 1 ms.

```
yarn run v1.22.18
$ jest App-test
 PASS  __tests__/App-test.tsx
  countSessionDuration
    ✓ same dates (4 ms)
    ✓ minutes (1 ms)
    ✓ hours + minutes (1 ms)
    ✓ days + minutes
    ✓ days + 0 minutes (1 ms)
    ✓ days + hours + minutes
  parkingLotAPI
    ✓ get lot address where id is 1 (119 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        0.895 s, estimated 1 s
Ran all test suites matching /App-test/i.
Done in 2.03s.
```

Joonis 9 Üksuse- ja integratsioonitestide edukas läbimine.

3.2.4 End-to-End näidistest

Püramiidi juurutamise lõpetamisel on vaja läbida suurema osa - E2E testid, selleks tuleb projekti installida detox. Paigaldamine ei olnud lihtne, oli palju vigu, mis tuli järk-järgult ametlikku dokumentatsiooni kasutades lahendada [8].

Pärast detox'i edukat lisamist on vaja androidi emulaatorile rakendada kõik seaded, selleks käivitama käsu „*detox build --configuration android*“.

Näidis E2E testi jaoks loodi stsenaarium, kus kasutaja lisab uue sõiduki (vt Joonis 10). Kuna Zipstalli rakendus on seotud parklatega, siis see stsenaarium on rakenduses üks olulisemaid ja seetõttu tuleks seda esmalt testida.

```
describe('Initial Setup', () => {
        beforeAll(async () => {
          await device.launchApp();
        });

        beforeEach(async () => {
          await device.reloadReactNative();
        });

        it('add new vehicle', async () => {
          await element(by.text('MENU')).tap();
          await element(by.id('scrollView')).scroll(1000, 'down');

          await element(by.text('Vehicles')).tap();
          await element(by.text('ADD VEHICLE')).tap();
          const enterLicensePlate = element(by.id('number-plate'));
          await enterLicensePlate.typeText('322ABC');
          await enterLicensePlate.tapReturnKey();
          await element(by.id('add-vehicle')).tap();

          await expect(element(by.text('322ABC'))).toBeVisible();
        });
});
```

Joonis 10 E2E näidistest, kus kasutaja lisab uue sõiduki.

Nagu testi käigus näha, avaneb esmalt Menüü, seejärel kerib alla ja klõpsab nupul, kus kuvatakse kõik kasutaja sõidukid. Järgmisena leiab testija sõiduki numbri sisestamise välja ja kinnitab lisamise. Pärast seda toimub test ise, kus kontrollitakse, kas kuvatakse uus sõiduk.

Tasub märkida, et E2E testide kirjutamine võttis väga vähe aega, kuna need on kõik sarnased, kuna põhimõtteliselt testid peavad kuskil klõpsama või teksti sisestama. Sellest hoolimata on detoxil piisavalt suur funktsionaalsus, mis võimaldab testida peaaegu kõiki

mobiilirakenduse aspekte. Samuti E2E testid ei muuda põhikoodibaasi, välja arvatud see, et mõnikord tuleb mõne komponendi jaoks lisama täiendava parameetri testId.

Nii et selles näites tuvastati uue sõiduki lisamisel nupp „*ADD VEHICLE*“ esmalt teksti kaudu. Seejärel oli vaja testId kaudu ära tunda sarnane nupp nimega „*ADD VEHICLE*“. Sest nupu nimi on täpselt sama, mis ekraani pealkiri – „*ADD VEHICLE*“. Selleks oli vaja selle nupu lisaparameetrina lisada testId.

Sellest järeldub, et turvalisem on alati kasutada elementide tuvastamist testId kaudu kui teksti kaudu. Kuigi see veidi halvendab koodi loetavust ebavajaliku teabega.

3.3 Testide kasutamise juhendi koostamine

Enne testide loomist oli eesmärk teha kõik selleks, et kolleegid saaksid võimalikult kiiresti ja lihtsalt teste kasutama hakata. Selleks tuleb koostada testide kasutamise juhend.

Väga oluline on märkida, et hea juhendi koostamiseks pole ühte õiget viisi. Kuid on üks väga vale viis ja see on juhendi puudumine [9]. Isegi kui kõikidele projektiarendajatele on testide kasutamist lihtsam selgitada kui käsiraamatut kirjutada, siis varem ja hiljem tulevad meeskonda uued arendajad, kes vajavad juhendamist.

Juhendi loomisel oli peamine ülesanne muuta see võimalikult lihtsaks ja arusaadavaks (vt Joonis 11). Selleks kirjutati ainult vajalik teave ja täiendavaid üksikasju testide loomise kohta leiate ametlikust dokumentatsioonist.

Joonis 11 Testide kasutamise juhend, vasakul lähtekood, paremal juhend ise.

4 Testide analüüs

Sellest jaotisest lugeja saab teada, kui oluline on testitava koodi kirjutamine. Samuti saab teada saadud tulemustest ja edasisest arengust.

Selles lõputöös toodi püramiidist iga testimise tüübi kohta ainult üks testinäide. Sellest saab juba teha peamised järeldused, kuid analüüsi usaldusväärsuse tagamiseks loodi täiendavad testid. Selle tulemusel tuleb 7 Snapshot testi (see on maksimum, mis võib olla), 3 ühikutesti, 5 integratsioonitesti ja 10 end-to-end testi.

4.1 Koodi testitavus

Esimene asi, millega autor testide loomisel kokku puutus, oli see, et projekti kood polnud testitav. See tähendab, et enamiku ühiku- ja integratsioonitestide puhul tuli testimise võimaldamiseks teha muudatusi põhikoodibaasi. Näiteks valiti vana funktsioon, mis on testimisel olnud juba ammu prioriteetne (vt Joonis 12).

```
export const Nearest30Date = () => {
        const now = new Date();
        const minutes = now.getMinutes();
        if (minutes >= 0 && minutes <= 15) {
           now.setMinutes(30);
           return now;
        } else if (minutes > 15 && minutes <= 30) {
           now.setMinutes(45);
           return now;
        } else if (minutes > 30 && minutes <= 45) {
           now.setMinutes(0);
           now.setHours(now.getHours() + 1);
           return now;
        } else {
           now.setMinutes(15);
```

```
            now.setHours(now.getHours() + 1);
            return now;
          }
    };
```

Joonis 12 Funktsioon, mida pole võimalik testida.

See funktsioon on olnud kasutusel alates rakenduse loomisest ja seda on korduvalt muudetud ning tõenäoliselt muutub see ka edaspidi regulaarselt. Arvukate muudatuste tõttu ei tundu funktsioon " puhas" ja testitav.

Funktsioon „*Nearest30Date*“ peaks praegusele ajale lisama 30 minutit ja ümardama tulemuse nii, et minutid oleksid 0, 15, 30 või 45.

Seda funktsiooni ei saa sellisel kujul testida. „*new Date()*“on sisuliselt peidetud sisend, mis muutub programmi täitmise ajal või testkäivituste vahel. Seega annavad sellele järgnevad päringud erinevaid tulemusi.

Seda funktsiooni ei saa uuesti kasutada muudest allikatest hangitud või argumendina edastatud kuupäeva ja kellaaja töötlemiseks. Funktsioon töötab ainult konkreetse masina kuupäeva ja kellaajaga, mis koodi käivitab. Tihe sidumine (*Tight coupling*) on enamiku testitavuse probleemide peamine põhjus.

Kõige ilmsem ja lihtsaim viis probleemi lahendamiseks on lisada funktsioonidele argument (vt Joonis 13).

```
export const Nearest30Date = (now: Date) => {
        const minutes = now.getMinutes();
        if (minutes >= 0 && minutes <= 15) {
          now.setMinutes(30);
          return now;
        } else if (minutes > 15 && minutes <= 30) {
          now.setMinutes(45);
          return now;
        } else if (minutes > 30 && minutes <= 45) {
          now.setMinutes(0);
          now.setHours(now.getHours() + 1);
```

```
            return now;
          } else {
            now.setMinutes(15);
            now.setHours(now.getHours() + 1);
            return now;
          }
      };
```

Joonis 13 Funktsioon lisatud argumendiga.

Nüüd nõuab funktsioon väljakutsujalt kuupäeva argumendi esitamist, selle asemel, et seda teavet ise salaja otsida. Ja alles pärast neid muudatusi on võimalik seda funktsiooni testida.

Tasuks selle tarkvara kvaliteedi tagamise eest saame puhta, kergesti hooldatava, lõdvalt ühendatud (*loosely coupled*) ja korduvkasutatava koodi, mis ei kahjusta arendajate aju, kui nad püüavad seda mõista [10]. Lõppude lõpuks pole testitava koodi ülim eelis mitte ainult testitavus ise, vaid ka võimalus seda koodi hõlpsalt mõista, hooldada ja laiendada.

4.2 Automaattestimisest saadud tulemused

Peamine on arusaam, et tuleb kirjutada testitav kood, et tulevikus oleks võimalik rakendust testida. Tänu sellele muutub projekti kood palju "puhtamaks" ja arusaadavamaks.

Tööülesannete täitmine muutus lihtsamaks. Kuna kui on märgata, et arenduse käigus testitakse sama funktsiooni mitu korda käsitsi, siis algab selle funktsiooni testide loomine ja arendusprotsess läheb kiiremini.

Onboardingu probleem on lahendatud, nüüd ei pea arendaja muretsema, kui midagi selles äkki puruneb, testid näitavad seda kohe. Selgus, et testide loomine võtab vähe aega ega nõua palju pingutust.

End-to-end testid on rakenduste arendusprotsessi kindlasti kiirendanud. Kuigi need võtavad rohkem kui mõne sekundi, on see siiski kiirem kui käsitsi testimine. Samuti saab arendaja pärast testi käivitamist teha pausi ja lihtsalt protsessi jälgida.

Miinustest võib märkida, et integratsioonitestidel puudub täpne definitsioon, mitme funktsiooni korraga testimine on keeruline. Seetõttu näeb integratsioonitest välja, et kaks ühikutesti on üheks ühendatud.

Liiga vähe aega on möödunud, et kolleege testide kirjutamisega harjuda. See võtab aega, enne kui nad harjuvad rakenduses testide kasutamisega. Kuid kindlasti muutub see algajatele lihtsamaks, sest detox testid näitavad suurepäraselt, kuidas rakendus peaks töötama.

4.3 Edasised tegevused

Järgmine ülesanne on lisada rohkem teste, et saaksite testida kõiki rakenduse olulisi funktsioone. Samuti ei tohi unustada, et edaspidiseks kasutamiseks on oluline teste pidevalt täiendada.

Kaasake töösse teste, harjutades sellega kolleege looma teste nii uute funktsioonide arendamiseks kui ka testimiseks. On suur võimalus, et kui kolleegid õpivad Zipstalli arendamise käigus teste looma, hakkavad nad tegema teste ka teiste projektide jaoks. Kindlasti tõstab see kohati ka väikeettevõtte rakenduste arendamise efektiivsust.

5 Kokkuvõte

Käesoleva lõputöö eesmärgiks oli luua mobiilirakendusele automaatteste ja analüüsida nende efektiivsust idufirma jaoks. Selle jaoks anti ülevaade sobivaimateks testi tüüpidest, millest seejärel koostati püramiid. Autor selgitab, miks on oluline automatiseeritud testide loomine ning tegi selgeks, et Zipstalli mobiilirakenduse jaoks on testid vajalikud. Samuti valiti välja kaks kõige sobivamat raamistikku testide loomiseks - Jest ja Detox, mis olid samuti omavahel seotud.

Käesolevas lõputöös esitati iga testimise tüübi jaoks üks näidistest. Kus on üksikasjalikult kirjeldatud, milleks iga test loodi ja kuidas mobiilirakenduse arendamise käigus abiks on.

Pärast testide kasutamist autor jõudis järeldusele, et testitav kood on vaja kohe projekti sisse kirjutada. Seega muudab testitav kood koodibaasi puhtamaks ja arusaadavamaks. Aeg on näidanud, et testid on mõned probleemid juba lahendanud ja võib öelda, et töövoog on üldiselt kiirenenud.

Kuigi projekti testivahendite lisamine ja esimeste testide loomine võttis kaua aega. Selle tulemusena selgus, et testide edasine loomine võtab vähe aega ja vaeva, kuna paljud testid on üksteisega sarnased.

Kõige eelneva põhjal võib kindlalt väita, et testide loomine on väikeettevõtte jaoks tõstnud töö efektiivsust. Samuti tõuseb efektiivsus veelgi edaspidi, kui idufirmas tulevad uued töötajad ning tänu testidele kohanevad nad kiiremini tööle.

Täiesti loomulikult tuleb selleks, et teste edaspidi kasutada saaks, neid pidevalt juurde ja kaasajastada. Peamine on aga see, et selle lõputöö käigus loodud testid saaksid ettevõttes tõukejõuks, et kolleegid mõistaksid, et testid on iga rakenduse oluline osa.

Kasutatud kirjandus

[1] P. Hamill, Unit test frameworks, O'Reilly Media, Inc., 2004.

[2] Syllabus, ISTQB® Mobile Application Testing, 2019.

[3] V. Novak, „reactnative.dev,“ [Võrgumaterjal]. Available: https://reactnative.dev/docs/testing-overview. [Kasutatud 25 04 2022].

[4] S. Bekkhus, „jestjs,“ [Võrgumaterjal]. Available: https://jestjs.io/docs/snapshot-testing. [Kasutatud 25 04 2022].

[5] D. Knott, Hands-On Mobile A Guide for Mobile Testers and Anyone Involved in the Mobile App Business, Crawfordsville: 2015 Pearson Education, Inc., 2015.

[6] M. Cohn, „mountaingoatsoftware,“ [Võrgumaterjal]. Available: https://www.mountaingoatsoftware.com/. [Kasutatud 25 04 2022].

[7] A. KAPOOR, „netsolutions,“ [Võrgumaterjal]. Available: https://www.netsolutions.com/insights/mobile-app-testing-automation/. [Kasutatud 25 04 2022].

[8] „Detox,“ [Võrgumaterjal]. Available: https://wix.github.io/Detox/docs/introduction/getting-started. [Kasutatud 25 04 2022].

[9] H. Nyakundi, „How to Write a Good README File for Your GitHub Project,“ 08 12 2021. [Võrgumaterjal]. Available: https://www.freecodecamp.org/news/how-to-write-a-good-readme-file/. [Kasutatud 09 05 2022].

[10] S. Kolodiy, „Unit Tests, How to Write Testable Code and Why it Matters,“ [Võrgumaterjal]. Available: https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters. [Kasutatud 07 05 2022].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, "Niktia Putyatin"

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „React Native mobiilirakenduse automaattestimine Zipstall'i näitel - analüüs ning juurutamine", mille juhendaja on Maili Markvardt:
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

16.05.2022

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.