TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kevin Janson 179625IADB

Mobiilirakenduste testimise labori integreerimine õppetöösse

Bakalaureusetöö

Juhendaja: Meelis Antoi
Magistrikraad
Priit Pikk
Bakalaureusekraad

Tallinn 2021

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: Kevin Janson

14.05.2021

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on luua töötav mobiilirakenduste testimise labor ning see integreerida õppetöösse, et soodustada hindamismetoodikat ja parandada tudengite mobiilirakenduste kvaliteeti. Labor on avatud lähtekoodiga ning kogu intellektuaalne omand on Tallinna Tehnoloogiaülikooli oma. Lähtekood asub Lisa 6 peatükis.

Arendusprotsessi käigus luuakse terviklahendus, mis võimaldab tudengitel automaatselt kompileerida rakendusi, käivitada neid laboris olevate seadmete peal ning saada testimistulemustest konstantset tagasisidet tudengite etteantud emaili peale. Arendus on jagatud mitmesse osasse, mis käsitleb Java rakenduse loomist ning selle ühendamist erinevate osapooltega, mis on oluline osa CI/CD süsteemist.

Töö rõhk on kogu süsteemi töötamisel, kui ühtse sümbioosina erinevatest teenustest ning Java rakenduse arendamine, mille peamine ülesanne on töötada mobiilseadmete orkestraatororina. Java rakendus oskab anda tudengite kirjutatud mobiilirakendustele kahte tüüpi tagasisidet “Kuvatõmmiste test” ning “Kasutajaliidese test”. Viimane eeldab tudengite poolset testide kirjutamist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 27 leheküljel, 4 peatükki, 3 joonist, 0 tabelit.

Abstract

Integrating Mobile Application Testing Lab (Testlab) into Studies

The aim of this bachelor's thesis is to create a working mobile application testing laboratory, and to integrate it into the teaching in order to promote the assessment methodology and improve the quality of students' mobile applications. The laboratory is open source, and all intellectual property belongs to Tallinn University of Technology.

During the development process, a complete solution is created that allows students to automatically compile applications, run them on equipment in the laboratory, and receive constant feedback on the test results to the email provided by the students. The development is divided into several parts concerning the creation of a Java application and its integration with various parties, which is an important part of the CI / CD system.

The emphasis of the work is on the operation of the whole system as a unified symbiosis of different services, and also on the development of a basic service, ie a Java application, the main task of which is to work as an orchestrator of mobile devices. A Java application can provide two types of feedback to student-written mobile applications: "Screenshot test" and "User interface test". The latter requires students to write tests.

The thesis is in Estonian and contains 27 pages of text, 4 chapters, 3 figures, 0 tables.

Lühendite ja mõistete sõnastik

ADB	*Android Device Bridge* – Android käsurea tööriist
API	*Application Programming Interface* – rakendusliides
CD	*Continuous delivery* - pidev tarne
CI	*Continuous integration* - pidev integratsioon
CRON	*Command Run On* - ajapõhine tööplaneerija
cURL	*Client URL*, andmete edastamis utiliit
Espresso	Androidis kasutatav raamistik kasutajaliideste testide käivitamiseks.
Gitlab	Koodihoidla
GPS	*Global Positioning System* - globaalne positsioneerimis süsteem
Gradle	Automatiseerimis tööriist
JVM	*Java Virtual Machine* - Java virtuaalmasin
native	Spetsiifilisel platvormile kirjutatud kood. Ürgne
PDF	*Portable Document Format* - porditav dokumendivorming
SDK	*Software Development Kit* - tarkvaraarendus pakett
UI	*User Interface* - kasutajaliides
UML	*Unified Modeling Language* - unifitseeritud modelleerimiskeel
URL	*Uniform Resource Locator,* veebiaadress
XCRUN	IOS käsurea tööriist

Sisukord

Jooniste loetelu

Koodinäidete loetelu

1 Sissejuhatus

Hinnanguliselt kasutatakse täna maailmas 3.6 miljardit mobiiliseaded ning aastaks 2023 on ennustatud, et see arv kasvab 4.3 miljardi seadmeni [1]. Sellest tulenevalt on ka väga paljud rakendused ja mängud kolinud üle arvutitest nutitelefonidesse. Üha kasvav nutitelefonide turg on viinud paljud ettevõtted väljastama arvutite peal kasutatavaid tarkvarasid üle nutitelefonidesse. Kuna antud valdkonnas on nõudlus suur, siis on ka inimestel väga erinevad seadmed. Peamised parameetrid, mille põhjal seadmed erinevad ning mis on olulised kohad testimisel:

- Ekraani suurus
- Protsessori võimsus
- Seadme muutmälu
- Operatsioonisüsteemi tarkvara
- Graafikaprotsessor

See situatsioon on väga ebasoodne nutirakenduste arendajatele, sest rakendusi või mänge arendades peavad nad silmas pidama seadmete iseärasusi.

Käesolev lõputöö analüüsib probleemi, leides ülesse peamised takistuskohad nutitelefonidele mõeldud rakenduste arendamisel. Lõputöö käigus luuakse süsteem, kus tudengid saavad enda kirjutatud rakendusi testida erinevate seadmete peal ning saada tagasisidet. Lahendus hõlmab endas Javas kirjutatud programmi, mis orkestreerib seadmeid ning lisaks toetav CI/CD süsteem, mis hõlbustab ja automatiseerib kogu protsessi.

Lõputöö autor on töötanud mitu aastat Android platvormi arendajana ning lisaks olnud ka Tallinna Tehnikaülikoolis abiõppejõud aines “Platvormi põhised mobiilirakendused” (ICD0017). Autor tunnetas probleemi tudengite tööde hindamisel. Selleks, et testida tudengi rakendust ning anda tema tehtud tööle hinnang, pidi tegema antud sammud:

- Tudengi koodi alla laadimine arvutisse
- Koodi kompileerimine
- Telefoni emulaatorile/simulaatorile aretafakti installeerimine
- Tahvelarvuti emulaatorile/simulaatorile artefakti installeerimine

- Vaadete testimine
- Tagasiside andmine tudengile

Ülaltoodud nimekiri oli protseduuridest, mida pidi tegema iga tudengi rakenduse hindamiseks.

Lisaks on autorile huvi pakkunud erinevad CI/CD teenused. Autor lähtub eelkirjeldatud lõputöö süsteemi loomisel, enda kogemusel ning kooskõlastab loodud tulemuse Testlab autoriga, kelleks on Priit Pikk ning platvormi põhiste mobiilirakenduste õppejõu Andres Käveriga.

1.1 Metoodika

Käesoleva lõputöö käigus selgitatakse esmalt lahti eksisteerivat probleemi, eksisteerivaid lahendusi ja nende puudusi ning pakutakse välja probleemile sobiv IT lahendus, mis jääks lõputöö skoopi ning mille arendus ka realiseeritakse.

Lahenduse analüüsi käigus tuuakse välja erinevad kasutajagrupid ning neid puudutavad nõuded kasutajaloode kujul, mis on grupeeritud funktsionaalseteks ja mittefunktsionaalseteks nõueteks. Samuti analüüsitakse erinevaid tehnilisi lahendusi ning põhjendatakse tehtud valikuid.

Lahenduse valmimist on kirjeldatud erinevate osadena, mis käsitlevad süsteemi loomist ja kasutamist. Lahenduse käigu lõpu osas on kirjeldatud ka edasiseid samme väljaspool lõputöö skoopi.

1.2 Ülevaade probleemist

Peamiselt on maailmas hetkel kaks üldsusele tuntud mobiiliplatvormi Android ja IOS. Androidi turuosa on täna hinnanguliselt 71.9%, IOS 27.33%, alla ühe protsendi moodustavad ka muud platvormid, kuid antud lõputöö neid seadmeid ei käsitle [2].

Esimene probleem, mis koheselt tekib on see, et rakendus on vaja kirjutada kahele platvormile. Maailmas on hetkel tõusutrendil ristplatvorm tüüpi rakendused, mis proovivad lahendada ära probleemi, kus on vaja hallata kahte koodibaasi, ehk üks koodibaas, mis on kirjutatud Androidile ja teine IOSile. Kuid antud lõputöö käsitleb eelkõige *native* rakenduste testimist.

Teine probleem on eelkõige tulenev erinevatest seadme mõõtudest, suudetakse küll eristada erinevaid klasse, nagu: nutitelefone ja tahvelarvuteid, kuid kahjuks erinevad ka igas klassis olevate seadmete ekraani mõõdud üksteistest. Ekraanimõõtude erinevus teeb kasutajaliideste disainimise keeruliseks, sest ühe mõõduga midagi teha ei saa. Arendaja peab reeglina tegema kaks eraldi vaadet, üks siis telefoni vaatele ja teine tahvelarvuti vaatele. Kuid ka seal võivad tulla mängu erinevad tehnilised parameetrid nagu ekraani pikkus, laius, diagonaal ja pikslitihedus. Sinna hulka tuleb arvata ka tingimus, et mõne rakenduse loomisel on oluline teha režiim vaegnägijatele, mis omakorda vajab rohkem testimist.

Kui eelnevalt loetletud probleeme saab lahendada lihtsalt. Android seadmeid emuleerides ja IOS seadmeid simuleerides, siis väljakutset esitab Androidi platvorm. Väga paljud tootjad modifitseerivad Android operatsioonisüsteemi alust, ehitades selle peale oma operatsioonisüsteemi, mis võib segada näiteks mõne spetsiifilisema rakenduse või mängu loomist.

Emulaator lubab simuleerida Androidi seadet arvutis, kus kasutaja saab testida rakendusi, ilma et tal oleks selleks vaja füüsilist seadet [3]. Hetkel ei ole võimalik emulaatoris kasutada näiteks järgnevaid funktsioone: Sinihammas, NFC, mälukaart, USB ja kõrvaklappe [4].

IOSi maailmas proovitakse täna seadmeid simuleerida. Peamine vahe emuleerimisel ja simuleerimisel seisneb selles, et emulaatorid matkivad tarkvara ja riistvara, küll aga simulaatorid matkivad ainult tarkvara. Täna ei ole selget vastust, miks Android valis

emuleerise, kuid arvatakse, et see põhjus oli selles, kuna Android operatsioonisüsteem on mõeldud erineva riistvaraga seadmetel, küll aga on IOSil tegemist kindla riistvaraga.

IOSi maailmas on simuleerimisel rohkem puudusi, kuna kasutaja ei pääse ise ligi riistvara parameetritele. Peamised piirangud on erinevatel raamistikel põhinevate rakenduste käivitamine, näiteks puudus GPS toele ja paljule muule, mis on antud viites välja toodud [5]. See kõik selgitab, miks on vajalik füüsiliste seadmete olemasolu.

Ülal kirjeldatud probleeme lahendatakse täna nutitelefonide testimise farmide olemasolul [6]. Millest tuleb järgmises peatükis juttu.

1.3 Eksisteerivad lahendused

1.3.1 Pilves olevad testimise farmid

Pea igal suuremal pilveteenusied pakkuval ettevõtetel on täna olemas nii öelda testimise farmid. Sisuliselt on tegemist suurte ruumidega, kus asetsevad kapid, mis on täidetud nutitelefonidega ning vastava liidese kaudu saab neile ka ligi. Täna pakkuvad sellist lahendust näiteks: Samsung, Google, Amazon. Liidese kaudu on võimalik oma ettevõtte arendusprotsess väga lihtsasti ühendada. Näiteks kui arendaja soovib oma koodi testida, piisab tal ainult vastav artefakt läbi liidese üles laadida ning tulemused jõuavad kohale üsna pea. Selle lahenduse puudus on füüsiline ligipääs seadmetele, sest teatud juhtudel on vaja ka seadmeid ise käes hoida. Näiteks GPS signaalist või mobiili levist sõltuva rakenduse testimine.

1.3.2 Seadmete emuleerimine ja simuleerimine

Võimalik on oma lokaalses arvutis käivitada mitu instantsi seadmetest, kuid see ei taga 100% kattuvust füüsilise seadmega. Seadmete emuleerimine/simuleerimine on ka väga ressursirohke tegevus. Kuna tegemist on graafilise liidesega, siis kurnavad need arvutit. Kuigi see protsess on läinud sujuvamaks, on nendes virtuaal seadmetes rakenduste testimine endiselt kohmakas. Jällegi ei asenda nad füüsilist seade, kui see peaks vajalik olema.

1.3.3 Raamistikud

Androidi maastikul on olemas raamistik, nimega “Roboelectric”, mis lahendab eelmises punktis kirjeldatud emulaatorite aeglust kompenseerida sellega, et teste käivitatakse JVMis [7]. Roboelectricu loojad lõid selle raamistiku, et seda oleks hea juurutada testimisel põhinevasse arendusse, sest emulaatorite käivitamine võtab aega. IOSis on täna testide käivitamine ilma graafilise liideseta vaikimisi lubatud ning kasutaja saab tulemust näha läbi käsurea. Taustal käivitatakse küll simulaatorid, kuid kasutaja ei näe graafilist liidest, mis aitab hoida kokku arvuti ressursside kasutamist ning katab võimalikult suure erinevate parameetritega seadmete hulga.

1.4 Uue lahenduse skoop

Olemasolevate lahenduste suurim piirang on füüsiline ligipääsmatus seadmetele, mis ei ole reeglina probleem, kui ei arendata väga spetsiifilist rakendust. Reeglina omab Android arendaja ise Android telefoni ja IOS arendaja omab iPhone. Probleem on aga selles, et aines “Platvormi põhilised mobiilirakendused” peab tudeng omandama mõlema platvormi arendamise oskused ning kahte seadet tudengitel reeglina ei ole. Eelpool toodud probleemi lahenduseks oli luua Tallinna Tehnikaülikooli seadmete labor “Testlab”.

Testlab on Tallinna Tehnikaülikooli neljandal korrusel asuv labor, kus asub kapp erinevate nutiseadmetega, millele tudengid ligipääsevad, kui nad soovivad oma rakendust füüsiliselt testida. Hetkel on seal 20 Android seadet ning 5 IOS seadet. Vanemad seadmed asuvad ka kastides, mida saavad tudengid kasutada Testlabi igapäevatööd segamata. Näiteks vahetada seadmete operatsioonisüsteemi või võtta seadmed tükkides lahti ning hiljem kokku panna. Mis oligi üks ajend Testlabi loomiseks, et ülikoolis oleks oma labor, kus saaks neid toiminguid teha. Lisaks ei saa olemasolevates testimisfarmides kasutada kuvatõmmiste testimist, kuna vastavate testide tegemiseks on vaja käsklusi jooksutada käsurealt. Labori seadmete parki üritatakse iga aastaselt täiendada, valides seadmeid, mis on nii Eestis ja mujal maailmas kõige rohkem kasutusel. Laborit näeb peatükis Lisa 2.

Antud diplomitöö pakub lahenduseks labori seadmetega ning teenuse, mis lubab *native* rakendusi testida kõikide laboris olevate Android ja IOS seadmete peal. Lahendus piirdub Javas kirjutatud orkestraatoriga ning Jenkins CI/CD serveriga, mis ühendab omavahel Tallinna Tehnikaülikoolis kasutuses olevaid tudengite koodirepositooriume Testlabiga.

Antud teenus on küll koodirepositooriumist sõltumatu, kuid kuna täna on Tallinna Tehnikaülikoolis standard Gitlab, siis käsitletakse selle kasutamist. Tulevikus on plaanitud luua ka API, mis lubaks kasutada ka teisi koodirepositooriume või siis laadida vastav artefakt kohe Testlabi.

Android rakenduste testimiseks käsitletakse kuvatõmmiste testimist ning kasutajaliidese testimiseks mõeldud raamistiku Espresso. IOSi puhul piirdutakse antud skoobis ainult kuvatõmmiste testimisest. Tulenevalt osapoolte soovist, saadetakse testimise tulemused, konkreetse kasutaja profiilis seadistatud meili peale, kui tudeng ei ole oma emailiks märkinud muud. Seda parameetrit saab tudeng CI/CD skriptist konfigureerida.

Testlabi eesmärgid on järgnevad:

- Luua terviklikum ülevaade testimisest ja selle vajalikkusest
- Anda ülevaade CI/CD süsteemide juurutamisest
- Seadmete laenutus
- Seadmetel rakenduste testimine kasutades raamistike
- Seadmete operatsioonisüsteemi modifitseerimine

Antud lõputöö, ei käsitle muid testimis raamistike, kuid kindlasti tekib tulevikus ka võimalus kasutada muid raamistike.

2 Teenuse analüüs

2.1 Nõuete määramine

Nõuete määramisel on eelkõige arvesse võetud labori juurutamist õppetöösse, kuigi laborit saab kasutada ka teiste ettevõtete ärivajadustes.

Nõuded põhinevad kasutajalugudel, kus on põhirollideks testija (*tudeng*) ja õppejõud, kaasa arvatud abiõppejõud ehk tulemuste hindajad.

Nõuded, mida antud diplomitöös hetkel ei käsitleta on väljatoodud töö lõpus.

2.1.1 Funktsionaalsed nõuded

Testija-põhised funktsionaalsed nõuded:

- Testijana soovin valida, millist tüüpi teste käivitan
- Testijana soovin, et testitav rakendus luuakse koodist automaatselt
- Testinaja soovin saada testimistulemusi omale e-mailile
- Testijana soovin ma laenata omale seadmeid füüsiliseks testimiseks
- Testijana soovin ma võimalust vahetada seadme operatsioonisüsteemi
- Õppejõuna soovin lihtsat ülevaadet tudengite testimiste tulemustest
- Testinajana soovin ma testimistulemused kätte saada ühtses ja lihtsas formaadis
- Testijana soovin ma võimalust kasutada Testlabi ka väljaspool Gitlabi
- Testijana soovin Testlab teenust käivitada Gitlabist

2.1.2 Mittefunktsionaalsed nõuded

Testija-põhised funktsionaalsed nõuded:

- Õppejõuna tahan laenutada seadmeid pikemaks ajaks
- Õppejõuna soovin näha statistikat labori kasutamisest
- Labor on füüsiliselt ligipääsetav selleks volitatud isikutel

- Labor on loodud avatud lähtekoodiga

2.1.3 Tulevikus loodavad funktsionaalsused

Käesoleva töö raames loodava lahenduse arendusmahust on jäetud välja järgmised funktsionaalsused. Siia hulka kuuluvad eelkõige funktsioonid:

- Õppejõuna soovin ma testimistulemusi näha automaatselt Moodle keskkonnast.
- Tudengina soovin ma laiemat valikut testimis raamistiku valimiseks.

2.2 Erinevad testid

Käesoleva töö raames loodav lahendus toetab järgmisi variante:

- Androidi kuvatõmmiste testid
- Android Espresso testid
- IOS kuvatõmmiste testid

IOS kasutajaliide testimine jääb arendavate funktsioonide mahust välja. Kuvatõmmiste testimise vajalikkus tuleneb õppeaine vajadustest. Hetkel on aine "Platvormi põhised mobiilirakendused" (ICD0017) 6 EAPi. Kus käsitletakse nii Android, kui IOS arendust vastavalt 3 EAPi ja 3 EAPi, sellest tulenevalt, ei käsitleta aines testimist kohustuslikus korras. Aasta 2021 sügisel muutub vastav aine 9 EAPliseks ning selle sees käsitletakse ka kasutajaliideste testide kirjutamist.

2.2.1 Kuvatõmmiste testid

Kuvatõmmiste testid (*screenshots test*) on Testlabi teenus, mis ei eeldata kasutaja poolset testi kirjutamist. Automaattestimise käigus rakendust keeratakse ja tehakse kuvatõmmised automaatselt. Antud õppeaine raames kirjutatakse 3 Android ja IOSi rakendust, millest 2 on reeglina ainult ühe vaate põhised rakendused, reeglina on selleks mõni mäng. Antud testide puhul installitakse rakendus kõikidesse laboris leitavatasse seadmetesse, tehakse kuvatõmmis, keeratakse seadmeid 90° kraadi ning tehakse uuesti kuvatõmmis. See test annab tudengile hea ülevaate, kuidas tema kirjutatud rakendus või mäng näeb erinevates seadmetes ning erineva ekraani orientatsiooniga välja. Ühtlasi on

see väga oluline testimiseks, kas tudengi poolt kirjutatud rakendus või mäng üldse kõikidesse seadmetesse installeerub. Android raport (Lisa 3). IOS raport (Lisa 4) .

2.2.2 Espresso testid

Espresso on kasutaja liidese testimise raamistik, mis on mõeldud Androidi rakenduste testimiseks [8]. Espresso puhul on tegemist küllaltki lihtsa raamistikuga, mis lubab testida UI elemente. Android Studios on võimalik ka teste lindistada, ehk kasutaja saab vajalikud UI elemendid läbi vajutada, mida soovib testida ning sellest tehakse automaatselt kood. Koodi saab seadmete peal käivitada. Antud skoobi raames kasutatakse teeki, nimega Spoon [9]. Spoon käivitab Espresso teste seadmete peal üksteisest sõltumatult paralleelselt ning loob ka nendest ülevaate.

2.3 Tehnoloogia valik

Tehnoloogia valik tehti eelkõige autori poolsest kogemusest Java programmeerimiskeelega. Kuna enamus Android rakendusi on kirjutatud Javas, leidus häid teeke, mida programmis kasutada.. Testlab rakenduse koodi kirjutamisel lähtus autor Effective Java 3rd Edition väljapakutud nippidest ja kooditavadest. Lisaks hõlmab kirjutatud rakendus ka käsurea skripteerimist, mis oli oluline osa suhtlemaks teatud operatsioonisüsteemi tasandil protsessidega, väga palju sai seda kasutatud, et luua käsurea skripte, millel on oluline osa IOS rakenduste kompileerimisel. Kuna autori kogemus oli varasem Android rakenduste kirjutamisega sai tehtud ka automatiseerimis tööriista valik Gradle kasuks. Mis on hetkel Android maailmas tavaks.

2.4 Arenduskeskkonna valik

Arenduskeskkonna valikud võib jagada järgnevalt: koodihaldus, koodi arenduskeskkond ning CI/CD server.

Koodirepositooriumiks valiti TalTechis kasutav Gitlab. Valikut toetas Testlabi jaoks vajalike funktsionaalsuste ja integratsiooni võimekuste olemasolu Gitlabis. Sarnane valik sai tehtud ka valides IntelliJ integreeritud programmeerimise keskkonnaks [10]. Sellel on Tallinna Tehnikaülikoolis kasutamiseks olemas litsents.

CI/CD serveriks sai valitud Jenkins [11]. CI/CD serveri valiku tegemine oli lõputöö kirjeldatud skoobis üks raskemaid valikud, seda nimelt põhjusel, et algul sai kasutatud Gitlabi enda CI/CD serverit, põhjusena mitte ajada tehnoloogia pinu liiga suureks. Kuid töökäigus ilmnesid puudused, et antud lahendusse see ei sobi. Peamine põhjus oli turvalisuse küsimus, sest Gitlabi enda featuurid ei pakkunud piisavalt turvalisust. Gitlabi ametlik leht kinnitab, et *shell executor* tüüpi on mõtet kasutada, kui koodi käivitatakse turvalises keskkonnas [12]. Probleem oli selles, et tudengitel oli ligipääs serveri käsureale. Kasutajale käsurea loa andmine on aga turvarisk. Antud keskkonnas olid nad operatsioonisüsteemi õigustega. Gitlabi ametlik leht soovitab kasutada Docker tüüpi lahendusi, kuid antud lahendusse ei sobinud see sellepärast, et teatud protsesse oli vaja kasutada operatsioonisüsteemi tasandil ning see on vastupidine sellega, mida Docker teha üritab. Koodinäitest 1 on näha *script* sektsioon, kus kutsutakse välja Tesltabi põhiprogramm, mis tegeleb testide käivitamise ja orkerestreerimisega. Kuna see fail asub iga tudengi koodi hoidlas, on neil võimalus seda sektsiooni ise muuta. Kogu kompileerimis protsess on eraldatud Dockeri konteineris, piirduvad tudengi õigused ainult konteineris olevate protsessidega ning tudengi tehtud muudatused, ei kajastu väljaspool konkreetset konteinerit.

Jenkins lahendab ära selle probleemi, et kõik detailid saab kasutaja eest ära peita. Kasutaja peab vastavat URLi ainult välja kutsuma ning andma kaasa paar parameetrit, et CI/CD server käivitaks teste.

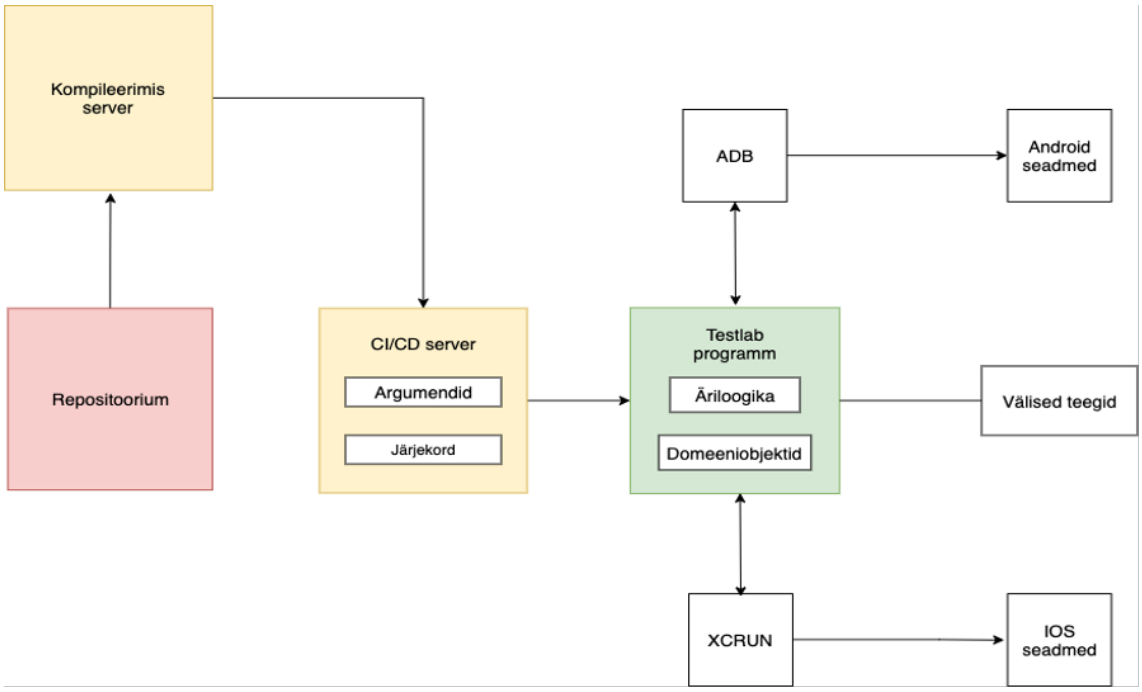
```
install:
  variables:
    GIT_STRATEGY: none
  stage: test
  dependencies:
    - assemble
  tags:
    - testlabRunner
  script:
    - java -jar $SPOON_RUNNER_PATH --apk
"$BUILDS_PATH/app/build/outputs/apk/debug/app-debug.apk" --debugApk
"$BUILDS_PATH/app/build/outputs/apk/androidTest/debug/app-debug-
androidTest.apk" --outputFolderPath "$BUILDS_PATH/outputs/" --api "0" --
report "$REPORT_TYPE"

  artifacts:
    paths:
      - $BUILDS_PATH/outputs
```

Koodinäide 1. Lõik CI/CD skriptist

2.5 Arhitektuur

Rakenduse arhitektuur koosneb pilve serverist, kus testija kood kompileeritakse artefaktiks. Peale seda edastatakse see GET päringuga Jenkins CI/CD serverisse, mis tegeleb järjekorra hoidmisega. Korraga saab käivitada ainult ühe kasutaja teste, kuna kõik seadmed on hõivatud konkreetse testi käivitamisega hõivatud. Peale seda käivitatakse kasutaja poolt etteantud testid seadmetel ning testija saab tulemused meili peale.



Joonis 1. Testlab arhitektuur

Tudengi jaoks piisab selle süsteemi käivitamiseks, kui ta lisab vastava skripti enda koodi hoidlasse. Peatükis 5 on vastav skript koos kommentaaridega välja toodud. Antud skript on jagatud kahte sektsiooni, millest esimene tegeleb koodist artefakti loomisega ning teine sektsioon tegeleb artefakti edastamisega CI/CD serveriga, mis omakorda edastab vajaliku info koos argumentidega Testlab programmi, mis tegeleb testide jooksutamisega. Antud koodilõigus on seadistatud süsteem nii, et peale koodi üleslaadimist koodihoidlasse, peab tudeng selle skripti käivitama käsitsi, see on mõeldud sellepärast, et peale igat *push*-i ei koormataks üle laborit ning serverit. Tudengile on antud vabadus vastavat skripti ka muuta, kuid kommentaaridena on antud märku, mida ja kuidas oleks sobilik muuta. Antud skript on küll defineeritud kahe etapina, kuid seda on võimalik teha ka ühe etapina, kui selleks peaks olema soovi.

2.6 Analüüsi kokkuvõte

Analüüsis sai käsitletud erinevaid kasutajagruppe ning nende kohta käivaid funktsionaalseid ja mittefunktsionaalseid nõudeid. Lisaks sai välja pakutud tulevasi arendusi ja kirjeldatud arhitektuuri.

Tehnoloogia poole pealt sai eelistatud Android platvormile omaseid tehnoloogiaid. Mis tulenes eelkõige autori poolsetest kogemusest Androidiga. Antud arhitektuuri loomisel sai silmas peetud ka platvormi sõltumatust, taoline lahendus ja arhitektuur peaks töötama väljaspool arenduskeskkonda. Antud juhul sai seda arendatud Mac operatsioonisüsteemi peal, kuid taoline lahendus peaks töötama ükskõik, millise Linux arhitektuuri peal (välja arvatud IOS testimine), kui on olemas vajalikud teegid.

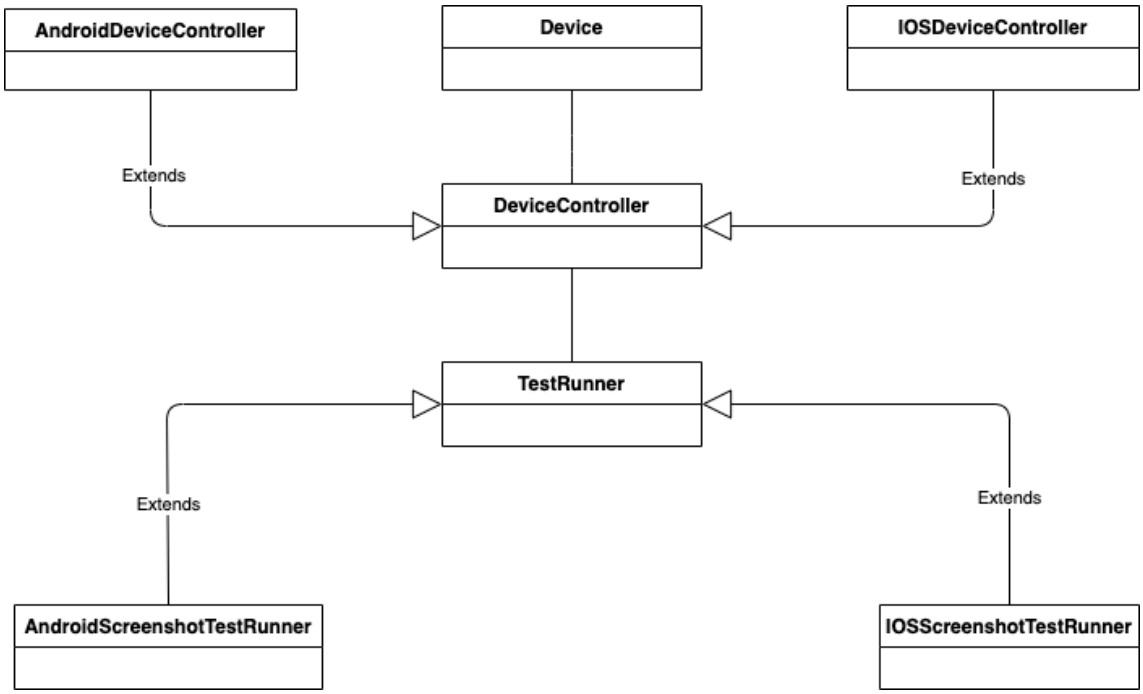
Koodi kirjutatakse IntelliJ IDEA programmis, koodihoidlaks kasutatakse Gitlabi, peamine arenduskeel on Java ning CI/CD serveriks kasutatakse Jenkinsi.

3 Programmi arendus

Arendus on jaotatud kolme suuremasse peatükki: Androidi orkestraator, IOSi orkestraator ning CI/CD serveri ühildumine programmiga.

3.1 Sissejuhatus peatükki

Programmi arendamisel sai eelkõige silmas peetud selle sobivust õppetöösse. Programm "*testlab-runner*" on loodud kasutamiseks nii IOS, kui Androidi puhul. Seda sellepärast, et neil on väga palju ühisosa abstraktse klassi ning utiliit klasside näol. Joonisel 2 on näha klassidiagrammi.



Joonis 2. Klassidiagramm

Ühisosa on abstraktne klass, nimega "Device", sisuliselt loeb programm nii Android, kui IOS seadmete info programmi ning konverteerib nad "Device" klassiks. Device klass koosneb isendiväljadest:

- UUID
- name

- apiLevel
- deviceClass

Loetletud väljad on sama ühisosa, mis on nii Android kui IOS seadmetel. DeviceClass defineerib, kas tegemist on tahvelarvutiga või telefoniga. Androidi puhul ei ole väli kasutuses, kuid oluline on see IOSi puhul, et teada milline artefakt seadmele installeerida. Lisaks kasutatakse seda klassi, et genereerida PDF tüüpi raportid. Programmi peamine ülesanne on sisse lugeda kasutaja poolt sisestatud argumente, selle põhjal teha valikud, mis platvormi ja milliseid teste käivitada. Peale seda jõuda vastava tulemuseni, milleni kasutaja jõuda soovib. Väga palju argumente peab kasutaja ise ette andma, et tagada programmi sõltumatus operatsioonisüsteemist. Autor toob välja tähtsamad koodilõigud, kuid mitte kõike. Vastav lähtekood on saadaval ka lisades. IOSi peatükis käsitletakse ka kompileerimise faasi. Seda ei tehta Androidi juures, sest vastav artefakt antakse ette juba ette. Lisaks ei käsitleta erinevaid utiliit meetodeid, sest need on standardsed koodilõigud, mis sobivad oma olemuselt igasse programmi, sõltumata konkreetsest ülesandest.

Autor on üritanud koodis hoida ühtset struktuuri, ehk erinevad paketid sisaldavat sama struktuuri erinevatele seadmetele. Mõlemates pakettides on olemas klassid, mis tegelevad seadmete lugemisega, seadmete kontrollimisega ning klassid, kus on kirjeldatud, mis samme tehakse iga seadme kohta asünkroonselt. Pakettides asub ka kontroller klass, mis sisaldab erinevaid meetodeid, mida kutsutakse välja seadmete peal. Seal sisalduvad meetodid oskavad suhelda konkreetse seadme meetodeid välja kutsuda. Need on maskeeritud kasutama vastavaid APIsid või siis kutsutakse välja käsurealt koodilõike seadmetega suhtlemiseks.

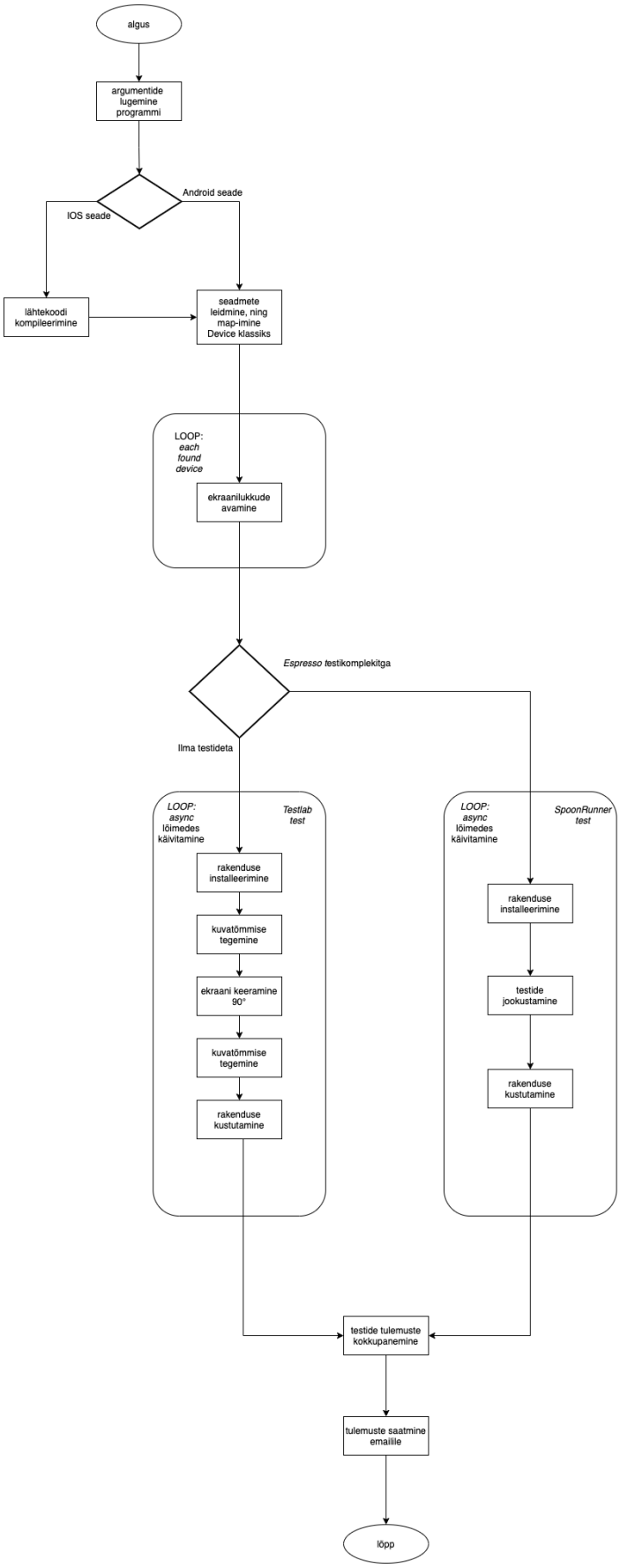
Sama struktuur on ka runner tüüpi klasside puhul. Vastav runner klass omab endas teadmisi, milliseid käske on vaja jooksutada, et saada vastav tulemus. Arhitektuur on loodud pidades silmas tulevasi platvorme, kui neid peaks juurde tekkima. Siis piisab ainult luua uus runner klass, mis laiendab TestRunner klassi.

Emailide saatmiseks kasutab autor Javasse sisse ehitatud paketti, milleks on *javax.mail*. Argumenditest loetakse sisse Gmaili argumendid ning need sisestatakse vastavasse klienti. Ekraani kuvatõmmiste raportite puhul saadetakse välja PDF raport. Kuna SpoonRunner genereerib palju rohkem infot, siis selle edastamiseks on vaja kasutada failide salvestamise hoidlat . Antud valik sai tehtud Google Drive kasuks. Espresso

testidest tulenev sisu, pakitakse kokku ning pakitud fail laetakse ülesse Google Drivesse. Vastav link saadetakse tudengile emaili peale, kus ta saab ka lühikese ülevaate teksti kujul, kus on kirjas kui palju seadmeid on laboris, kui palju teste õnnestus ning kui palju ebaõnnestus. Google Drivesse laetakse, aga SpoonRunneri poolt genereeritud pakitud fail, mis sisaldab infot:

- Logid
- Ülevaade
- Testimiseks kulunud aeg
- Seadmete ülevaade

Joonisel 3 on näha, milline näeb Testlab programmi kood välja blokkdiagrammina. Teatuid portsesse tehakse kõikidel seadmetel ning need on ka antud graafikul väljatoodud.



Joonis 3. Testlab programmi visuaalne selgitus

Rakenduse alguspunktiks on Main.java klass, kus loetakse sisse argumendid ning vastavalt esimesele argumendile tehakse esimene valik, millise platvormi peal test käivitada. Antud hetkel on tegemist esimese etapiga ning kogu programm käivitatakse käsurea pealt, ehk antud programmi käivitab Jenkinsi töö. Tulevikus on plaanis see teha kasutatavaks veebiteenusena.

```
public class Main {
   static Logger log = Logger.getLogger(Main.class.getName());

   public static void main(String[] args) throws InterruptedException,
GeneralSecurityException, IOException {

       CLIArgs commandLineArguments = new CLIArgs(args);
       log.info(commandLineArguments.toString());

       switch (commandLineArguments.getPlatform()) {
           case ANDROID:
               AndroidRunner.start(commandLineArguments);
               break;
           case IOS:
               IOSRunner.start(commandLineArguments);
               break;
           default:
               throw new RuntimeException("Platform is not implemented
yet");
       }
   }
}
```

Koodinäide 2. Main klass

3.2 Android orkestraator

Android orkestraatori loomine oli kõige väljakutsuvam osa, just kuvatõmmiste testide koodibaas. Kuna laboris on erineva operatsioonisüsteemiga seadmeid, siis pidi tegema koodilõike, mis töötaksid kõikide seadmete peal olenemata SDK versioonist. Androidi puhul on olemas kahte tüüpi testimist. Kuvatõmmiste testi kirjutas autor ise. Espresso testide jaoks aga käivitatakse väline teek. Antud lõputöö skoobis ei vaadelda Spoon teeki süvitsi. Kuid sisuliselt on tegemist .jar teegiga, millele tuleb ette anda kaks Androidi artefakti teekonda, SDK teekond ja teekond, kuhu genereeritakse raport. Peale selle käivitamist käivitatakse seadmete peal teste paralleelselt ning väljastatakse vastav aruanne, mis saadetakse tudengi profiilis märgitud aadressile.

Kuvatõmmiste testimine vajab rohkem koodi kirjutamist, sest asünkroonsete lõimude kontrollimiseks vajalik kood tuleb ise implementeerida. Järgnevalt on kirjeldatud, kuidas leitud seadmete peal pannakse käima lõim, mis tegeleb vastavate protsessidega:

- Avatakse seadme ekraan
- Installeeritakse vastav rakendus
- Avatakse rakendus
- Tehakse kuvatõmmis
- Keeratakse ekraani 90 kraadi võrra
- Tehakse kuvatõmmis
- Laetakse rakendus maha
- Ekraan suletakse

Antud paralleelsusest saavutan kasutades Java guru Joshua Bloch raamatus pakutud lahendust. Seal on soovitatud kasutada *newCachedThreadPool*-i, kui tegemist on lihtsa ülesandega. Ning lõimudest, ei looda järjekorda [13]. Antud koodilõigus on näha, et iga seadme kohta pannakse käima eraldiseisev lõim, mis tegeleb testimisega. Ning hiljem kui kõik lõimud on oma töö lõpetanud liigutakse edasi järgmise koodilõigu juurde, mille ülesanne on testimistulemused kokku panna ning need edastada.

```
//Effective Java item 68 (https://github.com/HugoMatilla/Effective-JAVA-
Summary#68-prefer-executors-and-tasks-to-threads)
ExecutorService executorService = Executors.newCachedThreadPool();

foundDevices.parallelStream()
        .forEach(device -> executorService.execute(new
AndroidScreenshotTestRunner(new AndroidDeviceController(device,
commandArgs.getApkPath(), commandArgs.getAdbPath(),
commandArgs.getOutputFolderPath(), packageName, debugPackageName))));

executorService.shutdown();
```

Koodinäide 3. Paralleeluse saavutamine

Järgnev koodilõik kirjeldab, milline näeb välja lõim. Igas leitud seadmes käivitatakse neid käsklusi. DeviceController klass sisaldab meetodeid, mida saab igas seadmes välja kutsuda. Seal suheldakse läbi ADB kasutades käsurida. ADB on käsurea tööriist, mis võimaldab seadmetega suhelda [14]. Kuna Androidi operatsioonisüsteem kasutab Linuxi kerneli, saab ka iga seadmega suhelda eraldi läbi käsurea. Sisuliselt on paljud ADB käsud

maskeeritud arendaja jaoks API liideseks, mis võimaldab saada paremat tagasisidet ning tegeleda veateadetega, kui seda lugeda otse käsurealt

```
public class AndroidScreenshotTestRunner extends DeviceRunner {

    public AndroidScreenshotTestRunner(DeviceController deviceController) {
        super(deviceController);
    }

    @Override
    public void runTest() {
        deviceController.wakeUpScreen();

        deviceController.installApp(Optional.empty());
        deviceController.openApp();

        deviceController.takeScreenshot(Orientation.PORTRAIT);
        deviceController.rotateScreen();
        deviceController.takeScreenshot(Orientation.LANDSCAPE);

        deviceController.uninstallApp();
        deviceController.closeScreen();
    }
}
```

Koodinäide 4. Android kuvatõmmise lõimu töö

3.3 IOS orkestraator

IOS orkestraatori peamiseks probleemiks oli füüsiliste seadmetega suhtlemine. Lisaks suhtlemisele peab ka iga IOS seadme unikaalse ID registreerima vastavas portaalis. Kuna antud skoop käsitleb kuvatõmmiste teste, siis erinevalt Androidist ei ole IOS maailmas olemas nii head liidest. ADB ekvivalent on IOS maailmas täna XCRUN [15]. Kõige suurem probleem oli seadmete keeramine. Seda on hetkel võimalik teha IOS simulaatorite peal, kuid puudub võimalus anda see käsklus ette füüsilistele seadmetele. Selle probleemi lahendamiseks manipuleeritakse failiga “Info.plist” [16]. Tegemist on võti väärtus paaride failiga, mis sisaldab erinevaid metaandmeid rakenduse kohta. Ekvivalent Androidi maailmas AndroidManifest.xml faili kohta. Üks selline võti väärtus paar defineerib ka selle, kas rakendus tuleks käivitada vertikaalses või püsti asendis. Selleks, et seda faili muuta kasutab autor tööriista nimega Plutil, mis on lihtne käsurea utiliit muutmaks võti väärtus paare [17]. Antud failis muudetakse erinevaid parameetreid põhjusel, et need on erinevad telefonide ja tahvelarvutite peal. Koodist kompileeritakse järgnevad tulemid:

- iPhonele loodud vertikaalne artefakt
- iPhonele loodud horisontaalne artefakt
- iPadile loodud vertikaalne artefakt
- iPadile loodud horisontaalne artefakt

Peale mida toimub sama protsess, mis Androidi puhul. Rakendused installeeritakse, tehakse kuvatõmmised ning rakendus laetakse maha. Koodinäitest 5 on näha, kuidas rakendus dünaamiliselt valmis ehitatakse. Sisuliselt antakse ette DeviceClass klass, mis on siis kas iPhone või iPad ning orientatsioon. Kas tegemist on vertikaalses asendis oleva artefaktiga või horisontaalses asendis. Dünaamiliselt ehitatakse valmis vastav skript, mis siis käsureal välja kutsutakse.

```
public void buildAppFor(IOSDeviceClass iosDeviceClass, Orientation
orientation) {
    StringBuilder script = new StringBuilder();
    if (iosDeviceClass.equals(IOSDeviceClass.IPHONE)) {
        script.append(String.format("plutil -replace
UISupportedInterfaceOrientations~ipad -json '[]' %s%s/Info.plist\n",
projectPath, scheme));
        if (orientation.equals(Orientation.PORTRAIT)) {
            script.append(String.format("plutil -replace
UISupportedInterfaceOrientations -json
'[\"UIInterfaceOrientationPortrait\"]' %s%s/Info.plist\n", projectPath,
scheme));
        } else {
            script.append(String.format("plutil -replace
UISupportedInterfaceOrientations -json
'[\"UIInterfaceOrientationLandscapeRight\"]' %s%s/Info.plist\n",
projectPath, scheme));
        }
    } else {
        //For some reason for Ipad this line needs to be removed from
"Info.plist" so that screenshots would work
        script.append(String.format("plutil -remove
UIApplicationSceneManifest %s%s/Info.plist\n", projectPath, scheme));
        script.append(String.format("plutil -replace
UISupportedInterfaceOrientations -json '[]' %s%s/Info.plist\n",
projectPath, scheme));
        if (orientation.equals(Orientation.PORTRAIT)) {
            script.append(String.format("plutil -replace
UISupportedInterfaceOrientations~ipad -json
'[\"UIInterfaceOrientationPortrait\"]' %s%s/Info.plist\n", projectPath,
scheme));
        } else {
            script.append(String.format("plutil -replace
UISupportedInterfaceOrientations~ipad -json
'[\"UIInterfaceOrientationLandscapeRight\"]' %s%s/Info.plist\n",
projectPath, scheme));
        }
    }
    script.append(String.format("xcrun xcodebuild clean -project
%s%s.xcodeproj -scheme %s| xcpretty\n", projectPath, scheme, scheme));
    script.append(String.format("xcrun xcodebuild -project %s%s.xcodeproj -
scheme %s DEVELOPMENT_TEAM=%s SYMROOT=%sbuild/%s/%s  build | xcpretty\n",
projectPath, scheme, scheme, developmentTeamId, projectPath,
iosDeviceClass, orientation));

    ProcessBuilder pb = new ProcessBuilder("/bin/bash");
    try {
        Process bash = pb.start();
        PrintStream ps = new PrintStream(bash.getOutputStream());
        ps.println(script);
        ps.close();
        bash.waitFor();
    } catch (IOException | InterruptedException e) {
        log.error("An exception!", e);
    }
}
```

Koodinäide 5. Dünaamiline artefakti ehitus IOSile

Täpselt nagu Androidi seadme lõime puhul tehakse samad protsessid ka IOSi puhul läbi. Androidi puhul installeeritakse rakendus seadmesse ühe korra, kuid kuna IOSi puhul defineerib seadme ekraani orientatsiooni artefakt, siis installeeritakse iga kuvatõmmise jaoks vastav artefakt eraldi ning laetakse see hiljem maha.

```
public class IOSScreenshotTestRunner extends DeviceRunner {

    public IOSScreenshotTestRunner(DeviceController deviceController) {
        super(deviceController);
    }

    @Override
    public void runTest() {
        deviceController.installApp(Optional.of(Orientation.PORTRAIT));
        deviceController.takeScreenshot(Orientation.PORTRAIT);
        deviceController.uninstallApp();

        //Rotation
        deviceController.installApp(Optional.of(Orientation.LANDSCAPE));
        deviceController.takeScreenshot(Orientation.LANDSCAPE);
        deviceController.uninstallApp();
    }

}
```

Koodinäide 6. IOS kuvatõmmise lõimu töö

3.4 Jenkins

Jenkinsi kasuks tehtud valikut, kirjeldas autor juba arenduskeskkonna peatüki all. Antud peatüki all kirjeldab autor, kuidas Jenkins on kogu protsessi sisse juurutatud ning mis on selle ülessane.

Jenkins on avatud lähtekoodil baseeruv CI/CD server [18]. Selle ülesanne on manageerida ning käivitada erinevaid arendusest tulenevaid töid. Jenkinsi all on defineeritud kaks *pipeline*. Üks on mõeldud teenindamaks Androidi töid ja teine IOSi omasid. Sisuliselt on tegemist funktsiooniga, mis kuulab vastava URLi pealt päringuid ning lisab iga päringu järjekorda ning serveerib ükshaaval tehtavaid ülesandeid. Igasse konveier meetodisse on lisatud ka argumendid, mis on vastavad keskkonna väärtused, mida vajab Testlab programm töötamiseks. Lõppkasutaja, ehk tudengi eest on need ära peidetud ning neid

saab muuta ainult administraator. Tudengi CI/CD skript annab ette ainult, millist tüüpi testi soovitakse käivitada ning millisele e-mailile antud tulemused tagastatakse. Selleks kasutatakse Curli [19]. Curl on klient, mis lubab saata andmeid, kasutades erinevaid protokolle. Antud juhul tehakse GET päring Jenkinsi URLi peale, kasutades lippu "-F", mille kaudu edastatakse vastavad argumendid Jenkinsile.

Jenkinsis on olemas ka funktsioon manuaalseks testimiseks, mis lubab URLile etteantavad argumendid ka anda sisse graafilise liidese kaudu, peale mida on ka vastav järjekord jooksvatest töödest näha ning kui midagi peaks minema valesti, on vastav logi ka kohe käepärast. See annab võimaluse ka administraatoril sekkuda ehk siis parandada vastav *pipeline* ära, kui selleks tekib tahtmine. Tudeng ehk lõppkasutaja seda logi ei näe, sest kui tema rakenduses on sees vead kajastuvad need Espresso raportis.

3.5 Edasised arendused

Ülal kirjeldatu on vaid osa funktsioonidest, mida Testlab hetkel pakub. Tulevikus on plaanis ka luua virtuaalne labor, kus tudengid saavad üle interneti seadmeid juhtida. Tudengile jääb mulje nagu oleks tegu virtuaalse seadmega, kuid tegelikud tegevused käivitatakse, aga samade füüsiliste seadmete peal. Antud skoobis seda ei käsitleta, kuna see funktsionaalsus alles valmib. Planeeritava lahenduse nimi on openSTF [20].

Lisaks on ka mõeldud selle peale, et kui peaks juhtuma olek, kus seadmed on päevasel ajal kasutuses ja ei ole ühendatud laadimiskaabli kaudu laboriga. Siis võib tekkida probleem, kus tudeng on parasjagu käivitamas kuvatõmmiste või Espresso teste ning vastavaid seadmeid ei ole laboris saadaval. Sellest tulenevalt võib saadav raport tulla puudulik. Selleks on planeeritud lahendus, mis hõlmab Jenkinsi CRON lahenduse kasutamist. Sisuliselt pannakse tööle taimer, mis käivitab teste öösel, kui seadmed on laboris tagasi.

Plaanimisjärgus on ka loodav funktsionaalsus käivitamaks IOSis kasutajaliidese teste, kasutades ekvivalentset teeki SpoonRunnerilele. Kuid kuna Testlabi arvuti on hetkel kasutuses ka IOS rakenduse kompileerimis etapina, siis tuleks tulevikus see protsess viia kuskile mujale, et mitte koormata arvutit testide käivitamise ajal. Samamoodi nagu Androidile on ka IOSile planeeritud luua virtuaalne keskkond, kust saab üle interneti seadmetele ligi.

Õppejõudude ja abi õppejõudude abistamiseks on planeerimis järgus ka API liidese kasutamine Moodlega. Kui tudeng on oma loodud tulemusega rahul saadetakse vastav raport otse Moodle keskkonda, kus siis õppejõud saab vastavat raportit hinnata või lisada sellele kommentaari.

4 Kokkuvõte

Antud diplomitöö eesmärk oli luua nutitelefonide rakenduste ja mängude testimiseks mõeldud labor. Töö kirjutamisel lähtuti eelkõige õppeaine “Platvormi põhised mobiilirakendused” sisust ning saadav tulemus kooskõlastati antud õppeaine õppejõuga ning teiste selle õppeainega seotud osapooltega.

Lõputöö raames analüüsiti olemasolevaid lahendusi ning võrreldi neid omavahel. Analüüsi käigus tuli välja, et ühte konkreetset lahendust ei ole olemas, mis tagaks kiire testimis lahenduse ning samal ajal üritaks ka katta võimalikult palju seadmeid.

Töö arhitektuuri analüüsi osa hõlmas endas tudengite koodihoidlate sidumist Jenkinsi CI/CD serveriga ning vastavate testimistulemuste raporti kätte toimetamist. Antud osas käsitleti erinevate programmide ja teekide koostööd ning vastav osa esitati ka blokkdiagrammi näol. Lisaks analüüsiti vastavaid turvariske, mis olid seotud CI/CD serveri käivitamisega.

Arenduse skoop hõlmas endas Java programmi, mille kaudu saavad tudengid käivitada teste füüsilistel seadmetel ning seadmeid ka vajadusel laenutada. Programm tegeleb käsurea argumentide tõlgendamisega, seadmete orkestreerimisega ning siis vastava raporti väljastamisega tudengi emaili peale. Tähtsamad koodilõigud toodi ka kirjalikult välja.

Tänu sellele diplomitööle on palju käsitsi tehtavat tööd tehtud automaatseks. Sellest tulenevalt on paranenud ka õppeaine kvaliteet ning ka hindamisemetoodika. Õppeaine läbinud tudengil on baas teadmised testimisest ning CI/CD skripti kirjutamisest. Lisaks tekib tudengile füüsiline kokkupuude erinevate seadmetega, mis paneb teda mõistma seadmete iseärasustest.

Projekti võib lugeda õnnestunuks, sest vastav lahendus on implementeeritud, vastavate osapooltega kooskõlastatud ja kasutatud täna ka põgusalt õppetöös. Välja on toodud ka lahendused tulevikuks, mille kallal autor hetkel töötamas on.

Kasutatud kirjandus

[1] Statista, “Number of smartphone users worldwide from 2016 to 2023,” Jaanuar 2021. [Online]. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[2] Statcounter, “Mobile Operating System Market Share Worldwide,” Jaanuar 2021. [Online]. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[3] A. Studio, “Android Studio,” Oktoober 2020. [Online]. https://developer.android.com/studio/run/emulator.

[4] A. Studio, „Android Studio,“ [Online]. https://developer.android.com/studio/run/emulator#limitations.

[5] D. W. Neal Goldstein, “Dummies,” Aprill 2021. [Online]. https://www.dummies.com/web-design-development/mobile-apps/the-ios-simulators-limitations/.

[6] S. Bose, „Why are Device Farms so important for Software Testing?,“ Oktoober 2019. [Online]. https://www.browserstack.com/guide/importance-of-device-farms.

[7] Robolectric, “What is Roboelectic?,” Veebruar 2021. [Online]. http://robolectric.org/.

[8] A. Developers, “Espresso,” Oktoober 2020. [Online]. https://developer.android.com/training/testing/espresso.

[9] Square, “Distributing instrumentation tests to all your Androids,” Aprill 2021. [Online]. https://github.com/square/spoon.

[10] JetBrains, “What is IntelliJ,” 2021. [Online]. https://www.jetbrains.com/idea/.

[11] Jenkins, “What is Jenkins,” Aprill 2021. [Online]. https://www.jenkins.io/.

[12] Gitlab, “Security of running jobs,” Aprill 2021. [Online]. https://docs.gitlab.com/runner/security/.

[13] J. Bloch, „Item 80: Prefer executors, tasks, and streams to threads,“ *Effective Java 3rd edition*, 2018, p. 324.

[14] A. Developers, “Android Debug Bridge (adb),” Veebruar 2021. [Online]. https://developer.android.com/studio/command-line/adb.

[15] Manpagez, “Manpagez,” 2021. [Online]. Available: https://www.manpagez.com/man/1/xcrun/.

[16] Apple, “Documentation Archive,” Aprill 2018. [Online]. https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html.

[17] A. Briegel, “Editing Property Lists with plutil,” November 2016. [Online]. https://scriptingosx.com/2016/11/editing-property-lists/.

[18] M. Heller, “InfoWorld,” Märts 2020. [Online]. https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html.

[19] Curl, “What is cURL,” Aprill 2021. [Online]. https://curl.se/.

[20] CyberAgent, “OpenSTF,” 2018. [Online]. https://openstf.io/.

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

Mina, Kevin Janson

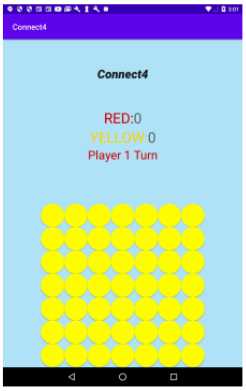
1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose MOBIILIRAKENDUSTE TESTIMISE LABORI INTEGREERIMINE ÕPPETÖÖSSE, mille juhendaja on Meelis Antoi
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse 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.

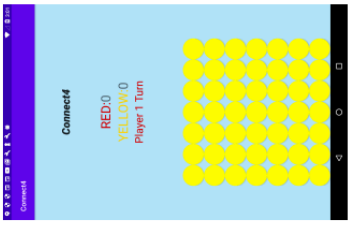
14.05.2021

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.

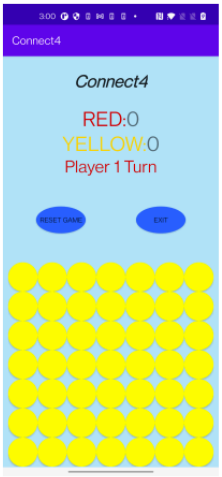
Lisa 2 – Testlab labor

Lisa 3 – Android test



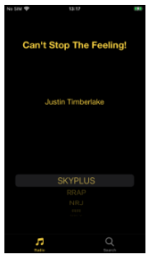


Nexus 7
| 23

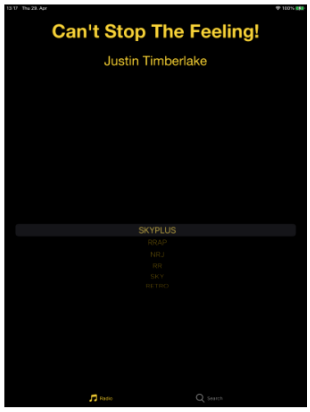


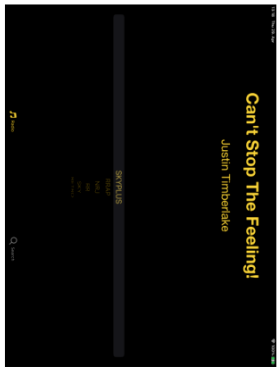
IN2023
| 30

Lisa 4 – IOS test

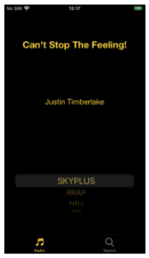


TalTech's iPhone 8 | 14.5





Taltech's iPad | 14.5



Lisa 5 – CI/CD skript

```
#Alus kettatõmmis, mida kasutatakse rakenduse kompileerimisel
image: jangrewe/gitlab-ci-android:latest

#Siin sektsioonis on defineeritud vastavad muutujad, mida kasutame
variables:
 #Kasutaja vahetab selle vastavalt $SCREENSHOT või $ESPRESSO
 REPORT_TYPE: $ESPRESSO
 #Kasutajal on võimalus vahetada järnev muutuja välja enda emaili vastu,
vastasel juhul saadetakse testimistulemused, üliõpilaskoodi emaili peale.
 EMAIL: ${GITLAB_USER_EMAIL}

SCREENSHOT: "screenshot" #Seda kasutatakse kuvatõmmiste testiks
ESPRESSO: "espresso" #Seda kasutatakse Espresso tüüpi testiks

#Järgnev sektsioon on selleks, et järgnev jooksutamine oleks kiirem
cache:
 key: ${CI_PROJECT_ID}
 paths:
   - .gradle/

#Siin sektsioonis tegeletakse koodis artefakti kompileerimiseks. Seda tehakse
kompileerimis serveris Dockeri konteineris
assembleArtifacts:
 stage: build
 allow_failure: false
 when: manual
 script:
   #Seadistakse keskkond ja luuakse kaks artefakti, mis on vajalikud Testlab
programmiks
   - export GRADLE_USER_HOME=$(pwd)/.gradle
   - chmod +x ./gradlew
   - ./gradlew assembleDebug
   - ./gradlew assembleAndroidTest
 #Defineerime millised failid soovime edastada järgmisesse sektsiooni
 artifacts:
   paths:
     - app/build/outputs/apk

#Siin sektsioonis edastakse vajalik info Jenkins CI/CD serverile.
testArtifacts:
 stage: test
 dependencies:
   - assembleArtifacts
 script:
  - curl -X GET
http://smartlab.ttu.ee:9090/buildByToken/buildWithParameters?job=Android -F
token=${CI_TOKEN} -F projectId=${CI_PROJECT_ID} -F
testingType=${REPORT_TYPE} -F email=${EMAIL}
```

Lisa 6 – Lähtekood

https://gitlab.cs.ttu.ee/kejans/kevin-janson-thesis