TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Karl Jääts

IntelliJ IDEA-le testide loomise töövahendi Test-motor toe lisamine

Bakalaureusetöö (9 EAP)

Juhendaja(d): Vesal Vojdani

Tartu 2019

IntelliJ IDEA-le testide loomise töövahendi Testmotor toe lisamine

Lühikokkuvõte:

Käesoleva bakalaureusetöö eesmärgiks on luua IntelliJ IDEA-le pistikprogramm, mis lubaks mugavalt IDEA seest käivitada testide loomise töövahendit nimega Testmotor. Testmotor ei genereeri käivitatavaid teste, vaid sisendeid koodis defineeritud funktsioonidele, mille põhjal saaks kasutaja ise vabalt valitud raamistikus teste kirjutada. Selle tõttu on loodav pistikprogramm võimeline ka nende sisendite põhjal genereerima käivitatavaid teste, et kiirendada testide loomise protsessi. Peamine eesmärk on teha testide genereerimine võimalikult kiireks ning kasutajale mugavaks.

Võtmesõnad:

Java, IntelliJ, automaattestid, pistikprogramm, peegeldus

CERCS: P175

Adding support for the test generation tool Testmotor to IntelliJ IDEA

Abstract:

The aim of this thesis is to create a plugin for IntelliJ IDEA that allows running the test generation tool called Testmotor from the IDE. Testmotor generates inputs for the functions to be tested that can then be used as a bases for generating the actual tests. Thus, the plugin is also capable of generating the actual tests given test inputs and a reference implementation. The main goal is to make creating tests as fast and easy for the user as possible.

Keywords:

Java, IntelliJ, tests, plugin, reflection

CERCS: P175

Sisukord

1. Sissejuhatus 4
2. Töö taust 5
2.1 Töö eesmärk 5
2.2 Automaattestid 5
2.3 Testmotor 5
2.4 IntelliJ IDEA 6
2.4.1 Olemasolevad testide genereerimise pistikprogrammid IntelliJ's 6
EvoSuite 6
TestMe ja Squaretest 6
2.5 Apache Freemarker 6
2.6 Gradle 7
3. Meetodid 8
3.1 Probleem 8
3.2 Lahendus 8
3.2.1 Testide genereerimine kahes osas 8
3.2.2 Testide genereerimiseks vajalikku infot hoidev fail Java failina 9
3.2.3 Faili nõutav formaat 9
3.3 Testide genereerimise väljund 10
4. Implementatsioon 11
4.1 Üldine töö struktuur 11
4.2 IntelliJ pistikprogrammi üles seadmine 11
4.3 Väljundist testide genereerimine 11
4.3.1 Mall 11
4.3.2 Sisendfailist teksti kätte saamine 12
4.3.3 Testide kontrollväärtuste genereerimine 13
4.4 Testmotori käivitamine 14
5. Tulemused 16
5.1 Pistikprogrammi seisukord 16
5.2 Jäänud probleemid 16
5.3 Tulevikuvõimalused 17
6. Kokkuvõte 18
7. Viidatud kirjandus 19
I. Litsents 20

1. Sissejuhatus

Koodi automaatne testimine on tarkvaraarenduse lahutamatu osa. Võrreldes käsitsi testimisega säästab see tohutult aega ja on mõnes mõttes ka töökindlam, kuigi teisi testimise meetodeid täielikult muidugi ei asenda. Automaattestimist saaks aga veelgi kiirendada, kui vähemalt osa testidest saaks mingi programmi abil genereerida, kuna nende kirjutamine on aeganõudev ning tihti programmeerijatele üsnagi igav või tüütu. Testide automaatseks genereerimiseks pole aga palju tööriistu loodud, sest need pole võimelised kuigi kasulikke teste genereerima, sest neil on aluseks võtta ainult testitav kood, millest vigade leidmine ongi kogu testimise peamine mõte. Teste genereeriv programm aga ei saa kuidagi teada, mis on viga ja mis töötab nii nagu peab, kui süntaksivead ning muud sellised välja jätta, sest need tulevad muul moel niigi välja.

Testide genereerimine on küll üldjuhul vastunäidustatud, kuid on mõned erandid. Näiteks programmeerimise või muu sellise õpetajad kasutavad palju automaattestimist õpilaste tööde kontrollimiseks, kuid peavad kulutama palju aega nende testide kirjutamisele, eriti kuna tööde eri variandid vajavad eri teste. Õpetajatel on aga tihti juba olemas töötav programm, mis teeb seda, mida õpilastelt tahetakse. Kuna nende koodi saab usaldada ning programm, mille testimiseks loodud teste kasutatakse, pole see sama, mille põhjal testid genereeriti, siis saab nende koodi põhjal edukalt täiesti kasulikke teste automaatselt genereerida.

Selliseid testide genereerimise tööriistu, mis sobiksid algõppe programmeerimisülesannete kontrollimiseks, palju ei eksisteeri, nii et selle tööga paralleelselt lõi Ergo Nigola tööriista nimega Testmotor. Testmotor analüüsib koodi ning genereerib selle põhjal hulga sisendeid koodis esinevatele meetoditele, mille põhjal loodud testid peaksid koodi üsnagi põhjalikult testima.

Töö eesmärk oli leida mugavam viis programmeerimisülesannete testide loomiseks ja haldamiseks, kasutades automaatse testi genereerimise töövahendit Testmotor. Lahendusena loodi IntelliJ IDEA-le pistikprogramm, mis laseb Testmotorit IntelliJ seest käivitada ning on võimeline Testmotori poolt loodud sisendite põhjal teste genereerima.

Töö on jagatud neljaks peatükiks. Esimeses peatükis antakse lühiülevaade kasutatud tehnoloogiatest. Teises peatükis kirjeldatakse lahendatavat probleemi ning lahenduse ideeni jõudmist. Kolmandas peatükis seletatakse lahenduse implementatsiooni ning põhjendatakse implementatsioonis tehtud valikuid. Neljandas peatükis kirjeldatakse pistikprogrammi hetkeseisu, selle probleeme ning tulevikuvõimalusi.

2. Töö taust

2.1 Töö eesmärk

Loodud pistikprogramm[1] nimega „*Testmotor plugin*“ on mõeldud eelkõige Java programmeerimiskursuste, olgu need siis MOOC-id või tavapärased kursused, õpetajatel, eesmärgiga kiirendada hindeliste tööde parandamisele kuluvat aega. Kuigi automaattestide olemasolu juba kiirendab oluliselt igasuguste hinnatavate tööde kontrollimist, kulub testide enda kirjutamisele siiski üsna palju aega, eriti kuna neid on vaja iga töö variatsiooni jaoks eraldi kohendada või kui töös on midagi vaja muuta, isegi kui see on midagi väga väikest, näiteks ühe arvu muutmine, siis peab ka testid käsitsi uuesti üle tegema. Testmotori ja selle töö käigus loodud pistikprogrammi koostöös testide genereerimine püüab seda protsessi lihtsustada ning kiirendada.

2.2 Automaattestid

Tarkvara testimise eesmärgiks on vigade leidmine koodist ning tarkvara kvaliteedi kohta info andmine. Automaattestid on üks viis selle saavutamiseks. Need on programmijupid, mida kasutatakse koodi automaatseks kontrollimiseks ning nad on eriti kasulikud loogiliste vigade leidmiseks. Need on vead, mis koodi kompileerimisel küll veateadet ei anna, kuid programm ei käitu nii nagu nende programmeerija soovis.

Automaattestidel on koolitööde hindamise protsessis lisaboonuseks see, et õpilased saavad oma töö kohta kohest tagasisidet, mis laseb neil oma tööd veel parandada ning enne töö tähtaega see uuesti esitada.

Selles töös tegeletakse põhiliselt ühiktestidega, mis on automaattestimise üks peamisi meetodeid. Ühikteste kasutatakse lähtekoodi komponentide ükshaaval testimiseks. Täpsemalt kasutatakse loodavas pistikprogrammis automaattestideks JUnit raamistikku, mis on raamistik ühiktestimiseks Javas.

Ühiktestides kasutatakse tihti mingit funktsiooni, mis kutsub välja testitavat komponenti, mis tavaliselt on funktsioon, mingite sisenditega ning seejärel võrdleb testitava funktsiooni tagastatud väärtust testi kirjutaja poolt antud oodatava väärtusega. Kui need on samad, siis on test läbitud, kui ei ole, siis on kas koodis või testis endas mingi viga. JUnitis on selleks funktsiooniks *assertEquals* ning mõned muud, mis on mugavuseks lisatud, nagu näiteks *assertTrue*.

2.3 Testmotor

Testmotor[2] on Java koodi jaoks testide loomise töövahend [1]. Seda arendab selle tööga paralleelselt Ergo Nigola. Testmotori tööpõhimõte seisneb selles, et see analüüsib olemasolevat koodi, ning seejärel genereerib koodi põhjal hulga sisendeid koodis defineeritud funktsioonidele, mille põhjal loodud testid peaksid katma suurema osa koodi funktsionaalsusest nii nagu see genereerimise hetkel oli. Testmotor ei ole mõeldud iseseisvaks kasutamiseks, vaid vajab midagi, mis kasutaks selle genereeritud sisendeid testide genereerimiseks. Põhjus, miks see loob ainult sisendeid ning mitte otse käivitatavaid teste seisneb selles, et see jagab testide genereerimise kahte ossa nii, et kui funktsiooni siseselt midagi muutud, siis ei pea kogu testide genereerimise protsessi uuesti läbi tegema, vaid saab lihtsalt samade sisendite järgi uued testid genereerida. Samuti annab see Testmotorile paindlikust, sest seda saab siis kasutada eri testimisraamistikes ilma, et Testmotorit ennast muutma peaks. Lisaks

[1] https://github.com/karljaats/test-motor-plugin
[2] https://bitbucket.org/ENigola/testmotor/src/master/

on selliseid sisendeid lihtsam muuta, juhuks kui automaatne genereerimine ei andnud päris õigeid tulemusi või kasutaja tahab sinna midagi lisada või nende järjekorda muuta.

2.4 IntelliJ IDEA

IntelliJ IDEA [2] on integreeritud programmeerimiskeskkond (IDE ehk *integrated development environment*) põhiliselt programmeerimiskeelele Java, kuid tänu pistikprogrammidele toetab IntelliJ ka mitmetes muudes keeltes programmeerimist. IntelliJ looja on JetBrains ning IntelliJ platvormi põhjal on JetBrains loonud ka mitmeid teisi integreeritud programmeerimiskeskkondi nagu näiteks PyCharm Phytonile ning WebStorm JavaScriptile.

IDE-ks, millele pistikprogrammi teha, valiti IntelliJ ja mitte näiteks Eclipse, lihtsal põhjusel, et nii selle töö tegija kui juhendaja kasutavad eelkõige IntelliJ'd ning tundub, et Tartu Ülikoolis üldisemalt on IntelliJ palju levinum. Lisaks kuna teised JetBrains-i poolt toodetud IDE-d on sama raamistiku peale ehitatud [3], siis peaks saama loodavat pistikprogrammi väikese vaevaga ka neis töötama, mis võib tulevikus kasuks tulla kui peaks tekkima tahtmine laieneda ka teistele programmeerimiskeeltele peale Java.

2.4.1 Olemasolevad testide genereerimise pistikprogrammid IntelliJ's

Hetkel on IntelliJ's kolm pistikprogrammi, mis mingil kujul automaatteste genereerivad. Nendeks on EvoSuite, TestMe ja Squaretest. Kõik need on Java testide genereerimiseks. Kõigi nende töös jääb aga midagi soovida ning ükski neist ei kasuta sellist kahesammulist genereerimise viisi nagu selles töös kasutatakse.

EvoSuite

EvoSuite on sellele tööle sarnaseim programm, sest sellel on samuti väline töövahend, mida pistikprogramm käivitab. EvoSuite [4] genereerib JUnit 4 raamistikus teste nii, et maksimeerida erinevaid testide kasulikkuse kirjeldamiseks kasutatavaid parameetreid nagu palju koodist on testidega kaetud, ning minimiseerida testide hulka. Selle kasulikkuse idee seisneb selles, et programmi planeeritud käitumisest kõrvalekaldeid oleks lihtsam märgata testidest kui koodist endast. EvoSuite-i probleem seisneb selles, et genereeritavaid teste ei panda projekti kaustas õigesse kohta, mis tekitab mitmeid probleeme ning muudab selle kasutamise tüütuks. Lisaks on selle üles seadmine keeruline, sest see tuleb pistikprogrammist eraldi installeerida ning kasutab testides oma siseseid funktsioone, mis muudab selle kasutamise tööde kontrollimiseks keeruliseks. See on oluline probleem, sest meil on vaja, et tudengid saaksid testidest aru ja saaksid nende põhjal oma koodi parandada.

TestMe ja Squaretest

TestMe ja Squaretest on omavahel väga sarnased ning ei kasuta mingeid väliseid programme testide genereerimiseks. Nende probleemiks on aga see, et nende poolt genereeritud testide kvaliteet on küllaltki madal ja mõned testid isegi ei tööta, nii et kasu nendest eriti tegelikult pole. Mõlemad toetavad mitmeid eri testide raamistikke [5, 6] ning TestMe toetab ka programmeerimiskeeli Scala ja Groovy [5].

2.5 Apache Freemarker

Apache Freemarker on avatud lähtekoodiga Java teek mallide põhjal teksti genereerimiseks. Seda saab kasutada igasuguse teksti, kaasa arvatud lähtekoodi, genereerimiseks. Mallide jaoks kasutab Freemarker spetsiaalset programmeerimiskeelt nimega *FreeMarker Template Language*, mis lubab, lisaks lihtsalt muutujate tekstis õigesse kohta paigutamisele, muu hulgas mallides teha tehteid, valida muutujate alusel tingimuslikult erinevaid osi tekstist mida

kasutada, ning itereerida läbi seda lubavate muutujate, näiteks massiivide, mida mallile on käivitav programm andnud. Freemarkerit saab kasutada eri programmeerimiskeeltes, kuid kuna tegemist on Java teegiga, siis kasutatakse seda eelkõige Javas. [7]

2.6 Gradle

Gradle on avatud lähtekoodiga tööriist programmide koostamise automatiseerimiseks [8]. See hõlmab endas muu hulgas vajalike teekide alla tõmbamist, lähtekoodi kompileerimist ning automaattestide käivitamist. Gradle on laialdaselt kasutusel olles loodud varasemate sarnaste tööriistade, nagu Apache Maven, põhjal, kuid olles uuem ning parandatud. Gradle'i käivitamine käib läbi ülesannete, inglise keeles *task*, mis lasevad kasutajal käivitada vaid neid töö etappe, mida parajasti vaja.

3. Meetodid

3.1 Probleem

Probleemiks on see, et programmeerimisalaste kursuste õpetajatel kulub palju aega hindeliste tööde jaoks automaattestide kirjutamisele. Automaattestide kirjutamist saab kiirendada genereerides teste vähemalt osaliselt automaatselt.

Õpetajad on selle jaoks unikaalses positsioonis, sest neil peab juba niikuinii korrektselt töötav kood olemas olema, mida õpilased peavad põhimõtteliselt järgi tegema. Kuigi automaattestid on tarkvara arenduses laialdaselt kasutusel pole nende automaatseks genereerimiseks palju vahendeid. Selle põhjuseks on see, et üldiselt on automaattestide eesmärgiks leida koodist kohad, mis töötavad teisiti kui nende programmeerija tahtis. Teste genereeriv programm ei saa aga kuidagi teada, mida programmeerija tahtis, see näeb ainult seda, kuidas kood tegelikult käitub, mis teeb kasulike testide genereerimise üldjuhul peaaegu võimatuks. Õpetajatel on aga korrektselt töötav kood juba olemas, nii et teste genereeriv programm saab nende koodi lihtsalt aluseks võtta ning eeldada, et see töötabki nii nagu peab.

Teiselt poolt on aga õpetamisel ka oma nõudmised loodud testidele. Loodud testid peaksid olema õpilastele loetavad, sest mõnedes ainetes antakse teste ette, et nende põhjal ülesanne järk-järgult lahendada. Selleks on vaja ka teste pedagoogiliselt üles ehitada ja mõelda selle üle, kuidas on mõistlik teste rühmitada ja järjestada, et seda saaks töö hindamisel kasutada. Automaatselt testide genereerimisel on tulemus hulk erinevaid teste, mis testivad koodi eri osi üsnagi segiläbi. Õpetajatel on aga vaja, et testid oleks korrastatud ning juppideks jaotatud, nii et saaks näha, mis ülesanded õpilane ära tegi, ning, et anda õpilasele kasulikku, lihtsasti mõistetavat tagasisidet. Samuti võivad kasutajad tahta lisada oma teste juurde ning iga väikeste muutuse pärast pole tingimata vaja kogu testide genereerimise protsessi uuesti läbi teha.

3.2 Lahendus

3.2.1 Testide genereerimine kahes osas

Ülalmainitud probleemide lahendamiseks töötab testide genereerimine kahes osas. Esiteks genereerib Testmotor koodi põhjal konkreetses formaadis Java faili, milles on testitav meetod ning meetodi parameetrite jaoks sisendid. Seda faili saavad siis kasutajad muuta, tõstes sisendeid ümber, lisades enda omi, ning nimetades ümber muutujaid, mille põhjal nimetatakse genereeritavad testid. Seejärel genereeritakse selle vahepealse faili põhjal tegelikud testid, mille sisuks on kontrollid kas meetod testitavas koodis annab sama tulemuse kui see meetod andis testi genereerimise ajal. See vaheetapp tähendab, et kui muutub midagi väikest, mis lisa testide kirjutamist ei nõua, näiteks muudetakse mingit konstanti, siis saab lihtsalt selle vahepealse Java faili põhjal testid uuesti genereerida, mis uuendab teste, kaasa arvatud kasutaja poolt lisatud teste, kuid jätab kasutaja tehtud muudatused vahepealses failis alles. Kui vahepealset faili poleks vaid genereeritaks otse teste, siis kirjutaks testi genereerimine kasutaja tehtud muudatused lihtsalt üle, ning kasutaja peaks need iga kord uuesti tegema. Nii jääks loodud pistikprogrammi kasutegur küllaltki väikeseks, kuigi otse testide genereerimine teeks testide genereerimise kasutajale mugavamaks, kuna seda saaks teha vaid ühe nupu vajutusega, ning ei tekiks kasutaja projekti lisa faile.

Algselt oligi plaanis korraldada töö nii, et Testmotor genereerib otse teste ning pistikprogramm lihtsalt käivitab seda ning võib-olla annab Testmotorile ette projekti kohta infot, näiteks, mis kaust on testide jaoks, et Testmotor saaks testide failid paigutada õigesse kohta ja kasutaja ei peaks neid käsitsi liigutama. Otsustati, aga et kuna programmi sihtgrupp on

õpetajad, kes tahavad niikuinii teste faili siseselt ümber tõsta, muuta ja lisada, siis on oluline see neile lihtsaks teha ning proovida, nii palju kui võimalik, vähendada juhuseid, kus nende muudatused üle kirjutatakse. See oligi peamine põhjus, miks otsustati teha testide genereerimine kahes osas, et vähemalt väikeste muudatuste korral, nagu näiteks kontrolltöö teise variandi jaoks mingis meetodis tehte muutmine, oleks võimalik teste uuesti genereerida ilma, et testides käsitsi tehtud muudatused kaduma läheks.

3.2.2 Testide genereerimiseks vajalikku infot hoidev fail Java failina

Testmotori poolt genereeritud vahepealse failina, mille põhjal pistikprogramm teste genereerib, otsustati kasutada spetsiifilises formaadis Java faili. Algselt pidi failis olema ainult meetodi jaoks sisendid, mille põhjal teste genereerida. Neid sisendeid kasutatakse kontrollimiseks, kas testitava projekti meetod annab sama sisendiga sama tulemuse kui testide genereerimise ajal. Kaalumisel oli ka teha sisendite faili näiteks XML kujul või siis kas või lihtsalt tekstifailina, kasutades mingit spetsiifilist notatsiooni, kuid Java fail tundus parima variandina, sest sisenditeks võib olla ka näiteks meetodite kutsed või muud Java keele spetsiifilist, ning sel juhul oleks mugav kasutada IntelliJ's olemas olevaid automaatse lõpetamise ning süntaksi kontrollimise funktsionaalsusi, mis kindlasti tõstaks kasutajamugavust. Java faili valik muutus veelgi kindlamaks, kui sai selgeks, et vaja on faili jäädvustada ka muud informatsiooni, peale lihtsalt sisendite. Samuti oleks vaja faili panna, mis testide faili ning klassi nimi võiks olla ning on oluline, et teste saaks grupeerida eri meetoditesse, mida peaks samuti saama kasutaja nimetada, vastavalt oma soovile. Veel ilmnes, et testitav meetod võiks olla selles samas failis defineeritud, et lubada kasutajale ning Testmotorile suuremat vabadust sellega ümber käimisel. See kõik tähendas, et kõne all olev fail peaks kindlasti olema Java fail, sest faili kirjutatakse päris palju Java koodi, nii et IntelliJ's olemasolev Java koodi kirjutamist hõlbustav funktsionaalsus on tingimata vajalik. Java faili valik tuli ka implementatsioonis kasuks.

3.2.3 Faili nõutav formaat

Soov IntelliJ Java funktsionaalsust ära kasutada seadis aga faili formaadile mitmeid piiranguid, sest need töötavad ainult siis, kui fail on Java kompilaatorile arusaadavalt ning korrektselt vormistatud. See tähendab, et kood peab olema mingi klassi sees, ning testide jaoks mõeldud sisendid peavad olema defineeritud, kas igaüks eraldi muutujana, või siis mingi andmestruktuuri sees, ja mitte näiteks lihtsalt komaga eraldatult reas.

Formaadi nõutavas kujus lepiti kokku neid piiranguid arvesse võttes. Näide nõutavale formaadile vastavast sisendite failist on toodud joonisel 1. Faili alguses peab olema Java paketti nimi, kuhu nii see klass, kui ka sellest loodav testide klass kuuluvad. Kood peab olema klassis, mille nime kasutatakse ka testide klassi nimeks lisades selle lõppu sõna „Test". Klassis on meetod nimega „f", mida kutsutakse välja antud sisenditega, et saada testidele kontroll väljundid. Meetodi f olemasolu eesmärk on anda Testmotorile ja kasutajale paindlikkust testimisel, näiteks võimalus sisenditega midagi täiendavat teha, enne kui nad testitavale meetodile antakse või ka, et kasutada samu sisendeid mitme meetodi korraga testimiseks. Meetodil peab olema kindel nimi, et

```
package lihtne;

public class BasicTestSisend {

    public static int f(int input) {
        Basic basic = new Basic();
        return basic.inc(input);
    }

    private static int x = 10;
    private static int computeValue() {
        return 100;
    }

    // sisend:
    public static int data1[] = {
            10,
            50,
            x,
            computeValue(),
            x+10
    };
```

Joonis 1. Sisendite faili näide.

pistikprogramm selle üles leiaks ning selleks valiti „f“, sest see on äratuntavalt erinev, sest koodi konventsioonide järgi pannakse meetoditele tavaliselt pikemad ning rohkem kirjeldavad nimed. Iga soovitava test meetodi jaoks on klassis massiiv, mille nimi algab sõnaga „*data*“, mis teste genereerides asendatakse sõnaga „test“, et saada loodava meetodi nimi. Massiivi liikmete tüüp peab olema sama, mis meetodi f tagastustüüp. Sisendite kujutamiseks valiti massiiv, sest mingis andmestruktuuris peavad need olema ja nende eraldi muutujatena kirjutamine oleks esiteks pikk, teiseks võiks tekkida segadus sisendite ning muude muutujatega, kolmandaks peaks testide meetodite nimede jaoks midagi muud kasutama, ning neljandaks, mis on kõige olulisem põhjus, oleks keeruline neid selgelt grupeerida, et kasutaja saaks ühetaolised testid panna ühte kohta. Põhjus, miks valiti just massiiv ning mitte näiteks *ArrayList*, on see, et massiivi defineerimine on süntaktiliselt lühem ning seega on sisendite loetavus parem ning samuti ei pea massiivi kasutamiseks midagi importima. Samuti võib, aga ei pruugi, klassis olla igasugu muutujaid ning meetodeid, mida saab siis sisenditena või meetodis f kasutada. Need muutujad ning meetodid peavad kõik olema staatilised.

Peamine probleem faili jaoks välja mõeldud formaadiga on see, et massiivi kasutamise tõttu ei saa meetodil f olla rohkem kui üks parameeter. See tekitaks sisenditega loetavuse probleeme, ning samuti tekitaks pistikprogrammi koodis probleeme. See tekitab veidi ebamugavust, kuid programmi kasulikkust ei piira, sest mitme parameetriga meetodite testimeseks saab sisendid meetodi f jaoks lihtsalt mingisse vabalt valitud andmestruktuuri panna, ning seejärel meetodi koodis need jällegi lahti pakkida. Osalt selle jaoks meetod f eksisteeribki.

3.3 Testide genereerimise väljund

Joonisel 2 on toodud joonisel 1 asuva sisendite faili põhjal pistikprogrammi poolt genereeritud testide fail. Nagu näha on fail küllaltki sisendite failiga sarnane. Suurim muutus on, et sisendite massiivi asemele on massiivi elemente kasutades genereeritud test meetod, kus iga sisendi kohta on *assertEquals* kutse. Funktsioon *assertEquals* kontrollib, kas tema esimene ning teine argument on sama väärtusega. Esimeseks argumendiks on väärtused, mis saadi jooksutades funktsiooni f antud sisendiga testide loomise käigus. Teiseks argumendiks on funktsiooni f kutse selle sama sisendiga. Nii kontrollitakse, kas testitav kood töötab sama moodi kui kood, mille põhjal need testid genereeriti.

```
package lihtne;

import org.junit.Test;
import static org.junit.Assert.*;

public class BasicTestSisendTest {


    public static int f(int input) {
        Basic basic = new Basic();
        return basic.inc(input);
    }

    private static int x = 10;
    private static int computeValue() { return 100; }

    // sisend:

    @Test
    public void test1() {
        assertEquals( expected: 11, f( input: 10));
        assertEquals( expected: 51, f( input: 50));
        assertEquals( expected: 11, f(x));
        assertEquals( expected: 101, f(computeValue()));
        assertEquals( expected: 21, f( input: x+10));
    }
}
```

Joonis 2. Pistikprogrammi poolt genereeritud testide fail

4. Implementatsioon

4.1 Üldine töö struktuur

Loodud pistikprogrammil on kaks poolt. Esimene pool tegeleb Testmotoriga. See annab kasutajale lihtsa ning mugava viisi Testmotorit käivitada soovitud kohas ning laseb tal Testmotorit konfigureerida. Teine pool tegeleb Testmotori väljundist tegelike, jooksutavate JUnit testide genereerimisega.

4.2 IntelliJ pistikprogrammi üles seadmine

Intellij's on pistikprogrammide loomiseks pistikprogramm nimega *gradle-intellij-plugin*, mille abil saab luua pistikprogrammi projekti ning see loob Gradle projekti, mis teeb kasutaja eest kogu vajaliku seadistamise ära ning loob eraldi Gradle'i ülesanded IntelliJ käivitamiseks loodava pistikprogrammiga. See lubab kasutajal saab kohe oma pistikprogrammi arendama hakata ilma seadistusele mõtlemata.

Pistikprogrammi projekt on põhimõtteliselt tavaline Gradle projekt, millele on lisatud spetsiaalne XML-fail nimega „plugin.xml", kus on defineeritud kogu informatsioon, mida IntelliJ'l vaja on, alustades pistikprogrammi nimest ja kirjeldusest ning lõpetades lisatavate nuppude ja muu sellise definitsioonidega.

IntelliJ pistikprogrammid töötavad peamiselt *Action'*ite kaudu, mis on klassid, mis laiendavad *AnAction* klassi. Failis „plugin.xml" nuppe ja muid selliseid elemente defineerides saab IntelliJ'le öelda, mis *Action'*it see käivitama peaks. Samuti saab neid muude IntelliJ siseste sündmuste külge haakida. *Action'*i *actionPerformed* meetodis saab pistikprogramm siis teha, mida vaja ja muid funktsioone välja kutsuda. Sellele meetodile antakse kaasa ka *AnActionEvent* tüüpi muutuja, mis peaks sisaldama kõike vajalikku, nagu projekti asukoht, lahti olev fail, faili tekst ja nii edasi. Testmotori pistikprogrammis on samuti *Action'*id kasutusel – mõlema nupu, „*Run Testmotor*" ning „*Generate Tests*" jaoks.

4.3 Väljundist testide genereerimine

4.3.1 Mall

Dokumendimall või dokumendipõhi on põhi mingi teksti koostamiseks kuhu saab märgitud kohtadesse andmeid sisestada. Nende eesmärk on lihtsustada ning kiirendada kasutaja tööd.

Mallide kasutamise tarkvaraks, mida pistikprogrammis kasutada, oli algselt 2 valikut – Apache Freemarker ja Apache Velocity. Põhjus, miks just need kaalumisel olid, oli see, et need on põhilised malli tarkvarad, mida IntelliJ toetab. IntelliJ tugi oli oluline, et pistikprogrammi arendustööd lihtsustada, sest pistikprogrammi arendati IntelliJ's. IntelliJ tugi oli oluline ka sellepärast, et on võimalik, et millalgi tulevikus lisatakse pistikprogrammile võimalus kasutajatel oma malle kasutada, ning sel juhul oleks IntelliJ's mallide kirjutamise tugi kasutajatele mugav. Lõpuks otsustatu valida Apache Freemarker, lihtsal põhjusel, et see on uuem ning paistis, et seda uuendatakse energilisemalt.

Seda tööd oleks saanud ka täiesti ilma malle kasutamata teha kirjutades vajalikud osad sõne kujul otse pistikprogrammi Java koodi. Mallide kasuks otsustati, et lubada tulevikus kasutajatel enda mallide lisamist, lihtsustada ning kiirendada arendustööd ja et selgelt eraldada genereeritavate failide üldine struktuur failide muutuvatest osadest, seeläbi muutes pistikprogrammi koodi loetavamaks. Samuti oleks malli otse Java koodi kirjutamine tulevikus potentsiaalselt probleeme tekitanud kui pistikprogrammil peaks vaja olema kasutada rohkem kui ühte malli, näiteks eri tüüpi failide genereerimiseks.

Töö käigus loodud pistikprogramm kasutab malli käivitatava automaattestide faili loomisel alusena, kuhu on vaja vaid konkreetsed andmed sisestada. Vajalikeks andmeteks on paketti nimi, kuhu nii testitav fail kui loodav testide fail kuuluvad, kõik impordid, mis sisendfailis on, testklassi nimi, kogu kood, mis sisendis jääb klassi deklaratsiooni alguse ning esimese meetodite sisendite massiivi vahele ning viimaks mitmetasemeline andmestruktuur, mis hoiab testmeetodite nimesid ning kõiki meetodite sisendeid ning oodatavaid väljundeid neile. Mall võtab need andmed ning paigutab need õigetesse kohtadesse loodavas failis ning tulemuseks on käivitatav ning tööks valmis automaattestide fail.

Joonisel 3 on kujutatud pistikprogrammis testfailide loomiseks kasutusel olev mall Freemarkeri keeles FTL. Väiksem kui ja suurem kui märgid märgivad FTL koodi, mis hõlmab mitmeid tavalisi programmeerimiskeelte võimalusi nagu valiklaused, tsüklid ja muidugi muutujad. Näiteks joonise 3 esimesel real on valiklause, mis kontrollib, et muutuja nimega *package*, mis hoiab paketi nime, pole tühi, sest kui see on tühi, siis peab järgneva rea ära jätma. Muutujaid märgib ka *${...}* süntaks nagu on näha näiteks joonise 3 kuuendal real. Muu on lihtsalt tekst, mis kopeeritakse otse loodud faili. FTL kood ise malliga loodud faili edasi ei lähe. Muutujaid saab defineerida ka malli siseselt, kuid enamasti, nagu ka loodud pistikprogrammis, antakse need mallile väliselt, malli kasutavast koodist.

```
1  <#if package != "">
2  package ${package};
3
4  </#if>
5  <#if imports != "">
6  ${imports}
7  </#if>
8  import org.junit.Test;
9  import static org.junit.Assert.*;
10
11 public class ${class_name} {
12 ${rest_of_code}
13 <#list tests as test>
14     @Test
15     public void ${test[0]}() {
16     <#list test[1..] as data>
17         assertEquals(${data.expected}, f(${data.input}));
18     </#list>
19     }
20
21 </#list>
22 }
```

Joonis 3. Testfailide mall.

4.3.2 Sisendfailist teksti kätte saamine

Selleks, et andmeid saaks malli sisestada, peab need kõigepealt sisendfailist kätte saama. Selleks on mitmeid viise, näiteks saaks kasutada IntelliJ tarkvaraarenduskomplekti poolt pistikprogrammidele antud meetodeid, mille abil IntelliJ ise üldiselt oma tööd teeb. IntelliJ on aga eelkõige jooksva koodikirjutamise hõlbustamiseks ning selle tarkvaraarenduskomplekti on tehtud seda silmas pidades. See tähendas, et kuigi IntelliJ läbi oleks olnud võimalik saada mõningaid vajalikke andmeid, nagu näiteks klassi ning paketti nimi ning impordi nimekiri, siis kõiki andmeid sealt poleks olnud võimalik saada, vähemalt ilma muid võimalusi kasutamata. Vähemalt osaliselt oleks pidanud kasutama tekstitöötlust ning sel juhul oli lihtsam juba kõiki andmeid läbi tekstitöötluse otsida.

Teine võimalus, mida samuti kaaluti, oli kasutada Java võimalust, mida kutsutakse peegelduseks, mis laseb kasutada kompileerimisel loodud *.class* faile, et sealset koodi käivas programmis kasutada. Selle lähenemise põhiliseks positiivseks küljeks oli, et peegeldust pidi niikuinii kasutama, et sisendite põhjal oodatavaid väljundeid genereerida, sest selleks on vaja kompileeritud koodi. Peegeldust andmete failist saamiseks kasutada polnud aga võimalik, sest kui testimiseks vajalikes sisendites esineb meetodite väljakutseid, siis hoiustatakse massiivi vaid nende väljund, kuid testide loomiseks oleks vaja jätta see väljakutseks, nii nagu see väljakutse koodis esineb. Teiseks probleemiks oli ka see, et peegelduse kasutamine vajab osade tegevuste jaoks otsitava elemendi nime, kuid osaks otsitavatest andmetest ongi need samad nimed.

Veel üks võimalus oli kasutad IntelliJ sisest PSI (*Program Structure Interface* ehk Programmi struktuuri liides) süsteemi. PSI kaudu kujutab IntelliJ lahti oleva faili elemente ning faili kohta käivat informatsiooni. PSI kaudu oleks loodaval pistikprogrammil võrdlemisi lihtne kätte saada osa vajalikust informatsioonist, näiteks klassi nime ning imporditud klasside nimekirja, kuid ülejäänud infost, nagu sisendite massiivi liikmed, jaoks oleks ikkagi pidanud kasutama mingit muud lähenemist ning otsustati, et pole mõtet hakata ühte probleemi korraga kahte moodi lahendama. Seega jäi üle vaid saada IntelliJ'lt soovitud faili kogu tekst, ning seda siis töödelda, kasutades selleks regulaaravaldisi, et vajalikud andmed teksti kujul kätte saada.

```
"class (?<className>.*?) \\{"
```

Joonis 4. Klassi nime leidmise regulaaravaldis

Regulaaravaldis on otsingumuster, mida kasutatakse sõnedest mingi mustri leidmiseks. See koosneb tähemärkidest, millest osadel on erilised tähendused. Näiteks regulaaravaldis, mida kasutatakse pistikprogrammis klassi nime leidmiseks, on kujutatud joonisel 4. „*class*" tähendab lihtsalt sõna *class*. Sulgudes olev „*classname*" on nimi nendes sulgudes leitud regulaaravaldise jupile, mis on selle konkreetse jupi kättesaamiseks koodis pärast regulaaravaldise kasutamist. Punkt kujutab igasugust märki peale reavahetuste ning „*" signaliseerib, et sellele eelnev märk esineb 0 või rohkem korda ning küsimärk annab teada, et antud jupp peaks olema võimalikult lühike. Seejärel „*{" tähistab lihtsalt märki "*{*", kuid kuna sellel on regulaaravaldistes eriline tähendus siis peab seda kaldkriipsudega märkima, et programm teaks, et antud juhul on mõeldud lihtsalt looksulgu. Kokkuvõttes otsitakse kohta, mis on „*class* " ja „ *{*" vahel ning koosneb mistahes tähemärkidest, välja arvatud reavahetused.

Teksti töötlemiseks kasutati regulaaravaldisi, sest just tekstist mingite konkreetsete mustrite leidmiseks regulaaravaldised ongi. Ilma nendeta on tekstist programmaatiliselt info otsimine küllaltki töömahukas ning võrdlemisi keeruline. Lisaks töötavad regulaaravaldised eriti hästi struktuurete tekstidega nagu seda on koodifail, sest seda struktuuri saab korrektse koha üles leidmiseks ära kasutada.

Leiti, et kuna vaja on leida vaid mõningaid konkreetseid kohti tekstist, siis kogu teksti pole mõtet hakata struktuurselt läbi käima ning grupeerima, piisab kui iga andmete osa eraldi tekstist üles leitakse. See hoidis kokku suure hulga suuresti ebavajalikku tööd, kuid tekitas ka mõningaid probleeme. Mitte kogu tekstist läbi käimine nagu süntaksianalüsaator ehk parser seda teeb, tähendas, et mustriotsing jäi konteksti ära tundmise osas mõnevõrra nõrgaks. Näiteks ei tunne loodud regulaaravaldised ära kui hetkel vaatluse all olev koodijupp on Java mitmerealist kommentaari ära kasutades välja kommenteeritud, mis võib mõnel juhul viia probleemini, et pistikprogramm paneb genereeritud testide faili teste ka välja kommenteeritud sisendeid kasutades.

4.3.3 Testide kontrollväärtuste genereerimine

Testides võrdluste tegemiseks, kas meetod antud sisenditega tagastab ootuspärase väärtuse, on kõigepealt vaja teada, milline see ootuspärane väärtus sellise sisendi korral on. Tänu ebatavalisele situatsioonile, milles loodud pistikprogramm on mõeldud töötama, saab pistikprogramm lihtsalt jooksutada testitavat meetodit antud sisenditega ning tagastatud väärtus ongi ootuspärane väärtus, sest testid on mõeldud õpilaste poolt tehtud samasuguste tööde testimiseks. Pistikprogrammil pole aga niisama lihtne neid meetodeid jooksutada, sest need pole sama programmi osad ning vaatluse all olev kood ei ole isegi käivitatud programm. See ei pruugi isegi kompileeritud olla. Pistikprogramm näeb vaid kasutaja avatud projekti asukohta arvutis ning lahti olevat koodifaili tekstilisel kujul.

Üks võimalus vajalikele meetoditele ligi pääseda on kasutada objektorienteeritud programmeerimiskeelte, kaasa arvatud Java, võimalust kasutada peegeldust. Peegeldus on programmeerimisvõte, mis on laseb programmil ennast käitusajal muuta. Antud juhul tähendab see, et pistikprogramm otsib üles vaadeldava koodi kompileerimisel loodus *.class* failid ning otsib nendest üles vajaliku funktsiooni ning seejärel jooksutab seda funktsiooni vajalike sisenditega. Kuna see funktsioon pole pistikprogrammi osa loetakse seda programmi käitusajal muutmiseks kuigi jäädavaid muutuseid see programmi koodis ei tee.

Peegelduse implementeerimisel tekkisid aga mõned probleemid. Kuna peegelduse teostamiseks on vaja kompileeritud koodi *.class* laiendiga faile, siis on vaja need üles leida, kuid nende asukoht erineb sõltuvalt sellest, kas kasutaja kasutab oma projektis Gradle'it või mitte. Samuti tekkis probleem kui samu *.class* laiendiga faile esines mõlemas kohas, sest tekkis küsimus kumbasid peaks eelistama. Veel üks suur probleem oli, et kasutaja ei pruugi olla oma projekti kompileerinud, mis tähendaks, et kas otsitavaid faile pole olemas, mis tekitaks veateate või on nad vananenud, mis tekitaks ebakorrektseid tulemusi. Nende probleemide lahendamiseks otsustati, vähemalt selle töö lõpuks valminud pistikprogrammi versioonis, eeldada, et kasutaja kasutab Gradle'it. Seda otsust soodustas see, et Gradle'i olemasolu projektis suuresti hõlbustab testide kasutamist. See lubas kindlust *.class* failide asukoha suhtes ning lubas pistikprogrammil hõlpsalt Gradle'i kaudu vaatluse all olevat projekti kompileerida, lahendades sellega mõlemad probleemid.

Peegeldusel on mitmeid limitatsioone. Esiteks peab peegeldust teostavale programmile kõik kasutatav kood olema kättesaadav, kas siis *.class* või *.jar* failide kaudu, ehk siis peab programm teadma kus need asuvad. Sellest tekkis probleem, mis töö lõpuks jäi veel lahendamata, kus kui Testmotori genereeritud sisendite failis oli kasutusel mingi väline teek, mis oli näiteks Gradle'iga lisatud, siis peegelduse teostamisel seda teeki üles ei leitud ning testide genereerimine katkestati. Teiseks limitatsiooniks on, et mingi elemendi üles leidmiseks on selle nime vaja, mis tõttu tekkis ka sisendite failile limitatsioon, et testide jooksutamisel kasutatav funktsioon pidi olema kindla nimega, milleks valiti „f". Viimaks ilmnes probleem, et kuna IntelliJ kasutab ise Java 8-t, siis kui pistikprogrammi kasutaja projekt kasutab näiteks Java 11-st, siis peegeldust teha ei saa, sest vanem Java versioon uuema poolt kompileeritud koodi ära ei tunne. Kui kasutaja kasutab vanemat Java versiooni, siis probleemi ei teki, sest see on tagurpidi ühilduv. Õnneks tuli just 06.veebruaril 2019 välja uus IntelliJ versioon, mis lubab eksperimentaalselt kasutada uut *JetBrains Runtime 11*-st [9], mis töötab Java 11-st põhjal. See aga lahendas probleemi vaid ajutiselt, sest tulevikus, kui uued Java versioonid välja tulevad või kui Java 12-st laialdaselt kasutama hakatakse, tuleb see probleem uuesti esile.

4.4 Testmotori käivitamine

Pistikprogrammil on ka võimalus IntelliJ seest lihtsa nupuvajutusega Testmotorit käivitada. Töötavad ainult failid, mis vastavad nõutud formaadile, mis on varem täpsustatud sisendite faili formaat, kuigi seal ei pea sisendite massiive olema. Lühidalt on nõutavaks formaadiks mingi klass, mis asub samas paketis kui testitav klass ning millel on meetod nimega „f". Testmotori käivitamisel kõigepealt kompileeritakse Gradle abil kasutaja kood, et teha kindlaks, et vajalikud Java *.class* failid on olemas ja uuendatud. Seejärel otsitakse peegelduse abil lahti oleva faili *.class* vastest üles meetod f ning antakse see Testmotorile, et see f-ile sisendeid genereeriks. Testmotori enda käivitamiseks vajalik kood saadi selle repositooriumi README.md failist[3]. Vajalikuks juhuslike arvude genereerimise seemneks antakse Testmotorile käivitamise aegne kellaaeg millisekundites, nii et loodud sisendid peaks iga

[3] https://bitbucket.org/ENigola/testmotor/src/master/README.md

kord olema erinevad. Saadud sisendid lisatakse massiivina selle sama faili lõppu, kuid muidugi klassi sisse. Sellega ongi fail testide loomiseks valmis, kuigi kasutaja võib lasta Testmotoril veel sisendeid genereerida.

5. Tulemused

5.1 Pistikprogrammi seisukord

Töö lõpuks on valminud pistikprogramm, mis on võimeline Testmotorit valitud kohas käivitama ning selle väljastatud sisendid sinna samasse koodi lisama. Samuti saab seda pistikprogrammi kasutada Nende sisendite põhjal JUnit automaattestide genereerimiseks, kuigi sellel on mõned kasutust piiravad asjaolud.

5.2 Jäänud probleemid

Testmotori käivitamisel hetkel teada olevaid tõsiseid probleeme pole, kuid on paar väiksemat ebamugavust. Esimene on see, et iga kord kui sisendite massiivi Testmotoriga luuakse, siis pannakse selle nimeks „*data1*“, mis tähendab, et kui kasutaja tahab Testmotorit samas failis mitu korda kasutada peab ta ise vähemalt ühe nende nimedest ära muutma, muidu seda koodi kasutada ei saa. Teine väike ebamugavus on, et seda massiivi tehes pannakse selle tüübiks tüübi kogu nimetus koos paketiga kust seda leida. Näiteks *String* tüüpi väärtuseid hoidva massiivi tüübiks kirjutatakse *java.lang.String*. Koodis probleeme see ei tekita, kuid pole loetavuse küljest kuigi hea ning enamus kasutajaid tõenäoliselt tunneb vajadust see käsitsi ära muuta.

Testide genereerimisega on probleeme rohkem, mis tähendab, et koodil, mida sellega testida saab on mõned piirangud.

- Suurim piirang on, et testida saab ainult meetodeid, mille tagastusväärtus on kas primitiiv, näiteks *int*, või siis sõne. Väärtusi, mille loomiseks peab kasutama võtmesõna *new*, nagu objektid või massiivid ja muu selline, kasutada ei saa. Põhjuseks on, et seda on vaja testi sees kohapealt luua, kuid mingi juhusliku klassi objekti täpse koopia loomine nullist on väga keeruline, kui mitte täitsa võimatu, vähemalt viisil, mis töötaks kõikide klassidega.
- Veel üks suuremat sorti probleem on, et funktsioonis f ei saa kasutada koodi teekidest, mis on väliselt lisatud, näiteks Gradle’i kaudu. Põhjuseks on, et peegelduse kood ei leia neid teeke üles.
- Sisendite failis peab kõik vajalik kood peale sisendite massiivide olema enne sisendite massiive, sest muidu seda koodi testide faili üle ei kopeerita, sest regulaaravaldis, millega üle kopeeritavat koodi otsitakse eeldab, et sisendite massiivid on klassi lõpus.
- Kasutaja projekt peab kasutama Gradle’it ja kasutaja projekti struktuur peab olema see, mida Gradle vaikimisi teeb, muidu peegelduseks vajalikke kompileeritud faile üles ei leita.
- Sisendite failis olevate kommentaaridega peab olema natuke ettevaatlik, sest kui mingi jupp koodi on Java mitmerealise kommentaariga välja kommenteeritud, siis võivad kasutatud regulaaravaldised võtta seda kui tavalist koodi.
- Enne testide genereerimist kompileeritakse Gradle’it kasutades projekti klassid, kuid kasutajale sellest kuidagi teada ei anta.

Nendest probleemidest hoolimata töötab testide genereerimine sujuvalt niikaua kuni jäädakse standardteekide juurde ning kontrollitavate meetodite tagastusväärtused on primitiivid või sõned, mis on suurem osa juhtudest, eriti lähtudes õpetajate vajadustest. Ning eks see töö oligi mõeldud peamiselt prototüübina, sest kõikide võimalike testide võimaldamine ning genereerimine on väga suur töö.

5.3 Tulevikuvõimalused

Kui keegi seda pistikprogrammi edasi arendama hakkab, siis esiteks peaks muidugi nimetatud probleemid lahendama. Samuti võiks lisada mõningaid väiksemaid kasutajamugavust parandavaid asju, nagu näiteks genereeritud faili automaatne avamine pärast selle genereerimist või menüüst nuppude ära peitmine kui kasutaja pole kohas kus neid kasutada saaks.

Võimalikke teid edasi on muidugi veel. Näiteks võiks võimaldada pistikprogrammi kasutada ka projektides, mis Gradle'it ei kasuta. See lisaks kindlasti terve hulga potentsiaalseid kasutajaid. Samuti peaks tegema kogu asja rohkem paindlikumaks ning konfigureeritavamaks. Oleks hea kui saaks näiteks pistikprogrammi sätetes kasutaja ise täpsustada, mis võiks olla meetodi nimi, mida testides kasutatakse, praegu on see „f“, kuid see on lühike ning pole kirjeldav, nii et paljudele see tõenäoliselt ei meeldi. Samuti saaks kasutaja täpsustada kuidas loodavate elementide nimed saadakse.

Töö algstaadiumites oli ka mõte, et mall, mida kasutatakse testide genereerimiseks võiks vähemalt mingil määral olla kasutaja poolt muudetav. Esialgu kasvõi lihtsalt pakkuda mitut võimalust, mille vahel valida, pärast võib isegi lubada kasutajal oma malle kirjutada, mingite reeglite järgi.

Veel üks võimalik muudatus, mis tõenäoliselt tuleks kasuks on sisendfailist info kätte saamisel regulaaravaldiste asemel kasutada IntelliJ PSI-d nendes kohtades kus see on võimalid. PSI kaudu info saamine oleks tõenäoliselt robustsem kui regulaaravaldiste kaudu. Selles töös seda ei tehtud, sest kogu vajalikku infot PSI kaudu kätte pole võimalik saada ja otsustati, et aja kokkuhoiu mõttes pole mõtet seda kahte moodi teha, kuid kui aega on rohkem, siis see pole probleem.

Võimalik oleks ka laieneda teistele programmeerimiskeeltele nagu näiteks Python. Kuna PyCharm, mis on IDE Pythonile, töötab sama platvormi peal kui IntelliJ peaks olema üsna lihtne osa pistikprogrammist kasutada ka seal. Testmotor Pythonit ega mingit muud keelt peale Java ei toeta, nii et enne kui see muutub pole võimalik laieneda ning nii palju pistikprogrammi tööst on Java keskne, et kui tahta mõnele muule keelele sama pakkude oleks võib-olla mõistlik lihtsalt üsnagi otsast peale alustada.

6. Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli luua IntelliJ IDEA-le pistikprogramm, mis võimaldaks IntelliJ seest käivitada testide loomise töövahendit Testmotor, mis loob testide jaoks meetoditele sisendeid, ning selle väljundite põhjal automaatselt genereerida automaatteste. Selline pistikprogramm ka valmis.

Testmotori käivitamine töötab hästi, kuigi sellel on paar väikest ebamugavust kasutajale, mida kasutaja peab käsitsi parandama, näiteks, et loodud sisendite massiivi nimi on alati sama.

Testide genereerimine töötab samuti, kuigi sellega testitavate meetoditel on piirangud, hoolimata sellest, et testide genereerimise koodile kulus suur osa sellele tööle kulutatud ajast. Testitavad meetodid ei saa kasutada lisatud teekide koodi ning nende tagastustüüp peab olema kas primitiiv või sõne. Samuti on testide genereerimisel mõned väiksemad probleemid, kuid neid piiranguid järgides see töötab ning on täiesti kasutatav testide genereerimiseks. Kuna see pistikprogramm oli niikuinii mõeldud eelkõige õpetajatele, pole need piirangud ka kuigi olulised, sest nad kasutavad niikuinii õpetamisel peamiselt primitiive ja hoiavad oma projektid võrdlemisi lihtsad, et see oleks õpilastele arusaadav.

Osa sellest tööst oli ka sisendite faili jaoks sobiva formaadi välja mõtlemine, mida pistikprogramm saaks kasutada nii testide genereerimiseks kui Testmotorile vajaliku meetodi ette andmiseks.

7. Viidatud kirjandus

[1] E. Nigola, Testmotor repository. https://bitbucket.org/ENigola/testmotor/src/master/. (10.05.2019).

[2] Intellij IDEA website, Jetbrains. https://www.jetbrains.com/idea/. (15.01.2019).

[3] Intellij Platform, JetBrains. http://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html. (10.05.2019).

[4] EvoSuite plugin page. https://plugins.jetbrains.com/plugin/7985-evosuite-plugin. (15.01.2019).

[5] TestMe plugin page. https://plugins.jetbrains.com/plugin/9471-testme. (15.01.2019).

[6] SquareTest homepage, SquareTest. https://squaretest.com/. (15.01.2019).

[7] What is Apache FreeMarker™?, Apache Software Foundation. https://freemarker.apache.org/. (22.04.2019).

[8] Gradle User Manual. https://docs.gradle.org/current/userguide/userguide.html. (04.05.2019).

[9] Z. Kalyuzhnaya, "What's new in IntelliJ IDEA 2019.1 EAP 3?," JetBrains. https://blog.jetbrains.com/idea/2019/02/whats-new-in-intellij-idea-2019-1-eap-3/. (04.01.2019).

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Karl Jääts**,
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
IntelliJ IDEA-le testide loomise töövahendi Testmotor toe lisamine,
(*lõputöö pealkiri*)

   mille juhendaja on **Vesal Vojdani**,

   (*juhendaja nimi*)

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi

Tartus, **10.05.2019**