TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Ergo Nigola

Programmeerimisülesannetele puupõhiste testandmete genereerimise süsteem TestMotor

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2019

Programmeerimisülesannetele puupõhiste testandmete genereerimise süsteem TestMotor

Lühikokkuvõte:

Automaatsed testid on hea viis programmeerimisülesannete kontrollimiseks. Need vähendavad õppejõu tööd ning võimaldavad õpilasel saada kohest tagasisidet. Automaattestide kirjutamine võib aga olla raske ja üksluine töö, kuna tuleb välja mõelda hulk argumente, millega vaatlusaluseid meetodeid kontrollida. Selle protsessi lihtsustamiseks loodi programm TestMotor, mis võimaldab automaatselt genereerida argumente testitavatele Java meetoditele ja konstruktoritele. Loodud programm genereerib argumente puu kujul, mis võimaldab neid puu kõrguse järgi erinevateks raskusastmeteks grupeerida. Lisaks loodud programmi kirjeldusele, sisaldab töö ka selle analüüsi ning olemasolevate tehnoloogiate kirjeldusi.

Võtmesõnad:

Ühiktestimine, testandmete genereerimine, programmi analüüs, Java

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

TestMotor: A System for Tree-Based Test Data Generation for Programming Assignments

Abstract:

Automated tests are a good way for checking programming assignments. They decrease lecturer’s work and allow students to get immediate feedback. But writing automated tests can be difficult and repetitive, because a number of arguments need to be created, with which the methods under test will be tested. To ease this process, the program TestMotor was created, which allows automatic argument generation for Java methods and constructors. The program generates arguments as trees, which allows grouping them by the tree height into different levels of difficulty. In addition to the description of the created program, the thesis also contains its analysis and descriptions of some existing technologies in automatic test generation.

Keywords:

Unit testing, test data generation, program analysis, Java

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Tarkvara testimine on tänapäevase tarkvaraarenduse oluline osa, millega tagatakse tarkvara vastavus nõuetele. Testimist saab jaotada manuaalseks ja automaatseks. Automaatsed testid kujutavad endast programme, mis kontrollivad tarkvara käitumist automaatselt. Vähemalt osa automaattestide koodist peab testija ise kirjutama, et spetsifitseerida mida ja kuidas testida. Edaspidi saab neid jooksutada korduvalt, ilma, et testija peaks mingit tööd tegema. Üks testimise alamliike on ühiktestimine (ingl *unit testing*). Ühiktestid testivad tarkvara vähimate ühikute kaupa, milleks on näiteks meetodid ja konstruktorid [1]. Kuna käesoleva töö raames vaadeldakse ainult automaatseid ühikteste, siis nimetatakse lihtsuse mõttes neid edaspidi lihtsalt testideks.

Peale tarkvaraarenduse on testidest kasu ka programmeerimise õppetöös. Programmeerimisülesannetele saab kirjutada teste, mis kontrollivad automaatselt õpilase lahendust. Sellised testid võimaldavad õpilasel saada kohest tagasisidet oma lahendusele ning lihtsustavad õppejõu tööd. Programmeerimisülesannetele testide kirjutamine võib aga ise olla mahukas ülesanne. On vaja luua hulk testitavate funktsioonide kutseid erinevate argumentidega. Need argumendid võivad olla väga keerulised objektid, mille loomiseks tuleb välja kutsuda hulk meetodeid ja konstruktoreid.

Käesoleva töö eesmärk on luua programm, mis võimaldaks automaatselt genereerida testandmeid Java programmeerimisülesannete testidele, lihtsustades seeläbi õppejõudude tööd. Antud kontekstis on testandmeteks testitavate meetodite ja konstruktorite argumendid, kuna ühiktestid testivad just meetodite ja konstruktorite tööd. Töö teises peatükis kirjeldatakse testide automaatse genereerimise teoreetilist tausta ning kolmandas peatükis olemasolevaid tehnoloogiaid. Neljandas peatükis tuuakse praktiline näide programmeerimisülesandest ning sellele automaatselt genereeritud testidest. Viies peatükk sisaldab loodava programmi kirjeldust ning kuues peatükk tulemuste analüüsi.

2. Testide automaatne genereerimine

Manuaalse testide kirjutamise puhul kirjutab testija ise koodi, mis kontrollib programmi funktsionaalsuse vastavust nõuetele. Kuna paljudes testimise raamistikes omavad erinevad testid sarnast struktuuri ning samu koodi elemente, siis on võimalik lasta mõnel programmil automaatselt luua testi põhistruktuur ning seeläbi vähendada testijalt nõutavat manuaalset tööd. Seda teeb näiteks programm Squaretest[1] ning lihtsamal kujul on selline funktsionaalsuse juba osadel arenduskeskkondadel sisse ehitatud, näiteks IntelliJ IDEA-l [2]. See pole siiski täiesti automaatne testide genereerimine, kuna testide sisu tuleb testijal endal kirjutada. Kuid leidub ka programme, mis suudavad automaatselt genereerida täielikult valmis teste. Seda teevad näiteks programmid Randoop, EvoSuite ja IntelliTest, mida kirjeldatakse pikemalt järgmises peatükis. Testide koodi võib mõtteliselt jaotada kaheks: kontrollitava funktsionaalsuse rakendamine ja tulemuste kontrollimine. Automaatse testide genereerimise puhul tuleb mõlemate osade loomine automatiseerida. Algoritmiliselt on aga keeruline otsustada, mis järjekorras milliseid funktsioone välja kutsuda ning milliseid väärtusi lõpus võrrelda, et testist oleks ka kasu.

Üks testjuhtum (ingl *test case*) kontrollib tavaliselt vaid ühte programmi funktsiooni, mistõttu võib piirduda ainult selle väljakutsetega. Sellegipoolest võib olla vajalik mõne objekti kontrollimise puhul kutsuda välja suurem hulk erinevaid meetodeid, et viia objekt mõnda keerulisemasse seisundisse, kus võivad vead peituda. Keerulisem ülesanne on sobivate argumentide valimine funktsioonide kutsetele. Tuleb ka arvestada, et argumendid ei pruugi olla vaid primitiivset tüüpi (ingl *primitive type*), nagu arvud ja tõeväärtused, vaid võivad olla ka keerulised viittüüpi (ingl *reference type*) objektid, mille automaatne loomine ei ole enam triviaalne ülesanne. Argumentide valimiseks on olemas hulk erinevaid strateegiaid. Lihtsaimaks võib lugeda juhuslikku genereerimist, mis valib argumentideks näiteks juhuslikke täisarve, juhusliku pikkuse ja sisuga sõnesid ning juhuslike konstruktorite ja meetodite kutsete abil loodud objekte [3]. Enamus programme võimaldavad testijal seada piiranguid argumentidele, näiteks arvuliste väärtuste maksimaalne suurus, sõne minimaalne pikkus ja nullobjektide lubamine või mitte lubamine [3]. On ka teisi funktsioonide kutsete genereerimise strateegiaid, mis näiteks analüüsivad programmi koodi või iteratiivselt parandavad funktsioonidele ette antavaid argumente, analüüsides varasemate argumentidega saadud tulemusi.

[1] https://squaretest.com/

Teine osa testidest on tagastatud väärtuste ja objektide seisundite korrektsuse kontrollimine. Arvutile on aga keeruline kirjeldada, milline programmi käitumine olema peaks. Üks võimalus on lugeda õigeks need tulemused, mida testide genereerimise ajal saadi. Selliseid teste saab kasutada regressiooni testimisel (ingl *regression testing*). Need testid ei aita küll koheselt avastada vigu, kuid aitavad ära fikseerida koodi käitumise, et tulevikus avastada funktsionaalsuse muutusi, mis võivad tuleneda kas antud koodi enda muudatustest või mõne muu programmi osa muutmisest [4]. Osad testide automaatse genereerimise programmid võimaldavad siiski mõningal määral kirjeldada, kuidas funktsioonid peaksid käituma [3]. Lihtne kirjeldus, mida võib tihti vaja minna, on, et funktsioon ei tohi ühtegi erindit visata.

Kuna teste kasutatakse ka programmeerimisülesannete kontrollimisel, on ka seal automaatsest testide genereerimisest kasu. Ülalkirjeldatud strateegiad ei ole selle jaoks siiski alati parim lahendus, kuna tarkvaraarenduses ja programmeerimisülesannetes kasutatavatel testidel on erinevad eesmärgid ja eeldused. Kui programmeerimisülesande kõik testid ebaõnnestuvad mõne väikese vea tõttu koodis, on õpilasel raske seda üles leida. Seetõttu on kasulik, kui iga test kontrollib vaid väikest osa kogu nõutud funktsionaalsusest ning kui testid lähevad järk-järgult raskemaks. Programmeerimisülesannetele testide genereerimisel tuleb silmas pidada ka asjaolu, et genereerimine teostatakse näidislahenduse põhjal, kuid testid rakendatakse õpilase lahendusel. Seetõttu ei pruugi testitava funktsiooni koodi analüüsimine kasulikku informatsiooni juurde anda.

3. Olemasolevad tehnoloogiad

Selleks, et paremini mõista käesoleva töö raames valminud programmi ning selle uudset funktsionaalsust, kirjeldatakse selles peatükis lühidalt kolme juba olemasolevat rakendust, mis automaatselt teste genereerivad. Alustuseks antakse ka lühike ülevaade ühiktestimise raamistikust JUnit. Selle töö raames valmiv programm ei sõltu küll kasutatavast testimis-raamistikust, kuna programm genereerib vaid Java avaldisi, kuid kuna need avaldised tuleks mingi testraamistiku abiga rakendada, on asjakohane siin ühte neist lühidalt kirjeldada.

3.1 JUnit

JUnit[2] on enim levinud Java ühiktestide raamistik [5]. Joonisel 1 on toodud näide lihtsast JUnit (versioon 4) testklassist. Selles klassis on kolm testmeetodit, mis on tähistatud *@Test* annotatsiooniga (ingl *annotation*) ning *@Before* annotatsiooniga tähistatud meetod, mida kutsutakse enne iga individuaalset testmeetodit. Esimene testmeetod kontrollib, et klassi *Inimene* meetod *getNimi* tagastab õige nime objekt *juku* puhul. Kui peaks tagastatama ebakorrektne väärtus, väljastatakse sõnum „Jukul on vale nimi“. Teine testmeetod kontrollib, et objektil *juku* oleks korrektne vanuse väärtus ning et seda saaks ka muuta. Kolmas testmeetod kontrollib, et klassi *Inimene* objekt ei aktsepteeriks negatiivset vanuse väärtust. Kuna ootuspärane käitumine sellel juhul on erindi viskamine, on see kirjeldatud meetodile eelnevas annotatsioonis.

[2] https://junit.org/junit4/

```
public class InimeneTest {

   Inimene juku;

   @Before
   public void before() {
      juku = new Inimene("Juku", 14);
   }

   @Test
   public void testNimi() {
      assertEquals("Jukul on vale nimi", "Juku", juku.getNimi());
   }

   @Test
   public void testKorrektneVanus() {
      assertEquals(14, juku.getVanus());
      juku.setVanus(18);
      assertEquals(18, juku.getVanus());
   }

   @Test(expected = IllegalArgumentException.class)
   public void test() {
      juku.setVanus(-20);
   }
}
```

Joonis 1. JUnit näidiskood.

Kokku on näites kolm kontroll-lauset, mis antud juhul kujutavad endast erinevate *assertEquals* nimega meetodite kutseteid. Testitakse kolme meetodit: *getNimi*, *getVanus* ja *setVanus*. Esimesed kaks neist ei võta argumente, seega neile ei saaks testandmeid genereerida, kuid saaks genereerida nende kutseid ja kontrollida tagastatud väärtusi.

3.2 Randoop

Randoop[3] on Java koodi jaoks JUnit testide genereerimise programm. Randoop genereerib juhuslikke teste, kuid kasutab tagasiside süsteemi, et lõpptulemusena väljastada ainult testid, mis võivad potentsiaalselt anda uut informatsiooni programmi kohta [6]. Randoop loob juhuslikke järjestusi meetodite ja konstruktorite väljakutsetes, kuid kontrollib pärast iga uue väljakutse lisamist, et tulemusena saadud programmi seisund vastaks filtrite nõuetele ja ei oleks üleliigne ega vigane [6]. Randoopis on kirjeldatud vaikimisi hulk nõudeid, millele programmi seisund peab vastama, kuid seda on võimalik testijal täiendada. Randoopi loojad testisid programmi Java ja .NET teekidel, nii kolmandate osapoolte omadel kui ka standard-teekidel, mille tulemusena avastati 60 reaalset viga [6].

[3] https://randoop.github.io/randoop/

3.3 EvoSuite

EvoSuite[4] on JUnit testide automaatse genereerimise programm, mis kasutab geneetilist algoritmi. Geneetilise algoritmi mõte on iteratiivselt luua olemasoleva isendite populatsiooni põhjal paremaid isendeid, kasutades nende muteerimist ja ristamist ning seejärel parimate väljasorteerimist [7]. EvoSuite genereerib kandidaattestidest uusi teste, kasutades samuti ristamist ja muteerimist [8]. Alles jäetakse ainult parima headusmõõduga isendid. EvoSuite võimaldab valida erinevaid headusmõõtusid, näiteks koodi harude katvust (ingl *branch coverage*) ja lausete katvust (ingl *statement coverage*) [8].

Peale koodi katvuse (ingl *code coverage*) suurendamise, üritab EvoSuite optimeerida ka kontroll-lauseid. Selleks, et test kontrolliks programmi korrektsust võimalikult suure kindlusega, on vaja piisavalt põhjalikku tulemuste kontrolli. Samas loob suure juhusliku hulga kontroll-lausete valimine palju ebaolulisi kontrolle või võib funktsionaalsust üle-spetsifitseerida. Oluliste kontroll-lausete välja filtreerimiseks kasutab EvoSuite mutatsioon-testimist (ingl *mutation testing*) [8]. Mutatsioontestimise puhul luuakse testitavast programmist hulk mutante, millesse on tehislikult lisatud väikeseid vigu [9]. Kui test suudab tuvastada piisavalt palju ebakorrektse käitumisega mutante, võib seda pidada heaks testiks. Samas kui mõni testi kontroll-lausetest ei avasta ühtegi mutanti või avastab ainult vähesed, siis sellest võib järeldada, et see kontroll-lause ei ole informatiivne ega kasulik ning seega jäetakse see testist välja. Tulemusena jäävad alles vaid rohkem informatiivsed kontroll-laused.

3.4 IntelliTest

IntelliTest on Microsofti poolt välja töötatud testide automaatse genereerimise süsteem C# keele jaoks. Genereeritud testid võivad olla erinevate raamistike omad, sealhulgas MSTest, NUnit ja xUnit.net [10]. IntelliTest kasutab sümbolipõhist koodi täitmist (ingl *symbolic execution*), et saavutada võimalikult suur koodi katvus [11, 12]. Sümbolipõhise koodi täitmise mõte on omistada funktsiooni argumentidele konkreetsete väärtuste asemel sümbolid ning erinevate hargnemiskohtade juures teha kindlaks, millised tingimused peavad argumentide kohta kehtima, et koodi täitmine jätkuks ühes või teises suunas [13]. Sümbolipõhise koodi täitmise tulemuste põhjal valib IntelliTest sellised argumentide väärtused, et kogu testitav kood saaks kaetud. Genereeritavate testide jaoks on muuhulgas võimalik spetsifit-

[4] http://www.evosuite.org/

seerida, milliste erindite puhul testid läbi lähevad ja seada eeldusi funktsioonide argumentidele [10]. IntelliTest suudab ise luua objekte, mille vajalikud konstruktorid või meetodid on testklassidest nähtavad [11].

4. Praktiline näide

Järgnevalt tuuakse näide programmeerimisülesandest, millele võiks automaatselt teste genereerida. Seejärel näidatakse ja võrreldakse millised näevad välja õppejõu poolt käsitsi kirjutatud ja olemasolevate tehnoloogiate ning TestMotori poolt loodud testid.

Vaatleme programmeerimisülesannet, mille raames tuleb õpilasel kirjutada klassis *Evaluator* olev meetod *eval*, mis võtab argumendiks *LetAvaldis* isendi. *LetAvaldis* esitab mingi programmeerimiskeele *let*-avaldist, mille väärtustamisel *eval* meetodiga saadakse tulemuseks täisarv.

LetAvaldis on abstraktne klass, millel on järgmised alamklassid:

- *LetArv* – täisarv
- *LetMuutuja* – vaba muutuja
- *LetVahe* – kahe avaldise vahe
- *LetSidumine* – muutuja sidumine teatud avaldise piires
- *LetSumma* – tsüklis väärtuste kokku liitmine

Lisaks konstruktoritele, saab nende klasside isendite loomiseks kasutada klassi *LetAvaldis* vastavaid abimeetodeid:

- `num(int arv)`
- `var(String nimi)`
- `vahe(LetAvaldis vasak, LetAvaldis parem)`
- `let(String muutujaNimi, LetAvaldis muutujaSisu, LetAvaldis keha)`
- `sum(String tsüklimuutujaNimi, LetAvaldis low, LetAvaldis high, LetAvaldis keha)`

Nende klasside täpsete tähenduste mõistmisest olulisem on tähele panna, et need esitavad puu struktuuri: *LetVahe*, *LetSidumine* ja *LetSumma* tüüpi avaldiste loomiseks on vaja alam-avaldisi.

Õppejõu poolt kirjutatud testid sellele ülesandele võiks välja näha sellised, nagu joonisel 2. Erinevad testandmed on raskuse ja kontrollitava funktsionaalsuse järgi grupeeritud.

```
@Test
public void test01_eval_literaalVahe() {
   checkEval(num(0), 0);
   checkEval(num(97519), 97519);
   checkEval(vahe(num(10),num(5)), 5);
   checkEval(vahe(vahe(num(10),num(5)),num(9)), -4);
}

@Test
public void test02_eval_let() {
   checkEval(let("x",num(5),var("x")), 5);
   checkEval(let("x",num(5),vahe(var("x"),num(10))), -5);
   checkEval(let("x",num(100),num(5)), 5);
   checkEval(vahe(num(55),let("x",num(100),num(5))), 50);
   checkEval(var("a"), null);
}

@Test
public void test03_eval_multiLevel() {
   checkEval(let("x",num(1),let("y",num(5),sum("i",var("x"),var("y"),var("i")))), 15);
   checkEval(let("x",sum("i",num(1),num(4),var("i")),var("x")), 10);
   checkEval(sum("i",sum("j",num(1),num(100),num(0)),num(5),var("i")), 15);
   checkEval(let("x",vahe(var("x"),num(11)),var("x")), null);
}
```

Joonise 2. Väljavõte õppejõu poolt loodud testidest.

Joonisel 3 on toodud osa Randoopi poolt antud ülesandele genereeritud testidest. Randoopi loodavad testid on mõeldud põhiliselt regressiooni testimiseks. Need ei ole mõeldud olema inimesele kergesti loetavad ja mõistetavad. Randoop võimaldab spetsifitseerida nimekirja testitavatest meetoditest. See aga tähendab, et muude meetodite kutsed on keelatud. Kui tahetakse otseselt testida ainult ühte meetodit, aga lubada testandmete genereerimisel ka teiste kasutamist, siis seda ei võimaldata. Näites toodud testmeetod kutsub testitavat meetodit *eval* välja ainult ühe korra ja vigase sisendiga, ülejäänud kood kujutab endast muude ebaoluliste väärtuste kontrollimist.

```
@Test
public void test109() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test109");
    avaldis.ast.LetAvaldis letAvaldis2 = avaldis.ast.LetAvaldis.num((int) ' ');
    avaldis.ast.LetAvaldis letAvaldis4 = avaldis.ast.LetAvaldis.num((int) '4');
    avaldis.ast.LetAvaldis letAvaldis5 = avaldis.ast.LetAvaldis.vahe(letAvaldis2, letAvaldis4);
    avaldis.ast.LetAvaldis letAvaldis8 = avaldis.ast.LetAvaldis.num((int) (short) 1);
    avaldis.ast.LetAvaldis letAvaldis9 = null;
    avaldis.ast.LetAvaldis letAvaldis11 = avaldis.ast.LetAvaldis.num((int) (short) 100);
    avaldis.ast.LetAvaldis letAvaldis12 = avaldis.ast.LetAvaldis.vahe(letAvaldis9, letAvaldis11);
    avaldis.ast.LetAvaldis letAvaldis13 = avaldis.ast.LetAvaldis.let("hi!", letAvaldis8,
letAvaldis9);
    avaldis.ast.LetAvaldis letAvaldis14 = avaldis.ast.LetAvaldis.let("", letAvaldis2,
letAvaldis9);
    try {
        int int15 = avaldis.Evaluator.eval(letAvaldis9);
        org.junit.Assert.fail("Expected exception of type java.lang.IllegalArgumentException;
message: Unknown expression: null");
    } catch (java.lang.IllegalArgumentException e) {
    }
    org.junit.Assert.assertNotNull(letAvaldis2);
    org.junit.Assert.assertNotNull(letAvaldis4);
    org.junit.Assert.assertNotNull(letAvaldis5);
    org.junit.Assert.assertNotNull(letAvaldis8);
    org.junit.Assert.assertNotNull(letAvaldis11);
    org.junit.Assert.assertNotNull(letAvaldis12);
    org.junit.Assert.assertNotNull(letAvaldis13);
    org.junit.Assert.assertNotNull(letAvaldis14);
}
```

Joonis 3. Väljavõte Randoopi genereeritud testidest.

Joonisel 4 on toodud näide EvoSuite'i genereeritud testidest. EvoSuite ei võimalda spetsifitseerida testitavat funktsionaalsus meetodi haaval, vaid ainult klasside või pakettide kaupa. Samas on genereeritud testid kompaktsed ja inimesele lihtsamini mõistetavad. Kuna EvoSuite optimeerib koodi katvust, siis on enamus argumentide tüüpe testidega kaetud. Kuid nagu ka Randoopi puhul, puudub võimalus teste kontrollitava funktsionaalsuse või raskusastme järgi klassifitseerida.

```
@Test(timeout = 4000)
public void test0()  throws Throwable  {
  LetAvaldis letAvaldis0 = LetAvaldis.num(1);
  int int0 = Evaluator.eval(letAvaldis0);
  assertEquals(1, int0);
}

@Test(timeout = 4000)
public void test1()  throws Throwable  {
  LetAvaldis letAvaldis0 = LetAvaldis.num(2627);
  LetVahe letVahe0 = new LetVahe(letAvaldis0, letAvaldis0);
  LetAvaldis letAvaldis1 = LetAvaldis.vahe(letVahe0, letAvaldis0);
  int int0 = Evaluator.eval(letAvaldis1);
  assertEquals((-2627), int0);
}

@Test(timeout = 4000)
public void test2()  throws Throwable  {
  LetArv letArv0 = new LetArv(20);
  LetAvaldis letAvaldis0 = LetAvaldis.sum("R(D}nVP%d)&W3Kb WdB", letArv0, letArv0,
letArv0);
  LetVahe letVahe0 = new LetVahe(letAvaldis0, letArv0);
  int int0 = Evaluator.eval(letVahe0);
  assertEquals(0, int0);
}
```

Joonis 4. Väljavõte EvoSuite'i genereeritud testidest.

Erinevalt programmidest Randoop ja EvoSuite, ei genereeri TestMotor valmis teste, vaid testandmeid, mis esitavad testitava meetodi kutset. Testandmete faili kirjutamine ei ole samas keeruline ülesanne ning TestMotor pakub ka võimalust genereeritud avaldiste väärtustamiseks, et teada saada oodatav tulemus. Joonisel 5 on toodud näide TestMotori genereeritud testandmetest.

```
// num, vahe; kõrgused 2, 3, 4
eval(num(35))
eval(vahe(num(95), num(227)))
eval(vahe(vahe(num(107), num(357)), vahe(num(-65), num(283))))

// num, var, vahe, let; kõrgused 3, 4, 5
eval(let("7", num(66), num(27)))
eval(let("z", num(36), vahe(num(231), num(222))))
eval(vahe(let("u", num(70), vahe(num(265), num(242))), let("p", num(-24), var("p"))))

// num, var, vahe, let, sum; kõrgused 4, 5, 6
eval(let("4", num(204), sum("6BPzyF2K", num(265), num(131), var("E"))))
eval(sum("7", num(124), num(-15), vahe(num(98), vahe(num(155), num(347)))))
eval(let("u", num(263), vahe(num(-113), sum("7cs", num(-4), vahe(num(-18), num(152)),
let("9", var("b"), var("drlqsBGu"))))))
```

Joonis 5. TestMotoriga genereeritud testandmed.

TestMotor kasutab sisemiselt puu struktuuri genereeritud testandmete esitamiseks. See võimaldab määrata soovitud puu kõrguse, millega saab reguleerida testide keerukust. Samuti saab määrata, milliseid meetodeid ja konstruktoreid avaldiste genereerimisel kasutada võib, tänu millele saab testandmeid grupeerida kontrollitava funktsionaalsuse järgi. Joonisel 5 on kirjutatud iga testandmete grupi juurde, mis meetodeid lubati genereerimisel kasutada ning mis kõrgust igalt avaldiselt nõuti. Need testandmed on juba võrdlemisi sarnased õppejõu poolt loodutega.

5. Implementatsioon

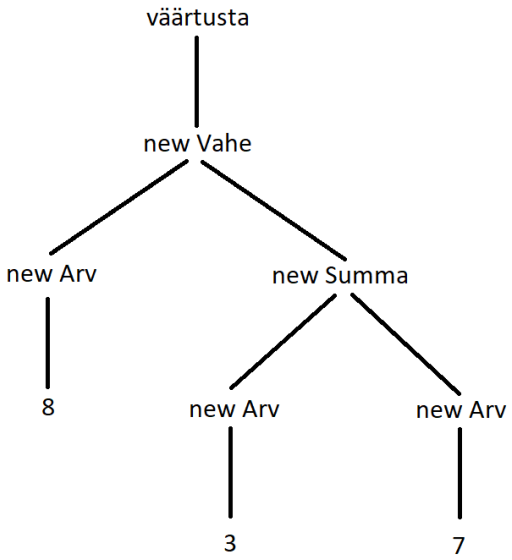
Käesoleva uurimustöö raames valmis programm TestMotor, mis on mõeldud programmeerimisülesannete testidele testandmete genereerimiseks. Testandmeid genereeritakse Java meetoditele ja konstruktoritele ning ka programm ise on kirjutatud Javas, sest selle standardteek juba sisaldab vajalikke klasse ja meetodeid testitava programmi meetodite ja konstruktorite signatuuride analüüsimiseks. Programm ei vaata meetodite ega konstruktorite sisse, vaid valib genereeritavad väärtused ainult signatuurides nõutavate argumentide tüüpide põhjal. TestMotor on mõeldud kasutamiseks teegi või raamistikuna, mis tuleks oma projekti lisada ning seejärel saab selle vajalikke klasse ja meetodeid kasutada. Programmi testimise ja katsetamise eesmärgil on seda võimalik ka käsurealt käivitada, andes vajalik informatsioon edasi programmi argumentidena, kuid sellel moel on ligipääsetav ainul osa funktsionaalsusest. Viide programmi lähtekoodile koos käivitamise instruktsioonidega on lisades. Lähtekood on dokumenteeritud Javadoc kommentaaridega. Järgnevalt antakse ülevaade loodud programmi disainist, funktsionaalsusest ja tööpõhimõttest.

5.1 Meetodi kutse kui puu

TestMotor käsitleb iga genereeritud meetodi või konstruktori kutset kui puud. Puu tippudeks on meetodite ja konstruktorite kutsed, loendtüübid (ingl *enumerated type* või *enum*), literaalid ja massiivid. Loendtüübid ja literaalid on alati lehttipud. Meetodite ja konstruktorite kutsed ning massiivid võivad samuti olla lehttipud, aga üldjuhul omavad nad alamtippe, mis esitavad kutse argumente, massiivi elemente või isendimeetodite (ingl *instance method*) puhul objekte, mille peal meetodit kutsutakse. Iga puu esitab korrektset Java avaldist. Joonisel 6 on toodud näide kunstliku meetodi *väärtusta* kutsest ning joonisel 7 näide selle esitusest puuna.

```
väärtusta(new Vahe(new Arv(8), new Summa(new Arv(3), new Arv(7))))
```

Joonis 6. Näide Java avaldisest.



Joonis 7. Joonisel 6 toodud avaldise puu.

Programmis on puu tippude ühiseks ülemklassiks abstraktne klass *ValueNode*. Sellel klassil on kolm alamklassi: *LeafNode*, mis esitab literaale ja loendtüüpe, *ExecNode*, mis esitab meetodi või konstruktori kutset, ning *ArrayNode*, mis esitab massiivi. Puu tippe esitavad klassid ei oma klassikalisi puule omaseid meetodeid, millega saaks näiteks puu kõrgust arvutada või alamtippe kätte, kuna nende jaoks pole otsest vajadust. Puu struktuuri on aga oluline silmas pidada, sest genereerimise protsessis arvestatakse genereeritava puu kõrgusega. Klass *ValueNode* ja tema alamklassid omavad see-eest meetodeid *stringify* ja *invoke*. Meetod *stringify* teisendab puu legaalseks Java avaldiseks, näiteks nagu joonisel 6, mida saab testmeetodi koodi asetada. Erinevalt joonisel 6 toodust, on kõik nimed täielikult kvalifitseeritud (ingl *fully qualified*) ehk neile eelneb ka Java paketi ja klassi nimetus, et Java kompilaator saaks aru, millele täpselt viidatakse. Meetod *invoke* teeb põhimõtteliselt sama, mida Java standardteegi klassi *Method* samanimeline meetod ehk kutsub meetodi välja ja tagastab väärtuse. Kuna tipud võivad esitada ka muid avaldisi peale meetodi kutsete, näiteks literaale ja konstruktorite kutsed, siis on tegelikult see meetod sarnasem avaldise väärtustamisele. Argumente meetodile *invoke* andma ei pea, kuna tipu alamtipud ongi meetodi või konstruktori argumendid. Testandmete genereerimise protsessi käigus luuaksegi selline puu struktuur, mille juurtipuks on testitava meetodi või konstruktori kutse. Pärast testandmete (ehk puu) genereerimist tagastatakse loodud puu juurtipp.

5.2 Programmi rakendamine

Programmi keskne klass on *TestMotor*, mis omab vajalikke meetodeid testandmete genereerimiseks ning koordineerib seda protsessi. Testandmete genereerimiseks tuleb kõigepealt luua *TestMotor* isend. Konstruktorile võib argumendina anda listi kasutaja poolt loodud väärtuste generaatorite klassidest. Konstruktoris initsialiseeritakse juhuarvugeneraator, väärtuste generaatorid ja *Reflections* isend, kahte viimast kirjeldatakse pikemalt allpool. Pärast *TestMotor* isendi loomist saab selle peal kutsuda meetodit *generateTestData*, mille kaks olulisemat argumenti on testitav meetod või konstruktor ning genereeritava puu kõrgus. Kui meetodi või konstruktori kutse genereerimine õnnestub, tagastatakse tulemuseks puu juurtipp tüübiga *ValueNode*. Kui genereerimine ei õnnestu, visatakse *GenerationException*, mis on kontrollitud (ingl *checked*) tüüpi erind. Näide testsisendi genereerimisest on toodud joonisel 8 ning joonisel 9 on selle koodi võimalik väljund.

```
try {
    Executable executable = Date.class.getMethod("after", Date.class);
    TestMotor tm = new TestMotor();
    tm.setSeed(42);
    for (int i = 0; i < 3; i++) {
        ValueNode testCall = tm.generateTestData(executable, 2, null, false);
        System.out.println(testCall.stringify());
    }
} catch (GenerationException e) {
    System.out.println("Meetodi kutse genereerimine ebaõnnestus");
} catch (NoSuchMethodException e) {
    throw new RuntimeException(e);
}
```

Joonis 8. TestMotoriga testsisendi genereerimine.

```
new java.util.Date(53, 276, 182, 199, -58).after(new java.sql.Date(89, 320, -17))
new java.util.Date().after(new java.sql.Timestamp(-284L))
java.util.Date.from(java.time.Instant.now()).after(new java.sql.Date(-347L))
```

Joonis 9. Joonisel 8 toodud koodi väljund.

Kuigi toodud näites antakse programmile ette fikseeritud juhuarvugeneraatori seeme, võivad tulemuseks genereeritud meetodi kutsed siiski erinevates süsteemides erineda, kuna see oleneb ka jooksmise ajal kättesaadavatest klassidest ja Reflections teegist. Meetod *generateTestData* võtab lisaks ülal mainitud argumentidele veel hulga lubatud konstruktoritest ja meetoditest ning tõeväärtuse, mis näitab kas eriendeid viskavate kutsete genereerimine on lubatud. Lubatud konstruktorite ja meetodite listi spetsifitseerimine on kasulik, kui tahetakse, et teatud klasside isendeid loodaks ainult neid kasutades. Kuna klassi

Date meetod *after*, mille jaoks testandmeid genereeritakse, on isendimeetod, siis sellele genereeritakse lisaks argumentidele ka isend, mille peal seda meetodite kutsutakse.

5.3 Testandmete genereerimise protsess

Testandmete genereerimine koosneb kahest alamprotsessist: meetodite ja konstruktorite kutsete genereerimine ja nõutud tagastustüübiga avaldiste genereerimine. Genereerimise käigus kutsuvad need protsessid enamasti üksteist ja iseennast korduvalt esile. Kui on vaja genereerida testandmeid mingile meetodile, siis on vaja genereerida selle meetodi kutse. Selleks on aga vaja genereerida selle meetodi iga argumendi jaoks avaldis, mille tagastustüüp ühtiks argumentides nõutuga. Need avaldised võivad aga omakorda olla meetodite kutsed ning nende loomiseks on jällegi argumendid vaja genereerida. Selline protsesside vaheldumine kestab kuni saavutatakse nõutud puu kõrgus ning ollakse jõutud lehttippudeni välja, kuna need ei vaja enam alamatippude genereerimist.

Meetodite ja konstruktorite kutsete genereerimise loogika asub klassi *TestMotor* meetodis *generateCall*. See meetod võtab argumentidena meetodi või konstruktori, mille kutse on vaja genereerida, ning puu nõutava kõrguse. Meetodite ja konstruktorite esitamiseks on Java standardteegis klassid *Method* ja *Constructor*, mis omavad ühist ülemklassi *Executable*, tänu millele ei pea programm tingimata meetoditel ja konstruktoritel vahet tegema, kuigi osades kohtades on see endiselt vajalik. Meetodi *generateCall* ülesanne on vaadata, mis tüüpi väärtusi on argumentideks vaja genereerida ning kui kõrged peaksid nende puu esitused olema. Selleks, et meetodi või konstruktori kutset esitava tipu kõrgus oleks *h*, peab vähemalt üks tema alamtippudes olema kõrgusega (*h*-1) ning ülejäänud maksimaalselt kõrgusega (*h*-1). Programm üritab võimaldada võimalikult erinevate puude genereerimist ning tagada võimalikult suurt genereerimise õnnestumise tõenäosust. Selleks alustatakse juhusliku alamtipu genereerimisega, millelt nõutakse kõrgust (*h*-1), ülejäänud alamtipud üritatakse genereerida juhusliku kõrgusega 0 kuni (*h*-1). Kui mõnda alamtippu ei suudeta genereerida esialgselt valitud kõrgusega, proovitakse mõnda muud kõrgust. Kui ei suudeta ühtegi alamtippu genereerida kõrgusega (*h*-1) või mõnda alamtippu kõrgusega 0 kuni (*h*-1), siis visatakse erind.

Nõutava tagastustüübiga avaldise genereerimisega tegeleb meetod *generateValue*, mis võtab argumentideks tüübi ja kõrguse. Tüüpe esitatakse Java standardteegi liidest *Type* implementeerivate klassidega. Enamasti on selleks klass *Class*, millega saab esitada kõiki Java niiöelda tooreid tüüpe, milleks saavad olla muuhulgas massiivitüübid, loendtüübid, primitiivsed

tüübid, liidesed ja klassid. Üldisema tüübi *Type* kasutamine on vajalik geneeriliste tüüpide puhul. Näiteks kui mõne meetodi argumendi tüüp on *List<String>*, siis seda ei saaks klassiga *Class* esitada. Meetod *generateValue* ise väärtusi tegelikult ei genereeri. Selle meetodi ülesanne on leida üles õige väärtuste generaator, mis suudab nõutud tüüpi väärtusi genereerida, ning siis sellele töö üle anda.

5.4 Väärtuste generaatorid

Nõutavat tüüpi väärtuste genereerimiseks on programmis generaatorklassid. Täpsemini loovad need generaatorid puu esituses Java avaldisi, mille tagastustüüp on see, mis argumentides nõutud. Lisaks programmiga kaasa tulevatele generaatoritele saab kasutaja lisada enda omi. Kõik väärtuste generaatorid on abstraktse klassi *ValueGenerator* alamklassid. Sellel klassil on kaks abstraktset meetodit: *canGenerate* ja *generate*. Esimene neist peab vastama, kas generaator suudab argumendina spetsifitseeritud tüüpi väärtusi genereerida. Teine meetod genereerib päriselt neid väärtusi, mis tähendab, et selle tagastustüüp on *ValueNode*. Argumentideks võtab *generate* genereeritava väärtuse tüübi ja puu nõutava kõrguse.

Lisaks meetoditele *canGenerate* ja *generate*, on klassis *ValueGenerator* ka meetodid *canGenerateGeneric* ja *generateGeneric*, mis on analoogsed esimese kahega, aga suudavad ka geneerilist tüüpi väärtusi luua. Meetoditele *canGenerate* ja *generate* antakse tüüp ette *Class* isendina, vastavatele geneeriliste tüüpide versioonidele aga üldisema *Type* liidest implementeeriva klassi isendina. Kuna paljude tüüpide puhul, nagu primitiivsed tüübid ja mittegeneerilised klassid, ei seostata nendega geneeriliste tüüpide informatsiooni, piisab lihtsama *Class* isendi kasutamisest, mis võimaldab lühemat ja lihtsamat meetodi koodi. Meetodid *canGenerateGeneric* ja *generateGeneric* ei ole abstraktsed ning esimene neist tagastab alati tõeväärtuse väär ja teine viskab erindi. Kui generaator toetab geneerilisi tüüpe, siis tuleks nendele meetoditele tegelik implementatsioon kirjutada.

Programmiga tulevad kaasa järgmised generaatorid:

- *BooleanGenerator* – tõeväärtuste generaator
- *CharGenerator* – trükimärkide generaator
- *NumberGenerator* – igat tüüpi numbrite generaator
- *StringGenerator* – sõnede generaator
- *EnumGenerator* – loendtüüpide generaator

- *ArrayGenerator* – massiivide generaator
- *CollectionGenerator* – kogumike generaator
- *ObjectGenerator* – *Object* klassi isendite generaator
- *AnyClassGenerator* – üldine objektide generaator

Literaalide (primitiivsed väärtused ja sõned) generaatorid valivad juhuslikke väärtusi teatud vahemikest. Massiivide generaator loob juhusliku pikkusega massiive, mis koosnevad nõutud tüüpi objektidest. Massiivi elementide genereerimiseks kasutatakse klassi *TestMotor* meetodit *generateValue*. Kogumike generaator on hulkade, listide ja järjekordade loomiseks ning genereerib analoogselt massiividega juhusliku suurusega kogumikke. Massiivide ja kogumike generaatorid on ainukesed programmiga kaasa tulevad generaatorid, mis toetavad geneerilisi tüüpe. Massiivide puhul tähendab see seda, et massiivi elemendid võivad olla geneerilist tüüpi.

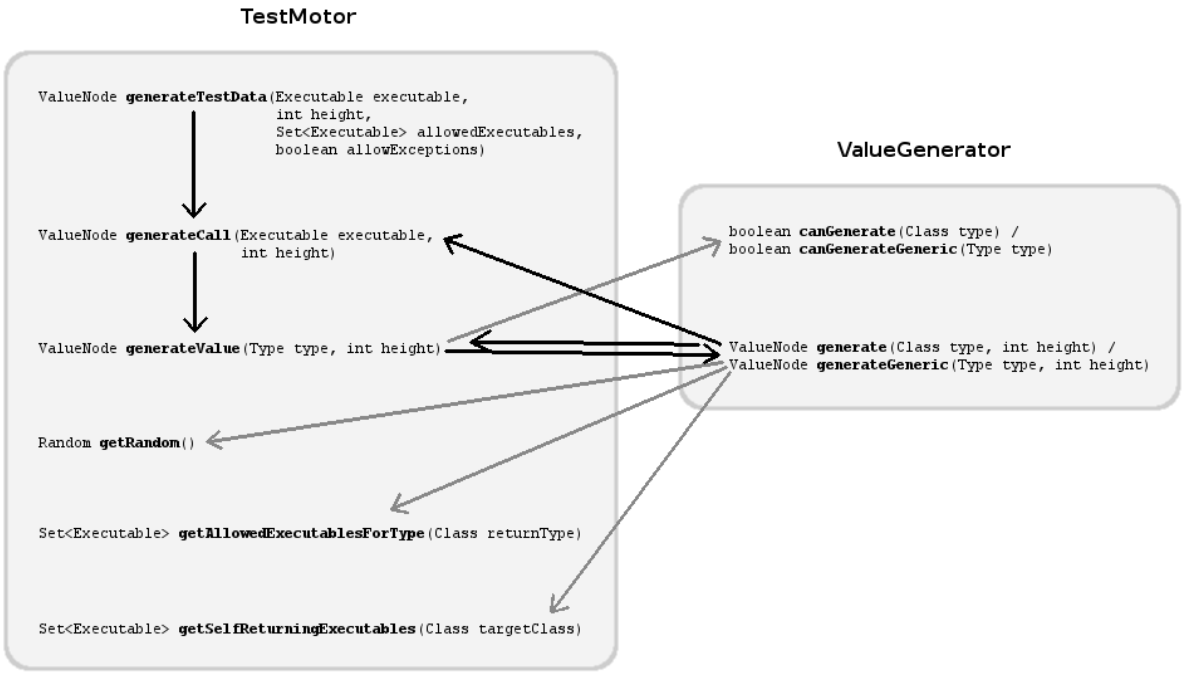
AnyClassGenerator on kõige üldisem generaator, mis suudab genereerida igat tüüpi väärtusi peale primitiivsete ja massiivitüüpide, kuid see ei arvest geneeriliste tüüpide tüübi-argumentidega. Selleks, et genereerida avaldis tüübiga *A*, kus *A* on kas klass või liides, tuleb valida mõni konstruktor tüübist *A* või selle alamtüüpidest (ehk alamklassidest ja -liidestest) või meetod, mille tagastustüüp on *A*. Testandmete genereerimist alustades on võimalik meetodile *generateTestData* anda hulk lubatud meetodite ja konstruktoritega. Kui see hulk sisaldab tüübi *A* isendit tagastavaid meetodeid või konstruktoreid, siis tuleb kasutada ühte nendest. Kui aga selles hulgas ei ole ühtegi sellist meetodit ega konstruktorit, kasutatakse klassis *TestMotor* olevat abimeetodit *getSelfReturningExecutables*. See meetod otsib tüübist *A* ja selle alamtüüpidest üles kõik avalikud konstruktorid ja meetodid, mille tagastustüüp on *A*. Kuna Javas ei oma tüübid informatsiooni oma alamtüüpide kohta, siis tuleb selle informatsiooni leidmiseks klassid ja liidesed programmi jooksmise ajal ära indekseerida [14]. Selleks kasutatakse teegi Reflections[5] abi. Selle teegi abiga saab kõik programmile kätte-saadavad tüübid ära indekseerida, misjärel saab iga tüübi alamtüüpe pärida. Kui generaator on hulga kasutatavaid meetodeid ja konstruktoreid kätte saanud, valitakse neist juhuslikult üks ning proovitakse genereerida selle kutse, kasutades klassi *TestMotor* meetodit *generateCall*. Kui valitud meetodi või konstruktori puhul ei õnnestu õige kõrgusega kutset genereerida, valitakse juhuslikult järgmine. Loendtüüpide generaator võib luua otsese viite

[5] https://github.com/ronmamo/reflections

loentüübi konstandile, kuid võib ka käituda sarnaselt nagu *AnyClassGenerator* ning luua meetodi kutse, mis tagastab selle loendtüübi isendi.

Klass *ObjectGenerator* on mõeldud *Object* tüüpi isendite loomiseks. Tehniliselt on kõik Java klassid tüüpi *Object*, aga seda generaatorit rakendatakse ainult juhul, kui nõutav tüüp on täpselt *Object* ehk ei ole mõni selle alamklass. See generaator eksisteerib genereeritavate testandmete lihtsuse ja programmi optimeerimise eesmärgil. Kui *Object* tüüpi väärtuste genereerimiseks kasutataks klassi *AnyClassGenerator*, läheks kaua aega, sest sobivaid konstruktoreid ja meetodeid otsitaks kõigist klassidest. Lisaks oleks siis genereeritav väärtus mõnest täiesti suvalisest klassist, mis võiks testandmetes segadust tekitav välja näha. Selle asemel genereerib *ObjectGenerator* ainult sõnesid ja klassi *StringBuilder* isendeid. Sõnedest üksinda ei piisaks, sest *StringGenerator* genereerib ainult sõne literaale, mis on puu mõttes alati kõrgusega 0. Seega juhul, kui nõutav puu kõrgus on suurem kui 0, genereeritakse klassi *StringBuilder* isend. *Object* tüüpi väärtuste genereerimise vajadus tekib enamasti juhtudel, kui käsitletakse geneerilisi klasse, mille jaoks tüübiargumentidega arvestamise tuge ei ole. Näiteks Java standardteegi klassil *Stream<T>* on meetod *of(T t)*. Kuna selle klassi jaoks tüübiargumentidega arvestamise tuge ei ole, siis näeb programm seda meetodit kujul *of(Object t)*.

Joonisel 10 on diagramm testandmete genereerimise protsessis väljakutsutavate olulisemate meetoditega. Mustad nooled näitavad, kuidas meetodid puu tippude ja tipu alamtippude genereerimist üksteisele delegeerivad. Hallid nooled näitavad genereerimise protsessis kasutatavate olulisemate abimeetodite kutseid. Milliseid meetodeid klassi *ValueGenerator* meetodid *generate* ja *generateGeneric* täpselt välja kutsuvad, oleneb kasutatavast generaatorist.



Joonis 10. Testandmete genereerimise protsessi meetodite kutsed.

Lisaks programmiga kaasa tulevatele väärtuste generaatoritele, saab kasutaja lisada enda omi. Selleks tuleb luua klassi *ValueGenerator* alamklassid ning anda list nende klasside viidetega *TestMotor* isendi loomisel konstruktorile. Klasside viidete andmist kasutatakse mugavuse eesmärgil, generaatori isendi loob programm automaatselt. Peale klassis *ValueGenerator* olevate abstraktsete meetodite implementeerimise, peab generaatorklass omama ka avalikku konstruktorit, mis võtab ainsa argumendina *TestMotor* isendi. Viidet *TestMotor* isendile on programmil vaja, et pääseda ligi selle klassi isendimeetoditele, nagu *generateCall*, *generateValue* ja *getRandom*. Kui mitu generaatorit suudavad genereerida sama tüüpi väärtusi, siis kasutatakse alati seda, mis esines generaatorklasside listis esimesena, kusjuures kasutaja poolt loodud generaatorid on alati suurema prioriteediga kui programmiga kaasatulevad.

6. Tulemuste analüüs

Selles peatükis analüüsitakse pikemalt tehtud töö tulemusi. Kirjeldatakse programmi testimist, testandmete genereerimise ajakulu, programmi nõrkusi ja tugevusi ning edasi-arendamise võimalusi.

6.1 Rakenduse testimine

Rakenduse korrektsuse tagamiseks teostati nii arenduse käigus kui ka pärast programmi valmimist testimist. Suur osa programmi funktsionaalsusest on kaetud automaatsete ühik-testidega, mis tagavad programmi korrektse töö ka tulevikus tehtavate muudatuste ja täienduste puhul. Automaattestid on kirjutatud JUnit 5 raamistikus. Joonisel 11 on toodud nimekiri kirjutatud testklassidest ja -meetoditest. Meetodite nimed annavad aimu, mida iga meetod testib.

- TestMotorTest
  - testAllowedExecutables()
  - testBasicTestDataGeneration()
  - testExceptionsNotAllowed()
  - testGenericCollectionsGeneration()
  - testCombinatorialTestDataGeneration()
  - testImpossibleGeneration()
- ArrayNodeTest
  - testStringify()
  - testInvoke()
- ExecNodeTest
  - testInstanceMethodWithNoParent()
  - testStringify()
  - testInvoke()
- LeafNodeTest
  - testNonMatchingTypes()
  - testStringify()
  - testInvoke()

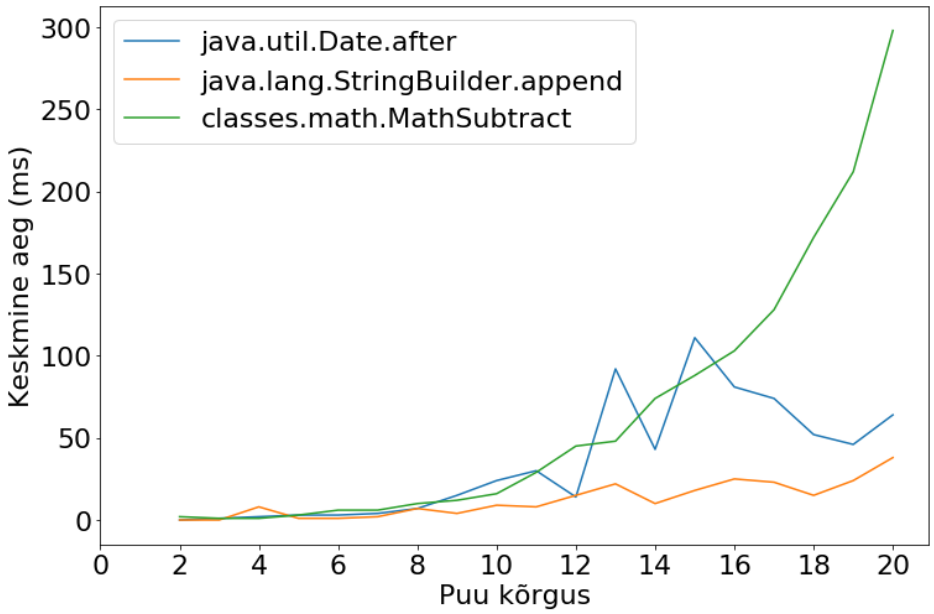
Joonis 11. Programmi automaatsed ühiktestid.

Programmi automaatse testimise teeb keeruliseks asjaolu, et testandmete genereerimine ei pruugi anda alati samu tulemusi. Genereerimise tulemust mõjutavad näiteks programmile kättesaadavad klassid ja Reflections teegi poolt tagastatavate klasside järjekord. Seetõttu kontrollivad automaattestid pigem väikeseid individuaalseid komponente. Ülejäänud funktsionaalsus kontrolliti manuaalse testimisega. Sealhulgas veenduti, et programm suudab genereerida igat tüüpi väärtusi ning et genereeritavad puud on alati õige kõrgusega.

TestMotor on kirjutatud Java versioonis 8. Kuna programmi koodis kasutatakse versiooniga 8 tulnud funktsionaalsust, siis peab koodi jooksutama minimaalselt Java versiooniga 8. Käesoleva töö kirjutamise ajal ei tööta Reflections teek (versioon 0.9.11) hilisemate Java versioonidega kui 8 korrektselt, mistõttu on kogu programmi Java versioonide ülempiir samuti 8. Meeldetuletuseks: Reflections teeki kasutatakse klasside ja liideste alamklasside ja alamliideste leidmiseks. TestMotori enda kood on Java versiooniga 11 ühilduv ja testitud, kuid kuni ei ole leitud asendust Reflections teegile, ei ole Java versioonist 8 hilisemate versioonidega programmi jooksutamise tulemused alati ootustpärased. Testandmeid endiselt genereeritakse, aga nende otsinguruum on piiratud, mistõttu genereerimine ebaõnnestub tihedamini ning loodavad testandmed on üksluisemad. TestMotorit on võimalik jooksutada iga arvuti peal, mis toetab vastava Java versiooni programmide jooksutamist.

6.2 Genereerimise ajakulu

Rakenduse efektiivsuse hindamiseks mõõdeti testandmete genereerimiseks kuluvat aega erinevate puu kõrguste puhul. Mõõtmised teostati kahe Java standardteegi klassi *Date* ja *StringBuilder* meetoditel *after* ja *append* ning töö autori poolt loodud klassi *MathSubtract* konstruktoril. Iga testitava meetodi või konstruktori puhul genereeriti puid kõrgustega 2 kuni 20, iga kõrguse puhul 50 korda. Tulemused on toodud joonisel 12, kus x-teljel on puu kõrgus ning y-teljel selle kõrguse jaoks kulunud keskmine aeg millisekundites. Siinkohal tasub meelde tuletada, et enne testandmete genereerimisega alustamist, tuleb luua *TestMotor* isend, mille konstruktoris luuakse omakorda Reflections teegist pärit klassi *Reflections* isend. Selle klassi loomise ajal indekseeritakse ära kõik programmile saadavad klassid ja liidesed, mis testimiseks kasutatud arvutil võttis keskmiselt aega umbes 5 sekundit. Seda tuleb teha vaid ühekordselt programmi käivitamise ajal. Nagu jooniselt näha, oleneb genereerimiseks kuluv aeg suuresti meetodist või konstruktorist, mille jaoks testandmeid genereeritakse. Mõõtmised teostati arvutil protsessoriga Intel Core i7-4600U.



Joonis 12. Genereerimiseks kuluv aeg olenevalt puu kõrgusest.

Klass *MathSubtract* esitab matemaatilise avaldise miinusmärki. Selle konstruktor võtab argumentideks kaks teist matemaatilist avaldist, mis võivad olla kas binaarsed operatsioonid või arvud. Arvu esitava klassi konstruktor võtab argumendiks ühe täisarvu, mistõttu on selle klassi isendi kõrgus alati üks. Kui on vaja genereerida kõrgemat puud, tuleb kasutada mõnda binaarse operatsiooni klassi. Kuna binaarsed operatsioonid nõuavad kahte alamavaldist, kasvab nende kasutamisel puu laius. Seetõttu kasvab genereerimiseks kuluv aeg puu kõrguse kasvades kiiresti. Kuna programm ei nõua, et genereeritavad puud oleks tasakaalus ehk et puu alamtipud oleks sarnaste kõrgustega, siis ei ole genereerimine ruutkeerukusega, kuid keskmiselt puu laius iga järgmise kõrgusega siiski kasvab. Kõrgete matemaatiliste avaldiste puu genereerimisel on samas heaks küljeks asjaolu, et arvu esitava klassi konstruktor on ainsana ebasobiv. Kui juhuslikult valitakse esimesena see konstruktor, siis visatakse praktiliselt kohe erine, sest sellega ei ole võimalik kõrgeid puid genereerida, ning valitakse mõne binaarse operatsiooni konstruktor, mis igal juhul sobib. Mõnda teist tüüpi väärtuste genereerimisel võidakse aga proovida läbi palju suurem hulk ebasobilikke meetodeid ja konstruktoreid, enne kui sobivani jõutakse.

Klass *StringBuilder* on mõeldud sõnede nii-öelda ehitamiseks. Selles on hulk isendi-meetodeid nimega *append*, mis lisavad argumendina antud väärtuse ehitatava sõne lõppu ning tagastavad selle sama *StringBuilder* isendi, mille peal meetodit kutsuti. Erinevad versioonid meetodist *append* võtavad erinevat tüüpi argumendi, näiteks sõne, numbri või trükimärgi. Mõõtmiste käigus kasutati versiooni, mille argument on tõeväärtus. Lisaks on selles klassis

ka hulk muid meetodeid, mis võtavad mõne lihtsat tüüpi argumendi, näiteks primitiivse väärtuse, ning tagastavad isendi, mille peal meetodit välja kutsuti. Kuna enamuste meetodite argumendid on liiga lihtsat tüüpi, et nende jaoks kõrgeid puid genereerida, tuleb kõrguse saavutamiseks kasutada isendeid, mille peal meetodeid kutsutakse. Näiteks kõrguse 6 puhul võiks tulemuseks olla midagi sellist:

```
new StringBuilder().append('a').append(3).insert(1,"str").reverse().append(true)
```

Selle avaldise puhul oleks juurtipuks kõige viimane meetodi *append* kutse. Selliste avaldiste puhul püsib puu laius üsna väiksena, mistõttu kasvab genereerimiseks kuluv aeg kõrguse suhtes praktiliselt lineaarselt.

Klassis *Date* asuv isendimeetod *after* võtab argumendiks sama klassi, *Date*, isendi. See klass omab palju alamklasse ning erinevaid meetodeid ja konstruktoreid, mida selle loomiseks saab kasutada. Seetõttu erinevad tulemuseks genereeritavad puud üksteisest palju ning ka genereerimisajad varieeruvad. Sama kõrgusega puude genereerimise aeg võib erineda üle kümne korra. Joonisel toodud väärtused tähistavad 50 korra keskmist, kuid ka seal on märgata, et genereerimiseks kuluv aeg ei kasva alati sujuvalt.

Praktikas tekib harva vajadus suurte puu kõrguste järele. Kõrguse 10 puhul võib kirjeldatud meetodi *after* jaoks genereeritav avaldis olla üle 500 tähemärgi pikk ning *MathSubtract* konstruktori puhul juba üle 3000. Töö autori hinnangul peaks piisama maksimaalselt kõrguse 7 kasutamisest, sest selle puhul on testandmed juba piisavalt keerulised, et testitavas programmis vead üles leida.

6.3 Tugevused ja nõrkused

TestMotor genereerib testandmeid ilma meetodite ja konstruktorite sisse vaatama, analüüsides ainult nende signatuure. Seetõttu ei mõista programm argumentide tähendust. Kui mõni meetod lubab näiteks ainult positiivseid täisarve vahemikus 0 kuni 100, genereeritaks sellele meetodile palju vigaseid argumente. Juhul kui see meetod viskab erindi ebakorrektse argumendi puhul, saab ära keelata erindeid viskavad kutsed. Sellel juhul genereerib programm testandmeid korduvalt, kuni luuakse selline, mis erindeid ei viska, või jõutakse fikseeritud arvu lubatud katseteni, mis juhul visatakse erind. Vaikeväärtusena on selleks arvuks 30, kuid kasutaja saab seda muuta.

Kui meetod nõuab väga spetsiifilisi argumente, näiteks kindla struktuuriga sõne, siis erindite keelamisest ilmselt ei piisa. On väga ebatõenäoline või osadel juhtudel isegi võimatu, et sõnede generaator loob juhuslikult mingi spetsiifilise struktuuriga sõne, näiteks sellise, mis esitab korrektset kuupäeva. Sellel juhul saab probleemi lahendada enda generaatori kirjutamisega. Kui teatakse, et sõnesid läheb vaja ainult ühe meetodi jaoks, võib kirjutada enda sõnede generaatori. Kui sõnesid võidakse ka mujal kasutada, siis on parem kirjutada generaator, mis loob otse selle meetodi kutseid, andes argumendiks korrektse struktuuriga sõned. Enda generaatorite kirjutamine on aga võrdlemisi keeruline. Lihtsam lahendus võib olla programmile selliste lubatud meetodite ja konstruktorite ette andmine, mille puhul programm peaks suutma adekvaatseid argumente genereerida.

Programmi üks suurimaid puudujääke on geneeriliste tüüpide tüübiargumentidega mitte arvestamine. Kuigi programm suudab luua geneerilist tüüpi Java kogumikke, puudub universaalne lahendus. Teine puudus on, et programmi disainist tulenevalt, genereerib programm testandmeid ainult avaldiste kujul. See tähendab, et testitava meetodi või konstruktori argumentide loomine peab toimuma ühel real. Näiteks sellise koodi genereerimist programm ei võimalda:

```
Set<String> sõned = new HashSet();
sõned.add("sõne1");
sõned.add("sõne2");
testitavMeetod(sõned);
```

Klassi *HashSet* ja muude Java kogumike loomiseks kasutab programmiga kaasatulev *CollectionGenerator* klassi *Arrays* meetodite *asList*. Näiteks hulga loomine võiks välja näha selline:

```
new HashSet(Arrays.asList("sõne1", "sõne2"))
```

Seega Java kogumike puhul on nende mittetühjade versioonide loomine ühe avaldisega võimalik. Kuigi enamuste klasside puhul on nende isendite loomine ühe avaldisega võimalik, nõuab nende mingisse kindlasse olekusse viimine tihti paljude isendimeetodite kutseid. Selleks oleks aga vaja kasutada mitut individuaalset Java lauset, mida TestMotor ei võimalda.

Tarkvaraarenduse jaoks kasutatavate testide loomisel jääb programm olemasolevatele tehnoloogiatele alla, kuid TestMotor on disainitud just programmeerimisülesannetele testide loomiseks. Nende puhul on kasulik, kui testandmeid saab grupeerida keerukuse ja kontrollitava funktsionaalsuse järgi. Oletame, et testandmeteks on vaja genereerida matemaatilist

avaldist esitav objekt. Selle keerukuse määramiseks saab kasutada puu kõrgust. Kõrguse 2 puhul võiks tulemus olla selline:

```
new Liitmine(new Arv(4), new Arv(5))
```

Kõrguse 4 puhul aga selline:

```
new Korrutamine(new Arv(6), new Lahutamine(new Jagamine(new Arv(3), new
Arv(11)), new Arv(-20)))
```

Keerukuse ja kontrollitava funktsionaalsuse määramiseks võib kasutada ka lubatud meetodite ja konstruktorite määramist. Kui tahetakse luua lihtsamaid testandmeid, võib näiteks lubada ainult klasside *Arv* ja *Liitmine* konstruktorite kasutamist. Tulemus võiks kõrguse 3 puhul välja näha selline:

```
new Liitmine(new Liitmine(new Arv(15), new Arv(-10)), new Arv(21))
```

Tänu kõrguse ning lubatud meetodite ja konstruktorite määramisele, on raskuse ja kontrollitava funktsionaalsuse järgi testandmete grupeerimine võimalik.

TestMotori tugevaim külg on puu struktuuri esitavate klasside isendite genereerimine. Näiteks matemaatilised avaldised ja formaalsed grammatikad on üldjuhul esitatavad puuna. Programm kasutab ise ka sisemiselt puusid ning testandmete genereerimine toimub rekursiivselt, mis on väga sobilik puude loomiseks. Kuna programm ei nõua, et genereeritud puud oleks tasakaalus, ning kasutatavad meetodid ja konstruktorid valitakse juhuslikult, siis luuakse väga mitmekülgseid ja erinevaid puid.

Praktikas võib TestMotori kasutamine olla tülikas, kuna puudub kasutajaliides. Kasutajal tuleb esmakordsel kasutamisel selgeks õppida programmi klasside ja meetodite tähendused ning funktsionaalsus. Samuti teeb programmi rakendamist keerulisemaks asjaolu, et programm genereerib vaid testandmeid, mitte teste. Testandmete faili kirjutamise loogika tuleks kasutajal endal luua. Nende ebamugavuste vältimiseks on käesoleva tööga paralleelselt valmimas IntelliJ IDEA pistikprogramm[6].

6.4 Edasiarendamise võimalused

Käesoleva töö raames valminud programm täidab oma eesmärki ja võimaldab genereerida testandmeid igasugustele meetoditele ja konstruktoritele. Sellegipoolest on võimalusi selle

[6] IntelliJ IDEA TestMotori pistikprogrammi repositoorium: https://github.com/karljaats/test-motor-plugin

täiustamiseks. Näiteks eelmises peatükis välja toodud nõrkusi saab vähendada või elimineerida. Võiks luua süsteemi, mis võimaldaks luua testandmeid mitme lausega. Samuti saaks implementeerida universaalse lahenduse geneeriliste tüüpide loomiseks. Lisaks võiks ka testandmete genereerimiseks kuluvat aega vähendada. Nagu ülalpool kirjeldatud, genereerib programm mõistliku kõrgusega puid piisavalt kiiresti, kuid veel lühem genereerimise aeg suurendaks programmi kasutamise mugavust. Kasu võiks olla ka programmi detailsema seadistamise võimalustest.

Meetodite ja konstruktorite argumentide piirangute analüüsimine oleks juba keerulisem ja mahukam ülesanne. Selleks saaks kasutada näiteks sümbolipõhist täitmist, mis analüüsib meetodite ja konstruktorite koodi. Tänu sellele, võiks programm genereerida vähem ebakorrektseid argumente ning rohkem vigu leidvaid. Selle metoodik kasulikkust vähendaks siiski asjaolu, et programmeerimisülesannete puhul genereeritakse testandmed näidislahenduse põhjal, kuid rakendatakse õpilase lahendusel. Kui näidislahendus ja õpilase lahendus erinevad oma struktuuri poolest, siis ei pruugiks sümbolipõhise täitmise abiga saadud argumendid kasulikumad olla.

7. Kokkuvõte

Programmeerimisülesannetele testide kirjutamine võib olla ajamahukas töö. Selle üks keerulisemaid osasid on testitavatele meetoditele argumentide välja mõtlemine. Sellest tulenevalt, seati töö eesmärgiks luua programm, mis lihtsustaks programmeerimis-ülesannetele testide kirjutamist, automatiseerides meetodite ja konstruktorite argumentide loomise. Alustuseks kirjeldati testide automaatse genereerimise teooriat ning vaadeldi lähemalt kolme olemasolevat tehnoloogiat: Randoop, EvoSuite ja IntelliTest. Toodi ka praktiline näide programmeerimisülesandest ning sellele nii käsitsi kui automaatselt genereeritud testidest.

Töö käigus loodi programm TestMotor, mille disainimisel arvestati programmeerimis-ülesannete testide omapäradega, nagu vajadus teste keerukuse järgi grupeerida. Kõige sobilikumaks peeti puupõhist lähenemist. Loodud programm genereerib automaatselt meetodite ja konstruktorite argumente, mida esitatakse puu kujul. Kasutaja saab muuhulgas määrata genereeritavate testandmete kõrguse ning lubatud meetodid ja konstruktorid. Loodud puid saab kasutaja teisendada Java avaldisteks, mida on võimalik kirjutada testmeetoditesse. Samuti saab neid väärtustada, et leida testandmete oodatav tulemus.

Kvaliteedi tagamiseks testiti programmi põhjalikult. TestMotor saab hakkama igat tüüpi väärtuste genereerimisega, puudujäägiks on küll geneeriliste tüüpide tüübiargumentidega mitte arvestamine. Testandmete genereerimiseks kuluv aeg on piisavalt lühike, et tagada kasutamise mugavus. Testimise põhjal saab öelda, et TestMotor täidab oma eesmärki. Tulevikus on võimalik programmi veel täiustada, et suurendada kasutusmugavust, vähendada genereerimiseks kuluvat aega või parandada testandmete kvaliteeti.

8. Viidatud kirjandus

[1] Pittet S. The different types of software testing. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (07.05.2019)

[2] Creating Tests. https://www.jetbrains.com/help/idea/create-tests.html (07.05.2019)

[3] Ahmad M. A. New Strategies for Automated Random Testing. University of York, Department of Computer Science, PhD thesis. 2013. https://pdfs.semanticscholar.org/5dd9/ef9378a4e04968d7deb417c55df48ffab7a8.pdf

[4] Artzi S., Kiezun A., Pacheco C., Perkins J. Automatic Generation of Unit Regression Tests. 2005. https://people.csail.mit.edu/akiezun/unit-regression-tests.pdf (07.05.2019)

[5] Ryan F. A Look at Unit Testing Frameworks. 2018. https://redmonk.com/fryan/2018/03/26/a-look-at-unit-testing-frameworks/ (07.05.2019)

[6] Pacheco C., Lahiri S. K., Ernst M. D., Ball T. Feedback-directed Random Test Generation. *Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA, 2007, p. 75-84.

[7] Whitley D. A genetic algorithm tutorial. *Statistics and Computing*, 1994, p. 65-85.

[8] Fraser G., Arcuri A. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged, Hungary, 2011, p. 416-419.

[9] Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*. New York, NY, USA, 2011, Issue 5, p. 649-678.

[10] Generate unit tests for your code with IntelliTest. 2015. https://docs.microsoft.com/en-us/visualstudio/test/generate-unit-tests-for-your-code-with-intellitest?view=vs-2017 (07.05.2019)

[11] Input generation using dynamic symbolic execution. 2017. https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/input-generation?view=vs-2017 (07.05.2019)

[12] Lakshman P. Smart Unit Tests – a mental model. 2014. https://blogs.msdn.microsoft.com/devops/2014/12/11/smart-unit-tests-a-mental-model/ (07.05.2019)

[13] King J. C. Symbolic Execution and Program Testing. *Communications of the ACM*. New York, NY, USA, 1976, Issue 7, p. 385-394.

[14] Berre D. Java Tip 113: Identify subclasses at runtime. 2001. https://www.javaworld.com/article/2077477/java-tip-113--identify-subclasses-at-runtime.html (05.04.2019)

Lisad

I. Lähtekood

Programmi lähtekood on saadaval avalikus repositooriumis aadressil https://bitbucket.org/ENigola/testmotor

Avalehel näidatavas failis „README.MD“ on ka link allalaaditavale JAR faili ning instruktsioonid selle käivitamiseks.

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Ergo Nigola**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   Programmeerimisülesannetele puupõhiste testandmete genereerimise süsteem TestMotor,
   mille juhendaja on Vesal Vojdani, 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**