TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITV40LT

Andres Antonen 134220IAPB

TUDENGITE KOODI AUTOMAATTESTIMISSÜSTEEMI ARENDUS

bakalaureusetöö

Juhendaja: Ago Luberg
MSc
assistent

Tallinn 2016

Autorideklaratsioon

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

Autor: Andres Antonen

23.05.2016

Annotatsioon

Antud lõputöö eesmärgiks on arendada rakendus, mis võimaldab integreeritud arenduskeskkonda kasutamata testida programmikoodi sellele kirjutatud ühiktestidega ning läbi viia automatiseeritud koodianalüüsi. Rakendus peab usaldusväärselt töötama erinevate sisenditega, sh olukordades, kus testitav programmikood on puudulik või mõnel muul viisil mittetöötav. Rakenduse kasutaja peab testraporti kaudu lihtsalt nägema, millised ühiktestid ebaõnnestusid ja millised mitte, ning saama lisainfot probleemide parandamise kohta.

Töö tulemusena valmis prototüüp, mis kasutab Java programmikoodi testimiseks TestNG raamistikku ning koodianalüüsiks Checkstyle tarkvara. Valitud testraamistikku laiendatakse mitmete atribuutidega testraporti detailsemaks ja mõistetavamaks muutmiseks. Prototüüp leiab sellele antud kaustateedest automaatselt üles testitava koodi ja ühiktestid, kompileerib need, analüüsib stiilivigade suhtes ning tagastab kasutajale lihttekstis testraporti või JSON-formaadis objekti, mida saab kasutada muudes süsteemides töö hindamiseks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 36 leheküljel, 5 peatükki, 9 joonist, 4 tabelit.

// Abstract

The development of automatic testing system for students' code

The aim of this thesis is to develop an application that can automatically verify, through the use of unit testing and automated code review, the style and validity of arbitrary program code without using an integrated development environment. Different aspects of the problem are analysed and a plan is laid out how to implement such prototype. A testing framework and a code analysis tool are chosen by comparing different implementations and then extended by adding additional metadata, such as unit test weights, messages, descriptions and options, to the unit tests. The results can then be used, for example, to assess students' code writing skills. The prototype should accept various inputs, including broken code and/or missing data, displaying the results in a human-readable form that is either sent to the lecturer or the student.

The result of this thesis is a prototype codenamed StudentTester that uses the Java programming language to test code. The testing framework of choice is TestNG, a highly customisable framework that can run unit tests written in JUnit as well. For code analysis, a popular tool called Checkstyle is used. The default unit tests are extended using custom Java annotations, which define weights, comments, output verbosity etc. The prototype scans directories given and recursively finds all files containing Java source code, compiles them together with the unit tests, invokes both Checkstyle and TestNG and finally prints the results on the screen. Since the prototype is primarily intended to be used in conjunction with already implemented systems in the IT Faculty of Tallinn University of Technology, it can output compatible JSON-formatted results as well.

The thesis is in Estonian and contains 36 pages of text, 5 chapters, 9 figures, 4 tables.

Lühendite ja mõistete sõnastik

Git	Versioonihaldustarkvara, mis algselt loodi Linuxi tuuma arendamiseks
TDD	Testjuhitud arendus (ingl k *test-driven development*); arendamise viis, kus eelnevalt kirjutatakse testid ning seejärel programmiloogika
JDK	Java Development Kit; Java arenduskeskkond
XML	EXtensible Markup Language; märgistuskeel, millega saab vahetada infot infosüsteemide vahel
JSON	Andmevahetusvorming, kus andmed on salvestatud võti-väärtus (sõne, number) paarina või jadana
HTML	Hypertext Markup Language; märgistuskeel, milles kirjutatakse veebilehti
IDE	Integreeritud arenduskeskkond (*integrated development environment*); rakendus, milles on võimalik arendada teisi rakendusi
API	Rakendusliides (*application programming interface*); kogum reeglitest, kuidas suhelda mõne rakendusega funktsioonide, sõnumite vm kaudu
VM	Virtuaalmasin (*virtual machine*); abstraktse käsustikuga arvuti või virtuaalne keskkond, kus jookseb operatsioonisüsteem

Sisukord

1 Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Ülesande püstitus 11
1.3 Ülevaade 12
2 Analüüs 13
2.1 Ühiktestid 13
2.2 Ühiktestide laiendamine 14
2.3 Koodistiili valideerimine 16
2.4 Programmikoodi kompileerimine 17
2.5 Visuaalne tagasiside 17
2.6 Integreerimine teiste süsteemidega 18
3 Java testimisraamistikud 20
3.1 JUnit ja TestNG 20
3.2 JUniti ja TestNG tehniline võrdlus 22
3.3 Testraamistiku valik 23
4 Prototüüp StudentTester 24
4.1 Rakenduse funktsionaalsus 24
4.2 Prototüübi käsurea argumendid 25
4.3 Testide loomine 26
4.4 Dünaamiline kompileerimine 29
4.5 Checkstyle 30
4.6 Rakenduse väljundi vormindamine IReporter liidese abil 31
5 Prototüübi testimine 33
6 Kokkuvõte 35
Kasutatud kirjandus 36
Lisa 1 – Näidis tudengitele saadetavast testraportist aines Programmeerimise põhikursus Javas (ITI0011) 2016. a kevadsemestril 37
Lisa 2 – TTÜ IT teaduskonna automaattestimissüsteemi JSON-formaadis väljundi skeem 38

Lisa 3 – Prototüübi StudentTester annotatsioonidega täiustatud testikomplekt......... 40
Lisa 4 – Prototüübi StudentTester tüüpiline väljund.. 41
Lisa 5 – viide StudentTester avalikule repositooriumile 43

Jooniste loetelu

Joonis 1. Kompilatsioonivead pooliku lahenduse saatmisel või tahtlikul manipuleerimisel programmikoodiga. Tudengile saadetavas e-kirjas on näha, millist lahendust ühiktestis eeldatakse. 11
Joonis 2. Annotatsiooni näide Javas. Objektile „book“ on lisatud annotatsiooniga metaandmed autori kohta. 16
Joonis 3. Võimalik tagasiside formaat testerile programmikoodi saatnud kasutajale. ... 18
Joonis 4. JUnit'i ja TestNG võrdlus otsingutermini populaarsuse põhjal. Märksõna “java” on lisatud korrektsemate tulemuste saamiseks. 21
Joonis 5. Näide TestNG testklassist. Testi „testSimpleMath“ õnnestumiseks peab olema täidetud tingimus, et muutuja „answer“ omab arvulist väärtust 4. 27
Joonis 6. Näidis TestNG testide konfiguratsioonist. Käivitatakse ainult ühikteste sisaldavad klassid, mis on defineeritud märgendiga `<class>`. 27
Joonis 7. Osa rakenduse StudentTester väljundist puuduva meetodi korral tudengi programmikoodis. Viga tekkis testifailis, seega ei kuvata meetodi väljakutset. 30
Joonis 8. Väljavõte Checkstyle'i konfiguratsioonifailist. Kood kirjeldab reeglit, mis juhatab kasutaja tähelepanu rea lõpus olevatele liigsetele tühikutele............................ 31
Joonis 9. Näidis StudentTesterit testivast ühiktestist. 34

Tabelite loetelu

Tabel 1. Raamistike JUnit ja TestNG tähtsamate käskude/toimingute võrdlus 22

Tabel 2. Prototüübi StudentTester käsurea argumentide selgitus................................... 25

Tabel 3. Annotatsiooni `@Gradeable` väljad ja nende kirjeldused. 28

Tabel 4. Annotatsiooni `@GlobalConfiguration` väljad ja nende kirjeldused. 28

1 Sissejuhatus

Infotehnoloogiaga seotud erialade õppimisel on iseenesestmõistetav, et tudengid peavad paljudes õppeainetes kirjutama programmikoodi ning veenduma, et see töötab vastavalt ülesande nõuetele ning selle struktuur järgib teatud stiilinõudeid. Sõltuvalt ainest võivad tudengi koodi kontrollida näiteks õppejõud, selleks määratud abilised või automaattestimissüsteem (edaspidi nimetatud lihtsalt „süsteem“).

Süsteemi eesmärgiks on jooksutada automaatselt tudengi programmikoodi peal ühikteste (ingl k *unit tests*), mis kontrollivad, kas ettemääratud sisendite korral vastab meetodi/meetodite väljund oodatavatele tulemustele. Sellist testimisviisi nimetatakse ka musta kasti testimiseks (ingl k *black-box testing*), kuna testija ei ole teadlik meetodi sisemisest struktuurist ja selle toimimisest [1]. Kui testimisel leitakse viga, st väljund ei vasta oodatud tulemusele, teavitatakse kasutajat esinenud probleemist.

1.1 Taust ja probleem

TTÜ IT teaduskonnas on mitmeid õppeaineid, kus kasutatakse tudengite tunniülesannete, kodutööde, kontrolltööde vms ülesannete kontrollimiseks automaattestimissüsteemi. Nendest võib nimetada aineid Programmeerimise põhikursus Javas (ITI0011), Loogiline Programmeerimine (ITI0021), Algoritmid ja andmestruktuurid (ITI0050) ja Programmeerimise süvendatud algkursus (ITI0140). Nendes ainetes on kasutusel süsteemid programmeerimiskeeltele Java, Prolog ja Python.

Kui tudeng soovib teada saada enda töö vastavust ülesande tingimustele, laeb ta selle üles Git repositooriumisse, misjärel lisatakse töö ootejärjekorda. Pärast automaattesteri kontrolli saadetakse tudengi e-mailile kiri testitulemustega, mille näidis on toodud Lisa 1 – Näidis tudengitele saadetavast testraportist aines Programmeerimise põhikursus Javas (ITI0011) 2016. a kevadsemestril.

Aine Programmeerimise põhikursus Javas (ITI0011) näitel on näha, et tudengile saadetav kiri koosneb kahest osast – koodistiili valideerimisest Checkstyle teegi (täpsem kirjeldus

alampeatükis 4.5) abil ning ühiktestide tulemustest, mida hetkel jooksutatakse JUnit testraamistikus. Kui tudengi kood mõnes ühiktestis annab väära tulemuse, teavitatakse sellest kirjas ebaõnnestunud testi(de) nimega.

Praegust implementatsiooni võib lugeda töötavaks, ent mittetäielikuks, sest automaattester küll teavitab ebaõnnestunud ühiktestidest, kuid iseseisval lahendamisel võib jääda testmeetodite nimedest probleemi mõistmisel väheseks ning teatud juhtudel on võimalik ära kasutada Java kompilaatori veateateid testfailide sisu nägemiseks. Joonis 1 on näha situatsioon, kus tudeng on jätnud implementeerimata osad ülesandes defineeritud klassid/meetodid, mille tõttu programmikoodi kompilatsioon ebaõnnestub ning kasutajale tagastatakse osa testfailide sisust, andes ülesande lahendamiseks mitteplaneeritud eelise.

```
Test: EX09TestBonus
Compile error:
/tmp/tmpj17w3fm8/tmp/tests/EX09TestBonus.java:29: error: cannot find symbol
        f5 = new FairTradeRose(5, true);
                 ^

(...)

  symbol:   class FairTradeRose
  location: class EX09TestBonus
/tmp/tmpj17w3fm8/tmp/tests/EX09TestBonus.java:123: error: cannot find symbol
        assertEquals("Gandalf",
Order.findTheMostCaringCustomer(Arrays.asList(orderOne, orderTwo,
orderThree)));

(...)
```

Joonis 1. Kompilatsioonivead pooliku lahenduse saatmisel või tahtlikul manipuleerimisel programmikoodiga. Tudengile saadetavas e-kirjas on näha, millist lahendust ühiktestis eeldatakse.

1.2 Ülesande püstitus

Kuna praegune süsteem omab mitmeid probleeme, siis on vaja seda täiendada ning vähemalt eelmises alampeatükis nimetatud puudused kõrvaldada. Lisaks süsteemi jätkusuutlikkuse ja kasutusmugavuse parandamiseks võiks olla implementeeritud järgnev funktsionaalsus:

- Ühiktestidele kaalude lisamine, st lõpliku testitulemuse kujunemisel annavad osad testid rohkem punkte (hetkel loetakse iga ühiktesti väärtuseks 1 punkt ning läbitud ühiktestide arv jagatakse kõigi ühiktestide arvuga, tulemused kuvatakse e-mailis);
- Kirjelduste lisamine testidele, et nende ebaõnnestumise korral saab kasutaja rohkem infot, mida antud ühiktest kontrollib;
- Testide konfigureerimine näitamaks kasutajale enam- või vähemdetailsemaid veateateid;
- Üldine konfigureerimisvõimaluste laiendamine;
- Võimekus töötada nii Java, Pythoni, Prologi vm programeerimiskeelega;
- HTML-väljund tulemuste paremaks vormindamiseks ja sidumiseks TTÜ graafika/stiilinõuetega.

Antud lõputöö raames realiseeritakse kõik ülaltoodud punktid va kaks viimast; esialgu on planeeritud luua ainult Java keeles kirjutatud koodiga töötav prototüüp.

1.3 Ülevaade

Teine peatükk analüüsib, millised nõuded peaksid olema loodaval prototüübil. Kolmandas peatükis antakse ülevaade kahest suuremast Java testimisraamistikust JUnit ja TestNG ning selgitatakse, miks raamistikest osutus valituks viimane. Neljas peatükk keskendub loodud prototüübi dokumentatsioonile ning viiendas peatükis on põgusalt välja toodud rakenduse testimine. Kokkuvõttes analüüsitakse lühidalt tehtud tööd, vaadeldakse, mis sai tehtud ja mis mitte, ning tuuakse välja, kuidas saab valminud prototüüpi veel edasi arendada.

2 Analüüs

Prototüübi, mis peaks tulevikus hakkama asendama TTÜ IT teaduskonnas kasutusel olevat ühiktestimise süsteemi, loomisele eelnevalt tuleb välja selgitada, millistele nõuetele peaks see vastama, ning leidma, kuidas neid nõudeid kõige paremini implementeerida. Järgnevalt antakse ülevaade komponentidest, millest loodav süsteem koosnema peaks ning kuidas need omavahel seotakse.

2.1 Ühiktestid

Loodava prototüübi põhieesmärgiks on jooksutada teste, mis annavad tagasisidet testitud programmikoodi kohta. Kuna eelkõige on testitavaks koodiks tudengite kodutööd ja harjutused, mille sisendid ja väljundid on rangelt ülesandes defineeritud, siis on kasulik luua ühikteste, mida on võimalik jooksutada muutmata kujul iga lahenduse peal. Korrektselt kirjutatud ühiktestid peaksid teoreetiliselt garanteerima, et sajaprotsendilise tulemuse saab ainult ülesannet täpselt järgiv programmikood. Selleläbi vähendatakse lisatööd vigade otsimisel ja õppetöö puhul manuaalset koodi kontrollimist.

Suur osa tänapäevastest ühiktestimise raamistikest omab kollektiivset nimetust xUnit (kus x asendatakse programmeerimiskeele esitähega, näiteks Java puhul JUnit), mis on oma struktuuris ja funktsioonaalsuses eeskuju võtnud SUnit'ist[1]. SUnit on 1998. aastal Kent Becki poolt loodud testraamistik Smalltalki programmeerimiskeele jaoks. xUnit'i arhitektuuri järgivad raamistikud iseloomustavad järgmised komponendid [2]:

- Testikomplekt (*test suite*) – kogum ühiktestidest, mis jagavad sama konteksti ning mille järjekord võib olla suvaline;
- Testikontekst (*test fixture, test context*) – keskkond, milles ühe testikomplekti testid käivituvad. Enne testide käivitamist seatakse üles isoleeritud keskkond (tavaliselt `setup()` meetodiga) ning peale testide jooksutamist see

[1] http://sunit.sourceforge.net/

elimineeritakse (tavaliselt `teardown()` meetodiga). Selline struktuur on eriti kasulik „kallite“ objektide loomisel nagu server, ühe testikonteksti käigus luuakse see vaid üks kord;

- Testjuht (*test case*) – väikseim iseseisev osa testist, mis kontrollib testitava koodi ühte aspekti (antud sisendi korral vastab väljund oodatavale tulemusele);
- Väide (*assertion*) – lause/funktsioon, mis kontrollib testjuhu olekut; tavaliselt võrreldakse muutuja/funktsiooni väärtust eeldatava või väljaarvutatud väärtusega ning erinevuse korral „kukutatakse“ testjuht läbi;
- Testide jooksutaja (*test runner*) – käivitab testid ning teavitab kasutajat tulemustest teatud formaadis, tavaliselt XML-failis või lihttekstina.

2.2 Ühiktestide laiendamine

Üldiselt on testimisraamistikud mõeldud arendajatele tarkvara korrektse funktsioneerimise kontrollimiseks ning seetõttu sisaldab raamistiku väljund palju tehnilist informatsiooni, sealhulgas silumisinfot (*debug).* Kui testraporti saajaks on tudeng, siis on tähtis, et väljundis puudub info testitavate andmete kohta; vastasel juhul võib tudeng minna kergema vastupanu teed ning testerit petta vastuste programmikoodi sissekodeerimise teel. Mõnel juhul on detailsema veainfo näitamine jällegi vajalik.

Kui arendada süsteemi tudengite koodi testimise eesmärgil, on loogiline järeldada, et teatud juhtudel saaks testimise tulemusi kasutada ülesannete hindamisel. Hetkel genereeritakse osades TTÜ IT teaduskonna õppeainetes kasutusel oleva süsteemi tulemuste põhjal normaliseeritud punktisumma, mille kinnitab ja vajadusel korrigeerib õppejõud. Hindamisel on ilmnenud probleem, et osad ühiktestid katavad suuremat osa programmikoodi funktsionaalsusest ning seetõttu võiksid need lõpphindes mängida suuremat rolli. Seni on proovitud olukorda lahendada kaalukamate ühiktestide mitmekordse kopeerimisega, kuid on selge, et antud lahendus on äärmiselt kohmakas ja ebaefektiivne.

Tudengite suulise tagasiside põhjal on leitud puudusi ka testide tagasiside osas – ühiktesti ebaõnnestumise korral ei ole selle nimi (näiteks `testPrinting`) alati piisav vihje vea parandamiseks ning abi saamiseks pöördutakse õppejõu või kaastudengi poole. Kuna

ühiktesti meetodi nime ei ole otstarbekas kasutada pikema info edastamiseks, tekib vajadus luua eraldi andmevälju.

Eeltoodud probleemide lahendamiseks on vajalik kaasata metaandmeid käivitatavatele testidele, mis kas kajastuks testraporti struktuuris, saadavas hindes või mõnes muus aspektis. Üks võimalustest oleks defineerida testikomplekti klassis muutujad vajaliku infoga, näiteks täisarvu tüüpi muutuja testi kaalu ja sõne tüüpi muutuja testi kirjelduse lisamiseks. Selline lahendus muutub aga kiiresti kohmakaks, kuna 10 või enama ühiktestiga tuleb koos luua mitukümmend lisamuutujat, mis ei oma programmeerimiskeele tasemel testmeetoditega mingit seost.

Mõnedes programmeerimiskeeltes on olemas konstruktsioonid muutujatele, meetoditele ja klassidele metaandmete lisamiseks, mis võivad olla saadaval nii kompileerimise hetkel kui ka programmi töötamise ajal. Java programmeerimiskeeles nimetatakse selliseid konstruktsioone annotatsioonideks (*annotation*), C#'s atribuutideks (*attribute*).

Java annotatsioonid on metaandmete vorm[1], mida saab lisada näiteks meetodi, parameetri, muutuja või klassi definitsiooni ette eelmisele reale ning mis algavad @-märgiga. Annotatsioonide tugi lisati Java programmeerimiskeelde 1.5 JDK'ga[2]. Nende eesmärgiks on pakkuda lisainfot objektidele, millele neid on rakendatud, ning seeläbi muuta programmikoodi käitumist või lihtsustada metaandmetega seotud toiminguid. Enimlevinud näide annotatsioonist on `@Override`, mis instrueerib Java virtuaalmasinat otsima superklassist samanimelist meetodit, millele alamklassis annotatsioon rakendati. Kui alamklassis olev meetod nö ei „kirjutanud üle“ nimetatud meetodit, genereeritakse viga.

[1] https://docs.oracle.com/javase/tutorial/java/annotations/

[2] https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/package-summary.html

```java
@Author(firstName = "Mats", lastName = "Traat")
Book book = new Book();

public @interface Author {
    String firstName();
    String lastName();
}
```

Joonis 2. Annotatsiooni näide Javas. Objektile „book“ on lisatud annotatsiooniga metaandmed autori kohta.

Annotatsioonid võivad olla refleksiivsed, st neid on võimalik lisada Java klassifailidesse ning need on rakenduse jooksutamise ajal leitavad ning andmed loetavad. Täpsemalt on reflektsioon programmikoodi võime jälgida ja muuta iseenda koodi, ka nimetatud koodi jooksutamise ajal. Programmeerimiskeelt nimetatakse refleksiivseks, kui see võimaldab pakkuda selles keeles kirjutatud programmidele reflektsiooni [3].

Analüüsist selgub, et kui tahetakse luua automaattestimissüsteemi, mis võimaldaks eeltoodud funktsionaalsust saavutada, siis tasub uurida kasutatava programmeerimiskeele võimalusi meetoditele metaandmete lisamiseks, mida on võimalik rakenduse jooksutamise ajal reflektsiooni abil pärida.

2.3 Koodistiili valideerimine

Programmeerimiskeeli iseloomustab süntaks, mis on kogum reeglitest, kuidas ning milliste võtmesõnadega saab kirjutada korrektset ning töötavat koodi. Valdav enamus tänapäevastest keeltest kasutab C-laadset süntaksit (C, C++, C#, PHP, Java, JavaScript)[1], kus plokke defineerivad loogelised sulud ({}), käsk lõpetatakse semikooloniga (;) ning üldine programmeerimisstiil tavaliselt vaba. Suurimaks erandiks populaarsete keelte seas on Python, mis kasutab koodistruktuuri defineerimisel tühikuid või tabulatsioonimärke ning keeldub käivitumast, kui neid on kasutatud läbisegi või vääralt. Selle tõttu on võimalik esimese rühma keeltes kirjutada palju lohakamalt koodi kui viimases. Halbade koodimisharjumuste hulka kuulub näiteks mitme lause ühele reale kirjutamine, tähtsate muutujate või funktsioonide kommenteerimata jätmine, raskestiloetavate ja ebaühtlase stiiliga meetodite või klasside kirjutamine.

[1] http://www.tiobe.com/tiobe_index

Loodavasse süsteemi oleks mõistlik integreerida ka tööriist, mis aitaks antud probleeme leida ning nendele tähelepanu pöörata. Sellise tööriista kasutamist nimetatakse automatiseeritud koodi ülevaatuseks (*automated code review*). Kuigi sellist tööd on võimalik teha manuaalselt, on automatiseerimine kiirem ja täpsem ning ei vaja keele kõiki peensusi teadvat spetsialisti [4]. Programmikoodi analüüsimine struktuuri- ja stiilivigade suhtes on eriti tähtis alustavatele programmeerijatele, kuna see aitab sisse harjutada hästi loetava koodi kirjutamist ning vältida võimalikke vigu.

2.4 Programmikoodi kompileerimine

Programmeerimiskeeled jagunevad käskude täitmismeetodi põhjal kaheks: interpreteeritavad keeled ja kompileeritavad keeled. Neist esimese puhul täidetakse käske rida-realt programmi lähtekoodi lugedes, viimase puhul teisendatakse lähtekood enne käivitamist masinkoodi, mis vastab kasutatava arvuti protsessori arhitektuurile ja käsustikule (*instruction set*). Olemas on ka vahepealsed keeled, mis kompileeritakse baitkoodi e käsustikule, mida seejärel interpreteerib virtuaalmasin (VM). Sellisesse kategooriasse kuulub näiteks Java.

Ühiktestide läbiviimine eeldab, et on võimalik käivitada nii teste kui ka testitavat programmikoodi. Kuna osad keeled vajavad kompileerimist, siis see tähendab, et nende puhul on vaja läbida see lisaetapp. Üldiselt toimub kompileerimine kas IDE's või käsureal, kuid loodava süsteemi iseseisvuse ja robustsuse tagamiseks tasub selgitada, kas kasutatav programmeerimiskeel omab liidest (API't) programmikoodi otseseks kompileerimiseks süsteemi käsurida kasutamata. Näiteks Javas on selliseks liideseks JavaCompiler[1], keeles C# aga CsharpCodeProvider[2].

2.5 Visuaalne tagasiside

Pärast ühiktestide ja stiilinõuete kontrolli läbiviimist on vajalik kasutajatele tulemusi esitada kergestimõistetaval kujul. See tähendab, et kuvatakse koodistiili valideerimise tulemused, ühiktestide läbimise protsent (mis võtab arvesse eelnevalt määratud kaalusid),

---

[1] https://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html

[2] https://msdn.microsoft.com/en-us/library/microsoft.csharp.csharpcodeprovider(v=vs.110).aspx

õnnestunud ja ebaõnnestunud ühiktestide nimed ning muu lisainfo, millest raporti saajal kasu olla võiks. Lihtsaim on sellist raportit implementeerida testraamistikuga, millel on liides testitulemuste tagastamiseks või nende faili kirjutamise võimekus. Vastasel juhul on vajalik teksti töötlemine süsteemi väljundvoost. Joonis 3 näitab, milline võiks olla testraporti struktuur lihttekstina. Kui kasutatav tehnoloogia lubab, võib raportit implementeerida ka keerukamas märgendikeeles, näiteks XML (mida paljud xUnit'i-laadsed testraamistikud teevad juba vaikimisi) või HTML, millest viimast saab saata e-mailina ning kuvada erinevate tekstistiilide ja graafikaga.

```
TEST RESULTS

<testi nimi>
<üldine sõnum raporti lugejale>

<koodistiili valideerimise tulemus>

<testikomplekti nimetus>
    <kirjeldus>

    SUCCESS: <õnnestunud ühiktesti nimi>
    <testi jaoks kulunud aeg, kaal jms info>
    (...)
    FAILURE: <ebaõnnestunud ühiktesti nimi>
    <testi jaoks kulunud aeg, kaal jms info>
    <ühiktesti kirjeldus>
    <vea kirjeldus>
    (...)

    <testikomplekti läbimise tulemus>

<testikomplekti nimetus>
    (...)


<testide läbiviimise lõpptulemus e hinne>
```

Joonis 3. Võimalik tagasiside formaat testerile programmikoodi saatnud kasutajale.

2.6 Integreerimine teiste süsteemidega

Loodav süsteem ei ole mõeldud ainult käsureal käivitamiseks, vaid integreerimiseks juba olemasolevate süsteemidega, eriti praeguse TTÜ IT teaduskonnas kasutatava lahenduse täiustamiseks. Testitulemuste info edastamiseks on mitmeid viise, näiteks rakenduse väljundvoo suunamine teise rakenduse sisendvoogu või ajutisse faili kirjutamine, mida

kasutatakse hindeinfo sisestamiseks süsteemi või testraporti saatmiseks tudengi e-mailile. Lisa 2 – TTÜ IT teaduskonna automaattestimissüsteemi JSON-formaadis väljundi skeem kirjeldab, milline peab olema loodava prototüübi väljund JSONis, kui hakata sellega asendama praegust lahendust.

3 Java testimisraamistikud

Java programmeerimiskeeles kirjutatud rakenduste jaoks on mitmeid testimisraamistikke. Sõltuvalt testimise skoobist on võimalik kirjutada suuremaid integratsiooniteste, mis kasutavad nn *mock*-objekte ning kontrollivad rakenduse funktsionaalsust tervikuna. Üks populaarsematest sellistest raamistikest on Mockito[1], mida kasutatakse tarkvara testjuhitud arenduses (TDD, *test-driven development*). Kuna käesoleva projekti skoop on siiski väiksem ning eelkõige on vajadus testida tudengite kirjutatud programmikoodi väljundi õigsust, siis harilik ühiktestimine on sobivam. Kaks populaarsemat raamistikku, mis pakuvad Javas ühiktestide kirjutamist, on JUnit ja TestNG. Järgnevalt tuuakse välja mõlema raamistiku omadused ning võrreldakse nende funktsionaalsust.

3.1 JUnit ja TestNG

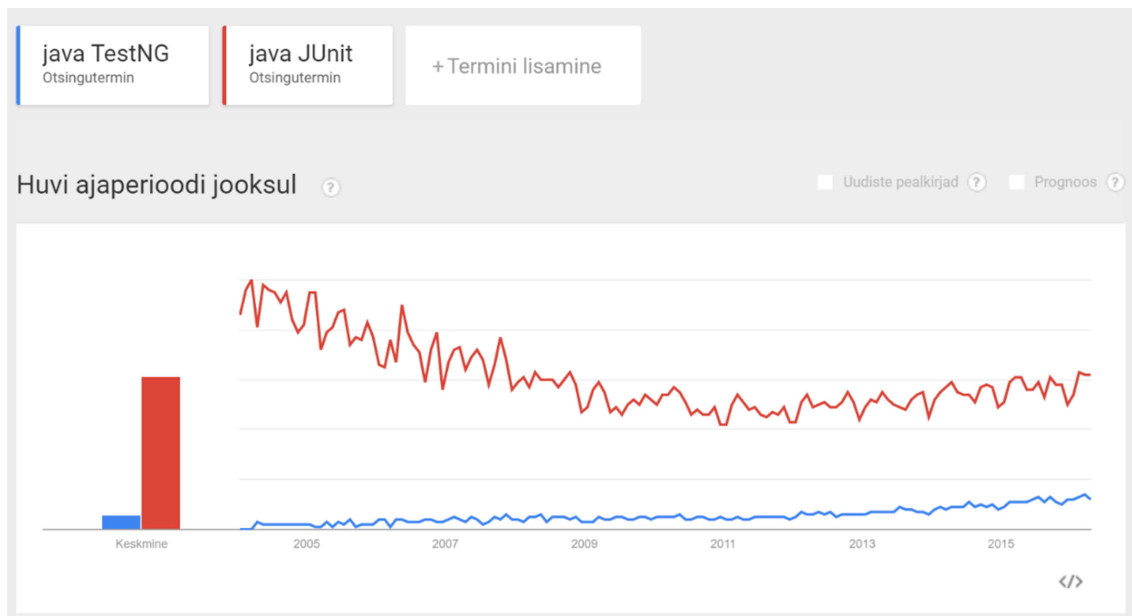
JUnit[2] ja TestNG[3] on kaks populaarsemat ühiktestide kirjutamise raamistikku Java programmeerimiskeelele. Mõlemad on üsna pika ajalooga: JUnit'i esimesed väljalasked olid enne aastat 2000 ja TestNG aastal 2004. Mõlemal raamistikul on pistikprogrammid ka laialdaselt kasutatud IDE'dele nagu Eclipse ja IntelliJ. Kuna JUnit on ~4 aastat pikema ajalooga kui TestNG, siis tõenäoliselt selle tõttu on hetkel eelmine suurema kasutajaskonnaga kui viimane, mida kinnitab ka Google Trends'i otsingute graafik Joonis 4.

---

[1] http://mockito.org/

[2] http://junit.org/

[3] http://testng.org/



Joonis 4. JUnit'i ja TestNG võrdlus otsingutermini populaarsuse põhjal. Märksõna "java" on lisatud korrektsemate tulemuste saamiseks.[1]

Otsingutulemuste põhjal on võimalik öelda, et JUnit'i kasutajaid on mitmekordselt rohkem kui TestNG kasutajaid, seda kinnitab Maven Repository statistika, mille järgi on JUnit raamistik kasutuses ~39 000 repositooriumis[2] ja TestNG ~3550 repositooriumis[3]. JUnit'i kasuks räägib ka fakt, et see on hetkel kasutuses TTÜ automaattestimissüsteemis ning õppejõud/tudengid on harjunud seda kasutama. Väiksema kasutajaskonna tõttu on loogiliselt igasuguse TestNG abimaterjali hulk tunduvalt tagasihoidlikum, kuigi see on tõusuteel. TestNG aeglast, kuid kindlat populaarsuse kasvu kinnitab Tomek Kaczanowski, raamatu „*Practical Unit Testing with TestNG and Mockito*" autor, kelle andmetel on JUniti arendus stagneerunud ning kommuun vaikseks jäänud (vahemikus 2009-2012 ainult 3 väljalaset, meililistis paari kuuga paarkümmend e-maili), samas kui TestNG arendus on palju intensiivsem (13 väljalaset ning üle 500 e-maili samadel perioodidel) [5].

---

[1] https://www.google.com/trends/explore#q=Java%20JUnit%2C%20Java%20TestNG&cmpt=q&tz=Etc%2FGMT-3

[2] http://mvnrepository.com/artifact/junit/junit

[3] http://mvnrepository.com/artifact/org.testng/testng

3.2 JUniti ja TestNG tehniline võrdlus

Tabelis Tabel 1 on toodud Java ühiktestimise tähtsamate käskude/tegevuste võrdlused.

Tabel 1. Raamistike JUnit ja TestNG tähtsamate käskude/toimingute võrdlus

	JUnit [6]	**TestNG** [7]
Testi annotatsioon	`@Test`	`@Test`
Ajapiirang	`@Test(timeout = 1000)`	`@Test(timeOut = 1000)`
***assertEquals*-lause**	`assertEquals(expectedResult, actualResult)`	`assertEquals(actualResult, expectedResult)`
Erindite eeldamine	`@Test(expected = ArithmeticException.class)`	`@Test(expectedExceptions = ArithmeticException.class)`
Enne / pärast meetodit käivitatavate meetodite annotatsioon	`@Before` `@After`	`@BeforeMethod` `@AfterMethod`
***Dependency*-testid**	ei	jah

Lisaks saab TestNG puhul välja tuua järgmised huvipakkuvad funktsioonid:

- automaatsed testraportid;
- laiendatavad liidesed;
- JUniti testide jooksutamise võimalus;
- konfigureerimine XML-faili kaudu.

3.3 Testraamistiku valik

Võrreldes JUniti ja TestNG funktsionaalsust külg-külje kõrval, võime näha, et kriitilise süntaksi osas on mõlemad raamistikud küllaltki sarnased ning testikomplektide „portimiseks“ ühest formaadist teise piisab tekstiredaktoris asenduste tegemise ja *import*-lausete muutmisest. Tähelepanu vajab *assert*-lausetes argumentide järjekord, kus TestNG puhul on eeldatav tulemus teisel kohal, JUniti puhul esimesel.

Antud lõputöö prototüübi aluseks sai valitud TestNG oma kasulike lisavõimaluste tõttu – raamistik toetab JUniti testide käivitamist, mistõttu ei ole seniste testide ümberkirjutamine vajalik. Lisaks on kasutajal võimalik laiendada pea iga raamistiku aspekti, sealhulgas väljatrükitava testraporti formaati, mistõttu ei ole praeguses lahenduses kasutatav väljundi manuaalne töötlemine TestNG raamistikku kasutusele võttes enam vajalik.

4 Prototüüp StudentTester

Antud lõputöö käigus valmis rakendus, mille esialgne nimi on StudentTester. StudentTesteri eesmärgiks on jooksutada tudengi saadetud programmikoodi peal selle jaoks kirjutatud ühikteste ning anda kasutajale ülevaatlikku tagasisidet testitulemuste kohta, olles samal ajal võimalikult kergesti konfigureeritav ning töökindel.

Rakendus StudentReporter koosneb neljast põhilisest elemendist:

- Java kompilaator, millele valmistatakse kompileerimiseks ette programmikood ning sellele koodile kirjutatud testid;
- Checkstyle[1]; analüüsimistööriist, mis kontrollib koodi vastavust eeldefineeritud stiilireeglitele;
- Testimisraamistik TestNG, mis käivitab testfailidesse kirjutatud ühiktestid;
- Reporteri klass, mis laiendab TestNG poolt defineeritud IReporter liidest ning koostab testitulemuste põhjal kasutajale tagasisideraporti.

4.1 Rakenduse funktsionaalsus

Rakendus töötab käsureal ning vajab tulemuse väljastamiseks järgnevat keskkonda:

- Olemas on kaust, kust tester saab testitava koodi (sisaldab .java faile);
- Olemas on kaust, kust tester saab testifailid (sisaldab .java faile);
- Testitava koodi kaustas asub TestNG konfiguratsioonifail testng.xml või teekond selleni on antud.

Ülaltoodud konfiguratsioon peab olema kaasa antud rakenduse käivitamisel argumentidega, mille selgitus on alampeatükis 4.2.

Korrektsete argumentide korral käivitub rakendus vaikimisi sätetega:

[1] http://checkstyle.sourceforge.net/

- Kood kontrollitakse Checkstyle'iga (selgitus alampeatükis 4.5);
- Kompileeritud koodi peal käivitatakse TestNG testid;
- Checkstyle ja TestNG tulemused kuvatakse kasutajale kasutades *StudentReporter* laiendatud klassi (selgitus alampeatükis 4.6).

4.2 Prototüübi käsurea argumendid

Tabel 2. Prototüübi StudentTester käsurea argumentide selgitus

Argument	**Selgitus**
`-contentRoot [kaustatee]`	Vajalik. Määrab kausta, kust tester leiab testitavate failide koodi.
`-testRoot [kaustatee]`	Vajalik. Määrab kausta, kust tester leiab testifailide koodi.
`-tempRoot [kaustatee]`	Määrab kausta, kuhu tester paigutab ajutised failid. Puudumisel üritatakse leida süsteemi ajutiste failide kaust.
`-verbosity [arv]`	Määrab testeri „jutukuse“ taseme. Tasemel 0 näidatakse ainult testraportit, tasemetel 1 ja 2 näidatakse rakenduse standardses veavoos (stderr) silumisinfot ja detailsemaid veateateid.
`-nocheckstyle`	Keelab Checkstyle käivitamise.
`-notestng`	Keelab TestNG käivitamise.
`-jsonoutput`	Väljastab tulemused JSON-formaadis edasiseks töötlemiseks. JSONi formaat vastab osaliselt struktuurile, mida kasutatakse lõputöö valmimise hetkel TTÜ IT teaduskonnas kasutatavas süsteemis (vt Lisa 2 – TTÜ IT teaduskonna automaattestimissüsteemi JSON-formaadis väljundi skeem)

-nomute	Vaikimisi eemaldatakse väljundist testitava koodi genereeritud väljund, ka siis, kui on kasutusel *-verbosity* argument. *-nomute* kasutamisel suunatakse see väljund standardsesse veavoogu (stderr).
-checkstylexml [failitee]	Määrab Checkstyle'i kasutatava konfiguratsioonifaili tee. Vaikimisi kasutatakse testide kaustas olevat faili checkstyle.xml. Selle puudumisel kasutatakse sisseehitatud valideerimisreegleid.
-testngxml	Määrab TestNG kasutatava konfiguratsioonifaili tee. Vaikimisi kasutatakse testide kaustas olevat faili testng.xml. Selle puudumisel genereeritakse viga.

Näide:

```
java -jar studentTester.jar -testroot "C:\test" -contentroot "C:\source"
```

käivitab rakenduse StudentTester, mis leiab kaustast „C:\test" ühiktestid ning kaustast „C:\source" testitavad failid. Koodi kontrollib Checkstyle, seejärel see kompileeritakse, TestNG käivitab ühiktestid, mis on defineeritud „testng.xml" failis, ning need kuvatakse käsureal inimloetavas formaadis.

4.3 Testide loomine

Testide kirjutamine TestNG's ja JUnitis on üldjoontes sarnane: luuakse Java klass, mille sisse kirjutatakse meetodid testitava koodiga. Koodi allikas võib asuda mõnes teises klassis, selle testimiseks luuakse vastava klassi instants ning kutsutakse välja vajalikud meetodid või siis tehakse seda staatiliselt. Et eristada meetodeid, mis on mõeldud testeri poolt käivitamiseks ja mis on muul eesmärgil, kasutatakse annoteerimist.

Testmeetodi annotatsioon on `@Test`, mis instrueerib testerit käivitama vastavat meetodit. Iga meetod töötab sarnaselt Java tavalistele funktsioonidele, ent lisaks kasutatakse käske, mis kontrollivad teatud hetkedel rakenduse seisu. Sellisteks käskudeks on näiteks *assert*-laused, millest täpsemalt võib nimetada *assertEquals, assertTrue, assertFalse* jt.

Vastavalt lauses toodud avaldise väärtusele seda lauset sisaldav meetod kas jätkab tööd või genereeritakse veateade ja test loetakse ebaõnnestunuks.

```
import static org.testng.Assert.*;
import org.testng.annotations.Test;

public class SimpleTest {

    int answer = 1 + 1;

    @Test
    public void testSimpleMath() {
        answer += 2;
        assertEquals(answer, 4);
    }
}
```

Joonis 5. Näide TestNG testklassist. Testi „testSimpleMath“ õnnestumiseks peab olema täidetud tingimus, et muutuja „answer“ omab arvulist väärtust 4.

TestNG vajab iseseisvaks testide jooksutamiseks XML-formaadis konfiguratsioonifaili, kus on defineeritud testikomplekt (*test suite*), testid (mitte segi ajada klassisisese ühiktestiga) ja klassid, kus ühiktestid asuvad. Joonis 6 on näide testikomplektist nimega EX19, kus on defineeritud kaks testi EX19Tests ja EX19TestsBonus. Mõlemad sisaldavad ühte samanimelist klassi testidega, sealjuures on märgitud, et test EX19Tests tuleks käivitada kasutades JUnit teeki. JUniti kasutamine ei muuda annotatsioonide kasutamise loogikat ning kasutada on võimalik modifitseerimata teste.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="EX19">
  <test name="EX19Tests" junit="true">
    <classes>
       <class name="EX19Tests"/>
    </classes>
  </test>
  <test name="EX19TestsBonus">
    <classes>
       <class name="EX19TestsBonus"/>
    </classes>
  </test>
</suite>
```

Joonis 6. Näidis TestNG testide konfiguratsioonist. Käivitatakse ainult ühikteste sisaldavad klassid, mis on defineeritud märgendiga `<class>`.

Prototüübis StudentTester on automaattestide täiustamiseks kasutusel kaks kohandatud annotatsiooni - `@Gradeable` ja `@GlobalConfiguration`. Neist esimene on mõeldud rakendamiseks testmeetodile (koos annotatsiooniga `@Test`) ning viimane neid meetodeid sisaldavale klassile. Annotatsiooni `@Gradeable` eesmärk on lisada metaandmeid testile, sh arvulist kaalu ning kirjeldust; `@GlobalConfiguration`'i abil on võimalik määrata kõikide testide väljastatava info hulka ning lisada sõnum väljundisse. `@GlobalConfiguration` annotatsiooni võetakse antud prototüübi versioonis arvesse vaid esimese klassi korral, mis on defineeritud TestNG konfiguratsioonifailis `<test>` märgendiga. Ülejäänud samasse testi kuuluvad klassid kasutavad eelnevaid väärtusi. Annotatsioonide väljade täpsem kirjeldus on välja toodud tabelites Tabel 3 ja Tabel 4.

Tabel 3. Annotatsiooni `@Gradeable` väljad ja nende kirjeldused.

Annotatsiooni väli	**Kirjeldus**
`int weight()`	Testi kaal. Suurem kaal tähendab, et test mängib lõpphinde arvutamisel teistest suuremat rolli. Vaikimisi on iga testi kaal 1.
`String description()`	Testi kirjeldus. Tavaolukorras trükitakse testraportis väärtus ainult siis, kui sellega seotud test ebaõnnestus või jäeti vahele.
`boolean printExceptionMessage()`	Kui väli määrata tõeseks, trükitakse kasutajale testi ebaõnnestumisel välja detailne veasõnum, mis võib sisaldada ka testi *expected-actual* väärtusi. Vaikimisi keelatud.
`boolean printStackTrace()`	Kui väli määrata tõeseks, trükitakse kasutajale testi ebaõnnestumisel välja programmiosa täitmise jäädvustis (*stack trace*). Vaikimisi keelatud.

Tabel 4. Annotatsiooni `@GlobalConfiguration` väljad ja nende kirjeldused.

Annotatsiooni väli	**Kirjeldus**
`ReportMode mode()`	Määrab, kui palju infot tudengitele väljastatakse testraportis. On suurema prioriteediga kui

	`@Gradeable` annotatsioonis defineeritud väärtused. Võib omada järgnevaid defineeritud väärtusi: **`ReportMode.NORMAL`**, mille puhul trükitakse testraport tavaliselt ning kasutatakse ka `@Gradeable` väärtusi; **`ReportMode.VERBOSE`**, mille puhul trükitakse detailsed veateated igale ebaõnnestunud testile; **`ReportMode.MAXVERBOSE`**, mille puhul trükitakse igale ebaõnnestunud testile ka *stack trace;* **`ReportMode.ANONYMOUS`**, mille puhul näidatakse ainult läbitud testide arv ja hinne, testide nimed on peidetud; **`ReportMode.MUTED`**, mille puhul ainult antakse teada testide kättesaamisest ja testide läbiviimisest.
`String welcomeMessage()`	Sõnum, mis lisatakse raportis iga testikogumiku ette, mis on TestNG konfiguratsioonis märgitud `<test>` märgendiga.

Eeltoodud annotatsioonide kasutamine on valikuline ning abivahendiks kaaludega ning kommenteeritud testide kirjutamiseks. Lisa 3 – Prototüübi StudentTester annotatsioonidega täiustatud testikomplekt toob näite testikomplektist, kus on kasutatud kohandatud annotatsioone testraporti väljundi muutmiseks.

4.4 Dünaamiline kompileerimine

Java Development Kit'iga (JDK) on kaasas Java Compiler API[1]; liides, mis võimaldab programmikoodi sees dünaamiliselt kompileerida Java klasse. Nimetatud funktsionaalsus

[1] https://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html

on kasulik prototüübi loomisel, kuna testerile etteantav kood on kompileerimata kujul (.java tekstifailid) ning neid ei ole võimalik otse käivitada.

Prototüübis StudentTester kasutatakse nimetatud liidest programmikoodi ja testifailide Java *class*-failideks kompileerimiseks, mida on saab hiljem käivitada. API võimaldab koguda diagnostikat kogu protsessi kohta ning kompilatsiooni ebaõnnestumisel teavitatakse kasutajat, millises failis mis tüüpi viga leiti. Diagnostiliste andmete töötlemise käigus selgitatakse, kas viga ilmnes testitavas programmikoodis või testifailis. Esimesele juhul näidatakse detailset infot, millisel real mis tüüpi viga leiti, viimasel juhul viidatakse vaid üldisele probleemile (näiteks ei ole implementeeritud funktsioon, mida testifailis välja üritatakse kutsuda). Selline lahendus on vajalik olukorras, kus testraporti saaja ei tohi näha testfaili sisu ning väärtusi, millega tester koodi kontrollib.

```
Compilation failed, cannot continue.
Error in EX19TestsBonus.java: cannot find symbol
  symbol:   method reverse(java.lang.String)
  location: class EX19
Hint: does the method exist?
Skipped 10 error(s) of the same type.
```

Joonis 7. Osa rakenduse StudentTester väljundist puuduva meetodi korral tudengi programmikoodis. Viga tekkis testifailis, seega ei kuvata meetodi väljakutset.

4.5 Checkstyle

Checkstyle[1] on ca ~15 aastase ajalooga vabatarkvara, mis võimaldab kontrollida programmikoodi vastavust teatud reeglitele. Nimetatud reegliteks võivad olla üldised Java keeles programmeerimise head tavad, aga ka ettevõtte, kus võib koos töötada kümneid või enam programmeerijaid, koodistiili ühtlustamiseks loodud nõuded.

Näited, mille kasutamist Checkstyle tavakonfiguratsioonis jälgib:

- Klassid, meetodid, muutujad, väljad jms on kommenteeritud JavaDoc'iga;
- Tab-märkide asemel kasutatakse tühikuid;
- Taanded, tühikud jms koodi struktuur on paigas;

---

[1] http://checkstyle.sourceforge.net/

- Kõik arvud peale -1, 0, 1 ja 2 tuleb defineerida eraldi muutujas ja kommenteerida.

Checkstyle'i reegleid on võimalik ka ise defineerida. Reeglite koostamine ei kuulu käesoleva lõputöö skoopi, ent olemasolevaid reegleid on võimalik sisse-välja lülitada modifitseerides XML-faili, mis antakse rakenduse käivitamisel kaasa argumendina. Reegli saab välja lülitada terve `<module>` märgendi väljakommenteerimisel või eemaldamisel failist.

```
<module name="RegexpSingleline">
   <property name="format" value="\s+$"/>
   <property name="minimum" value="0"/>
   <property name="maximum" value="0"/>
   <property name="message" value="Line has trailing spaces."/>
</module>
```

Joonis 8. Väljavõte Checkstyle'i konfiguratsioonifailist. Kood kirjeldab reeglit, mis juhatab kasutaja tähelepanu rea lõpus olevatele liigsetele tühikutele.

4.6 Rakenduse väljundi vormindamine IReporter liidese abil

TestNG pakub mitmeid liideseid, mida raamistiku autor nimetab *listener*'ideks, funktsionaalsuse laiendamiseks. *Listener*'e saab käivitada erinevates testimise staadiumites sarnaselt testifailide meetoditele, mis on näiteks annotatsioonidega `@BeforeMethod` või `@AfterClass`. Antud projektis kasutatakse TestNG *listener*'e programmikoodi soovimatu väljundi keelamiseks või mujale suunamiseks ning testraporti koostamiseks.

Kohandatud testraporti koostamiseks on TestNG'l olemas liides IReporter, mille kasutamisel on vaja realiseerida meetod `generateReport()`, millele antakse argumentidena kaasa nimekiri testikomplektidest. Antud lõputöös on loodud kohandatud reporter StudentReporter. Iga testikomplektis oleva testi (klasside ja neis olevate ühiktestide) kohta koostatakse raport ühiktestide nime, staatuse ja muu lisainfoga, kui ühiktestidele oli rakendatud `@Gradeable` annotatsioon ja vastavad metaandmed. Raporti koostamine sõltub ka klassile rakendatud `@GlobalConfiguration` metaandmetest; vastavalt testide kirjutaja soovile võivad olla eemaldatud testide nimed ja testitulemused või vastupidi, lubatud raporti lugejat abistavad detailsed veateated.

Võimalik on kirjutada ka enda reporter, mis väljastab soovitud struktuuriga testraporti. Selleks on olemas lisaks IReporter liidesele ka teine liides IBaseStudentReporter, mis

kohustab reporteri kirjutajat implementeerima avalikku meetodit `getResults()`, mis peab tagastama TestResults formaadis andmeobjekti. Andmeobjekti tagastamine on vajalik testraporti ja testitulemuste lisamiseks JSON objekti, mida saab kasutada näiteks praegusesse TTÜ IT teaduskonna hindamissüsteemi andmete saatmiseks. Praeguse implementatsiooni kohaselt peab reporter vajadusel tagastama TestResults objektiga:

- Testraporti üldväljundi (meetod `setOutput()`);
- Üldise protsentuaalse hinde (meetod `setPercent()`);
- SingleResult-tüüpi nimekirja osatestidest (meetod `addTest(kood, nimi, protsent)`), kus:
    - kood on osatesti järjekorranumber algusindeksiga 1;
    - nimi on osatesti nimi;
    - protsent on osatesti täitmise protsent.

Reporter StudentReporteri tüüpiline väljund on nähtav lõputöö lisas Lisa 4 – Prototüübi StudentTester tüüpiline väljund ning testikomplekti, millega selline tulemus saavutati, kirjeldab Lisa 3 – Prototüübi StudentTester annotatsioonidega täiustatud testikomplekt.

Loodud reporter kirjutab oma väljundi standardsesse väljundvoogu, kuid võimalik on ka testraporti kirjutamine faili. Näidiseid sellistest failidest võib leida pärast ühiktestide läbiviimist prototüübi käivitamise asukohas „test-output“ kausta nime all. Vaikimisi luuakse TestNG sisseehitatud *listener'*idega mitmeid HTML- ja XML-raporteid, millest võib olla kasu õppejõule saadetud koodi silumisel või keerukamate HTML-testraportite koostamisel. Antud lõputöö raames aga keerukama väljundi loomist ei vaadelda.

5 Prototüübi testimine

Iga loodava tarkvara puhul on tähtis enne selle laiemale kasutajaskonnale avaldamist teha kindlaks, et kõik selle komponendid töötavad ootuspäraselt ning võimalikult palju erijuhtumeid on kaetud testidega. Prototüüp StudentTester funktsionaalsusele mõeldes leiab palju juhtumeid, mida enne kasutuselevõttu kontrollida tuleks.

Testid peaksid kontrollima, kas prototüüp töötab:

- Puhta TestNG testikomplekti ja testitava programmikoodiga;
- Annotatsioonidega `@Gradeable` ja `@GlobalConfiguration`;
- Ülalnimetatud annotatsioonidega tühjade või puudulike metaandmete korral;
- Vigase programmikoodi korral;
- Vigase testi korral;
- Checkstyle või TestNG muu erindi korral.

Prototüübi jaoks on kirjutatud testid, mis kontrollivad väljundit konkreetsete juhtumite korral. Testid asuvad projektis tests.Tests klassis ning on käivitatavad kas läbi IDE või käsurealt:

```
java -cp studentTester.jar org.testng.TestNG -testclass tests.Tests -verbose
2
```

Testid on üles ehitatud järgnevalt:

- Testikomplektis luuakse testikontekst, st keskkond – süsteemi ajutisse kausta luuakse kaustad ühiktestidele ja kontrollitavale koodile (meetod `beforeClass`);
- Enne igat ühiktesti luuakse tühi ühiktestifail ning programmikoodifail, samuti luuakse testng.xml konfiguratsioonifail, mis viitab vastloodud ühiktestifailile (meetod `beforeMethod`);
- Käivitatakse ühiktest ning failidesse kirjutatakse simuleeritud ühiktestide ja programmikoodi sisu. Luuakse uus StudentTesterClass objekt, millele edastatakse loodud failid ning käivitatakse kirjutatud ühiktestid;
- Loetakse saadud tulemus JSONi kaudu ning seda võrreldakse eeldatava tulemusega;
- Pärast ühiktesti läbimist kustutatakse ajutised failid ning minnakse järgmise ühiktesti juurde (meetod `afterMethod`);

- Testikontekst elimineeritakse, kustutatakse ajutised kaustad (meetod `afterClass`).

```
@Test(description = "Check if a simple test can receive a result from a
class.")
public void testTrivial() {
    testFileWriter.write(String.format("import org.testng.annotations.Test;"
            + "public class StudentReporterTest%1$s {"
            +     "@Test\r\n"
            +     "public void testSanity() {"
            +         "StudentCode%1$s c = new StudentCode%1$s();"
            +         "assert c.onePlusOne() == 2;"
            +     "}"
            + "}", testCounter));
    studentCodeWriter.write(String.format("public class StudentCode%1$s {"
            +     "public int onePlusOne() {"
            +         "return 1 + 1;"
            +     "}"
            + "}", testCounter));
    // the first argument enables/disables Checkstyle, the second is for
TestNG
    JsonObject results = getTestResults(false, true);
    Assert.assertEquals(results.getInt("percent"), 100);
}
```

Joonis 9. Näidis StudentTesterit testivast ühiktestist.

6 Kokkuvõte

Osades TTÜ IT teaduskonna õppeainetes on kasutusel automaattestimissüsteem, mis jooksutab tudengite programmikoodi ühikteste, salvestab hinded ning saadab tudengile e-maili tulemustega. Senine süsteem töötab, ent sellele on võimalik juurde arendada palju funktsionaalsust ning eemaldada mõned puudused, sh testitulemuste napisõnalisus ja testfailide sisu näitamine.

Käesolevas lõputöös sai loodud prototüüp, mis lihtsustab integreeritud arenduskeskkonna välist ühiktestide läbiviimist. Esialgu sai implementeeritud testimine Java programmeerimiskeeles ning väljund on ainult lihttekstis, kuid tulevikus on prototüübi laiendamise käigus plaanis hakata toetama ka teisi programmeerimiskeeli ning parandada testraporti visuaalset poolt. Üleminek JUnit'ilt TestNG'le peaks olema valutu, kuna ühikteste ümber kirjutama ei pea ning olemasolevaid saab täiendada annotatsioonide lisamise teel (eeldusel, et uutel semestritel taaskasutatakse vanu ülesandeid ja teste).

Prototüüp peaks olema väikeste korrigeerimistega integreeritav praeguse TTÜ IT teaduskonna süsteemi asemele ning loodetavasti on see 2016 sügissemestriks juba kasutuses. Prototüübi JSON-väljundisse ei olnud võimalik kaasata kõiki välju, kuna need määratakse testeriväliselt ning selle kaasamine projekti oleks juba lõputöö skoobist väljas.

Kasutatud kirjandus

[1] ISO/IEC/IEEE, „Terms and definitions,“ *24765:2010 Systems and software engineering — Vocabulary*, 2010, p. 3.272.

[2] K. Beck, „Simple Smalltalk Testing: With Patterns,“ 1999. [Võrgumaterjal]. Saadaval: https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm. [Kasutatud 19. mai 2016].

[3] J. Malenfant, M. Jacques ja . F.-N. Demers, „A Tutorial on Behavioral Reflection and its Implementation,“ 1996. [Võrgumaterjal]. Saadaval: http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf. [Kasutatud 19. mai 2016].

[4] I. Gomes, P. Morgado, T. Gomes ja R. Moreira, „An overview of the Static Code Analysis approach in Software Development,“ 2009. [Võrgumaterjal]. Saadaval: http://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf. [Kasutatud 17. mai 2016].

[5] T. Kaczanowski, „Why TestNG and not JUnit?,“ 23. aprill 2012. [Võrgumaterjal]. Saadaval: http://kaczanowscy.pl/tomek/2012-04/why-testng-and-not-junit. [Kasutatud 21. mai 2016].

[6] K. Beck, E. Gamma ja M. Clark, „JUnit 4.12 API,“ 2016. [Võrgumaterjal]. Saadaval: http://junit.org/junit4/javadoc/latest/. [Kasutatud 18. mai 2016].

[7] C. Beust, „TestNG Documentation,“ 19 detsember 2015. [Võrgumaterjal]. Saadaval: http://testng.org/doc/documentation-main.html. [Kasutatud 18. mai 2016].

Lisa 1 – Näidis tudengitele saadetavast testraportist aines Programmeerimise põhikursus Javas (ITI0011) 2016. a kevadsemestril

```
Checkstyle errors:
Starting audit...
[ERROR] /tmp/tmp0raqa1ts/tmp/source/EX19.java:9:5: Missing a Javadoc comment.
[JavadocMethod]
[ERROR] /tmp/tmp0raqa1ts/tmp/source/EX19.java:19:21: Array brackets at
illegal position. [ArrayTypeStyle]
[ERROR] /tmp/tmp0raqa1ts/tmp/source/EX19.java:25:46: ')' is preceded with
whitespace. [ParenPad]
[ERROR] /tmp/tmp0raqa1ts/tmp/source/EX19.java:43:5: Missing a Javadoc
comment. [JavadocMethod]
Audit done.
Checkstyle ends with 4 errors.



Test: EX19Tests
Number of tests: 11
Number of failed tests: 0

Test: EX19TestsBonus
Number of tests: 13
Number of failed tests: 2
Failed tests:
testRandom1000
testRandom10x1000
------------------
Total tests passed: 22 / 24 (91.7%)

session: 30315425780
```

Lisa 2 – TTÜ IT teaduskonna automaattestimissüsteemi JSON-formaadis väljundi skeem

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "results": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "code": {
            "type": "integer"
          },
          "percent": {
            "type": "integer"
          },
          "output": {
            "type": "string"
          }
        }
      }
    },
    "percent": {
      "type": "integer"
    },
    "output": {
      "type": "string"
    },
    "extra": {
      "type": "string"
    },
    "version": {
      "type": "string"
    },
    "timestamp": {
      "type": "integer"
    },
    "-token": {
      "type": "string"
    },
    "session": {
      "type": "string"
    },
    "source": {
      "type": "array",
```

```
      "items": {
        "type": "object",
        "properties": {
          "path": {
            "type": "string"
          },
          "content": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

Lisa 3 – Prototüübi StudentTester annotatsioonidega täiustatud testikomplekt

```
import static org.testng.Assert.*;
import org.testng.annotations.Test;

import studenttester.annotations.*;
import studenttester.enums.*;

@GlobalConfiguration(mode = ReportMode.VERBOSE, welcomeMessage = "This is an
example test suite.")
public class ExampleTest {

    @Gradeable(description = "This is a successful test.")
    @Test
    public void exampleTest1() {
        assertTrue(true);
    }

    @Gradeable(weight = 2, printStackTrace = true)
    @Test
    public void exampleTest2() {
        assertFalse(true, "This fails on purpose.");
    }

    @Gradeable
    @Test(dependsOnMethods = "exampleTest2")
    public void exampleTest3() {
        throw new UnsupportedOperationException("This should never be
thrown.");
    }

    @Test
    public void exampleTest4() {
        int answer = 1 + 11;
        assertEquals(answer, 2);
    }

    @Gradeable(weight = 4, description = "This test should catch an
exception.")
    @Test(expectedExceptions = NullPointerException.class)
    public void exampleTest5() {
        throw new NullPointerException("This test ends abruptly.");
    }

}
```

Lisa 4 – Prototüübi StudentTester tüüpiline väljund

```
TEST RESULTS

Running Checkstyle...
Starting audit...
Audit done.



Test suite ThesisTest

Test context TestTest
This is an example test suite.
SUCCESS: exampleTest5
        0 msecs, unit test weight: 4 units
        Description: This test should catch an exception.
SUCCESS: exampleTest1
        4 msecs, unit test weight: 1 units
        Description: This is a successful test.
FAILURE: exampleTest2
        1 msecs, unit test weight: 2 units
        Description: No description.
        Exception type: class java.lang.AssertionError
        Detailed information:  This fails on purpose. expected [false] but
found [true]
        Stack trace:  java.lang.AssertionError: This fails on purpose.
expected [false] but found [true]
        at org.testng.Assert.fail(Assert.java:94)
        at org.testng.Assert.failNotEquals(Assert.java:513)
        at org.testng.Assert.assertFalse(Assert.java:63)
        at ExampleTest.exampleTest2(ExampleTest.java:19)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at
org.testng.internal.MethodInvocationHelper.invokeMethod(MethodInvocationHelpe
r.java:86)
        at org.testng.internal.Invoker.invokeMethod(Invoker.java:646)
        at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:823)
        at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:1131)
        at
org.testng.internal.TestMethodWorker.invokeTestMethods(TestMethodWorker.java:
128)
        at
org.testng.internal.TestMethodWorker.run(TestMethodWorker.java:112)
        at org.testng.TestRunner.privateRun(TestRunner.java:778)
        at org.testng.TestRunner.run(TestRunner.java:632)
        at org.testng.SuiteRunner.runTest(SuiteRunner.java:366)
```

```
        at org.testng.SuiteRunner.runSequentially(SuiteRunner.java:361)
        at org.testng.SuiteRunner.privateRun(SuiteRunner.java:319)
        at org.testng.SuiteRunner.run(SuiteRunner.java:268)
        at org.testng.SuiteRunnerWorker.runSuite(SuiteRunnerWorker.java:52)
        at org.testng.SuiteRunnerWorker.run(SuiteRunnerWorker.java:86)
        at org.testng.TestNG.runSuitesSequentially(TestNG.java:1225)
        at org.testng.TestNG.runSuitesLocally(TestNG.java:1150)
        at org.testng.TestNG.runSuites(TestNG.java:1075)
        at org.testng.TestNG.run(TestNG.java:1047)
        at
studenttester.classes.StudentTesterClass.runTestNG(StudentTesterClass.java:19
1)
        at
studenttester.classes.StudentTesterClass.run(StudentTesterClass.java:117)
        at
studenttester.classes.StudentTesterMain.main(StudentTesterMain.java:96)

FAILURE: exampleTest4
        0 msecs, unit test weight: 1 units
        Description: No description
        Exception type: class java.lang.AssertionError
        Detailed information:  expected [2] but found [12]
SKIPPED: exampleTest3
        Unit test weight: 1 units
        Description: No description.
        Test skipped because:  java.lang.Throwable: Method
ExampleTest.exampleTest3()[pri:0, instance:ExampleTest@4a3631f8] depends on
not successfully finished methods
        This unit test depends on tests: ExampleTest.exampleTest2

Passed unit tests: 2/5
Failed unit tests: 2
Skipped unit tests: 1
Grade: 55,6%

Overall grade: 55,6%
```

Väljund põhineb testikomplektil Lisa 3.

Lisa 5 – viide StudentTester avalikule repositooriumile

Prototüüp StudentTester on üles laaditud avalikku repositooriumi GitHub'is ning on kättesaadav aadressil https://github.com/ndrez93/StudentTester.

Prototüübi kloonimiseks enda arvutisse saab kasutada aadressi https://github.com/ndrez93/StudentTester.git.