TARTU ÜLIKOOL
Loodus- ja täppisteaduste valdkond
Arvutiteaduse instituut
Informaatika õppekava

Anette Taivere

Visual Studio Code pistikprogrammi testimine GobPie näitel

Bakalaureusetöö (9 EAP)

Juhendaja: Karoliine Holter, MSc

Tartu 2024

Visual Studio Code pistikprogrammi testimine GobPie näitel

Lühikokkuvõte:
Testimine on oluline osa tarkvaraarenduse protsessist. Selle töö eesmärgiks on uurida ja rakendada testimismeetodeid Visual Studio Code'i pistikprogrammile GobPie, et ennetada vigu ja muuta programmi töökindlamaks. Selleks uuriti GobPie pistikprogrammi, leiti selle funktsionaalsused, pandi kirja nõuded ning nõuete põhjal kirjutati automaattestid Java programmeerimiskeeles. Kirjutatud testide kvaliteedi hindamiseks kasutati koodi katvuse meetrikat. Tööd läbi viies avastati mitmeid vigu GobPie pistikprogrammis, mis parandati, ning koodi efektiivsuse suurendamiseks tehti koodihooldust.

Võtmesõnad:
Pistikprogramm, üksustestimine, GobPie, Goblint

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Testing the Visual Studio Code GobPie extension

Abstract:
Testing is an important part of the software development process. This work aims to explore and apply testing methods to the Visual Studio Code extension GobPie to prevent errors and make the program more reliable. To achieve this, the functionalities of the GobPie extension were studied, requirements were identified, and automated tests were written in Java programming language based on these requirements. Code coverage metrics were used to evaluate the quality of the written tests. Several bugs were discovered in the GobPie extension and fixed throughout the process, and code maintenance was performed to increase code efficiency.

Keywords:
Plugin, unit testing, Gobpie, Goblint

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

Sisukord

1. Mõisted ja terminid

Üksuste testimine - Riistvara- või tarkvaraüksuste testimine analüüsi- või teostusvigade avastamiseks [1].

Koodiredaktor (ingl *code editor*) - Programmeerimiseks mõeldud tööriist, mida kasutatakse programmeerimiskoodi kirjutamiseks, redigeerimiseks ja haldamiseks.

Integreeritud arenduskeskkond (ingl *integrated development environment*) - „Instrumentaalprogrammide komplekt tarkvara väljatöötamiseks, enamasti sisaldab teksti- või koodiredaktorit, vahendeid transleerimiseks, linkimiseks, silumiseks, kasutajaliidese loomise hõlbustuseks, võib sisaldada keelespetsiifilisi ja graafilisi abivahendeid“ [1].

Pistikprogramm - Riistvara- või tarkvaramoodul, mis on kergesti paigaldatav ja lisab senisele süsteemile võimalusi [1].

Goblint - Abstraktsel interpretatsioonil põhinev staatiline analüsaator mitmelõimeliste C-programmide jaoks [15].

GobPie - Tartu Ülikoolis arendatav pistikprogramm integreeritud arenduskeskkondadele, mille abil visualiseeritakse Goblint tööriista analüüse C-koodile [8].

Alljärgnevas loetelus on toodud selgitused nõuetes kasutatud terminitele.

Goblinti server - Goblint analüsaatoris teostatud andmevahetuse komponent, mis võimaldab Goblinti ja GobPie vahelist suhtlust.

MagpieBridge server - Raamistik, mis võimaldab sujuvat suhtlust erinevate analüsaatorite ja integreeritud arenduskeskkondade vahel.

Erand (ingl *exception*) - Tavalisest või normist lahknev olukord, väärtus, olem vms [1].

JSON - „Lihtne andmevahetusvorming, mis põhineb JavaScripti alamhulgal ja on hõlbus inimlugemiseks ja -kirjutuseks“ [1].

GobPie seadistus - JSON formaadis fail, mis võimaldab kasutajal kohandada pistikprogrammi seadeid.

Pistikprogrammile saab Gobpie seadistuses määrata järgnevaid seadistusi [2].

goblintConf - Suhteline tee analüüsitava projekti juurkaustast Goblinti seadete faili asukohani (nõutud).

goblintExecutable - Absoluutne või suhteline tee (analüüsitava projekti juurkaustast) Goblinti asukohani failisüsteemis (valikuline, mille vaikimisi väärtuseks on „goblint“, mis tähendab, et Goblinti asukoht on lisatud süsteemi *PATH* muutujasse nimega goblint).

preAnalyzeCommand - Enne analüüsimist käivitatav käsk (nt käsk kompileerimisandmebaasi koostamiseks/värskendamiseks, automatiseerimise eesmärgil) (valikuline, ilma seadistamata ühtegi käsku enne analüüsi ei käivitata).

abstractDebugging - Kas abstraktse silumise funktsionaalsus on sisse lülitatud (valikuline, vaikimisi välja lülitatud (*false*)).

showCfg - Kas koodiredaktoris kuvatakse kooditoimingud funktsioonide juhtvoograafide (*CFG*) vaatamiseks (valikuline, vaikimisi välja lülitatud (*false*)).

incrementalAnalyisis - Kas Goblint peaks kasutama järkjärgulist analüüsi (valikuline, vaikimisi sisse lülitatud (*true*)).

explodeGroupWarnings - Kas grupihoiatusi plahvatatakse, ehk näidatakse hoiatuste gruppi igal sellesse kuuluval individuaalse hoiatuse asukohal, või ei plahvatata, ehk grupihoiatust näidatakse vaid ühes defineeritud asukohas (valikuline, vaikimisi sisse lülitatud (*true*)).

2. Sissejuhatus

Tartu Ülikooli tarkvarateaduse uurimisrühmas arendatakse koostöös Müncheni Tehnikaülikooliga avatud lähtekoodiga abstraktsel interpretatsioonil põhinevat C-koodi staatilist analüsaatorit Goblint [3]. Goblintile on loodud pistikprogramm GobPie, et selle tulemusi integreeritud arenduskeskkonna Visual Studio Code graafilise liidese abil visualiseerida [8].

GobPie pistikprogrammi koodile pole aga seni automaatseid teste loodud, mistõttu ei pruugi programmi vastavus spetsifikatsioonile olla täielikult tagatud ning Goblinti kasutamine pistikprogrammi kaudu on veaohtlikum kui selle kasutamine käsurealt. Töö eesmärk on uurida GobPie pistikprogrammi ning teostada selle testimine, et tuvastada võimalikud vead ning parandada pistikprogrammi kvaliteeti. Testimise käigus leitud vead parandatakse, et tagada GobPie töökindlus ja pakkuda Goblintile kvaliteetsemat kasutajaliidese kogemust.

Testide haldamiseks on oluline koguda testitava programmi nõuded, luua testimismallid ning dokumenteerida testimistulemused [14]. Eesmärgi saavutamiseks uuritakse esiteks erinevaid testimismeetodeid ja testide kvaliteedi hindamise kriteeriumeid, mille põhjal valitakse sobilik testimismetoodika. Seejärel kogutakse GobPie tarkvara funktsionaalsed nõuded, mille põhjal valitud testimismetoodikat rakendada, et veenduda programmi vastavuses kogutud funktsionaalsetele nõuetele. Testimine viiakse läbi üksustestidena, mida kirjutatakse Java programmeerimiskeeles. Piisavuse hindamiseks kasutatakse JaCoCo ning IntelliJ IDEA testimistööriistu, kus vaadetakse peamiselt klasside, meetodite, ridade ning harude katvuskriteeriumeid.

Töö sisuline osa on jaotatud kolmeks peatükiks numbritega kolm kuni viis. Kolmandas peatükis antakse teoreetiline ülevaade nii testimise vajadusest, testimismeetoditest ja nende hindamise metoodikatest, kui ka Goblintist, ja selle kasutajaliidesest GobPie, millele testimist rakendatakse. Neljandas peatükis kirjeldatakse töö tehnilist teostust: nõuete kogumist, protsessi testide planeerimisest nende teostamiseni ning tuuakse välja loodud testid ja nende abil leitud vead. Viiendas peatükis hinnatakse loodud testide piisavust ning arutatakse töö tulemusi ja edasi arendamise võimalusi.

3. Teoreetiline ülevaade

Selles peatükis antakse ülevaade tarkvara testimise meetoditest ning vajadusest. Lisaks kirjeldatakse erinevaid testimise piisavuse hindamismeetrikaid ning tutvustatakse Goblint analüsaatorit ja GobPie pistikprogrammi.

3.1. Tarkvara testimise vajadus ja meetodid

Kasvava tarkvaraarendusega muutub tarkvara testimine järjest tähtsamaks, kuna koodi on palju, ning see suurendab vigade tekkimise riski. Taley [17] sõnul on testimine mõeldud tarkvara kvaliteedi parandamiseks ning lisaks sellele hoitakse testimisega kokku ka arenduskuludelt. Testimisel käsitletakse erinevaid testimistasemeid, millega tagatakse tarkvara vastavus nõuetele ja kliendi või kasutajate ootustele.

„Software Testing: An ISTQB-BCS Certified Tester Foundation Guide“ [6] kolmandas väljaandes on kirjeldatud erinevad testimistasemed, kus igal tasemel keskendutakse erinevatele aspektidele ning käsitletakse erinevat tüüpi vigu. Raamatus on välja toodud ülevaade järgnevatest testimistasemetest:

- üksustestimine;
- integratsiooni testimine;
- süsteemi testimine;
- vastavustestimine.

Esimese etapina kasutatakse üksusteste kirjutatud koodi üksuste testimiseks, et veenduda selle korrektsuses ning vastavuses spetsifikatsioonidele. Peale üksustestide loomist ühendatakse testid, et luua süsteem ehk tehakse integreerimine. Integratsiooni testimisel vaadeldakse erinevate liideste ja komponentide või süsteemide omavahelist suhtlust. Peale komponentide kontrollimist vaadeldakse süsteemi funktsionaalsust. Süsteemi testimisel on eesmärgiks leida veel avastamata vigu, mis tulenevad süsteemi terviklikkuse mõjust ja keskkonnast. Viimasena viiakse läbi vastavustestimine, kus veendutakse, et süsteem toimib vastavalt kasutaja ootustele. Testimise esimeseks etapiks on üksustestimine ning kuna GobPie pistikprogrammi pole varasemalt testitud keskendutakse selles töös just üksustestide loomisele.

3.2. Testide piisavuse hindamise vajadus ja meetodid

Tarkvara testimise üheks väljakutseks on testjuhtude kvaliteedi hindamine. Mida kõrgema kvaliteediga testikomplekt luuakse, seda rohkem vigu on sellega võimalik tuvastada. Selliseid kvaliteedimeetmeid nimetatakse testimise piisavuse hindamise meetoditeks [7, 12]. Kirjanduses leidub mitmeid erinevaid meetodeid piisavuse hindamiseks [13], nagu näiteks koodi katvus, mutatsiooniskoor, pärimispuu sügavus jt. Enim viidatud metoodika testide piisavuse ja täielikkuse hindamiseks on koodi katvus [7, 13], millel on omakorda mitmeid alamkategooriaid. Koodi katvus on meetrika, mis näitab, milline protsent tarkvarakoodist on testsüsteemi poolt kasutatud [4]. Koodi katvuse meetrika alla kuuluvad näiteks:

- klasside katvus (ingl *class coverage*) [16]
- meetodite katvus (ingl *method coverage*) [16]
- ridade katvus (ingl *line coverage*) [18],
- lausete katvus (ingl *statement coverage*) [7, 9, 13],
- baitkoodi instruktsioonide katvus (ingl *bytecode instruction coverage*) [9],
- harude katvus (ingl *branch coverage*) [7, 13],
- muudetud tingimuse/otsuse katvus (ingl *modified condition/decision coverage*) [13],
- tsüklite katvus (ingl *loop coverage*) [7],
- määratlus-kasutus paari katvus (ingl *def-use pair coverage*) [7].

Obaid Barraood jt. poolt läbi viidud testimise kvaliteedi hindamise meetrikate süstemaatilise kirjandusliku ülevaate põhjal on kõige levinum testide kvaliteedi hindamise meetrika lausete katvus [13]. Selles töös kasutatakse testide piisavuse hindamiseks lisaks lausete katvusele ka sellest nõrgemaid meetrikaid, nagu klasside ja meetodite katvus, ning sellest tugevama meetrikana ka harude katvust.

3.3. Goblint analüsaator

Goblint[1] on avatud lähtekoodiga[2] C-programmide staatiline analüsaator, mille analüüsiraamistik kasutab abstraktse tõlgendamise metoodikat [15], Goblinti abstraktse tõlgen-

[1] `https://goblint.in.tum.de`
[2] `https://github.com/goblint/analyzer`

damise metoodikal põhinevad programmi andmevoo analüüsid vahetavad teavet ühise liidese kaudu [8]. Analüüside ühised tulemused võimaldavad programmi verifitseerida, ning verifitseerimise ebaõnnestumisel võimalikke vigu avastada.

Goblinti lähenemine programmeerimisvigade tuvastamisele ületab linterite tavalised tingimused, kuna see keskendub kogu programmi semantikale, et tuvastada kõik potentsiaalselt tekkivad vead [3]. Semantikale keskendumine aitab Goblintil avastada nii süntaktilisi vigu kui ka keerulisemaid programmeerimise loogikaga seotud probleeme. See aitab arendajal ennetada ja eemaldada vigu varajases arenguetapis, mis tagab programmide parema kvaliteedi.

Goblint analüsaator on realiseeritud OCaml programmeerimiskeeles ning sellel on modulaarne arhitektuur [15]. Viimane tähendab, et analüsaator on ehitatud üles moodulitena, mis võimaldab kerget analüüside lisamist ning olemasolevate analüüside paindlikku kohandamist ning sisse-välja lülitamist läbi Goblinti seadete.

3.4. GobPie pistikprogramm

GobPie[3] on Goblinti tulemuste koodiredaktoris visualiseerimiseks loodud pistikprogramm [8]. Kuna Goblintit arendatakse Tartu Ülikooli ja Müncheni Tehnikaülikooli koostöös, on tegemist rahvusvahelise projektiga ning selle kasutuskeeleks on inglise keel.

GobPie programm toetub MagpieBridge[4] raamistikule [8], mis on loodud lihtsustamaks erinevate staatiliste analüsaatorite tulemuste kuvamist integreeritud arenduskeskkondades ning nende koodiredaktorites [11]. GobPie eesmärk on parendada Goblinti kasutatavust, võimaldades selle kasutamist toetatud arenduskeskkondades. Praeguse seisuga on toetatud ainult Visual Studio Code[5], kuid tänu MagpieBridge raamistiku kasutamisele on GobPie programmil võimekus luua tugi võrdlemisi lihtsasti ka mitmetele teistele arenduskeskkondadele, nagu näiteks IntelliJ IDEA.

Goblinti ja GobPie parema ühenduse jaoks on Goblintile lisatud serverirežiim [5], mis

[3] `https://github.com/goblint/GobPie`
[4] `https://github.com/MagpieBridge/MagpieBridge`
[5] `https://code.visualstudio.com`

võimaldab GobPie'ga infot vahetada IPC-pesade kaudu. Sel viisil on võimalik hoida Goblint töös kogu pistikprogrammi kasutamise sessiooni aja ning vajalik infovahetus toimub Goblintit taaskäivitamata. Goblinti töös hoidmisel on võimalik uuteks analüüsideks kasutada inkrementaalset analüüsi [5], mille käigus analüüsitakse uuesti vaid viimastest muudatustest mõjutatud programmiosad, säästes seega nii aega kui ka analüüsimiseks kuluvaid ressursse. Samuti jäetakse serverirežiimi kasutades alles Goblinti seaded ning seetõttu ei ole vaja enne igat analüüsi Goblinti seadeid uuesti sisse lugeda. Küll aga peab GobPie olema suuteline tuvastama ja uuendama Goblinti seadeid, kui neid Goblinti seadete failis muudetakse.

4. Tehniline teostus

Selle peatüki eesmärk on anda ülevaade GobPie testimise tehnilisest teostusest. Peatükis keskendutakse nõuete kogumise protsessile, testide koostamise etappidele ning leitud vigadele.

4.1. Nõuete kogumine ja testimise planeerimine

Testitava programmi nõuete kogumine on osa tarkvara testimise protsessist, mis aitab püstitada eesmärke ning mõista süsteemi toimimist. Enne nõuete kogumist analüüsiti lähtekoodi ja kommentaare, et omandada sügavamat arusaama projektist.

Analüüsi käigus tuvastati GobPie põhifunktsioonid, mis moodustavad olulise osa rakenduse toimimisest. Põhifunktsionaalsuseks on näiteks pistikprogrammi võime käivitada programmi analüüs ning kuvada selle tulemused koodiredaktoris. Kuigi süsteemile saab defineerida nii funktsionaalsed kui ka mittefunktsionaalsed nõuded, keskenduti siin eelkõige funktsionaalsetele nõuetele.

Nõuete kogumise eesmärgiks oli süsteemi põhifunktsionaalsustele vastavate funktsionaalsete nõuete täielik kajastamine. Nõuded ei sisalda abstraktse silumise, MagpieBridge raamistiku, JSONRPC klasside laienduste ning HTTP serveri funktsionaalseid nõudeid, kuna need ei ole osa GobPie põhifunktsionaalsusest ning jäid seetõttu töö skoobist välja. GobPie funktsionaalsed nõuded jaotati selle põhifunktsionaalsuste kaupa järgnevalt:

A) analüüsi käivitamine;
B) GobPie konfiguratsiooni kohandamine ja haldamine;
C) JSON formaadis Goblinti serveri päringute vastuste teisendamine MagpieBridge serverile vastavateks objektideks;
D) Goblinti serveri käivitamine;
E) Goblinti konfiguratsiooni kohandamine ja haldamine.

Kirjutatud testid jaotati viite erinevasse faili, kus igas failis keskendutakse ühele viiest põhifunktsionaalsusest. Nõuete spetsifikatsioonid on vastavalt põhifunktsionaalsustele kirjeldatud tabelis 1, kus testgrupid on jaotatud nende teostamise järjekorras.

Tabel 1. Nõuete spetsifikatsioon

#		Nõue
GobPie ...		
1	A)	on võimeline eelanalüüsiks
2		teavitab kasutajat sõnumiga kui Goblinti analüüs ebaõnnestub
3		katkestab eelmise käimasoleva analüüsi kui uut analüüsi alustades ei ole eelmine analüüs lõpetanud
4		teavitab kasutajat sõnumiga kui eelmise käimasoleva analüüsi katkestamine ebaõnnestus
5		eelanalüüs ei käivitu kui sisend on tühiväärtus (ingl *null*)
6		eelanalüüs ei käivitu kui sisend on tühi sõnede jada
7		teavitab kasutajat sõnumiga kui GobPie eelanalüüs püüab erandi
1	B)	loeb GobPie konfiguratsiooni
2		loeb GobPie konfiguratsiooni kõik seaded koos nende võimalike väärtustega
3		täidab konfiguratsiooni lugemisel puuduvate valikuliste seadete väärtused vaikimisi
4		teavitab kasutajat sõnumiga kui GobPie konfiguratsiooni süntaks on vale
5		teavitab kasutajat kui GobPie konfiguratsiooni failis on seade, mida ei ole defineeritud
6		teavitab kasutajat sõnumiga kui GobPie konfiguratsiooni fail puudub
7		taaskäivitab Goblinti kui selle konfiguratsioon muutub
8		teavitab kasutajat kui GobPie konfiguratsioonis puudub parameeter
1	C)	parsib Goblintilt saadud (mitte plahvatavad hoiatuste) JSON tulemused MagpieBridge'ile sobivateks objektideks
2		parsib Goblintilt saadud (plahvatavad hoiatuste) JSON tulemused MagpieBridge'ile sobivateks objektideks
3		parsib Goblintilt saadud (piece) JSON tulemused MagpieBridge'ile sobivateks objektideks
4		parsib Goblintilt saadud (funktsioonide) JSON tulemused MagpieBridge'ile sobivateks objektideks
1	D)	pistikprogrammi käivitamisel käivitatakse Goblinti server
2		teavitab kasutajat sõnumiga kui Goblinti serveri käivitumine ebaõnnestub
3		teavitab kasutajat sõnumiga kui Goblint pole võimeline versiooni kontrollima
4		on võimeline Goblinti versiooni kontrollima
1	E)	loeb Goblinti konfiguratsiooni uuesti, kui see on muutunud
2		teavitab kasutajat sõnumiga kui Goblint pole võimeline konfiguratsiooni lugema

Nõuete haldamiseks ja testide kirjutamise protsessi jälgimiseks kasutati GitHubis pakutavat koodihoidlatele mõeldud teenust. GobPie testimiseks mõtestatud nõuded on GitHubi dokumenteeritud tabeli formaadis. Tabeli veergudeks on vaikimisi *Title* (ee tiitel), *Status* (ee staatus), *Assignees* (ee määratud isikud) ning tiitli ees ülesande number. Määratud isikud tabeli veergu ei kasutatud, kuna teste teostas üks isik. See-eest lisati veerg *Comment* kommentaaride jaoks. Ülesannetele saab lisada järgnevaid staatusi: *Todo* (ee teha), *In Progress* (ee pooleli), *Review* (ee ülevaade), *Not to do* (ee mitte teha), *Done* (ee tehtud). Testide loomise käigus muudeti staatuseid vastavalt nende arengule. Joonisel 1 on kujutatud osa protsessi jälgimise tabelist, kus on nähtavad ka töö käigus protsessi jälgimiseks kasutatud olekud.

	Title	Status	Comment
1	GobPie loeb GobPie konfiguratsiooni (readGobPieConfiguration)	Todo	GobPieConfTest.java
2	GobPie loeb GobPie konfiguratsiooni kõiki parameetreid ning erinevaid tõeväärtusi (readGob...	In Progress	GobPieConfTest.java
3	GobPie täidab nõutud väärtustega konfiguratsiooni lugemisel valikuliste konfiguratsiooni väär...	Review	GobPieConfTest.java
4	GobPie teavitab kasutajat sõnumiga kui GobPie konfiguratsioon süntaks on vale	Not to do	GobPieConfTest.java
5	GobPie teavitab kasutajat kui Gobpie konfiguratsioon failis on parameeter, mida ei tohiks ole...	Done	GobPieConfTest.java, Found bug

Joonis 1. GitHubis kasutatud olekud.

4.2. Testimisvõimekuse loomine

Üksustestimiseks oli esmalt vaja leida sobilik raamistik. Kuna GobPie projekt on teostatud Java programmeerimiskeeles, valiti üksustestimise raamistikuks Java koodi testimiseks laialt levinud raamistik JUnit5[6]. GobPie projekt kasutab projekti sõltuvuste ehitamiseks ning haldamiseks Maven[7] tööriista. Seega lisati JUnit5 kasutamiseks vajalikud sõltuvused projekti `pom.xml` faili.

Kuna GobPie hõlmab endas kahe erineva serveriga (Goblint ja MagpieBridge) suhtlemist, oli GobPie koodi üksustestimiseks vajalik serverite vaheline suhtlus minema abstraheerida. Selleks lisati projektile Mockito[8] raamistiku sõltuvus, mis võimaldab luua objektidest imitatsioone ning seega serveriga suhtlemist imiteerida.

Selleks, et jälgida ja testida programmi poolt väljastatud logisid, lisati projektile LogCaptor[9] ja LogBack[10] raamistiku sõltuvused, mis lihtsustavad logiandmete lugemist. System Stubs Jupiter[11] sõltuvuse lisamine hõlbustas programmi väljundi kontrollimist.

JUnit Jupiter sõltuvustest kasutati kahte moodulit: JUnit Jupiter API[12] ja JUnit Jupiter Engine[13]. JUnit Jupiter API lisamine võimaldas erinevate sisseehitatud funktsioonide kasutamist, nagu `@BeforeEach` ja `@Test`. Need funktsioonid mugavdasid üksustestide loomist ja aitasid vältida koodi kordamist. JUnit Jupiter Engine võimaldab JUnit5 teste käivitada Maven tööriista testimise käsuga, mida kasutatakse testide jooksutamiseks

[6] `https://junit.org/junit5`
[7] `https://maven.apache.org`
[8] `https://mvnrepository.com/artifact/org.mockito/mockito-core`
[9] `https://mvnrepository.com/artifact/io.github.hakky54/logcaptor`
[10] `https://mvnrepository.com/artifact/ch.qos.logback/logback-classic`
[11] `https://mvnrepository.com/artifact/uk.org.webcompere/system-stubs-jupiter`
[12] `https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api`
[13] `https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine`

GitHubi pidevintegratsiooni (ingl *continuous integration*) raamistikus. Kõigi eelnevalt mainitud sõltuvuste koostööna oli testide kirjutamine kvaliteetsem ja tõhusam.

4.3. Testide koostamine

Testide koostamisel võeti aluseks tabelis 1 kirjeldatud testimise nõuded. Alustati kõige olulisemast funktsionaalsusest, milleks on analüüsi käivitamine. Analüüsi funktsionaalsuse testidest on toodud näide joonisel 2, näitamaks testi, mis vastab nõudele A-3. See test kontrollib, kas analüüsi katkestus toimub ootuspäraselt.

```
1  @Test
2  void abortAnalysis() throws IOException {
3      CompletableFuture<GoblintAnalysisResult> runningProcess1 = new CompletableFuture<>();
4      CompletableFuture<GoblintAnalysisResult> runningProcess2 = new CompletableFuture<>();
5
6      doNothing().when(goblintServer).abortAnalysis();
7
8      when(goblintService.analyze(new AnalyzeParams(false))).thenReturn(runningProcess1);
9      goblintAnalysis.analyze(files, analysisConsumer, true);
10     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("---------------------- Analysis
       ↪  started ----------------------")));
11     systemOut.clear();
12
13     when(goblintService.analyze(new AnalyzeParams(false))).thenReturn(runningProcess2);
14     goblintAnalysis.analyze(files, analysisConsumer, true);
15
16     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("--------------- This analysis has
       ↪  been aborted -------------")));
17     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("---------------------- Analysis
       ↪  started ----------------------")));
18
19     verify(magpieServer, times(2)).forwardMessageToClient(new MessageParams(MessageType.Info, "GobPie
       ↪  started analyzing the code."));
20     verify(magpieServer, times(0)).forwardMessageToClient(new MessageParams(MessageType.Info, "GobPie
       ↪  finished analyzing the code."));
21
22     verify(goblintServer).abortAnalysis();
23
24     systemOut.clear();
25     runningProcess2.complete(new GoblintAnalysisResult(List.of("Success")));
26     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("---------------------- Analysis
       ↪  finished ----------------------")));
27     verify(magpieServer).forwardMessageToClient(new MessageParams(MessageType.Info, "GobPie finished
       ↪  analyzing the code."));
28 }
```

Joonis 2. Analüüsi katkestamise test.

Enne testi jooksutamist luuakse imiteeritud (ingl *mock*) objektid, et simuleerida testis vajalikku olukorda. Imiteeritud olukorras simuleeritakse nii Goblinti, kui ka MagpieBridge serverite tööd, mille mõlemaga GobPie suhtlema peab, et testis keskenduda vaid GobPie enda funktsionaalsusele. Lisaks kontrollitakse, kas Goblinti server ikka töötab.

Testi alguses luuakse kaks `CompletableFuture` objekti (read 3-4) kasutades sellist analüüsi protsessi, mis on käivitatud, kuid pole veel lõpule viidud. See võimaldab testis protsessi lõppu oodata ja selle lõpetamist juhtida. Meetodi `abortAnalysis` käivitamisel seatakse see simuleerima mitte millegi tegemist (rida 6), sest antud meetodi sisuks on Goblinti protsessile signaaliluure (ingl *sigint*) käsu saatmine, aga Goblinti serveri protsess testis päriselt ei jookse. Seejärel seadistatakse `GoblintService` analüüs tagastama lõpetamata protsessi (rida 8).

Analüüs käivitatakse esimest korda ning kontrollitakse, kas analüüsi käivitamisest teadaandev sõnum saadeti (read 9-10), mille järel süsteemi väljund tühjendatakse (rida 11). Selleks, et protsesse eristada, pannakse `GoblintService` analüüs tagastama erinevat lõpetamata analüüsi protsessi objekti (rida 13). Analüüsi katkestamiseks kutsutakse teist korda välja `analyze` funktsioon (rida 14), selleks et teine kutse peataks esimese analüüsi protsessi.

Analüüsi katkestamise ja automaatse taaskäivitamise kontrollimiseks loetakse süsteemi väljundit ja võrreldakse väljundeid (read 16-17). Lisaks sellele kontrollitakse, et GobPie analüüsi käivitamise sõnum edastatakse `magpieServer`'ile ning analüüsi lõpetamise sõnumit poleks serverile saadetud (read 19-20). Järgmisena kontrollitakse, kas meetodit `abortAnalysis` on välja kutsutud (rida 22) ning tühjendatakse süsteemi väljund uuesti (rida 24). Analüüsi lõpetamiseks lisatakse lõpetamata objektile loetelu (rida 25). Lõpetamise kontrollimiseks võrreldakse süsteemi väljundit ning kontrollitakse `magpieServer`'ile saadetud sõnumit (read 26-27).

Järgmisena kirjutati testid GobPie konfiguratsiooni failide töötlemise ning vea erindite jaoks. GobPie üheks põhifunktsionaalsuseks on selle seadete lugemine (vt peatükk 1). GobPie seadete lugemisel võib tulla mitmeid erindeid, näiteks võib seadetes olla kasutaja poolt defineeritud mõni seade, mida pole olemas (nõue B-5) või olla puudu nõutud väli (nõue B-8). Viimastest tuleb kasutajat informatiivse sõnumiga teavitada. Joonisel 3 on aga näitena esitatud test konfiguratsiooni faili eduka töötlemise kohta, mis vastab nõudele B-2. Testis kontrollitakse sellise GobPie seadistuse lugemist, kus kõik seaded on kasutaja poolt määratud.

```
1  @Test
2  void testReadCompleteGobPieConfiguration() {
3      GobPieConfReader gobPieConfReader = preFileSetup(2);
4      GobPieConfiguration expectedGobPieConfiguration =
5              new GobPieConfiguration.Builder()
6                      .setGoblintConf("goblint.json")
7                      .setGoblintExecutable("/home/user/goblint/analyzer/goblint")
8                      .setPreAnalyzeCommand(new String[]{"cmake", "-DCMAKE_EXPORT_COMPILE_COMMANDS=ON",
                       ↪  "-B", "build"})
9                      .setAbstractDebugging(false)
10                     .setShowCfg(false)
11                     .setIncrementalAnalysis(true)
12                     .setExplodeGroupWarnings(false)
13                     .createGobPieConfiguration();
14
15     GobPieConfiguration actualGobPieConfiguration = gobPieConfReader.readGobPieConfiguration();
16     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("Reading GobPie configuration from
       ↪  json")));
17     assertTrue(systemOut.getLines().anyMatch(line -> line.contains("GobPie configuration read from
       ↪  json")));
18     assertEquals(expectedGobPieConfiguration, actualGobPieConfiguration);
19 }
```

Joonis 3. GobPie seadete lugemise test.

Testi alguses luuakse objekt `GobPieConfReader` (rida 3), mis sisaldab vajalikku viidet failitee seadistuseni. Failitee leidmiseks kasutati abimeetodit `preFileSetup`. Konfiguratsiooni võrdlemiseks on `Builder`'i abil loodud oodatud väärtustega konfiguratsiooni objekt (read 4-13). Seejärel kutsutakse välja funktsioon `readGobPieConfiguration`, mis loeb sisse `resource` kaustas asuvast failist GobPie seadistuse. Selles testis tehakse ridadel 16-18 kolm erinevat kontrolli. Esiteks kontrollitakse, et logidesse ilmuksid read, mis annavad teada seadete lugemise alustamisest ja lõpetamisest. Viimasena võrreldakse loetud seadistuste vastavust oodatud seadistusele, et kontrollida, kas seadistamine toimub korrektselt.

GobPie pistikprogrammi üks põhifunktsionaalsustest on ka serveri päringute saatmine ja päringute vastuste töötlemine. Näiteks, kui GobPie pistikprogrammi seadistus `showCfg` on sisse lülitatud, küsib GobPie analüüsi käivitamisel Goblinti serverilt funktsioonide listi, et iga funktsiooni juurde lisada nupp selle juhtvoograafi vaatamiseks. Sel juhul vastab Goblinti server GobPie päringule JSON-formaadis sõnumiga, kus on kirjeldatud analüüsitud funktsioonide nimed ja asukohad. Kuna koodiredaktoris kuvab nuppe MagpieBridge server, on oluline, et Goblintilt saadud vastused oleksid õigesti teisendatud. Joonisel 4 esitatud testis, mis vastab nõudele C-4, kontrollitakse seega, kas Goblintilt saadud JSON-vormingus list analüüsitud funktsioonidest on õigesti teisendatud.

```
1  @Test
2  public void testConvertFunctionsFromJson() throws IOException {
3      List<GoblintFunctionsResult> goblintFuncResults = readGoblintFunctionsResponseJson();
4
5      when(goblintService.functions()).thenReturn(CompletableFuture.completedFuture(goblintFuncResults));
6      when(gobPieConfiguration.showCfg()).thenReturn(true);
7      when(goblintService.messages()).thenReturn(CompletableFuture.completedFuture(new ArrayList<>()));
8      goblintAnalysis.analyze(files, analysisConsumer, true);
9
10     URL emptyUrl = new File("").toURI().toURL();
11     GoblintPosition defaultPos = new GoblintPosition(1, 1, 1, 1, emptyUrl);
12     URL exampleUrl = new File("src/example.c").toURI().toURL();
13     List<AnalysisResult> response = new ArrayList<>();
14     response.add(
15             new GoblintCFGAnalysisResult(
16                     new GoblintPosition(8, 13, 0, 0, exampleUrl),
17                     "show cfg",
18                     "t_fun")
19     );
20     response.add(
21             new GoblintCFGAnalysisResult(
22                     new GoblintPosition(15, 23, 0, 0, exampleUrl),
23                     "show arg",
24                     "<arg>")
25     );
26     response.add(
27             new GoblintCFGAnalysisResult(
28                     new GoblintPosition(15, 23, 0, 0, exampleUrl),
29                     "show cfg",
30                     "main")
31     );
32     verify(analysisConsumer).consume(response, "GobPie");
33 }
```

Joonis 4. GobPie analüüsi tulemuse (funktsioonide list) lugemise ja teisendamise test.

Selle funktsionaalsuse testimiseks oli esiteks tarvilik imiteerida Goblinti serverit ning selle saadetud vastuseid. Imiteerimiseks lisati Goblintilt saadud analüüsi vastused JSON failidesse, kust need projektis kasutatud `Gson` teegi abil vastavateks Java objektideks teisendati (rida 3, meetod `readGoblintFunctionsResponseJson`). Seejärel imiteeriti, et funktsioonide listi küsides tagastatakse list sisse loetud funktsioonidest (rida 5), `showCfg` seadistus on GobPie's sisse lülitatud (rida 6) ja Goblint on vastanud ka analüüsi tulemuste päringule (rida 7), mis antud juhul võib olla tühi. Neid imitatsioone võetakse arvesse, kui GobPie's kutsutakse `analyze` meetodit (rida 8). Kui `showCfg` seadistus on sisse lülitatud, küsitakse `analyze` meetodis Goblintilt nuppude kuvamiseks funktsioonide list ning saadud vastus teisendatakse MagpieBridge serverile sobivaks.

Ridadel 10-31 luuakse oodatav list `AnalysisResult` objektidest, mis on selline list, mille GobPie vastavalt Goblinti tulemustele MagpieBridge serverile edastama peab. Näiteks lisatakse ridadel 14-19 oodatavasse listi uus `GoblintCFGAnalysisResult` tüüpi objekt,

mis defineerib, et nupu kuvamise asukohaks on määratud faili `example.c` read 8-13 (veergudeks 0). Nupu nimeks on *show cfg* ja funktsioon, mille kohta see käib, on `t_fun`.

Testi põhimõtteks on, et ühelt poolt on fail, kust on sisse loetud JSON-vormingus vastus Goblintilt (muutuja `goblintFuncResults`), ning teiselt poolt on olemas tulemus, mis kujul see peab olema saadetud edasi MagpieBridge serverile (muutuja `response`). Testi lõpus kontrollitakse, et analüüsi käigus teisendati JSON formaadis saadud tulemused õigesti ning saadeti MagpieBridge serverile. Selle kontrollimiseks verifitseeritakse (real 32), et testi käigus kutsus programm välja `consume` funktsiooni õige teisendusega, ehk tulemusega, mille väärtused on samad nagu on kirjeldatud muutujas `response`.

Kui Goblinti serveri vastuste teisendamise nõuded said testitud, kontrolliti ka Goblinti serveri endaga seotud funktsionaalsuseid. Joonisel 5 on toodud näide Goblinti serveri käivitamise testist, mis vastab nõudele D-1: „GobPie pistikprogrammi käivitumisel käivitatakse Goblinti server“.

```
1  @Test
2  public void testStartGoblintServer() {
3      MagpieServer magpieServer = mock(MagpieServer.class);
4      GobPieConfiguration gobPieConfiguration = mock(GobPieConfiguration.class);
5      GoblintServer goblintServer = spy(new GoblintServer(magpieServer, gobPieConfiguration));
6
7      doReturn(new String[]{"sleep", "10s"}).when(goblintServer).constructGoblintRunCommand();
8      goblintServer.startGoblintServer();
9      assertTrue(systemOut.getLines().anyMatch(line -> line.contains("Goblint run with command: ")));
10 }
```

Joonis 5. Goblinti serveri käivitamise test.

Testi alguses luuakse `MagpieServer` ja `GobPieConfiguration` võltsitud objektid (read 3-4). Seejärel luuakse osaliselt imiteeritud (ingl *spy*) `GoblintServer`'i objekt (rida 5), et mõningaid selle objekti käitumisi testi käigus juhtida. Näiteks oli vaja juhtida, et `constructGoblintRunCommand` välja kutsumisel tagastataks käsk (rida 7), mis paneb protsessi kümneks sekundiks ootama, selle asemel, et Goblinti protsessi päriselt käivitada. Käsk `sleep 10` imiteerib antud juhul Goblinti serverit. Järgmisena kutsutakse välja `startGoblintServer` funktsioon (rida 8), mis jooksutab serveri käivitamiseks mõeldud käsku. Selle funktsionaalsust ei taheta imiteerida, vaid päriselt kontrollida, mida *spy* objekti kasutamine võimaldab. Viimasel real kontrollitakse, kas süsteemi väljund vastab oodatule, ehk kas GobPie jooksutas serveri käivitamiseks antud käsku.

Viimasena kirjutati testid Goblinti seadete värskendamise kontrollimiseks (grupp E). Nende testide struktuur kasutab sarnaseid konstruktsioone nagu kasutati gruppide A-D testides. Käesolevas töös loodi kokku 27 testi, mis on saadavad GitHub repositooriumis. Lingid lähtekoodile on leitavad lisas I. Igale nõudele kirjutati vastav test, välja arvatud nõudele A-2, millele kirjutati kolm erinevat testi. Koodi paremaks haldamiseks kasutati tõmbetaotlusi (ingl *pull requests*), et võimaldada struktureeritud viisil testide esitamist, läbivaatamist ning integreerimist põhikoodiga. Nimekiri tehtud muudatustest, nendega seotud tõmbetaotlustest ja käsitletud nõuetest on saadaval lisas II.

4.4. Leitud vead

Pistikprogrammi testimise käigus leiti mitmeid vigu, mis parandati, et tagada testide edukas läbiviimine. Tehtud parandused saab kategoriseerida kaheks: veaparandus ning koodihooldus. Veaparanduse eesmärk on lahendada konkreetseid vigu süsteemis. Koodihooldus hõlmab tegevusi, mis on suunatud koodi kvaliteedi tõstmiseks, nagu koodi optimeerimine, funktsionaalsuse kaupa õigetesse klassidesse või pakettidesse ümber tõstmine ning kommentaaride lisamine. Alljärgnevalt on toodud ülevaade vigadest, mis avastati testide koostamisel ja millele tehti parandused.
Järgnevas loetelus on välja toodud vead ja nende parandused:

1) Funktsiooni `runCommand` kutsuti välja kahes erinevas kontekstis — nii eelanalüüsi kui ka Goblinti analüüsi protsesside käivitamiseks. Küll aga lisati mõlemal juhul loodud protsessile üks ja seesama `ProcessListener`, mis oli mõeldud vaid Goblinti protsessi jälgima. Selle parandamiseks tehti muudatused, mille tulemusena lisati mõlemale protsessile oma `ProcessListener`, et kasutajale saaks protsesside kohta käivad sõnumid õigesti edastatud.
2) GobPie seadeid lugedes ei hoiatatud kasutajat, kui seadetes on väli, mida `GobPieConfiguration` objektis ei ole. See tähendas, et ka juhtudel, kui kasutaja teeb seadistuse JSON välja nimes vea, ei hoiata programm, et seadet ei loetud. Selle vea parandamiseks loodi `GobPieConfValidatorAdapterFactory` klass, mis viskab tundmatut seadet nähes `JsonParseException` erindi, mis seadete lugemisel kinni püütakse ning selle tulemusel väljastatakse kasutajale sõnum tundmatu seade kohta.

3) Nagu defineeritud peatükis 1, on GobPie programmil seade (`explodeGroupWarnings`), mis plahvatab ühe grupihoiatuse erinevateks hoiatuste instantsideks. Kuigi see funktsionaalsus on põhjendatud vaid andmejooksude hoiatuste jaoks, rakendati seda kõikidel grupihoiatustel. Seega plahvatati ka sellised hoiatused, mille grupp oli loodud vaid statistika näitamiseks, näiteks surnud koodiosade ridade arvu kuvamiseks. See aga saastas pistikprogrammi hoiatuste paneeli duplikeeritud instantsidega. Vea parandamiseks lisati koodi kontroll, et grupihoiatuse sisu plahvatatakse vaid juhul, kui grupil on asukoht määratud, sest Goblintis on vaid andmejooksude grupihoiatustel asukoha väli väärtustatud.

Koodihooldus muudatused:

1) Osa eelanalüüsi funktsionaalsuse põhiloogikast – tühiväärtuse ja tühja sõne jada käsitlus – oli `getter` meetodis. See funktsionaalsus liigutati meetodi `preAnalyse` sisse, et `getter` meetodi eesmärgiks jääks vaid loetud väärtuse edastamine, ning kogu funktsionaalsus, sh kõikide juhtude käsitlemine, oleks ainuüksi `preAnalyse` funktsiooni lugedes selge.
2) Goblinti seadete jälgimiseks loodi `FileWatcher`, aga seda tehti `GoblintAnalysis` konstruktoris klassiväljade väärtustamisel. Selline koodi stiil raskendab testimist, sest konstruktori sees loodavat objekti ei saa lihtsasti imiteerida. Seega tehti uus klass `GoblintConfWatcher`, kuhu liigutati ka Goblinti seadete jälgimise funktsionaalsusega seotud meetod `refreshGoblintConfig`, mis oli ennist defineeritud analüüsi funktsionaalsusele keskenduvas klassis. `GoblintConfWatcher` objekti loomine liigutati omakorda `Main` klassi, kust see anti `GoblintAnalysis` konstruktorisse argumendina edasi. Refaktoreeritud koodiga võimaldati `GoblintConfWatcher` objekti testides imiteerimine ning eraldati erinevad funktsionaalsused eraldiseisvatesse komponentidesse.
3) `GobPieConfiguration` klassi loodi konstruktor, et testides oleks objekti loomisel võimalik selle väljad konkreetsete väärtustega algväärtustada. Lisaks teostati klassi väljade rohkuse tõttu `Builder` klass koos `setter`'itega, et testides oleks võimalik väärtustada vaid testimiseks vajalikke väljasid ning parandada sealhulgas testide loetavust.

4) Klassid (nt serveri vastuste tüübid), mille isendiväljasid väärtustati vaid korra ning mille järel väärtuseid vaid küsiti, muudeti kirjeteks (ingl *record*).
5) Mitmetesse klassidesse lisati `equals` meetod, et võimaldada testides kahe sama objekti võrdlemist nende objektiväljade väärtuste kaudu.
6) Mõned funktsioonid muudeti avalikuks, et oleks võimalik testides neid välja kutsuda.
7) Lisaks tõsteti mitmes kohas meetodeid ja klasse ümber, et kood oleks funktsionaalsuste kaupa klassidesse ning pakettidesse jaotatud. See refaktoriseerimine aitas funktsionaalsuste kaupa testide kvaliteeti paremini hinnata, sest nii muutus klasside ja pakettide katvus peaaegu üheselt korreleerituks funktsionaalsuse katvusega.

Testimise käigus leitud vead ja parandused aitasid süsteemi muuta stabiilsemaks ja töökindlamaks. Veaparandused keskendusid konkreetsete probleemide lahendamisele. Koodihoolduse muudatused parandasid koodikvaliteeti ja selgust, luues parema tarkvara, mida edaspidi arendada ja testida.

5. Tulemuste analüüs

See peatükk annab ülevaate koostatud testide piisavusest, tulemuste arutelust ning võimalikest arendamisvõimalustest.

5.1. Koostatud testide piisavuse hindamine

Testide piisavuse hindamiseks kasutati koodi katvuse hindamise metoodikat. Koodi katvuse hindamiseks on olemas erinevaid tööriistu, näiteks JaCoCo ning IntelliJ IDEA sisseehitatud funktsionaalsus. JaCoCo[14] on testimistööriist, mis võimaldab mõõta rakenduste koodi katvust Java keeles [10]. IntelliJ IDEA katvuse funktsionaalsus[15] võimaldab hinnata üksustestide efektiivsust, näidates kui palju koodist on kaetud.

Siinses töös kasutati testide piisavuse hindamiseks esiti IntelliJ IDEA't, kuna testide arendus toimus IntelliJ IDEA keskkonnas ning sisseehitatud funktsionaalsuse abil oli võimalik hõlpsasti jälgida testimistulemusi ja koodi katvust. IntelliJ koodi katvuse funktsionaalsus pakub automaatselt klasside, meetodite, ridade ning harude katvuskriteeriumite arvutamist, näidates kaetud koodi proportsionaalsust kogu koodi suhtes.

Joonisel 6 on arenduskeskkonnas jooksutatud testide koodi katvus. Esimeses tulbas olev klasside katvus näitab täidetud klasside osakaalu. Selle tulemuseks saadi 60 protsenti. Järgmises tulbas olev meetodi katvus näitab täidetud meetodite osakaalu ning selle tulemuseks on 54 protsenti. Jooksutatud ridade osakaaluks saadi 61 protsenti. Viimase tulba tulemuseks saadi 52 protsenti, mis näitab harude käivitamise osakaalu.

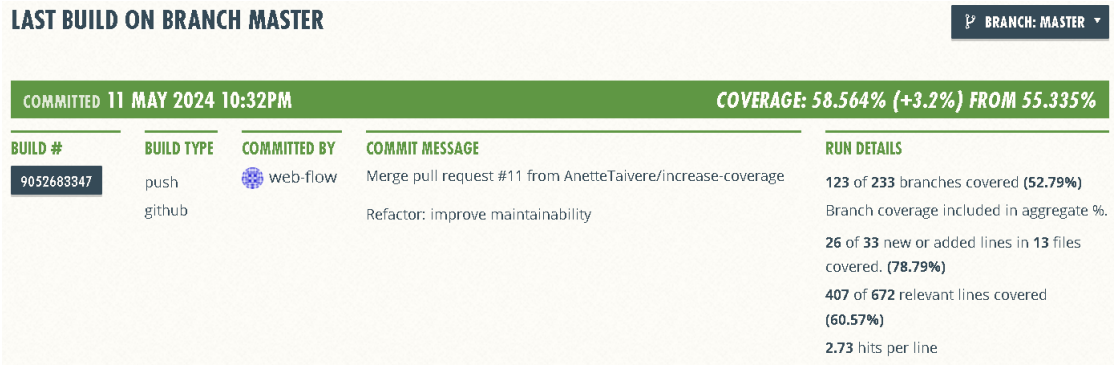
Element	Class, %	Method, %	Line, %	Branch, %
all	60% (30/50)	54% (91/167)	61% (373/606)	52% (121/229)
analysis	100% (3/3)	60% (21/35)	79% (82/103)	57% (30/52)
api	47% (17/36)	40% (33/82)	50% (125/249)	42% (48/112)
goblintserver	100% (4/4)	82% (14/17)	83% (77/92)	76% (13/17)
gobpie	100% (5/5)	100% (19/19)	92% (65/70)	95% (23/24)
util	100% (1/1)	100% (4/4)	68% (24/35)	50% (7/14)
Main	0% (0/1)	0% (0/10)	0% (0/57)	0% (0/10)

Joonis 6. IntelliJ IDEA koodi katvuse hinnang.

[14] https://www.eclemma.org/jacoco
[15] https://www.jetbrains.com/help/idea/code-coverage.html

Testide piisavuse hindamiseks valiti paketid ning failid, mida taheti analüüsida ning funktsionaalsus tuli käivitada kaustas, kus testid asuvad. Koodi katvuses ei hinnatud neid funktsionaalsuseid teostavaid pakette, mis olid nõuetest välja jäänud. Need paketid olid `abstractdebugging`, `HTTPserver`, `magpiebridge` ja `api` kaustas olev `jsonrpc` kaust.

Lisaks kasutati koodi katvuse mõõtmiseks JaCoCo tööriista. Selleks lisati see sõltuvusena projekti `pom.xml` faili. Koodi katvuse nägemiseks laetakse katvuse tulemused pidevintegratsiooni kaudu üles Coveralls[16] veebilehele. Veebilehele laetud raportite abil sai jälgida koodi katvuse muutusi ajas. Veebileht on kättesaadav lisas I. Joonisel 7 on näha JaCoCo tööriistaga arvutatud koodi katvuse protsenti, milleks on 59. Coveralls lehel kuvatakse koondsumma harude ja ridade protsentidest, mis on arvutatud valemiga, kus võetakse lisaks katvuse protsentidele arvesse ka ridade asjakohasust ja jooksutamiste arvu. Lisaks sellele on eraldi välja toodud ka haru katvus, milleks on 53 protsenti, ja rea katvus, mis on 61 protsenti.



Joonis 7. Coveralls koodi katvuse hinnang.

Testide piisavuse hindamiseks valiti vajalikud paketid ja failid ning hinnati nende koodi katvust vastavalt nõuetele. Koodi katvuse hindamisel kasutati kahte tööriista ning nende tulemused olid sarnased, mis kinnitasid hindamise usaldusväärsust.

5.2. Arutelu

Käesolevas töös loodi üksustestid GobPie pistikprogrammile. Selle tulemusena kaeti kõik 25 püsitatud nõuet ning kogu projekti koodi katvuseks saavutati 59 protsenti.

[16] `https://docs.coveralls.io`

Samas, testitavate põhifunktsionaalsustele otseselt vastavate klasside koodi katvused olid kõrgemad. Näitena, Goblint analüüsi funktsionaalsuse ridade katvus oli `GoblintAnalysis` klassis 88 protsenti, Goblinti serveri funktsionaalsuse ridade katvus oli klassis `GoblintServer` 75 protsenti ja Goblinti seadete haldamise ridade katvuseks saadi `GoblintConfWatcher` klassis 100 protsenti. GobPie seadete funktsionaalsusele otseselt vastavate klasside, `GobPieConfiguration` ja `GobPieConfReader`, ridade katvused olid vastavalt 100 ja 86 protsenti.

Mõningates põhifunktsionaalsustele vastavates klassides on ridade katvus madalam. Siinkohal tuuakse mõned põhjused, miks see nii on. Näiteks `GoblintServer` klassis on katmata meetod `abortAnalysis`, mis nagu ülal mainitud, saadab Goblinti protsessile signaaliluure käsu. Kuna Goblint analüsaatorit testides päriselt ei käivitata, ei saa antud meetodit testidega katta. `GobPieConfReader` klassis on madalama katvuse protsendi üheks põhjuseks see, et kuigi programmis eelnevalt kontrollitakse faili olemasolu, nõuab Java ühes sealsetest meetoditest, et faili lugemisel püütakse erind `FileNotFoundException`, kuigi tegelikult ei saa see erind enam tekkida ja seega on selle püüdmine surnud kood.

Suurim kadu kogu projekti koodi katvuses tuli klassidest `GoblintServiceLauncher`, `Main` ning mitmetest klassidest, mis vastavad Goblinti sõnumite struktuuridele ja abstraktse silumise funktsionaalsusele. Viimaseid ei saanud hõlpsasti katvuse analüüsist kõrvale jätta, kuna kõik (k.a. põhifunktsionaalsuse) Goblinti sõnumitele vastavad klassid asuvad ühes paketis.

Kuigi koodi katvuse meetrikad on kõige levinumad koodi kvaliteedi hindamise meetodid ning on heaks indikaatoriks testikomplektide kvaliteedi ja täielikkuse hindamiseks, ei soovitata siiski koodi katvust kasutada ainsa meetrikana, sest see ei ole piisav hindamaks testikomplekti tõhusust [13]. Testkomplekti tõhususe hindamiseks tasub piisavuse meetrikad kombineerida teiste meetrikatega, nagu näiteks mutatsiooni skooriga, mis on just hea meetrika hindamaks testkomplekti tõhusust [13]. Viimane on tingitud sellest, et mutatsiooniskoor näitab, kui suure osa koodis tehtud muudatustest ehk mutatsioonidest suudab testikomplekt ära tunda. Seega annab mutatsiooniskoor indikatsiooni selleks, kui kindel võib olla, et koostatud testikomplekt suudab koodis tehtud vigasid tuvastada ka tulevikus, kui koodi muudetakse.

5.3. Edasine arendamine

Üheks võimaluseks edasisel arendamisel on testimata jäänud programmiosade testimine. See suurendaks testide piisavust koodi suhtes ning muudaks ka teised funktsionaalsused, peale põhifunktsionaalsuste, programmis töökindlamaks.

Peale üksustestide loomist on võimalik edasiarendamisel kasutada teisi testimise meetodeid, näiteks integratsiooni, süsteemi ning kasutajaliidese testimist. Integratsiooni testimine võimaldab kontrollida serverite vahelist suhtlust ning kasutusjuhtude läbitegemine aitaks programmi muuta kasutajasõbralikumaks. Need arendused võimaldaksid kontrollida tarkvara komponentide omavahelist toimimist, süsteemi terviklikkust ning lõppkasutaja kogemust. See aitaks tuvastada võimalikke puudujääke koodis ning aitaks tarkvara kvaliteeti veelgi enam parandada.

6. Kokkuvõte

Bakalaureusetöö eesmärgiks oli uurida GobPie funktsionaalsusi ning süstemaatiliselt testida GobPie pistikprogrammi. Töö algas teoreetilise ülevaatega, kus selgitati erinevaid testimismeetodeid ning testimise vajalikkust, misjärel toodi välja ka Goblinti ja GobPie kirjeldused. Erinevaid testimismeetodeid uurides leiti sobiv testimismeetod GobPie pistikprogrammile. Valitud meetodiks osutus üksustestimine ning seejärel uuriti ka selle testimismeetodi hindamismeetrikaid.

Töö praktilisest osast kirjeldati nõuete kogumist ja planeerimist, testimisvõimekuse loomist ning testide koostamisprotsessi. Kokku koguti 25 funktsionaalset nõuet, mis jaotati vastavalt põhifunktsionaalsustele viieks kogumikuks. Nõuetele vastavalt kirjutati kogumikkude kaupa automaattestid. Loodud teste jooksutatakse GobPie rakenduse käivitamisel.

Töö tulemusena saadi koodi katvuseks 59 protsenti. Testimise käigus leiti koodist vigu, mis parandati ning see aitas tõsta koodi kvaliteeti. Koodi efektiivsuse jaoks tehti koodihooldust, mille käigus paigutati faile erinevatesse kaustadesse ning loodi abistavaid meetodeid ning objekte. Antud tööl on mitmeid võimalusi edasisteks arendusteks. Koodi katvuse suurendamiseks saab kirjutada juurde teste. Lisaks saab pistikprogammil rakendada teisi testimismeetodeid, nagu näiteks vastavustestimist.

Viidatud kirjandus

[1] Andmekaitse ja infoturbe portaal. Cybernetica AS. https://akit.cyber.ee.

[2] GobPie repositoorium. https://github.com/goblint/GobPie (04.12.2023).

[3] Goblint Webpage, „Overview". https://goblint.in.tum.de/home (04.12.2023).

[4] Kavitha Chellappan ja M.K.Jayanthi Kannan. *Code Coverage - Study on Applications of Code Coverage Metrics for Improving Test Effectiveness*. Mai 2022.

[5] Julian Erhard *et al. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap*. 2022. arXiv: 2209.10445 [cs.PL].

[6] Brian Hambling *et al. Software Testing: An ISTQB-BCS Certified Tester Foundation Guide 3rd ed*. BCS, 2015.

[7] Hadi Hemmati. „How Effective Are Code Coverage Criteria?" Teoses: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, lk. 151–156.

[8] Karoliine Holter ja Sarah Tilscher. *GobPie: An IDE Integration for Goblint Using MagpieBridge*. Slides presented at conference ECOOP. 2022. https://2022.ecoop.org/details/pride-2022/3/GobPie-An-IDE-Integration-for-Goblint-Using-MagpieBridge (04.12.2023).

[9] Ferenc Horváth *et al.* „Code coverage differences of Java bytecode and source code instrumentation tools". *Software Quality Journal* 27 (märts 2019), lk. 79–123. https://doi.org/10.1007/s11219-017-9389-z.

[10] Mountainminds GmbH & Co. KG ja Contributors. *JaCoCo Documentation, Coverage Counters*. https://www.jacoco.org/jacoco/trunk/doc/counters.html (10.04.2024).

[11] Linghui Luo, Julian Dolby ja Eric Bodden. „MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors". Teoses: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Toim. Alastair F. Donaldson. Köide 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 21:1.

[12] Aditya P. Mathur. *Foundations of Software Testing, 2/e*. Pearson Education India, 2013. https://books.google.ee/books?id=0UA8BAAAQBAJ.

[13] Samera Obaid Barraood *et al.* „Systematic Literature Review on Test Case Quality Characteristics and Metrics“. Teoses: *2023 3rd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*. 2023, lk. 01–08.

[14] Tauhida Parveen, Scott R. Tilley ja George Gonzalez. „A case study in test management“. Teoses: *Proceedings of the 45th Annual Southeast Regional Conference, 2007, Winston-Salem, North Carolina, USA, March 23-24, 2007*. ACM, 2007, lk. 82. http://doi.acm.org/10.1145/1233341.1233357.

[15] Simmo Saan *et al.* „Goblint: Autotuning Thread-Modular Abstract Interpretation“. Teoses: *Tools and Algorithms for the Construction and Analysis of Systems*. Toim. Sriram Sankaranarayanan ja Natasha Sharygina. (04.12.2023). Cham: Springer Nature Switzerland, 2023, lk. 547–552.

[16] Dr Muhammad Shahid, Suhaimi Ibrahim ja Mohd Mahrin. „A Study on Test Coverage in Software Testing“ (jaanuar 2011), lk. 211.

[17] Divyani Taley ja Dr. Bageshree Pathak. „Comprehensive Study of Software Testing Techniques and Strategies: A Review“. *International Journal of Engineering Research & Technology* vol 9 (2020), lk. 817–822.

[18] Juha Tauriainen. „Correlation of Unit Test Code Coverage with Software Quality“. Teoses: 2023. https://helda.helsinki.fi/server/api/core/bitstreams/d0e5a177-75dd-4a5a-b8c3-b506e2285dcb/content (06.04.2024).

Lisad

I. GobPie testide lähtekood ja koodi katvus

GobPie lähtekood koos selle töö raames loodud üksustestidega on kättesaadav aadressil `https://github.com/AnetteTaivere/GobPie`.

Selle töö käigus tehtud muudatuste ajalugu ning redigeerimistegevus on leitav aadressil `https://github.com/AnetteTaivere/GobPie/commits/master?author=AnetteTaivere`.

Testide jooksutamisel saadud koodi katvuse JaCoCo hinnang on saadaval aadressil `https://coveralls.io/github/AnetteTaivere/GobPie`.

II. GobPie testid ja vastavad tõmbetaotlused

Tõmbetaotlus	Testitud nõuded ja muudatused
#1	A-1; A-2; A-3
#2	A-3; A-4; A-5; A-6; A-7
#3	A-2; A-3; A-4; A-5; A-6; A-7
#4	B-1; B-2; B-3; B-4; B-5; B-6; B-7; B-8
#5	B-5
#6	C-1; C-2; C-4
#7	C-3; D-1; D-2; D-3; D-4; E-1; E-2
#8	veaparandus C-2 testile
#9	projekti rekonstrueerimine
#10	koodi katvuse töövoo lisamine GitHubi pidevintegratsiooni
#11	A-1; A-2; A-3; A-4; A-5; A-6; A-7; C-1; C-2; C-3; C-4 ning koodi hooldatavuse parandamine

III. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Anette Taivere**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   „Visual Studio Code pistikprogrammi testimine GobPie näitel",

   mille juhendaja on **Karoliine Holter**, 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 4.0, 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.

Anette Taivere

15.05.2024