TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika õppekava

Marten Siiber

Aine „Andmebaasid“ automaattestimise vahend

Bakalaureusetöö

Juhendaja: Vambola Leping

Tartu 2015

Aine „Andmebaasid“ automaattestimise vahend

Lühikokkuvõte:

Antud töö sisaldab kirjeldust Java programmi kohta, mille abil saab automaatselt testida kursuse “Andmebaasid” kasutatavat SQL Anywhere andmebaasi. Programm testib andmebaasi vastavust kursuse ülesannete nõuetele.

Töö esimeses osas kirjeldatakse ja analüüsitakse struktureeritud rakenduste ning andmebaasi rakenduste testimist. Seejärel tutvustatakse ja võrreldakse testimise raamistikke. Töö viimases osas kirjeldatakse testimisplaani, automaattestimise programmi arhitektuuri, samuti analüüsitakse loodud programmi vastavust püstitatud nõuetele.

Võtmesõnad:

Andmebaas, Automaattestimine, Java, TestNG

Course “Andmebaasid” automatic testing tool

Abstract:

This thesis contains a description of a Java program that allows automatically test SQL Anywhere database used in course “Andmebaasid”. The program tests whether database meets task requirements.

In the first part of thesis, structured and database application testing is described and analyzed. Next, testing frameworks are introduced and compared. Lastly, the thesis contains a description of the testing plan and architecture of the automatic testing program, also whether the program meets the requirements is analyzed.

Keywords:

Database, Automatic testing, Java, TestNG

Sisukord

1. Sissejuhatus

See bakalaureusetöö kirjeldab Java automaattestimise programmi loomist, mis lihtsustab kursuse "Andmebaasid" loodud andmebaaside nõuetele vastavuse kontrollimist.

Esimeses peatükis kirjeldatakse valdkonna tausta ja ülesande püstitust. Sealjuures tuuakse välja automaattestimise vahendi nõuded.

Teises peatükis kirjeldatakse rakenduste testimist, tuues välja ka andmebaasiga rakenduse testimise iseärasusi. Võrreldakse erinevaid testimistehnikaid, tuuakse välja andmebaasiga rakenduste testimisega seotud probleeme ja kirjeldatakse automaattestimist.

Kolmandas peatükis tutvustatakse ja võrreldakse omavahel populaarse automaattestimise raamistiku JUnit versioonide 3 ja 4 erinevusi. Tutvustatakse automaattestimise vahendi loomiseks kasutatud raamistikku TestNG ja võrreldakse seda JUnit 4'ga.

Viimases peatükis kirjeldatakse testimisplaani ja programmi arhitektuuri, analüüsitakse programmi koodi ja programmi vastavust püsitatud nõuetele. Lõpus tuuakse välja ka viisid edasiseks arenduseks.

2. Ülesande püstitus

Tartu Ülikoolis kasutatakse andmebaase tutvustavas kursuses relatsiooniliste andmebaaside haldamissüsteemi Sybase SQL Anywhere [1]. Kursuse praktikumiülesannete ja kodutööde tulemusena valmib andmebaas, kuhu on vajalik luua:

- Tabelid
- Seosed tabelite vahel
- Vaated
- Funktsioonid ja protseduurid
- Indeksid
- Trigerid

Lisaks sellele on vajalik veel andmete sisestamine failist erinevatesse tabelitesse. [2]

Probleemiks on see, et tööde kontrollimine ja hindamine toimub käsitsi. Selle teeb raskemaks ka see, et ülesanded on erinevad ja seega on nende kontrollimiseks vaja kasutada erinevaid viise. Kahjuks ei leidu vahendeid, mis võimaldaks testida kodutöödeks esitatud andmebaase automaatselt.

Andmebaasi loomisel tekitatakse andmebaasi- ja logifail [1]. Tekitatud andmebaasifail, mis koosneb kogu andmebaasi sisust, esitatakse tudengi poolt ja seejärel antakse automaattestimisel sisendiks. Kõigepealt on vajalik andmebaasi käivitamine, mis võimaldab andmebaasiga ühenduda. Peale andmebaasiga ühendamist on võimalik teha päringuid ja kontrollida andmebaasi vastavust ülesande nõuetele.

Sellise automaattestimise vahendi loomisel tuleks arvestada mitme asjaoluga. Esiteks sellega, et kuna kodutööd on jagatud viieks erinevaks ülesandeks, siis oleks võimalik testida ka viies erinevas järgus olevat andmebaasi. Teiseks sellega, et esitatud andmebaasides võib esineda lihtsamaid vigu, näiteks väikseid erinevusi tabelite nimedes ja sellega ei tohiks lugeda kogu tabelit valesti tehtuks. Kolmandaks sellega, et hinnatud andmebaasile antakse vastavalt

tagasisidet. Neljandaks sellega, et nõuete muutmisel oleks võimalik ka automaattestimise vahend vastavusse viia nende muudatusega.

3. Andmebaaside testimine

3.1 Testimise vajadus

Viimase kümnendiga on arvuti riistvara märgatavalt arenenud, pakkudes palju suuremat töötlemise võimet ja kiirust. Seega on ka tarkvara suurus ja keerukus kasvanud eksponentsiaalselt. Tarkvara on väga tähtis pea igale tööstusele ja inimesed on aina rohkem sõltuvad arvutitest igapäevaste ülesannetega täitmisel. Tarkvara suuruse ja keerukuse kasvamisega suureneb ka vajadus tagada tarkvara usaldusväärsus ning täpsus. Tarkvaravead võivad saatuslikuks saada mitmes tööstuses, sealhulgas näiteks lennunduses, meditsiinis ja sõjanduses. Tarkvarale ja selle testimisele ning hooldamisele kulutatakse aastas miljardeid. Tarkvaravead toovad aga endaga kaasa sadu miljoneid kahjumit aastas. [3]

Samuti pööratakse suurt tähelepanu tarkvara usaldusväärsuse, täpsuse, turvalisuse ja kasulikkuse tagamisse. Tarkvara elutsükli jooksul on oluline pühendada palju ressursse, sealhulgas aega, vaeva ja eelarvet. Nii on võimalik tagada, et lõpptoode on kasulik, kasutatav ja täidaks need ülesanded, milleks see loodi. [3]

3.2 Testimisviisid

Andmebaasi rakenduse testimist saab liigitada funktsionaalsuse testimiseks, jõudluse testimiseks (koormuse ja stressi, skaleeruvuse testimiseks), turvalisuse testimiseks, keskkonna ja ühilduvuse testimiseks ja kasutatavuse testimiseks. [4]

Eesmärgiks seatud automaattestimise vahend peaks kontrollima vastavust vaid funktsionaalsuse nõuetele, seega on järgnevad peatükid funktsionaalsuse testimise kohta.

3.3 Testimistehnikad

Funktsionaalsuse testimise lähenemist võib jagada kolme gruppi - kogemusepõhiseks testimiseks (ingl.k. *experience-based testing*), staatiliseks testimiseks (ingl.k. *static testing*) ja dünaamiliseks testimiseks (ingl.k. *dynamic testing*). Kogemusepõhine testimine tugineb testijate ja kasutajate oskusteabel. Staatilisel testimisel vaadatakse üle programmi kood ilma seda täitmata, et näiteks

leida algväärtustamata muutujaid. Dünaamilisel testimisel käivitatakse programm ja kontrollitakse, kas tagastatakse oodatud tulemus. Dünaamilisel testimisel on kaks põhilist lähenemist - musta ja valge kasti testimine. [5]

3.3.1 Musta kasti testimine

Musta kasti testimisel tuletatakse testjuhtumid (ingl.k. *test cases*) omamata ülevaadet programmi struktuurist [6]. Testjuhtumid luuakse ilma programmi lähtekoodi vaatamata, vaid analüüsides spetsifikatsiooni [5].

Musta kasti testimisel kasutatavad tehnikad hõlmavad endas näiteks rajaväärtusanalüüs (ingl.k. *boundary value analysis*), ekvivalentsiklasside testimist (ingl.k. *equivalence class testing*) ja otsustustabeltestimist (ingl.k. *decision table testing*). [3]

Funktsionaaltestimisele viidatakse vahel ka kui musta kasti testimisele. Funktsionaaltestimise nimetus tuleneb sellest, et tarkvara vaadeldakse kui funktsiooni, millele antakse sisendid ja saadakse väljundid. Nii saab testjuhtumeid kasutada ka siis, kui programmi lähtekood või struktuur peaks muutuma, või kui programm transleeritakse teise keelde. Funktsionaaltestimise eelis on ka veel see, et kuna see põhineb spetsifikatsioonil, siis saab testjuhtumid luua varases arendusfaasis, isegi enne programmi teostust. [3]
Funktsionaaltestimise puuduseks on see, et ilma programmi uurimata ei ole ka teada kui palju programmist testitakse. Funktsionaalsed nõuded on tavaliselt määratud loomuliku keelega ja seega võib tekkida probleeme üleliigse, vastuolulise või puuduliku spetsifikatsiooniga. [6]

Musta kasti testimisel jäävad leidmata teatud liiki vead, mis on tavalised andmebaasi rakendustes. Näiteks võib tekkida olukord kui üks funktsioon teeb andmebaasis muudatusi ja need muudatused mõjutavad teise funktsiooni tulemust ning nii võib mingis järjekorras erinevaid funktsioone rakendades jõuda ebaõnnestumiseni. [6]

3.3.2 Valge kasti testimine

Valge kasti testimisel, vastupidiselt musta kasti testimisele, analüüsitakse programmi lähtekoodi. Seda tehakse selleks, et leida programmis kõikvõimalikud täitmisele minevad harud ja

argumentide kogumid, millega tehakse kindlaks et ettenähtud täitmisele minevad harud programmis valitakse. [5]

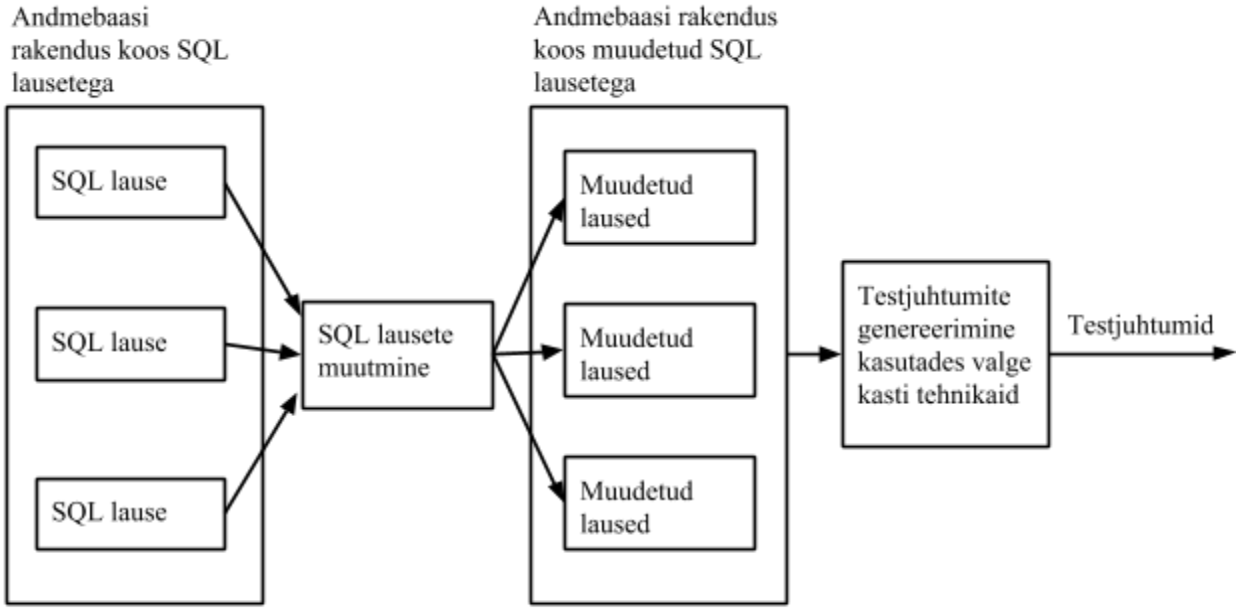
Valge kasti testimisel kasutatavad tehnikad on näiteks lausete testimine (ingl.k. *statement testing*), harupõhine testimine (ingl.k. *branch testing*), tingimuste katvuse testimine (ingl.k. *condition coverage*) ja teede testimine (ingl.k. *path testing*). [6]

Struktuurtestimisele viidatakse vahel kui valge või läbipaistva kasti testimisele. Struktuurtestimise eeliseks on see, et on võimalik kindlaks teha, et kõik harud programmi lähtekoodis läbitakse testjuhtumis. Nii saab näiteks struktuurtestimist kasutada kui katvuse maatriksit (ingl.k. *coverage matrix*), millega saab suunata funktsionaaltestimise tehnikaid. [3]

Traditsioonilistel valge kasti tehnikatel on oma piirangud andmebaaside rakenduste testimisel. Kõige olulisem piirang on see, et programmis leiduvaid SQL lauseid ei arvestata ja seega vaadeldakse neid SQL lauseid kui muste kaste. Tavaliselt luuakse vaid mõningad testjuhtumid, mis testivad SQL lauseid. Testjuhtumeid ei looda selleks, et need arvestaksid SQL lausete semantikaga, mis peegeldavad andmebaaside sisemisi muudatusi. Nii võib traditsioonilises valge kasti testimisel vaatamata jääda sellist tüüpi vead, mis on seotud andmebaasi sisemiste muudatusega. [6]

3.3.3 *WHODATE* tehnika

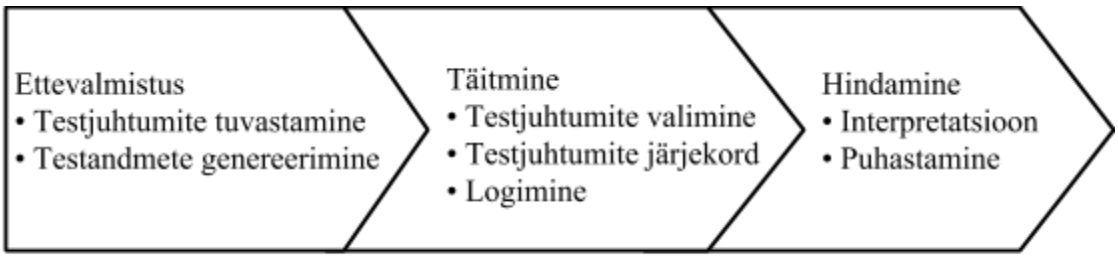
Valge kasti testimise tehnikad on välja kujunenud üldiste tarkvara rakenduste jaoks. Selleks, et neid tehnikaid saaks kasutada andmebaasi rakendustes, on loodud tehnika nimega *WHODATE*. Selle nimetus tuleb inglisekeelsest väljendist *WHite bOx Database Applicationg TEsting*. Selle tehnikaga luuakse traditsioonilises valge kasti tehnikates puuduvad testjuhtumid programmi SQL lausete testimiseks (vt. Joonis 1). Andmebaasi rakenduses kasutatavad SQL laused muudetakse üldotstarbelise programmeerimiskeele või lühendina GPL (ingl.k. *general-purpose programming language*) lauseteks. Peale seda kasutatakse juba tavalisi valge kasti testimise tehnikaid nii muudetud lausete kui ka teiste programmeerimiskeele lausete peal. [6]



Joonis 1. WHODATE lähenemine [6].

3.4 Testimise etapid

Andmebaasi rakenduste testimisprotsessi võib jagada 3 etapiks - ettevalmistus, täitmine ja hindamine (vt. Joonis 2). Ettevalmistuse etapi ülesanneteks on testjuhtumite tuvastamine ja testandmete genereerimine. Täitmise etapp koosneb 3 ülesandest. Esimeseks ülesandeks on valida asjakohased testjuhtumid regressioontestimise jaoks. Teiseks ülesandeks on leida optimaalne testjuhtumite täitmise järjekord, et vältida vajadust laadida sisse andmebaasi seis peale igat testjuhtumi täitmist. Kolmandaks on oluline logida täitmine, et ebaõnnestumisi oleks lihtsam pärast analüüsida. Hindamise etapis toimub testi tulemuste interpretatsioon ja andmebaasi puhastamine. [5]



Joonis 2. Testimise etapid [5].

3.5 Testimisega seonduvad raskused

Andmebaaside süsteemi testimine on keeruline ülesanne. Andmebaasi rakenduse jaoks testide loomisega tuleb arvestada mitme olulise erinevusega stuktureeritud ja andmebaasi rakenduse vahel.

Esiteks on tähenduslike ja efektiivsete testjuhtumite loomine raske. Näiteks peetakse keeruliseks andmebaaside semantika terviklikkuse kohustuse (ingl.k. *integrity constraint*) ja osaluskohustuse (ingl.k. *participation constraint*) testimist. Struktureeritud rakenduses testitakse kindlat algoritmi koos kindlat tüüpi sisendväärtustega, et leida vastus. Andmebaasiga rakenduses see algoritm asendatakse rakenduse poolt loodud SQL päringuga. Iga päringuga võib muutuda sisendväärtuste arv ja tüüp. Päringu loomisel tuleb arvestada andmebaasiskeemiga, kuna iga rakendus on ühendatud erineva skeemiga. Selle skeemi teadmine on vajalik testide loomisel, mis läbivad rakenduse, et jõuda mõtestatud tulemuseni. Kõikide SQL päringute hulk, mis on nii süntaktiliselt kui ka semantiliselt korrektsed, koosneb kahest alamhulgast - päringutest, mis on tähenduslikud antud rakendusele ja nendest päringutest, mis ei ole. [7]

Teiseks erinevuseks on oleku küsimus. Erinevalt struktureeritud rakendustest, on andmebaasi rakendustes vajaliku sisendina andmebaasi olek. Sellega on kaks erinevat mure - andmebaasi oleku kindlaksmääramine enne testjuhtumi käivitamist ja testjuhtumite järjestuse planeerimine nii, et muudatused andmebaasi olekus ei muudaks järgnevaid testjuhtumeid tähtsusetuks. Mittetriviaalsetel andmebaasidel, nagu on kasutusel pärismaailmas, pole mõistlik taastada andmebaasi olek peale iga testi jooksutamist. Igal testjuhtumil on potentsiaal muuta andmebaasi olekut ja seega ka järgmise testjuhtumi sisendit. Kuigi kogu andmebaasi võib pidada testjuhtumi sisendiks, siis võib teadmine, milliseid andmebaasi osasid jooksev ja eelnevad testjuhtumid mõjutavad, aidata eraldada omavahel seotud testjuhtumid ning nii teha minimaalselt andmebaasi oleku taastamisi testimise vältel. [7]

Kolmandaks erinevuseks struktureeritud ja andmebaasi rakenduse vahel on see, kuidas kontrollitakse testi õnnestumist peale testjuhtumi jooksutamist. Jällegi peetakse testjuhtumi üheks sisendiks kogu andmebaasi olekut, kuid nüüd on see ka koos päringu tulemusega väljundiks. Testjuhtumi korrektsuse kontrollimisega kaasneb sisend- ja väljundoleku muudatuste

võrdlemine ning päringu väljundi õigsuse kontrollimine. Samuti on oluline olekute võrdlemisel see, et toimuvad päringu poolt tehtud muudatused ning muid muudatusi ei toimu. [7]

Veel üheks andmebaasi süsteemide mureks on platformist sõltumatus, nii andmebaasi kui ka andmebaasile juurdepääsuga rakenduse puhul. Testjuhtumeid ja nende eeldatavat tulemust peaks pidama platformist sõltumatuks, alles testide jooksutamisel tuleks arvestada platformi. Testjuhtumid luuakse rakenduse tasemel ja neid peaks looma olenemata platformist, mille peal töötab andmebaas. Rakendusliides (API - ingl.k. *Application Programming Interface*), mida kasutatakse andmebaasi juurdepääsuks, peaks jääma samaks isegi kui andmebaas liigutatakse teise platformi peale, samuti peaks jääma samaks testjuhtumi tulemus antud andmebaasi olekuga. Juhul kui andmebaas on mingi kindla platformi peal ja juurdepääsuga rakendus liigutatakse teise platformi peale, siis sama põhimõte jääb kehtima. [7]

3.6 Automaattestimine

Automaattestimine on tarkvara rakendamine kontrollima testide käivitamist, võrdlema tegeliku väljundit ennustatud väljundiga, üles seadma testide eelduseid ja muid testide kontrollimise ning aruandluse funktsioone. Teisisõnu on testide automatiseerimine programmi kirjutamine, mille ülesandeks on testida, mis muidu toimuks käsitsi. [8]

Mitmeid automaattestimise tööriistu on juba arenduses, kuna automatiseerimisel on mitmeid eeliseid:

- Kiirus - testide automatiseerimine võib kiirendada testimist sadu tuhandeid kordi.
- Efektiivsus - automaattestimise tööriistade kasutamine on aitab aega säästa ja nii võib seda aega testimise plaani tegemiseks ja uute testjuhtumite kavandamiseks.
- Korrektsus ja täpsus - automatiseeritud testimise tööriist keskendub alati sellele, mis oli algselt kavandatud. Ükskõik mitu korda käsitletakse ühte testjuhtumit, ei teki iial ühtegi viga.
- Järjepidevus - automaattestimise tööriist ei väsi iial ega anna poole pealt alla. [8]

Kuid leidub ka märkismisväärseid puuduseid:

- Tarkvara muutub - näiteks kui muutub mõni nimi või funktsioon, muutub ka salvestatud tegevuste jada kehtetuks.
- Inimese vaatlemine ja intuitsioon on asendamatud - kui peaks juhtuma mõni õnnetus testimisprotseduuris, siis automaatset testimist pole võimalik pidevalt täita.
- .Kontrollimine on ebapraktiline - automaatse testimise metoodikad on tavaliselt üldised ja seega on peale muudatusi raske kontrollida korrektsust.
- Automaattestimise tööriistad pole võimelised leidma kõiki vigu täielikult, seega pole hea olla sõltuv automaattestimise tööriistast suurel määral.

Hoolimata nendest puudustest, on automaattestimise tööriistad palju abiks. [8]

Automatiseeritud testimisel on kaks lähenemisviisi - koodikeskne testimine (ingl.k. *code-driven testing*) ja graafilise kasutajaliidese testimine(GUI - ingl.k. *Graphical User Interface testing*). Koodikesksel testimisel testitakse avalikke liidesed klassidele, moodulitele või teekidele erinevate sisend argumentidega, et kontrollida kas tagastatud tulemused on korrektsed. Graafilisel kasutajaliidese testimisel jäljendatakse kasutaja hiire klõpsimist ja klahvivajutusi või ükskõik mida, mis viib graafilise kasutajaliidese muutuseni. [8]

4. Automaattestimise raamistikud

4.1 JUnit

JUnit on avatud lähtekoodiga Java raamistik, millega on võimalik kirjutada ja korduvalt jooksutada teste. JUnit autoriteks on Erich Gamma ja Kent Beck [9]. 2013. aastal tehtud uuringu järgi, milles oli vaatluse all 10000 versioonihaldusüsteemi GitHub projekti, leiti et JUnit on koos slf4j-api'ga kõige populaarsemad Java teegid [10]. JUnit'i võimaluste hulka kuuluvad:

- Väited (ingl.k. *assertion*) oodatavate tulemuste testimiseks.
- Testide alused (ingl.k. *test fixture*) ühiste testandmete jagamiseks.
- Testide täitjast (ingl.k. *test runner*) testide jooksutamiseks. [9]

4.1.1 JUnit4 võimalused ja võrdlus JUnit3'ga

JUnit'i neljas versioon tõi mitmeid suuri uuendusi, seda võib seetõttu ka pidada täiesti uueks raamistikuks.

JUnit 4 on edasi- ja tagasiühilduv. See tähendab, et JUnit 4 testide täitjaga on võimalik jooksutada JUnit 3'e teste ja sama on võimalik ka vastupidi. Tagasiühilduvuse mõttega on ka muudetud paketi struktuuri, JUnit 3 kuulub paketti *junit.framework* ja JUnit 4 paketti *org.junit*. [11]

Kui varasemalt oli testjuhtumi loomiseks vaja laiendada klassi *junit.framework.TestCase*, siis JUnit4 testjuhtumiks sobib iga avalik klass koos null argumendiga avalik konstruktor. JUnit3 testjuhtumi klassi laiendamisega tulevad kaasa ka väitemeetodid (ingl.k. *assert method*), JUnit4's saadakse need meetodid kasutades Java 5 staatilist importimist. [11]

Lisaks staatilisele importile kasutab JUnit4 Java 5 annotatsioone. Kui JUnit3 testmeetodi nimi peab algama sõnaga "test", siis JUnit4 kasutab testmeetodi märkimise jaoks annotatsiooni *@Test*. Näiteks JUnit3 testjuhtumi klassi loomisel vajaliku klassi laiendamisega on vaja ka üle katta meetodid *setUp()* ja *tearDown()*, mis käivitatakse vastavalt enne ja peale igat testmeetodit, mõne ühise loogika käivitamiseks. JUnit4'ga on sama lahendatud annotatsioonidega *@Before* ja

@After, kusjuures võib neid ühes testjuhtumi klassis mitu. Joonisel 3 on näha JUnit 3 testjuhtumi näide ja joonisel 4 vastav näide JUnit 4's. [11]

```
import junit.framework.TestCase;
public class AdditionTest extends TestCase {
   public void setUp() { … }
   public void tearDown() { … }
   public void testAddition() {
      assertEquals(3, 1+2);
   }
}
```

Joonis 3. JUnit 3 testjuhtumi näide.

```
import org.junit.Before;
import org.junit.After;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class AdditionTest {
   @Before
   public void initialize() { … }
   @After
   public void destroy() { … }
   @Test
   public void testAddition() {
      assertEquals(3, 1+2);
   }
}
```

Joonis 4. JUnit 4 testjuhtumi näide.

Lisaks sellele, et annotatsioonidega on asendatud mõned varasemad võimalused, on annotatsioonidega juurde tekkinud ka uusi võimalusi. Näiteks kasutab JUnit4 annotatsioone *@BeforeClass* ja *@AfterClass*, mis lubavad märgitud meetodit jooksutada vastavalt enne ja peale igat testjuhtumi klassi. Neid annotatsioone on kasulik kasutada mõne ressursinõudliku meetodi, näiteks andmebaasiga ühendamise jaoks. [11]

Erinevus on ka testide ignoreerimise vahel, JUnit3'ga saab testmeetodeid vahele jätta neid välja kommenteerides koodist või muutes nime mittevastavaks (nõutud on, et testmeetod algab sõnaga "test"). JUnit4 kasutab annotatsiooni *@Ignore*, mida saab ka alates versioonist 4.3 kasutada testjuhtumi klassi peal, jättes nii vahele kõik selles olevad testmeetodid. Peale testide jooksutamist on JUnit4 testide kokkuvõttes ära märgitud vahele jäetud testide arv, mida JUnit3 ei näita. [11]

Annotatsioonidele on võimalik anda ka parameetreid. Näiteks erindite testimisel tuleb JUnit3's kasutada *try-catch* konstruktsiooni ja kui erindit siiski ei tulnud, tuleb test muuta ebaõnnestuvaks. JUnit4 kasutab erindite testimisel testmeetodi annotatsiooni *@Test* parameetrit *expected*. Lisaks parameetrile *expected*, on testmeetodi annotatsioonis võimalik märkida ka testmeetodi maksimaalselt lubatud aeg parameetriga *timeout*. JUnit3's selline võimalus puudub.

JUnit3 tuleb koos Swing ja AWT testide täitjadega, mis pakuvad graafilist ülevaadet õnnestunud ja ebaõnnestunud testidest. JUnit4's puuduvad graafilised testide täitjad ja graafilist ülevaadet pakuvad kõik JUnit 4.x toega IDE'd. [11]

Testide kogumike (ingl.k. *test suite*) täitmiseks tuleb JUnit3's kirjutada meetod *suite()* ja sinna lisada manuaalselt kõik ettenähtud testid (vt. Joonis 5). JUnit4 pakub testide kogumike täitmiseks annotatsioone *@RunWith* ja *@SuiteClasses*. Esimesele annotatsioonile antakse parameetriga *value* edasi mõni olemas (testide kogumike puhul *Suite.class*, mis sisaldub JUnit4's) või ise defineeritud testide täitja klass. Teine annotatsioon võtab parameetriga *value* kõik antud testide kogumiku jaoks ettenähtud testjuhtumite klassid (vt. Joonis 6).

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class AllTests {
  public static Test suite(){
    TestSuite suite = new TestSuite();
    suite.addTestSuite(AdditionTest.class);
    return suite;
  }
}
```

Joonis 5. JUnit 3 testikogumiku loomise näide.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(value=Suite.class)
@SuiteClasses(value={AdditionTest.class})
public class AllTests {
 …
}
```

Joonis 6. JUnit 4 testikogumiku loomise näide.

JUnit4's sisalduvate testide täitja klasside hulka kuulub ka *Parameterized.class*, mille abil on võimalik täita samu teste erinevate sisendandmetega. Sisendandmete defineerimiseks tuleb klassi sees defineerida staatiline meetod koos annotatsiooniga *@Parameters*. [11]

JUnit4 sisaldab uusi väiteid. JUnit3's puudus võimalus võrrelda kahe massiivi võrdsust, kuid JUnit4's on see olemas ja versioonist 4.6 alates on võimalik võrrelda näiteks ujukomaarvude massiive, andes parameetrina kaasa ka lubatud täpsusvea. Alates versioonist 4.4 sisaldab JUnit ka Hamcresti raamistiku väiteid, millega on võimalik kirjutada paindlikemaid ja paremini loetavamaid testide väiteid. Samuti aitavad väiteid lihtsamini kirjutada Hamcrest'iga lisanduvad sobitajad (ingl.k. *matchers*) (vt. Joonis 7). [11]

```
import org.junit.Test;
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.not;
public class AdditionTest {
  @Test
  public void testAddition() {
    assertThat(2+3, is(5));
    assertThat(2+2, is(not(5)));
  }
}
```

Joonis 7. JUnit 4 Hamcresti väited ja sobitajad.

Versiooniga 4.4 tutvustati ka eeldusi (ingl.k. *assumption*), millega on võimalik täita testmeetod mingitel kindlatel eeldustel. Näiteks võib eeldada enne faili tee kontrollimist, et tee eraldaja sümbol on määratletud vastavalt (Windows'i puhul '\' ja Unix'i puhul '/'). Alles siis kui eeldus on täidetud, hakatakse kontrollima väidet. [11]

4.2 TestNG

TestNG on testimisraamistik, mis on inspireeritud JUnit'ist ja NUnit'ist ja mille autoriks on Cédric Beust. TestNG on disainitud, et lihtsustada laiemat testimise vajadust, alates ühiktestimisest (testides ühte klassi isoleeritud teistest) kuni integratsiooni testimiseni (testides terveid süsteeme, mis koosnevad mitmest klassist, paketist ja välisest raamistikust, nagu rakendusserverid). [12]

4.2.1 TestNG võimalused ja võrdlus JUnit4'ga

Võrreldes JUnit4'ga on TestNG võimalused vägagi sarnased.

Mõlemad raamistikud kasutavad annotatsioone, mis näevad välja sarnased, kuid leidub ka erinevusi. Näiteks mõlemas raamistikus olevate annotatsioonide *@BeforeClass* ja *@AfterClass* puhul nõuab JUnit4, et nendega märgitud meetod oleks staatiline, TestNG on meetodi deklareerimisel paindlikum ning seda ei nõua. Erinevusi on ka annotatsioonide olemasolus. Kui JUnit4 pakub võimalust käivitada mingit ühise loogikaga meetodit enne ja peale igat testmeetodit või testjuhtumi klassi, siis TestNG pakub lisaks nendele sama funktsionaalsust rohkematel tasemetel. Annotatsioonidega *@BeforeSuite* ja *@AfterSuite* on võimalik käivitada märgitud

meetod vastavalt enne ja peale testide kogumiku kõiki teste ning annotatsioonidega *@BeforeTest* ja *@AfterTest* enne ja peale testi. Gruppidesse jaotatud testmeetodite jaoks pakuvad annotatsioonid @BeforeGroup ja @AfterGroup käivitada mõnda märgitud meetodit vastavalt enne esimest ja peale viimast gruppi kuuluvat meetodit. Teiste annotatsioonide võrdlus on kirjeldatud töö lisades. [13]

Eelpool nimetatud testide gruppideks jaotamise võimalus oli algselt olemas vaid TestNG's, kuid alates versioonist 4.8 on see võimalus olemas ka JUnit's annotatsiooniga *@Categories* [14]. TestNG lubab mitte ainult deklareerida gruppidesse kuuluvaid meetodeid vaid saab ka määratleda gruppe, mis sisaldavad teisi gruppe. Nii saab valida, millised gruppide hulgad arvatakse kaasa ja millised gruppide hulgad välja testide käivitamisel. [15]

Üheks suuremaks erinevuseks on see, et TestNG kasutab XML faili testikogumike loomiseks (vt. Joonis 8). Kusjuures ei tähenda see seda, et testide käivitamiseks oleks vajalik XML faili olemasolu, sama on võimalik saavutada ka programmiliselt (vt. Joonis 9). Selleks, et testide kogumikku luua, võib kõige üldisemalt määratleda paketi, kus kõik testjuhtumite klassid asetsevad. Täpsemalt võib määratleda testide kogumikku ka konkreetseid klasse, klasside meetodeid või eespool kirjeldatud gruppe. [15]

```
<suite name="AllTests">
  <test name="Addition">
    <classes>
      <class name="AdditionTest">
        <methods>
          <include name="testAddition"/>
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

Joonis 8. TestNG testikogumik XML'is.

```
import org.testng.TestNG;
public class AllTests {
   public static void main(String[] args) {
      TestNG testng = new TestNG();
      testng.setTestClasses(new Class [] {
                                        AdditionTest.class
                                        });
      testng.run();
   }
}
```

Joonis 9. TestNG testide käivitamine programmiliselt.

Parameetritega testide loomine on samuti olemas mõlemas raamistikus, kuid mõlemad kasutavad selle teostusel väga erinevaid meetodeid.

JUnit4 kasutab parameetrite määramiseks varem kirjeldatud annotatsioone *@RunWith* ja *@Parameter*. Kusjuures peab annotatsiooniga *@Parameter* märgitud meetod tagastama parameetrite väärtused loendi massiivina ja parameetrid antakse edasi klassi konstruktorile

argumendina. See seab aga mitmeid piiranguid, näiteks peab konstruktorile antav argument algväärtustama mõne klassi muutuja, et seda parameetri väärtust saaks kasutada testimisel. Kuna parameetrite klassi tagastatav tüüp on loendite massiiv, on selles olevad parameetrid piiratud olema kas sõned või algtüübi väärtused. [13]

TestNG pakub mitmeid võimalusi parameetritega testide loomiseks.
Üheks võimaluseks on kasutada XML failis märgendit *parameter*, mille atribuutidele *name* ja *value* antakse parameetri nimi ning sellele vastav väärtus (vt. Joonis 10). Selleks, et seda parameetrit kasutada, tuleb testmeetodile märkida annotatsioon *@Parameters*, mille argumendi *value* väärtuseks tuleb anda XML failis määratud parameetri nimi (vt. Joonis 11). XML failiga parameetreid andes saab ühte testjuhtumit korduvalt kasutada erinevate andmetega hulgaga ja lisaks on ka lõppkasutajal võimalik testimiseks anda oma andmeid. [13]

```
<suite name="AllTests">
  <parameter name="number" value="2"/>
  <test name="Addition">
    <classes>
      <class name="AdditionTest"/>
    </classes>
  </test>
</suite>
```

Joonis 10. TestNG parameetrite andmine XML'is.

```
import org.testng.annotations.Parameters;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class AdditionTest {
  @Parameters({ "number" })
  @Test
  public void testAddition(int number) {
    assertEquals(1+1, number);
  }
}
```

Joonis 11. TestNG XML parameetrite kasutamine.

Teiseks võimaluseks on kasutada annotatsiooni *@DataProvider*, millega on võimalik testidele anda parameetriteks keerulisemaid tüüpe, kui seda lubab XML failiga parameetrite andmine. Näiteks on lubatud ka enda loodud objektide parameetriks andmine. Sellisel viisil parameetrite andmisel tuleb andmed tagastav meetod märkida annotatsiooniga *@DataProvider*, mille parameetrile *name* antakse nimi ja see nimi tuleb anda väärtuseks testmeetodi annotatsiooni *@Test* parameetrile *dataProvider* (vt. Joonis 12). [13]

```
@DataProvider(name = "numberProvider")
public Object[][] createNumbers() {
  return new Object[][] {
    { 2 },
    { 3 }
  };
}
@Test(dataProvider = “numberProvider”)
public void testAddition(int number) {
  assertEquals(1+1, number);
}
```

Joonis 12. TestNG @DataProvider.

TestNG’l on võimalus genereerida teste dünaamiliselt annotatsiooniga *@Factory*, millega saab luua mitmeid isendeid (ingl.k. *instance*) samast testjuhtumi klassist. Andes parameetri *dataProvider* väärtuseks mõne andmeid tagastava meetodi, on nii võimalik anda parameetreid ka testjuhtumi klassile. Selleks, et neid parameetreid saaks testmeetodid kasutada, tuleb klassi muutujad algväärtustada sisendiks saadud parameetritega. [15]

Viimaseks suuremaks erinevuseks JUnit4 ja TestNG vahel on see, et JUnit4 hoiab testid üksteisest sõltumatud (kuigi on võimalik saavutada testide sõltuvus kasutades varem kirjeldatud JUnit4 eelduseid) [13]. TestNG’s saab testmeetodeid muuta sõltuvaks nii teistest testmeetoditest kui ka gruppidest. Sõltuvust on võimalik kirjeldada nii XML failis kui annotatsiooni atribuutidega. XML failis tuleb sõltuvused kirjutada märgendis *dependencies*. Annotatsiooni atribuutide juhul tuleb testmeetodi annotatsioonile *@Test* anda atribuudile *dependsOnMethods* või *dependsOnGroups* väärtuseks vastava meetodi või grupi nimi, millest antud meetod sõltub (vt. Joonis 13). [15]

```
@Test
public void firstTest() { … }

@Test(dependsOnMethods = { “firstTest” })
public void secondTest() { … }

@Test(dependsOnMethods = { “secondTest” }, alwaysRun = true)
public void lastTest() { … }
```

Joonis 13. TestNG testmeetodite sõltuvused.

Sealjuures eristatakse kahte liiki sõltuvusi - tugevaid ja pehmeid. Tugevate sõltuvuste (ingl.k. *hard dependencies*) puhul peab testmeetodi käivitamiseks kõik sõltuvad testmeetodid olema õnnestunud. Kui vähemalt üks sõltuv testmeetod ebaõnnestub, siis sellest testmeetodist sõltuv meetod jäetakse vahele. Pehmete sõltuvuste (ingl.k. *soft dependencies*) puhul käivitatakse testmeetod hoolimata sellest, kas sõltuvad testmeetodid õnnestusid või mitte. See on kasulik testmeetodite käivitamise järjekorra andmiseks. Pehme sõltuvuse kasutamiseks tuleb annotatsioonis @Test parameetritesse lisada "*alwaysRun=true*". [15]

5. Automaattestimise vahendi teostus

Selles peatükis kirjeldatakse automaattestimise vahendi arhitektuuri. Kirjeldatakse üldist struktuuri, põhjendades erinevate komponentide eesmärke ja kuidas on need kirjutatud. Lisades on märgitud rakenduse, selle kasutusjuhendi, lähtekoodi ja testandmete asukoht.

5.1 Üldine ülevaade

5.1.1 Sisend ja väljund

Automaattestimise vahend saab sisendiks 2 argumenti, millest esimene on sisendandmete kaust ja teine tudengi poolt esitatud testitava andmebaasi fail (laiendiga .db). Sisendandmete kaustas asub korrektselt loodud andmebaasi fail, testide kogumiku XML fail, tabelite jaoks mõeldud andmed ja sõltuvalt testide kogumikust SQL skripti failidest. Automaattestimise vahend annab väljundiks tudengi pool esitatud andmebaasi failiga samasse kausta testimistulemuste raporti HTML kujul.

5.1.2 Testide ettevalmistus

Enne testide käivitamist toimub ettevalmistus. Selle käigus luuakse ühendus nii testitava andmebaasi kui ka sisendandmete kaustas oleva korrektse andmebaasiga. Testandmete ettevalmistamiseks tehakse SQL päringuid testitaval andmebaasil, mille tulemused salvestatakse muutujatesse, testmeetodite hilisemaks ligipääsuks. Korrektse andmebaasi testandmete mitteleidumisel tehakse päringuid ka sinna, salvestades tulemused faili, mida saab kasutada ka järgnevatel testimistel. Peale edukat andmebaasidega ühenduse loomist ja testandmete töötlemist lõpeb testide ettevalmistus.

5.1.3 Testide käivitamine

Peale ettevalmistuse lõppu hakatakse järjest käivitama testjuhtumeid. Testjuhtumi sees käivitatakse kõik selles olevad testmeetodid vastavalt määratud sõltuvustega tekkivas järjekorras. Täpsemalt testide jaotusest ja nende tööst on kirjutatud järgnevates peatükkides. Peale testide

kogumiku kõigi testide käivitamist suletakse ettevalmistusel loodud ühendus andmebaasidega ja hakatakse looma testmistulemuste raportit.

5.1.4 Testimistulemuste raport

Testimistulemuste raport on HTML faili kujul. Raportisse kirjutatakse testitud testikogumiku nimi, millel järgnevad testikogumikus sisalduvad testide nimetused. Iga testi kohta väljastatakse õnnestunud, ebaõnnestunud ja vahele jäetud testmeetodite arv (vt. Joonis 14). Lisaks väljastatakse ebaõnnestunud testide puhul vastava meetodi nimetus ja ebaõnnestumise põhjus (vt. Joonis 15).

```
Suites run: 1
Suite>TaskTestSuite
   Test>MetaDataTests
      Failed>0
      Passed>240
      Skipped>0
   Test>DataTests
      Failed>0
      Passed>39
      Skipped>0
```

Joonis 14. Õnnestunud testikogumik.

```
Suites run: 1
Suite>TaskTestSuite
  Test>MetaDataTests
     Failed>2
        tableColumnParameterTest
           Baastabeli 'inimesed' veeru 'perenimi' tüüp on
           vale pikkusega
           Oodati '100' aga leiti '50'
        tableColumnParameterTest
           Baastabelis 'inimesed' puudub veerg 'eesnimi'
     Passed>238
     Skipped>0
```

Joonis 15. Ebaõnnestumistega testikogumik.

5.2 Testimisplaani kirjeldus

Testikogumiku testid on üldiselt jaotatud kahte liiki - andmebaasi süsteemivaadetest kättesaadavate metaandmete testimine ja andmebaasi tabelites olevate andmetega testimine.

5.2.1 Metaandmete testimine

Andmebaasi süsteemivaadetest on võimalik saada kätte kõik antud andmebaasis loodud objektide metaandmed. Näiteks andmebaasi ühe põhiliste objektide, tabelite, metaandmed asuvad SQL Anywhere andmebaasis süsteemivaates *SYS.SYSTABLE*. Süsteemivaadetes leidub nii testimiseks vajalikku kui ka testimiseks üleliigset informatsiooni. Samas pole ühes süsteemivaates olemas kõike. Selleks, et kõik testimiseks kasutatav informatsioon kätte saada,

tehakse *SELECT* päring, millega küsitakse mitme süsteemivaate veerud. Näiteks tabelite kohta tehakse päring tabeli nime, tabeli tüübi, tabeli veergude arvu ja tabelis olevate andmete ridade arvu kohta. Täpsemalt leiab testimiseks kasutatud metaandmetest ülevaate lisades.
Metaandmete testimisel võetakse aluseks korrektse andmebaasi metaandmed. Testimisega tehakse kindlaks, kas samad metaandmed leiduvad ka testitavas andmebaasis. Kui samu metaandmeid testitavas andmebaasis ei leidu, loetakse test ebaõnnestunuks. Seejärel jätkatakse järgneva testiga vastavalt sellele, kas järgnev test sõltub ebaõnnestunud testist või mitte. Täpsemalt metaandmetega testimise meetoditest on kirjutatud järgnevas peatükis.
Kõike süsteemivaadetes saadavaid andmeid pole aga võimalik testimiseks kasutada. Näiteks salvestatakse süsteemivaates vaate objekti kohta selle definitsioon (tabelite veergude pealt kirjutatud SELECT päringuna). Kuna vaateid on võimalik päringuna kirja panna mitmetel erinevatel viisidel, siis oleks vajalik testida vaate funktsionaalsust ja mitte selle definitsioonile vastavust. Seda saab teha kui testida tabeliandmetega.

5.2.2 Tabeliandmetega testimine

Tabeliandmetega testimine toimub peale metaandmete testimist. Testitakse neid objekte, mida pole võimalik testida üksnes metaandmetest saadava informatsiooniga. Nendeks objektideks on vaated, funktsioonid, protseduurid ja trigerid. Testimisel tehakse kindlaks, kas antud objekt käitub vastavalt nii, nagu on nõutud.

Tabeliandmetega testimisel kustutatakse testiandmetega kaasasoleva SQL skriptiga ettenähtud tabelites kirjed ja lisatakse nendesse uued kirjed. Seda tehakse vaid tudengi poolt esitatud andmebaasis, korrektses andmebaasis peaksid andmed juba eelnevalt sisestatud olema. Peale tudengi andmebaasi uute andmete lisamist toimub testimine, mille võib jaotatud kaheks osaks.

Algul testitatakse neid objekte, mis tabelites olevaid andmeid ei muuda. Üheks nendeks objektideks on näiteks vaated. Vaadete korrektse käitumise testimiseks tehakse käitusaegselt samu SQL päringuid vaadete kohta mõlemasse andmebaasi. Seejärel kontrollitakse, kas testitava andmebaasi päringu väljund vastab korrektse andmebaasi tulemusele. Test loetakse ebaõnnestunuks, kui väljundid ei kattu ehk testitava andmebaasi vaade käitub teisiti kui nõutud. Sarnaselt testitakse ka funktsioone ja protseduure, mis tabelite andmeid ei muuda.

Seejärel testitakse neid objekte, mis tabelites olevaid andmeid muudavad. Nendeks objektideks on tabelites muudatusi tegevad protseduurid ja trigerid. Nende testide ettevalmistuseks kustutatakse testitavas andmebaasis olevate tabelite andmed. Testmeetodites lisatakse tabelitesse andmed, kutsutatakse välja testitav protseduur/triger ja seejärel võrreldakse oodatud muudatusi tegelike muudatustega. Kui tegelikud muudatused erinevad oodatud muudatustest, loetakse test ebaõnnestunuks.

5.3 Koodianalüüs

Testimisvahendi arendamiseks on kasutatud TestNG raamistikku. Algul valitud JUnit 4 raamistik osutus mittepiisavaks, kuna sellel puudub lihtne võimalus määrata testide järjekorda ja sõltuvusi. Lisaks pole parameetritega testide koostamine vajalikul viisil võimalik.

Testid on jaotud vastavalt andmebaasi objektidele eraldi testjuhtumiteks. Näitena kasutatud tabeli testjuhtumi klass on lisades.

Kasutades TestNG võimalust luua samu testjuhtumi isendeid erinevate parameetridega antakse näiteks tabeli metaandmeid testivasse klassi järjest tabeli kohta käivaid parameetreid. Need parameetrid salvestatakse konstruktoris klassimuutujatesse, testmeetodites kasutamiseks. Tabeli puhul testitakse esimesena seda, et kas see tabel üldse testitavas andmebaasis leidub. Kuna ülejäänud testmeetodid sõltuvad selle testist, siis tabeli mitteleidumisel jäetakse teised testmeetodid vahele ja liigutakse testimisega järgmiste testjuhtumite juurde. Tabeli leidumisel testitakse tabeli ülejäänud parameetreid (tabeli tüüp, tabeli veergude arv ja tabelis olevate andmete ridade arv). Kusjuures tabelis olevate andmete ridade testimisel ei testita täpset ridade arvu vaid seda, et tabelid poleksid andmetest tühjad (ridade arv on suurem kui 0). Samuti pole ülejäänud parameetrite testid kuidagi üksteisest sõltuvad.

Kasutades TestNG võimalust anda testmeetoditele erinevaid parameetreid, toimub tabeli testimisel ka vastava tabeli veergude testimine. Testmeetod saab järjest sisendiks lisaks tabeli nimele ka testitava tabeli veergude parameetrid. Selles testmeetodis testitakse kõigepealt veeru olemasolu antud tabelis ja seejärel sõltuvalt veeru olemasolust ülejäänud veeru parameetreid (veeru andmetüüp, andmetüübi pikkus, NULL väärtuse lubamine ja algväärtus).

Oodatava ja tegeliku tulemuse võrdlemiseks kasutatakse TestNG väiteid. Testi ebaõnnestumisel antakse väite argumendiga *message* kaasa ebaõnnestumise põhjus, mis kajastub hiljem testi raportis.

5.4 Ülesannete püstituse nõuete analüüs

Ülesande püstitusel sai eesmärgiks seatud neli nõuet, mida antud testimisvahend peaks täitma. Esimeseks nõudeks oli see, et testimisvahendiga oleks võimalik testida viies erinevas järgus olevat andmebaasi. Antud teostusega on see täiesti saavutatav. Viies erinevas järgus andmebaasi testimiseks läheb vaja erinevaid testandmeid koos viie korrektse andmebaasiga, mis vastavad ülesannete nõuetele.

Teiseks nõudeks oli see, et eiratakse väiksemaid vigu, näiteks väikeseid erinevusi nimetamisel või järjekorras. Antud teostuses ignoreeritakse erinevusi järjekorras, välja arvatud kohtades kus seda teha ei saa, näiteks protseduuri parameetrite järjekord. Nimetuste kontrollimisel ignoreeritakse väike- ja suurtähti, kuid ei eirata grammatikavigu. Väiksemate grammatikavigade ignoreerimiseks saaks kasutada Levenshteini distantsi [16] algoritmi, mille abil on võimalik arv, mitme muudatuse kaugusel on üks järjend teisest. Selle algoritmiga saaks lugeda õigeks korrektsest sõnast mingi kindla muudatuse kaugusel olevaid sõnu.

Kolmandaks nõudeks oli see, et testide kohta antakse vastavat tagasisidet. Antud teostuses toimub tagasiside testi raporti kaudu, kuhu kirjutatakse läbitud, õnnestunud, ebaõnnestunud ja vahele jäetud testide arv. Ebaõnnestunud testidel on lisaks ebaõnnestumise põhjus.

Neljandaks oli nõutud, et nõuete muutumisel oleks võimalik testimisvahend viia vastavusse nende muudatusega. Antud teostus ei täida seda nõuet täielikult. Kui näiteks tabelite testid genereeritakse dünaamiliselt, vastavalt sisendiks saadud testandmetele, siis kõikide protseduuride ja trigerite puhul seda teha pole võimalik. Seda seetõttu, et protseduuride ja trigeritega on võimalik muuta mitmeid objekte ja kõigi objektide muudatuste võrdlemine oleks liialt mahukas. Protseduuride ja trigerite testimiseks on loodud eraldi testjuhtumid, mis on antud nõutele spetsiifilised. Selleks, et siiski testimisvahend viia vastavusse, tuleb teha vajalikke muudatusi vastavates testjuhtumite klassides.

5.5 Edasine arendus

Testimisvahendi edasiseks arenduseks on mitmeid mõtteid ja muudatusi.

Metaandmete kättesaamiseks on hetkel kasutatud süsteemivaateid, mis on SQL Anywhere versioonist 12, kuid versioonis 16 on need märgitud iganenuks (ingl.k. *deprecated*) ja on mittesoovitatud kasutada. Tuleks kasutusele võtta versioonis 16 olevad süsteemivaated.

Testimise käigus tühjendatakse testitav andmebaas andmetest ja peale testide lõppu lisatakse uuesti testandmed. Selle asemel tuleks testitava andmebaasi tabelite andmed eksportida ajutisse faili, et peale testimise lõppu need andmed uuesti tabelitesse sisestada.

Hetkel testitakse andmebaase vastavust nõuetele ja väljundiks tuleb testi tulemuste raport. Kursuse raames oleks vajalik lisada testide juurde ka hindamisüsteem, mis näiteks vastavalt määratud ülesannete kaaludele annab lisaks tulemuseks punktisumma.

6. Kokkuvõte

Selle bakalaureusetöö tulemusena loodi Java automaattestimise vahend, mis lihtsustab kursuse "Andmebaasid" käigus loodava andmebaasi kontrollimist. Programm lubab õppejõul testida tudengite andmebaaside vastavust etteantud ülesannete nõuetele.

Bakalaureusetöö esimeses peatükis kirjeldati valdkonna tausta ja ülesande püstitust. Sealjuures toodi välja automaattestimise vahendi loomise nõuded. Teises peatükis kirjeldati testimist ja võrreldi erinevaid testimistehnikaid, toodi välja andmebaasiga rakenduste testimisega seotud probleeme ja kirjeldati automaattestimist. Kolmandas peatükis tutvustati ja võrreldi omavahel testimisraamistiku JUnit versioone 3 ja 4 ja automaattestimise vahendi loomiseks kasutatud raamistikku TestNG. Neljandas peatükis kirjeldati testimisplaani ja programmi arhitektuuri, analüüsiti programmi koodi ja programmi vastavust püsitatud nõuetele, toodi välja ka viisid edasiseks arenduseks.

Püstitatud eesmärgid said täidetud. Esialgu oli plaanis luua vahend, mis testib konkreetseid ülesandeid. Töö käigus ilmnes, et ülesannete muudatuse korral tähendaks see mitmete testmeetodite ümberkirjutamist ja tekkis vajadus genereerida testmeetodeid automaatselt. Valminud programm testib sisendandmetega ja ülesannete muudatuste korral on võimalik lihtsasti viia testid nendega vastuvusse.

Töö tegemine pakkus võimalust testimisplaani korduvalt ümber mõelda, proovides nii läbi mitmeid erinevaid lahendusi. Kuna sarnased testimisvahendid puudusid, siis tuli probleemile läheneda loovalt ja valminud tulemusega võib rahule jääda.

7. Viited

[1] Andmebaasid MTAT.03.105 Sissejuhatav praktikum. https://courses.cs.ut.ee/MTAT.03.264/2015_spring/uploads/Main/Materjalid%20praktikumi%20toetuseks (14.05.2015)

[2] Andmebaaside MTAT.03.264 ajakava. https://courses.cs.ut.ee/MTAT.03.264/2015_spring/uploads/Main/PIJUHIS.pdf (14.05.2015)

[3] Mikhail, R.F., Berndt, D., Kandel, A..”Automated Database Application Testing”, Series in Machine Perception and Artificial Intelligence, vol. 76. World Scientific, Singapore, 2010.

[4] Kai Pan , Xintao Wu , Tao Xie. “Generating program inputs for database application testing”, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, p.73-82, 2011.

[5] Haller, Klaus. “White-box testing for database-driven applications: a requirements analysis” In Proceedings of the Second International Workshop on Testing Database Systems, (DBTest '09), 2009.

[6] M. Y. Chan and S. C. Cheung, “Testing database applications with SQL semantics”. In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications, Springer, p. 363–374, 1999.

[7] H. Reza and K. Zarns. “Testing Relational Database Using SQLLint”. ;In Proceedings of ITNG, p. 608-613, 2011.

[8] W. Xu, D. Huang. “Automated testing for database system”, in: International Conference on Biomedical Engineering and Computer Science, p. 1-4. 2010.

[9] JUnit homepage http://junit.org/ (14.05.2015)

[10] The Takipi blog
http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/ (14.05.2015)

[11] P. Tahchiev, F. Leme, V. Massol, G. Gregory. JUnit in Action, Second Edition. Manning Publications Co. Greenwich, CT, USA, 2010.

[12] TestNG homepage
http://testng.org/doc/index.html (14.05.2015)

[13] JUnit4 Vs TestNG - Comparison
http://www.mkyong.com/unittest/junit-4-vs-testng-comparison/ (14.05.2015)

[14] JUnit Github Summary of Changes in version 4.8
https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.8.md (14.05.2015)

[15] TestNG documentation
http://testng.org/doc/documentation-main.html (14.05.2015)

[16] Levenshtein distance
http://rosettacode.org/wiki/Levenshtein_distance (14.05.2015)

Lisad

I. Annotatsioonide võrdlus

Annotatsiooni kirjeldus	JUnit 4	TestNG
Testmeetodi annotatsioon	*@Test*	*@Test*
Käivitada enne testide kogumiku kõiki teste	-	*@BeforeSuite*
Käivitada peale testide kogumiku kõiki teste	-	*@AfterSuite*
Käivitada enne testi	-	*@BeforeTest*
Käivitada peale testi	-	*@AfterTest*
Käivitada enne esimest gruppi kuuluva testmeetodi väljakutset	-	*@BeforeGroups*
Käivitada peale viimast gruppi kuuluvat testmeetodi väljakutset	-	*@AfterGroups*
Käivitada enne esimest klassi kuuluva testmeetodi väljakutset	*@BeforeClass*	*@BeforeClass*
Käivitada peale viimast klassi kuuluva testmeetodi väljakutset	*@AfterClass*	*@AfterClass*
Käivitada enne igat testmeetodit	*@Before*	*@Before*
Käivitada peale igat testmeetodit	*@After*	*@After*
Ignoreeri testmeetod	*@Ignore*	*@Test(enable=false)*
Oodatav erindi	*@Test(expected = ArithmeticException.class)*	*@Test(expectedExceptions = ArithmeticException.class)*

Testmeetodi ajalõpp	*@Test(timeout=1000)*	*@Test(timeout=1000)*

II. Rakendus, selle kasutusjuhend, lähtekood ja testandmed

Rakendus, selle kasutusjuhend, lähtekood ja testandmed on kättesaadav failis testingtool.zip.

III. Metaandmete tabel

Andmebaasi objekt	Objekti metaandmed	Selgitus
Tabel	table_name	Tabeli nimi
	table_type	Tabeli tüüp (baas, ajutine, vaade)
	column_count	Tabelis olevate veergude arv
	row_count	Tabelis olevate andmete rida arv
Veerg	table_name	Veeru tabeli nimi
	column_name	Veeru nimi
	column_type	Veeru andmetüüp
	column_type_length	Veeru andmetüübi pikkus
	nullable	NULL väärtuse lubamine
	default_value	Veeru vaikeväärtus
Primaarvõti	primary_key_name	Primaarvõtme nimi
	table_name	Primaarvõtme tabel nimi
	column_names	Primaarvõtme veergude nimed
Välisvõti	foreign_key_name	Välisvõtme nimi
	foreign_table_name	Alamtabeli nimi
	primary_table_name	Ülemtabeli nimi
	columns	Veergude nimed
	event	Sündmus

	action	Tegevus
Unikaalne võti	unique_key_name	Unikaalse võtme nimi
	table_name	Unikaalse võtme tabeli nimi
	column_names	Unikaalse võtme veeru nimed
CHECK kitsendus	check_definition	CHECK kitsenduse definitioon
	table_name	CHECK kitsenduse tabeli nimi
	column_name	CHECK kitsenduse veeru nimi
Protseduur	procedure_name	Protseduuri nimi
	parameter_name	Protseduuri parameetri nimi
	parameter_index	Protseduuri parameetri järjekorra number
	parameter_mode	Protseduuri parameetri tüüp (IN,OUT,INOUT)
	parameter_type	Protseduuri parameetri andmetüüp
	parameter_type_length	Protseduuri parameetri andmetüübi pikkus
Indeks	index_name	Indeksi nimi
	table_name	Indeksi tabeli nimi
	column_names	Indeksi veergude nimed koos suunaga (ASC,DESC)
Triger	trigger_name	Trigeri nimi
	table_name	Trigeri tabeli nimi
	event	Trigeri sündmus (INSERT,UPDATE,DELETE)
	trigger_time	Trigeri aeg (BEFORE/AFTER)

IV. Tabeli testjuhtum

```
public class TableTest extends BaseTest {
  private String tableName = null;
  private String tableType = null;
  private String columnCount = null;
  private SoftAssert softAssert = null;
  public TableTest(String tableName, String tableType, String columnCount, String rowCount) {
    this.tableName = tableName;
    this.tableType = tableType;
    this.columnCount = columnCount;
  }
  @Test
  private void tableExistsTest(ITestContext context) {
    context.setAttribute("tableName", tableName);
    boolean condition = TableTestMethods.existsTable(tableName);
    String message = TableTestFeedback.tableMissingMessage(tableName,tableType);
    assertTrue(message,condition);
  }
  @Test(dependsOnMethods = {"tableExistsTest"})
  private void tableParameterTest() {
    softAssert = new SoftAssert();
    tableTypeTest(tableName, tableType);
    tableColumnCountTest(tableName, tableType, columnCount);
    tableRowExistsTest(tableName, tableType);
    softAssert.assertAll();
  }
 @Test(dependsOnMethods={"tableExistsTest"},
        dataProviderClass=TestDataProvider.class,dataProvider="columnDataProvider")
  private void tableColumnParameterTest(String tableName,String columnName,String columnType,
                                        String typeLength,String nullable,String defaultValue) {
    softAssert = new SoftAssert();
    tableColumnExistsTest(tableName, tableType, columnName);
    tableColumnTypeTest(tableName, tableType, columnName,columnType);
    tableColumnTypeLengthTest(tableName, tableType, columnName, typeLength);
    tableColumnIsNullableTest(tableName, tableType, columnName, nullable);
    tableColumnDefaultValueTest(tableName, tableType, columnName, defaultValue);
    softAssert.assertAll();
  }
```

V. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina **Marten Siiber** (sünnikuupäev: 08.06.1993)
(*autori nimi*)
1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
Aine "Andmebaasid" automaattestimise vahend,
(*lõputöö pealkiri*)
mille juhendaja on Vambola Leping,
(*juhendaja nimi*)
1.1.reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
1.2.üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **14.05.2015**