TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Arvutitehnika instituut

IA40LT

Markus Põldmäe 142212IASB

AUTOMAATTESTIDE KIRJUTAMINE RUBY PROGRAMMEERIMISKEELES

Bakalaureusetöö

Juhendaja: Vladimir Viies

PhD

Allan Lahe

BSc

Tallinn 2017

Autorideklaratsioon

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

Autor: Markus Põldmäe

BAKALAUREUSETÖÖ ÜLESANNE

Üliõpilane: Markus Põldmäe

Üliõpilaskood: 142212

Lõputöö teema eesti keeles:

Automaattestide kirjutamine Ruby programmeerimiskeeles

Lõputöö teema inglise keeles:

Creating automated tests in Ruby programming language

Juhendaja (nimi, töökoht, teaduslik kraad, allkiri):

Vladimir Viies, Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, dotsent

Konsultandid: Allan Lahe(Mentor)

Lahendatavad küsimused ning lähtetingimused:

Ülevaade automaattestimisest kui tööriistast tarkvara arenduses, võrdlus automaat- ja manuaaltestide vahel. Ülevaade erinevatest automaattestimise tööriistadest.

Automaattestide loomine KE(kindla ettevõtte) osakonnale, kus Ruby programmeerimiskeel on valitud eelneva analüüsi põhjal. Lisaks tehakse automaattestide ülesehituse analüüs ning võrreldakse automaat- ja manuaaltestide ajalist vahet.

Eritingimused:

Nõuded vormistamisele: Vastavalt Infotehnoloogia teaduskonnas kehtivatele nõuetele

Lõputöö esitamise tähtaeg: 22.05.2017

Ülesande vastu võtnud: _____ kuupäeva: 22.02.2017
(lõpetaja allkiri)

Annotatsioon

Antud töö eesmärgiks on luua Ruby programmeerimiskeeles automaattestid KE osakonnale. Töös antakse ülevaade ka teistest automaattestide tööriistadest, mis on mõeldud veebikeskkonna, kasutajakeskkonna ja ühiktestide jaoks. Automaattestide loomine on KE kindla osakonna jaoks vajalik, sest kiirendab oluliselt arendusprotsessi ning annab võimaluse tekkinud vigu kiiremini avastada. Töö annab ülevaate ka loodud automaattestide tähtsusest ning välja tuuakse erinevused automaattestide ja manuaaltestide vahel. Sageli kiputakse automatiseerimisega üle pingutama ning ei osata hinnata automaattestide arendusele kuluvat ajalist osa.

Manuaal- ja automaattestide võrdluse juures tuuakse välja erinevad tingimused, millal peaks valima manuaaltestimist ning millal peaks automaattestimist. Antud töös on keskendutud regressioonitestidele, mida kasutatakse kindla ettevõtte jaoks loodavates automaattestides, samas on autor teadlik ka teistest testimise liikidest. Töös tuuakse erinevaid näiteid olukordadest, kus on automaattestimine oluliselt lihtsustanud testijate tööd. Ülevaade antakse ka testija rollist tänapäeval ning selle muutumisest aja jooksul. Lisaks jagatakse soovitusi õige testimise tüübi valimisel ja pakutakse välja automatiseerimise tööriista valikuks ette kõige populaarsemad automatiseerimise tööriistad.

Töö tulemusena luuakse KE kindlale osakonnale toimivad automaattestid, mis katavad andmebaaside vahelist XML tüüpi suhtlust ja kiirendavad seetõttu ka testimist. Töö lõpuks tuuakse välja ka manuaaltestide ning automaattestide ajaline erinevus konkreetse andmebaaside vahelise XML suhtluse kontrollimisel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 29 leheküljel, 23 peatükki, 12 joonist ja 1 tabelit.

Abstract

Creating automated tests in Ruby programming language

Test automation is becoming increasingly popular in every enterprise, which specializes in software development. Using automated tests saves a lot of time, which would be spent testing software manually. Automated tests replace standard and repeated testing done by humans. That is why automated tests help reduce the amount of testers and monetary cost spent on testing. Automated tests also boost development process - faster feedback for development quality. To use automated tests succesfully, they should cover most appearing test cases.

The thesis is divided into three bigger paragraphs. First paragraph gives an overview of automated tests as a tool that has been used for a while and what is the situation of automated tests nowadays. Pros and cons of automated tests are highlighted. The second paragraph is introduction to creating automated tests in specific big financial enterprise and the requirements for automated tests. A short overview will be given about the specific department and the position of created automated tests in development process. There will be also an evaluation about created automated tests future and new requirements for additional development.The third paragraph is concentrates on the practical part of test creation. This paragraph includes code examples of created automated tests and description how the tests work. The outcome of tests and testing user view are also described in third paragraph.

The aim of this thesis is creating a working automated testing system using big financial enterprise as an example and validating created tests while presenting outcome. As an extra to practical part there is also an overview about test automation benefits and different use cases for given big financial enterprise.

The thesis is in estonian and contains 29 pages of text, 23 chapters, 12 figures, 1 table.

Lühendite ja mõistete sõnastik

API - Application Programming Interface ehk rakendusliides. Reeglistik olemasoleva programmiga suhtlemiseks.

SOAP – Simple Object Access Protocol ehk lihtne objektipöördusprotokoll. Protokoll, millega veebiteenused vahetavad omavahel struktuurseid andmeid.

REST - Representational State Transfer ehk tarkvaraarhitektuuri stiil, mida kasutatakse hüpermeedia hajussüsteemide valdkonnas nagu näiteks veeb.

XML - eXtensible Markup Language. Andmete transportimise ja hoiustamise tekstiformaat, mis on loetav masina jaoks, kuid ka loetav inimese poolt.

XSD - XML Schema Definition. Fail, mis määrab ära kindla XML-i struktuuri, täpsustades elemendid, mis on oodatud XML failis.

teek - Eraldi paiknev koodikogum, mida saab kaasata kirjutatud koodi, et kasutada teegi funktsioone.

IDE - Integrated Development Environment. Integreeritud programmeerimiskeskkond, mis pakub tuge erinevatele programmeerimiskeeltele ning kontrollib süntaksit.

PL/SQL - Procedural Language/Structured Query Language. Protseduurne programmeerimiskeel, mis ühendab endaga SQL päringukeele.

Ruby - objekt-orienteeritud programmeerimiskeel.

RSpec - Ruby programmeerimiskeele jaoks loodud testraamistik.

LoC - Lines of Code. Allikkoodi read.

KE - Kindel ettevõte. Rahandussektoris tegelev suurettevõte.

virtuaalmasin – tarkvara ja/või seadme süsteem, mis emuleerib riistvara platvormi. Arvuti teise arvuti sees.

hash - Programmeerimises kasutatav võtmete ning nendele vastavate väärtuste kollektsioon.

proc - Ruby-s kasutatavad objektid sisemiste muutujatega, mida on võimalik välja kutsuda erinevates kontekstides, kasutades samu sisemisi muutujaid.

lambda - Ruby-s kasutatav objekt, mis on väga sarnane proc tüüpi objektiga, kuid samuti kontrollib sisendargumentide arvu, mida proc ei tee. Ehk tõstab esile ka vigu.

wrapper – Koodilõik, mis sisaldab endas teist programmeerimiskeelt, et originaalkoodis käivitada teist koodi.

Sisukord

Sissejuhatus 10
1. Automaattestimine 11
1.1 Automaattestimise erinevad tööriistad 12
1.2 Automaattestide eelised 17
1.3 Automaattestide puudused 19
1.3 Automaattestide tulevik 21
2. Automaattestide seostamine valdkonnaga 23
2.1 Ülesande püstitus 23
2.2 Raamistik 24
2.3 Nõuded automaattestidele 25
2.4 Dokumentatsioon 25
2.5 Keskkonnad ja kasutatavad programmid 26
2.6 Automaattestide tulemused 27
2.7 Edasiarendus 28
2.8 Ühiktestide valik 29
3. Automaattestide loomine KE-le 30
3.1 Automaattesti töö failide lugemisel ja salvestamisel 30
3.2 Ruby keele süntaks automaattestimisel 32
3.3 XML failistruktuur 33
3.4 RSpec raamistik 35
3.5 Suhtlus andmebaasiga 36
3.6 Testide käivitamine ja tulemused 36
3.7 Testide valideerimine 39
Kokkuvõte 40
Kasutatud kirjandus 41
Lisa 1 – automaattesti failide sisselugemise kood 42
Lisa 2 – XML väljade sisu kontrolli testikomplekti kood 43
Lisa 3 – andmebaasist vastuse saamise kood 43
Lisa 4 – automaattesti võrdlusfunktsioonide kood 44

Jooniste loetelu

Joonis 1 Testide automatiseerimise tsükkel 12
Joonis 2 Manuaal- ja automaattestide testimise maksumuse võrdlus ajas [6] 18
Joonis 3 XML suhtlus kahe süsteemi vahel 23
Joonis 4 Automaattestide arendamiseks vajalik keskkond 27
Joonis 5 Automaatiseeritav osa testimises 30
Joonis 6 Testi loomise tegevuste diagramm 31
Joonis 7 Kontrollkoodi näide 33
Joonis 8 XSD valideerimise kood 34
Joonis 9 Elementide arvu esinemise kontrolli kood 35
Joonis 10 Testide automaatse käivituse keskkond 37
Joonis 11 Jenkins keskkonna kasutaja vaade 38
Joonis 12 KE testide XML võrdluse HTML tulemus 38

Tabelite loetelu

Tabel 1 Automaattestimise tööriistad 16

Sissejuhatus

Automaattestimine on tänapäeval saanud väga populaarseks igas ettevõttes, mis tegeleb tarkvara arendamisega. Automaatestimise abil on võimalik hoida kokku suures koguses aega, mis kuluks tarkvara manuaalseks testimiseks. Automaatestid asendavad inimeste poolt tehtavat standartset ja korduvat testimist - tekib võimalus vähendada testijate arvu ning rahalist kulu, mis testimisele kulub. Lisaks aitavad automaattestid kiirendada arendusprotsessi - saadakse kiirem tagasiside arenduse kvaliteedile. Automaattestide edukaks kasutamiseks peavad need siiski katma ära enim esinevad olukorrad.

Töö on jaotatud kolmeks suuremaks peatükiks. Esimene peatükk annab ülevaate automaattestidest kui mõnda aega kasutusel olevast töövahendist, samuti nende olukorrast tänapäeval. Töös tuuakse välja nende eelised ja puudused võrreldes manuaaltestimisega. Teise peatükki eesmärgiks on tutvustada automaattestimist kindlas valdkonnas, mille jaoks automaattestid luuakse. Antakse lühiülevaade valdkonna tööst ning kirjeldatakse loodavate testide osa arendusprotsessis. Lõpetuseks hinnatakse loodud automaattestide tulevikku, tuues välja edasise arenduse nõuded ning nende täitmiseks vajaminevad lahendused. Kolmas peatükk keskendub rohkem automaattestide loomise praktilisele osale. Peatükis kirjeldatakse testide loomiseks kasutatavaid koodilõike ning antakse ülevaade testide toimimisest ja arendusprotsessist. Testide tulemused ning kasutajavaade on samuti selles peatükis lahti seletatud.

Töö eesmärk on töötava automaattestimise süsteemi loomine KE näitel ja loodud testide valideerimine ning tulemuste esitamine. Lisaks praktilisele osale on eesmärgiks anda üldine ülevaade automaattestimise kasulikkusest ning erinevatest kasutusvõimalustest ettevõtte näitel.

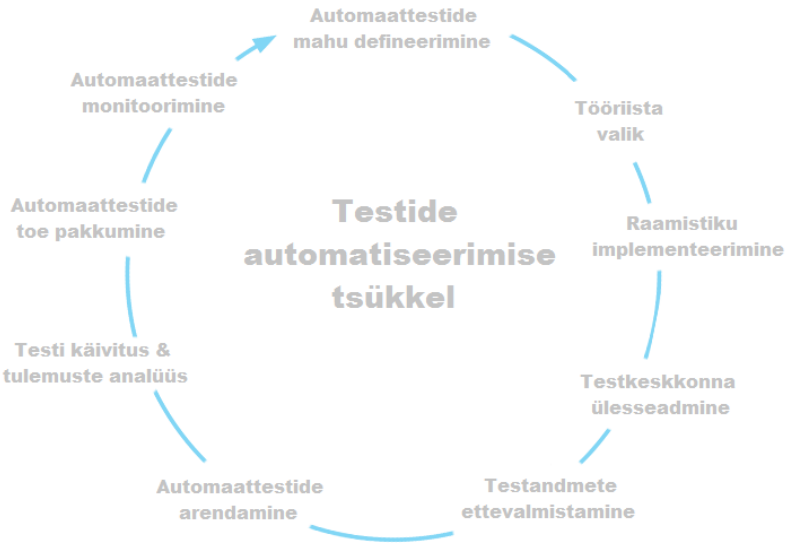
1. Automaattestimine

Tänapäeval on raske leida ettevõtet, kus ei kasutataks igapäeva töös erinevaid arvutiprogramme. Sellest tulenevalt on järjest suurem ootus ja nõue, et tööks vajalikud programmid funktsioneeriksid ilma vigadeta. Programmivigade avastamine võimalikult varajases arendusfaasis lubab vältida hilisemaid tõrkeid - sellele aitab kaasa testimine.

Programmide testimiseks on erinevaid võimalusi - manuaalne või automaatne.

Manuaaltestimine hõlmab endas testijat ning testitavat programmi. Manuaaltestimist tasub kasutada, kui testi ettevalmistamiseks on vähe aega või sisaldab test endas ainult mõningaid testimisringe. Manuaaltestimist kasutatakse enamjaolt testide juures kus toimub dünaamiliste lahenduste testimine ning ei teki palju kordusi [1]. Manuaaltestimine on ka siis hea valik, kui testimine toimub süsteemis, millele ei tule lisaarendusi.

Automaattestimine hõlmab endas koodilõiku ning testitavat programmi. Automaatsed testid erinevad manuaalsetest selle poolest, et automaatsete testide puhul tuleb kirjutada ka testimist teostav kood. Testide automatiseerimine on aeganõudev tegevus, kuid see tasub ära, kui testimisfaasis on tegevusi, mida manuaaltestija peaks kordama. Automaatteste tuleks kasutada siis, kui on tegemist suure infosüsteemiga, mille käsitsi testimine võib tekitada arenduses nn “pudelikaela” [1]. Järjest enam arendatakse automaatteste juba arenduse käigus selleks, et vältida hilisemat täiendavat arendusvajadust, samuti on võimalus saada kiiremini tagasisidet programmeeritud funktsionaalsuse kohta(Joonis 1).



Joonis 1 Testide automatiseerimise tsükkel

Automaattest on tarkvaratestimises kasutatav tarkvara, mis ei ole ühenduses testitava tarkvaraga. Eritarkvara ülesandeks on kontrollida testide käivitamist ning võrrelda nende tulemusi oodatud tulemustega [2]. Automaattestide abil on võimalik automatiseerida kindla formaadiga korduvaid teste või kontrollida süsteemisisest suhtlust, mida testijal oleks manuaalselt keeruline kontrollida. Automaattestid luuakse üldiselt kindla formaadiga manuaaltestide asemele, et kiirendada testimist. Automaattestimine on eriti vajalik agiilse tarkvara arendusprotsessi puhul, kus uue toodangu väljalaskmine toimub tihedamini.

1.1 Automaattestimise erinevad tööriistad

Tänapäeval on olemas suur hulk erinevaid tööriistu tarkvara automaatseks testimiseks. Automaattestimine on laienenud erinevatesse sektorisse, mis aitavad kaasa tarkvara arenduse kiiremale protsessile. Automaatteste on võimalik rakendada veebikeskkonnas, kasutajaliideses ning ka ühiktestimises(Tabel 1) [3].

Veebikeskkonna automaattestimine on hea viis kindlustada, et uus arendatud tarkvara versioon ei kaasa endas vigu. Veebirakenduste testimise automatiseerimine lubab arendusmeeskonnale kindlamat pinda muudatuste tegemisel ning kiiremat rakenduse funktsionaalsuse vigade tuvastamist peale igat muudatust. Muidugi on olemas keerulisi punkte veebirakenduste automaattestimises, sest on palju muutuvaid faktoreid, mis segavad testide kirjutamist. Näiteks võib esineda veebilehitsejast tingitud

kokkusobimatusi, mis on vältimatud, sest tuleb tagada tugi erinevatele veebilehitsejatele. Seetõttu ongi lai valik tööriistu, mis lihtsustavad automaattestide ühildumist erinevate veebiplatvormidega. Kuulsamad veebikeskkonna testimise tööriistad on alljärgnevad:

- **Selenium** - Populaarne automatiseeritud veebi testimise tööriist, mis abistab veebibrauserite automatiseerimisega erinevatel platvormidel. Seleniumil on mitme suurima veebibrauseri tugi, kes teevad koostööd Seleniumiga, et kaasata see enda brauserisse.
- **Watir** - Kooslus mitmest Ruby teegist veebibrauserite automatiseerimiseks, mille abil on võimalik kirjutada teste, mis on lihtsalt loetavad ning hooldatavad. Watir käitub veebibrauseritega samamoodi, nagu teeks seda inimkasutaja(vajutab linkidele, täidab vorme, vajutab nuppe jne). Samuti on võimalik jälgida, kas oodatud tekst ilmub lehel.
- **Windmill** - Veebitestimise tööriist, mis on loodud eesmärgiga aidata testijatel automatiseerida ning siluda veebirakendusi. Sellega kaasneb ka salvestusformaat, mille abil on võimalik salvestatud testi kasutada mitmes erinevas brauseris.
- **Ranorex** - Võimaldab veebirakenduse testimise automatiseerimist ning iminteraktsiooni salvestamist ja tagasimängimist testide jooksutamiseks. Ranorex on kõige populaarsem tööriist automatiseeritud testide loomiseks veebirakenduste ja kasutajaliideste jaoks.
- **SoapUI** - Mitme platvormi testimise tööriist, mis on loodud spetsifiilise funktsiooni täitmiseks. Selle ideeks on automaatselt testida API-sid (Application Programming Interface) nagu SOAP (Simple Object Access Protocol) ja REST (Represetional Stare Transfer)ning kindlustada koostalitlusvõime mitme erineva rakenduse vahel.
- **Sahi** - Tööriist veebirakenduste automaattestimiseks. Olemas nii avatud lähtekoodiga versioonis kui ka kommertsversioonis. Jääb oma funktsionaalsusega alla eelnevatele tööriistadele.
- **Tellurium** - Tööriist veebirakenduste testimiseks, mis võimaldab testide kirjutamist inglise keeles, ilma, et peaks kasutama mõnda skriptimiskeelt või omama programmeerimisega seotud teadmisi.

Kasutajakeskkonna testimine on samuti suures osas sõltuv operatsioonisüsteemist. Iga väiksem muudatus testitavas rakenduses võib lõhkuda automaattestid. Kasutajakeskkonna testimiseks on tänapäeval suur valik tööriistu nagu:

- **Squish** - Kasutajaliidese testimise tööriist mitme erineva platvormi jaoks. Squishi abil saavad testijad kirjutada automaatteste kasutades neile tuttavaid skriptimiskeeli nagu näiteks JavaScript, Perl, Python või Ruby
- **Ranorex** - Võimaldab arvutiprogrammide automaatset testimist ning kasutaja tegevuste salvestamist automaattestide tarbeks.
- **TestComplete** - Automatiseerimistööriist Windowsi platvormile, mis lubab salvestada, skriptida ning jooksutada kasutajaliidese teste rakenduste jaoks, mis on kirjutatud kasutades erinevaid raamistikke ja programmeerimiskeeli nagu .NET või C++
- **Test Studio** - Automatiseeritud funktsionaalsuse ja koormuse testimise tööriist, mis abistab rakenduste testimist mitmel erineva raamistiku ning programmeerimiskeelega platvormil.
- **eggPlant** - Loodud professionaalsete tarkvara rakenduste ja ettevõtete tiimide jaoks. Võimaldab erinevate rakenduste automatiseerimist nagu näiteks .NET, Java ning Flash rakendused.

Ühiktestimine on automaattestimises kõige detailsem tase, mis on loodud raamistikena olemasolevatele programmeerimiskeeltele. Ühiktestimise raamistikke saavad programmeerijad kasutada kindlate teekide ning rakenduste funktsionaalsuse testimiseks. Ühiktestide abil saab kontrollida suhtlusi rakenduste vahel, failide loomist, võrrelda faile ning muud. Ühikteste kasutatakse ka automatiseeritud arendussüsteemide kontrollimisel, näiteks on võimalik kontrollida andmebaasi täitmiseks genereeritud andmete sisu õigsust[3]. Tuntumad ühiktestimise raamistikud on:

- **NUnit** - Ühiktestimise raamistik kõikide .NET tüüpi keelte jaoks.
- **xUnit.net** - Ühiktestimise tööriist .NET raamistiku jaoks, mis on kirjutatud NUnit-i looja poolt. Tegemist on kõige uuema tehnoloogiaga, mille abil toimuvad ühiktestimised, C#, F#, VB.net ning teistes .NET tüüpi keeltes.
- **PyUnit** - Python-ile orienteeritud ühiktestimise raamistik. Osa Python-i raamistikust, mis toetab testide automatiseerimist.

- **JUnit** - Lihtne ühiktestimise raamistik korduvate testide kirjutamiseks Java-s. Üks populaarsemaid testimise raamistike Java arendajate jaoks.
- **TestNG** - Loodud JUnit-i ja NUnit-i põhjal, mis võimaldab rohkem funktsionaalsusi ning lihtsamat kasutust. Katab mitmeid erinevaid testi kategooriaid nagu ühiktestid, funktsionaalsustestid, täielikud testid ning integratsiooni testid.
- **PHPUnit** - Populaarne raamistik ühiktestimiseks PHP projektides. Pakub raamistikku ning lihtsasti loodavaid teste. Samuti on olemas funktsionaalsus testide ja tulemuste jooksutamiseks ja analüüsimiseks.
- **Symfony Lime** - Ühiktestimise raamistik, mis on loodud populaarse Symfony PHP veebirakenduse raamistiku jaoks. Raamistiku abil on võimalik näha väljundeid, mis omavad kindlaid värviformaate.
- **Test::Unit** - Ruby sisse ehitatud ühiktestimise raamistik, mille abil on võimalik luua lihttasemel läbiläinud või läbikukkunud teste. Samuti saab teste jooksutada üksikult või gruppidena.
- **RSpec** - Ühiktestimise tööriist Ruby programmeerimiskeele jaoks, mille kindlaks tuumaks on Behaviour Driven Development(BDD). Loodud selleks, et muuta Test Driven Development(TDD) rohkem produktiivsemaks ning meeldivamaks[3].

Tänapäeval on kujunenud mugavaks ka automatiseeritud testimise pilvteenused, mille kasutamisel ei pea looma kohalikku testimise infrastruktuuri ning on võimalus teha testimist pilvteenuse teel. Enamjaolt küsitakse pilveteenuste eest igakuiset tasu, kuid on olemas ka pilveteenuseid, mida saab kasutada tasuta. Pilvteenuste abil on võimalik jooksutada veebi-, mobiilseid- või ühikteste erinevates keskkondades. Tuntumad pilvteenuse testimiskeskkonnad on: Sauce Labs, TestingBot, Gridlastic, CircleCI, Tddium, CloudBees ning Mailosaur[3].

Tabel 1 Automaattestimise tööriistad

Tööriist	**Veebikeskkonna automaattestimine**	**Kasutajakeskkonna automaattestimine**	**Ühiktesti-mine**	**Automatiseeritud testimine pilvteenusena**
Selenium	X			
Watir	X			
Windmill	X			
Ranorex	X	X		
SoapUI	X			
Sahi	X			
Tellurium	X			
Squish		X		
TestComplete		X		
Test Studio		X		
eggPlant		X		
NUnit			X	
xUnit.net			X	
PyUnit/unittest			X	
JUnit			X	
TestNG			X	
PHPUnit			X	
Symfony Lime			X	
Test::Unit			X	
RSpec(Ruby)			X	
Sauce Labs				X
TestingBot				X
Gridlastic				X

Tööriist	**Veebikeskkonna automaattestimine**	**Kasutajakeskkonna automaattestimine**	**Ühiktesti -mine**	**Automatiseeritud testimine pilvteenusena**
CircleCI				X
Tddium				X
CloudBees				X
Mailosaur				X

Ühiktestidest on antud töös kasutusel RSpec raamistik, mis baseerub Ruby programmeerimiskeelel. Ruby keel on loodud jaapanlase Yukihiro Matsumoto poolt, kes on selle loomisel kasutanud oma lemmikosi teistest programmeerimiskeeltest nagu Perl, Smalltalk, Eiffel, Ada ning Lisp. Selle tulemusel on tekkinud Ruby programmeerimiskeel, mis on ülesehituselt lihtne ning naturaalne. Ruby eripäraks on tema ülesehitus, mis on välimuselt väga lihtne, kuid sisult väga keerukas. Ruby keeles programmeerimine on lihtsustatud eelkõige programmeerija enda jaoks, kuid kirjutatud koodi sisu on oluliselt keerulisem. Tegemist on objekt-orienteeritud programmeerimiskeelega, mis võimaldab lahendada ka keerulisemaid olukordi. Suur sarnasus on eelkõige programmeeriskeelega Python, mis on oma ülesehituselt samuti väga lihtne. Ruby on siiski objekt-orienteeritud ning Python seda pole[4]. Ruby on tasuta tarkvara, mida on lubatud tasuta kasutada, modifitseerida ning levitada.

1.2 Automaattestide eelised

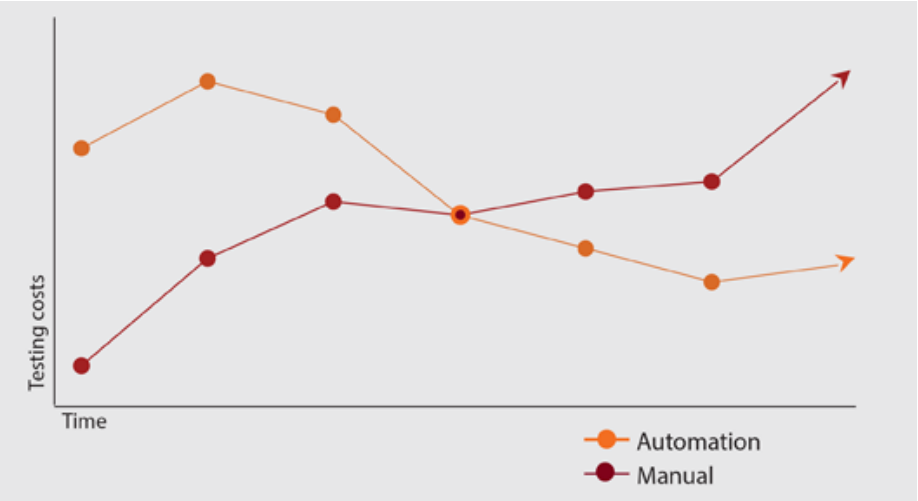
Automaattestimine on arvuti õpetamine, kuidas mingit infosüsteemi testida. Automaattesti loomise käigus kirjutatakse arvutile valmis programmi koodi lõigud, mille järgi arvuti käib läbi tegevused, mida testija peaks tegema käsitsi. Suurim kasu seisnebki üldiselt korduvate ülesannete automatiseerimises.

Automaattestide eelise võib välja tuua ka konkreetse näite põhjal tõsielust.

Ühes suures Wall Street'i ettevõttes oli ühe töötaja tööülesandeks igapäevaselt 8 tundi järjest läbida samu testilugusid. Olukorda otsustati muuta ning võeti ühendust testide automatiseerimist pakkuva firmaga. Firmal kulus antud testilugude automatiseerimiseks

3 päeva ning selle tulemusel kulus töötajal edaspidi sama arvu testilugude läbimiseks 15 minutit. Tänu tekkinud ajalisele kokkuhoiule oli testijal võimalik pühendada oluliselt rohkem aega uute funktsionaalsuste testimiseks[5]. See näide kirjeldab väga hästi esimest suurt automaattestide eelist, milleks on säästetud aeg.

Säästetud aeg on automaattestide juures esialgu suhteline mõiste, kuna see ei pruugi esile tekkida juba testide loomise perioodil. Üks tähtsamaid asju automaattestide loomise juures on analüüs, mis määrab kindlaks, kas tegelikkuses on üldse vajalik testide automatiseerimine. Testide automatiseerimine ei saa toimuda koheselt ning eelnev analüüs ongi eelkõige ettevõtte enda jaoks, et teha kindlaks, kas automaattestide loomine toob ettevõttele kasu või kahju(Joonis 2). Kui ettevõte peab kasulikuks automatiseerida testid, mille testimiseks on väga vähe aega, testimisringe on vähe või on tegu väga väikse süsteemiga, mis ei nõua lisaarendusi, siis reaalselt jääb ettevõte tarkvara arenduse käigus kahjumisse. Kahjum on tingitud automatiseerimise ajalise kulust. Ettevõttel tasuks mõelda automatiseerimise peale, kui kasutusel on olnud või äsja kasutusele võetud pikemad testilood, mis kipuvad olema sarnase struktuuriga ning korduvad. Sellisel juhul võib automatiseerimisele kuluda küll rohkem aega kui testimisele, aga siiski tulemus on hästi märgatav [5].



Joonis 2 Manuaal- ja automaattestide testimise kulu võrdlus ajas [6]

Teine automaattestide eelis seisneb väheses vajalikus väljaõppes. Nimelt on testimine omaette töö, mis nõuab samuti kindlaid teadmisi, loogilist mõtlemist ning tööstruktuuri tundmist. Kui manuaaltestimise osakaal suureneb, võib testijatel tekkida liiga palju tööd ning seetõttu võib vaja minna lisa tööjõudu. Lisa tööjõud aga tähendab uute testijate

väljaõpetamist ning see mõjutab omakorda testimise ajalist kulu. Seetõttu ongi suur roll automaattestimisel, mis on kirjutatud kindla inimese poolt, kellel on olemas vastavad teadmised olemas. Selle abil saab automaattestidega katta testijale tundmatu ala ning lihtsustada oluliselt testija tööd.

Üks suurimaid eeliseid tuleb automaattestimise juures välja pidevalt käiva arenduse puhul, kus on automatiseeritud regressioonitestid. Enamasti toimubki antud KE jaoks kirjutatud testide automatiseerimine regressioonitestide kujul. See võimaldab kiirelt näha arenduse käigus tekkinud vigu ning vähendab ooteaega, mis kulub arendajatel testija järgi ootamiseks. Automaattestimine on üks võimalus, kuidas süsteemi testimise protsessi kiirendada. Selle abil on võimalik ka lihtsamini püsida etteantud ajapiirangutes ning kontrollida süsteemi funktsionaalsust, võrreldes käsitsitestimisega [7].

Eelisena võib välja tuua ka tarkvara arenduse sujuva töökäigu. Nimelt on testimine suures osas kinni testijates ning automaattestide rakendamine igapäevases tarkvara arenduses kindlustab arenguprotsessi ühtlase voo. Testija rolli puhul on inimeste liikuvus väga suur ning uue inimese leidmine võtab samuti aega. Uue inimese leidmise puhul tuleb ta välja koolitada ning see pidurdab arendust. Automaattestide abil oleks võimalik eeltoodud ajakulu oluliselt vähendada.

1.3 Automaattestide puudused

Automaattestimine on nagu programmeerimine, sest testide loomiseks tuleb samuti kirjutada koodi ning kasutada erinevaid programme ja platvorme. Muidugi peab ettevõtetes eksisteerima ka korralik versioonihaldus, ehk testide jaoks loodud programmi koodi lõigud peavad olema organiseeritud. Tihti tuleb tegeleda ka testide hooldusega, sest arenduse käigus võivad tihti mõned juhud muutuda ning valmiskirjutatud skriptid võivad katki minna. Üldiselt sisaldab automatiseerimine endas palju tööd arendaja jaoks.

Testide automatiseerimisel tuleb alati olla kindel testide struktuurses ülesehituses, sest dünaamilisemate testide puhul võivad kirjutatud testid olla vigased. Testi kirjutaja poolt kaetud olukorrad ei pruugi alati ära katta dünaamiliste testide juures kõike ning seetõttu tuleb mõned osad süsteemist käsitsi üle testida, kuna masinat ei saa sajaprotsendiliselt usaldada.

Automatiseerimine on kulukas protsess. See on tingitud eelkõige pikast arendusprotsessist ning ka teadmatusest. Üheks osaks, mis muudab automatiseerimise kalliks on laialdane automatiseerimistööriistade valik, mis nõuab tugevat analüüsi, et teha kindlaks milline tööriist on ikkagi sobilik kindlate testide jaoks. Kindlasti hõlmab analüüs ka korralikku testianalüüsi - välja tuleb selgitada kindlad testid, mida peaks automatiseerima. Automatiseerimist ei tohi alustada suvalisest kohast, sest selline tegevus hägustab ülevaadet süsteemi kaetusele. Tihti võib testide automatiseerimise käigus tulla ka üllatusi, kus ilma tugeva analüüsita saadakse automatiseerimise ajal aru, et mõningaid osi ei ole siiski võimalik automatiseerida ning seetõttu võib arendus veelgi kallimaks minna. Samas on automatiseerimine kallis eelkõige alguse perioodil, kui teadmised on väiksemad ning kiputakse rohkem vigu tegema. Ettevõtetel, kes on tegelenud testide automatiseerimisega pikemalt, on lihtsam teste analüüsida ning seetõttu kujuneb ka testimine pikas perspektiivis odavamaks[7].

Üheks automaattestimise kindlaks puuduseks või ka rangeks nõudeks on ka vastava spetsialisti olemasolu. Selline roll langeb ettevõtetes tihti arendajatele, kes automaattestimisega saavad tegeleda omast vabast ajast. Seetõttu tuleb hoolikalt valida testid, mida automatiseerida. Arendajate ülejäänud tööülesanded on sageli tähtsamad ning seetõttu puudub automatiseerimise suhtes ka vajalik pühendumus. Selle puhul ongi tähtis teha vastavale äriosakonnale selgeks, et automaattestimine on vajalik ning see peaks olema arvestatud projekti eelarvesse. See motiveeriks arendajaid rohkem automaattestidega tegelema ning äri pool mõistaks, kuidas aitavad automaattestid kaasa kiiremale arendusele ja tagavad kindlama töötava funktsionaalsuse. Vastasel juhul on suur vastutus testijal, kellel aga võivad puududa vajalikud teadmised programmeerimisest, mis aitaks luua keerukamaid automatiseeritud teste. Kui automatiseerimise roll langeb testijale, siis tuleb kindlaks teha, et testijal on vajalikud teadmised tarkvara arenduse põhitõdedest, kuna vastasel juhul võib automaattestide kogum muutuda haldamatuks ja organiseerimatuks. Sellisel juhul tuleb samuti teha põhjalik analüüs, kus valitakse testijale jõukohane lahendus automaattestide loomiseks. Valitud lahendus peab olema selline, et oleks hiljem ka lihtsalt muudetav.

Sageli võib testija huvi keskenduda peamiselt automatiseerimisele ning siis kulub liigne aeg ka lihtsamate testide automatiseerimisele, mis tegelikkuses seda ei vaja. Automaatteste loov inimene peab jälgima vastavaid reegleid ning olema sellest samuti

huvitatud. Suureks probleemiks on ka testijatena töötavate inimeste pidev töökohavahetus, mis on tänapäeval suhteliselt levinud. Automatiseerimise õppimine võtab aega ning kui meeskonnaliikmed vahetuvad tihti, kaovad ka õpitud teadmised[7].

Kui manuaaltestija õpib oma tööülesanded selgeks ning teab ise kuidas testimine toimub, siis automaattestimise puhul on olukord keerulisem. Kuna tegemist on koodiga, siis on tähtis osa ka dokumentatsioonil või koodilugemise oskusel. Dokumentatsiooniga kindlustab testide arendaja, et loodud testiva programmi kood on põhjalikult kirjeldatud ning raamistik ja testid peavad arusaadavad olema ka automaattestide kasutajale. Dokumentatsioon ei ole siiski ainuke asi, millega testide loomisel piirdutakse - kirjutada tuleb nõuded, disainida raamistikud ning arhitektuur. Samuti on vajalik koodi haldamine ja versioneerimine. Nagu tarkvaras, esineb ka automaattestides bugisid ning seetõttu tuleb ka automaatteste ennast testida. Kõik need lisaülesanded muudavadki automatiseerimise pikemaks arendusprotsessiks, mis võib ära tasuda alles pikemas perspektiivis[8].

1.3 Automaattestide tulevik

Automaattestide tulevikku iseloomustab väga hästi praegune tarkvara arenduse muutumine agiilsemaks ning kiiremaks. Tarkvara ehitamine ja reliisimine on läinud kiiremaks kui kunagi varem. See aga mõjutab oluliselt testijat kui ametit. On levinud arvamus, et testija amet kaob lähiaastate jooksul. Tegelikkuses ei saa testija amet aga kuskile kaduda - muutuma hakkavad testija ülesanded. See ei tähenda, et iga testija peaks omama väga häid teadmisi programmeerimisest, vaid seda, et testijad peavad arendama rohkem tehnilist mõtlemist, et oma tööd edukalt teha. Samuti läheb testija rollis aina tähtsamaks automatiseerimist pakkuvate tööriistade kasutamine. Tihe koostöö arendajate ja testide vahel ongi see, mis tagab kõige efektiivsemate automaattestide loomise.

Testimisest ei tohi tulevikus saada pudelikael arendusprotsessis ning seetõttu peabki see juba toimuma paralleelselt tarkvara arendusega. 2020-ndaks aastaks peakski testija roll rohkem keskenduma pidevale monitooringule ja probleemidest teadaandmisele. Sellisel viisil on võimalik arendusprotsessi rohkem suunata ja kontrollida. Testija tehniline mõtlemine peaks pakkuma arendajale ka abi probleemi lahkamisel ning testide

automatiseerimisel. Kindlasti ei saa mõne aasta jooksul anda manuaaltestijatele üle tervet testide automatiseerimist, sest see nõuaks testijalt tohutult aega ning pühendumist. Automaattestide loomine peaks tulevikus hõlmama rohkem meeskonda, mitte ühte inimest[9].

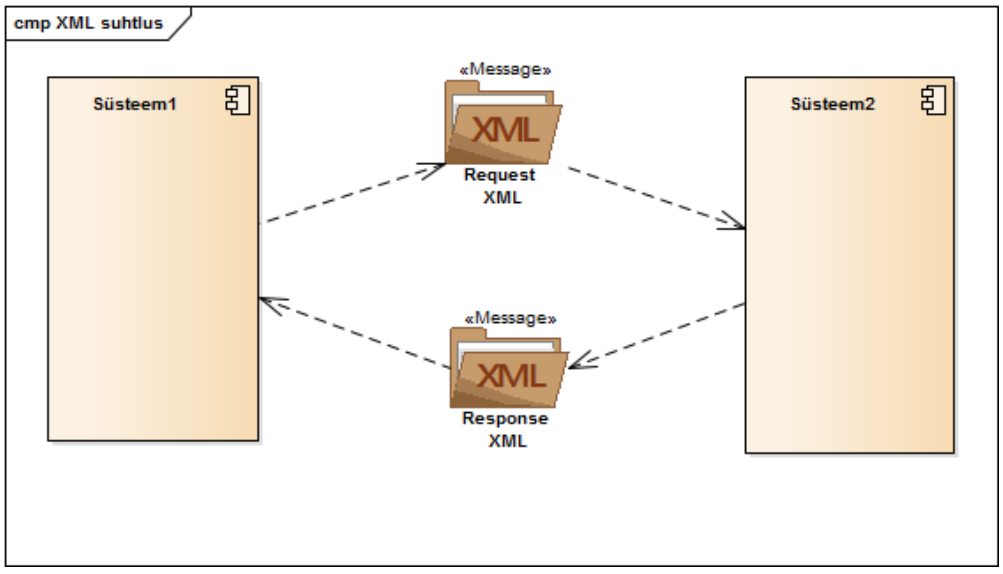
Ei saa väita, et lähiaastatel jääb automaattestide juures kõik samasuguseks. Automatiseerimise tööriistad on pidevas arengus ning kindlasti on näha muudatusi mõningate aastate jooksul. Tööriistad võivad muutuda lihtsamini mõistetavateks ning kasutada ka tavatestijale mõistetavaid skriptimiskeeli. Baastasemele ei saa automaattestimine kunagi jõuda, sest tegemist on ikkagi tarkvara loomisega. Samas on juba tänapäeval näha palju lihtsustusi, nagu näiteks testija tegevuste salvestamine ning skriptimiskeeled, milles kasutatakse baastasemel inglise keelt[9].

2. Automaattestide seostamine valdkonnaga

Bakaulaurusetöö tulemusena valmisid automaattestid (sh. keskkonnad, dokumentatsioon) KE valitud valdkonnale. Valdkonna vajadus automaattestide järgi oli tingitud sellest, et oleks võimalik kiirendada arendust ja saada kohest tagasisidet selle kohta, et uued arendused pole olemasolevat funktsionaalsust lõhkunud. Tegemist on automaatse regressioonitestiga. Kuna tegu on korduva tegevusega, siis on automaattestid õige valik funktsionaalsuse valideerimiseks. Andmebaaside sisu on võimalik lihtsalt manuaalselt kontrollida, kuid tegemist on korduva tegevusega, mis hakkab lõppkokkuvõttes kulutama testija aega ning pikendab ka üldist tarkvaraarenduse protsessi.

2.1 Ülesande püstitus

Bakalaureuse lõputöö käigus tuleb luua valdkonnale töötav automaattestide raamistik, mille abil on võimalik valideerida andmebaasis automaatselt genereeritud andmeid ning ka suhtlust erinevate andmebaaside vahel. Andmebaaside vaheline ning sisemine suhtlus toimub XML failide abil(Joonis 3).



Joonis 3 XML suhtlus kahe süsteemi vahel

Ülesanded:

1. Testimise raamistiku loomine – töö käigus tuleb luua automaattestidele raamistik, mis peegeldab KE-te sisest eelnevalt kasutatud raamistikku
2. Kirjeldada automaattestide nõudeid – anda ülevaade automaattestide jaoks püstitatud nõuetest ja defineerida kindlate nõuete skoop.
3. Dokumentatsiooni loomine – automaattestide loomise käigus on vajalik ka dokumentatsiooni loomine, mis kirjeldab testide ülesseadmist ning kirjutatud koodi.
4. Keskkonnad ja kasutatavad programmid – tuleb kirjeldada erinevate keskkondade põhimõtted ja kasutused. Samuti tuleb esile tuua kõik töö käigus kasutatud programmid.
5. Automaattestide tulemused - tulemuste salvestamine raamistikus paiknevasse kausta.
6. Testide edasi arendamine – tuleviku kindlustamine testide jaoks ning vajadusel ka uute testide loomine.

2.2 Raamistik

Automaattestide jaoks luuakse raamistik vastavalt KE-s kasutatavale eelnevale raamistikule. Automaattestide arendamine koosneb raamistiku loomisest Eesti, Leedu ja Läti jaoks. Kõik automaattestid käituvad sarnaselt, kuid omavad väikseid erisusi nagu näiteks kuupäeva formaat.

Raamistiku tööks on leida õigetesse kaustatesse paigutatud failide seest paarid ning need valideerida. Paaride leidmise süsteem töötab lihtsal request ja response süsteemil. Igal failil on nime ees “RQ_” või “RP_” vastavalt sellele, kas need on request või response. Automaattestide loomine hõlmab endas ka analüüsi erinevate olukordade üle, mida oleks vaja katta valdkonnas ning raamistiku loomist vastavalt KE standarditele.

2.3 Nõuded automaattestidele

Automaattestidele püstitatud nõuded on koostatud eelnevalt osakonna poolt ning samuti on nõudeid täiendatud töö käigus.

Nõuded automaattestidele:

1. Valideerida genereeritud andmed õigete tulemuste põhjal. See tähendab, et automaattestile antakse ette oodatud tulemus ning toimub oodatud tulemuse väljade võrdlemine genereeritud tulemusega;
2. Tuua välja üksikasjalikult erinevused oodatud ja genereeritud tulemuse võrdlemisel, mille abil on võimalik näha arendusest tingitud muutused;
3. Kontrollida genereeritud XML faili struktuurset ülesehitust, ehk valideerida fail XSD vastu. Selle abil on võimalik teada saada, kas genereeritud fail on loodud õige struktuuriga või on genereerimise käigus andmete struktuur poolikuks jäänud või katki läinud;
4. Iga XMLi faili kontrollimise lõpetades salvestada eraldi kausta oodatud ning genereeritud tulemused, et oleks võimalik tagantjärgi näha ka logi.

2.4 Dokumentatsioon

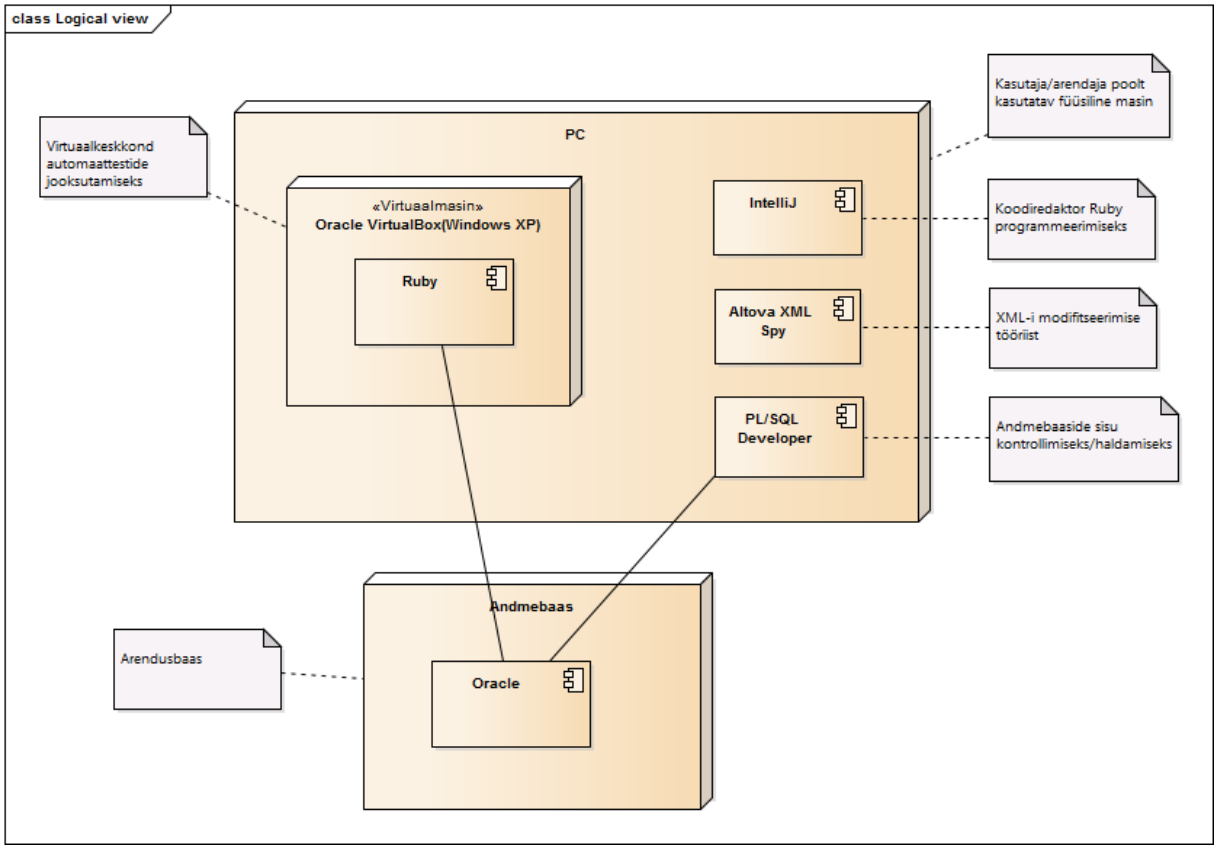
Üheks ülesandeks on luua paralleelselt automaattestidega ka põhjalik dokumentatsioon, mille abil on võimalik automaatteste teistel arendajatel edasi arendada. Dokumentatsioon automaattestide kohta paikneb valdkonna WIKI lehel, mis sisaldab endas lihtsustatud õpetusi testolukordade programmeerimisel. Lihtsustatud õpetused on loodud põhimõttel, et vähesemate programmeerimisteadmistega inimesed oleksid võimelised lisama ning eemaldama testolukordi vastavalt vajadusele. Dokumentatsiooni loomine automaattestidele toimub automaattestide arendamise käigus. Dokumentatsiooni sisu peab katma automaattestides kasutatud koodi ning kirjeldama üksikasjalikult kirjutatud koodiridu. Dokumentatsiooni põhiülesandeks on luua lihtne ja arusaadav dokument, mille abil on võimalik ka teistel arendajatel luua testolukordi ning kontrollida koodi tööd.

2.5 Keskkonnad ja kasutatavad programmid

Automaattestide käigus on vajalik ka manuaalne kontroll, mis hõlmab endas XML failide kontrolli. XML-i kontroll sisaldab endas XML failide sisu kontrolli, võrreldes täpselt elementide väärtusi oodatud tulemuse ja saadud tulemuse vahel. XML failide kontrollimine ning muutmine toimub ettevõttes kasutusel oleva tarkvara abil, milleks on Altova XMLSpy XML Editor[10]. Antud tarkvara abil on võimalik muuta XML failide sisu ning genereerida testolukordi, mis ei tohiks läbi minna, et kontrollida automaattestide tööd. Automaattestide loomine toimub Ruby programmeerimiskeeles, kasutades RSpec raamistikku, mis on samuti ettevõtte siseselt kasutuses mõningas valdkonnas. Integreeritud programmeerimiskeskkonnaks(IDE) on IntelliJ IDEA Ultimate versioon[11], mis toetab suurt hulka programmeerimiskeeli, kaasa arvatud Ruby. IntelliJ on vajalik, et testide kirjutajal oleks võimalikult hea ülevaade tekkivatest vigadest ning et kood oleks võimalikult puhas ja arusaadav. Programmeerimiskeskkond tegeleb pideva süntaksi kontrolliga ning pakub häid silumisvõimalusi. IntelliJ programmeerimiskeskkonda kasutab kokku üle kolme miljoni inimese.

Automaattestide arenduskeskkond on üles pandud virtuaalmasinasse, mille abil on automaattestide arendajal võimalikult vähe väliseid segavaid tegureid ning piiranguid(Joonis 4). Virtuaalmasinas puuduvad üldiselt kehtestatud piirangud, mis võiksid piirata testide loomist(näiteks lokaalsete adminiõiguste vajadus). Virtuaalmasina jaoks on kasutusel Oracle VM Virtualbox tarkvara[12], kus jooksutatakse automaatteste masinal, mille virtuaalseks operatsioonisüsteemiks on Windows XP.

Arenduse käigus on tähtis osa ka andmebaaside töö kontrollis, mis hõlmab endas PL/SQL tüüpi andmebaasi[13]. Andmebaasi kasutamine toimub läbi PL/SQL developer tarkvara abil. Andmebaaside poolse otsa kontroll on vajalik, et näha täpselt, milline on vastutulev XML fail, ning et kontrollida automaattestide õigesti töötamist. Üldiselt on andmebaaside manuaalne töö kontrollimine vajalik ainult automaattestidearenduse käigus, sest lõppetapis peavad testid olema autonoomsed ja kuvama kasutajale võimalikult palju infot testide kohta logifailides, et kasutaja ei peaks andmeid otsima andmebaasist.



Joonis 4 Automaattestide arendamiseks vajalik keskkond

2.6 Automaattestide tulemused

Automaattestide jooksutamise käigus saadud võrdluse tulemused tuleb salvestada kindlasse kausta, et oleks hea ülevaade tulemustest. Tulemustes peavad olema kuvatud kõik erinevused, mida on automaattest töö käigus leidnud. Kaetud peavad olema kõik XML elemendid, mis võivad muutuda arenduse käigus.

Tulemused peavad olema nähtaval ka eraldi html failina, terve testitsükli kohta. Selline html fail annab põhjaliku ülevaate testitsükli kohta ning võimaldab kasutajal kindlaks teha, et terve süsteem toimib korrektselt. Samuti saab tulemuste abil kasutaja kontrollida ka testi tööd, tekitades süsteemi valeolukordi või vigaseid andmeid, mida automaattest ei tohi läbi lasta. Läbi läinud ehk valiidsed testid tähistatakse rohelise värviga ning läbi kukkunud testid punase värviga. Testide tulemuste kuvamine toimub samuti RSpec raamistiku abil, mis võimaldab html faili genereerimist testi tulemuste põhjal. Bakalaureuse töö lõpuks peavad olema automaattestide tulemused kuvatavad osakonna

ruumi seinal paiknevale ekraanile, mille abil on pidev ülevaade arenduse käigus toimuvatest muutustest.

2.7 Edasiarendus

Loodavate automaattestid on orienteeritud põhiliselt XML failide ülesehituse ning sisu kontrolliks, kuid valitud raamistiku abil on võimalik oluliselt rohkem testimise tegevusi automatiseerida. Hetkeseisuga on automaattesti tööks kontrollida andmebaasi täieliku taaskäivituse puhul sinna genereeritud andmete tõesust. Kuna tegemist on lihtsa XML faili sisu kontrollimisega, siis on võimalik rakendada samalaadseid teste ka ülejäänud andmebaasis toimuvate protsesside jaoks, mis tegelevad XML failide genereerimisega. Sellisel juhul oleks rohkem kaetud arenduse käigus olukorrad, mis võivad tekkida.

RSpec raamistik ei piirdu lihtsamate failide sisu kontrollimisega ning võimalik on testidega katta ka teisi olukordi, kus võib esineda arenduse käigus muudatusi või puudujääke. Kuna Ruby on väga arenenud programmeerimiskeel, siis on võimalik kirjutada ka PL/SQL koodi wrapperi abil Ruby-sse. Selle abil on juba võimalik detailsemalt kontrollida andmebaaside sisemistest protsessidest tagastatud väärtusi, mis muudab automaattesti veelgi dünaamilisemaks. PL/SQL koodi abil on lihtsasti võimalik kontrollida olukordi, et kas genereerimisel tekib piisav arv kirjeid andmebaasi või mitte. Samuti on ligipääsetavad ka kõik andmebaaside sisesed parameetrid. Andmebaasiga ühenduse loomist võimaldab RSpec raamistik. Testide abil võib oodatud testi tulemused programmeerida staatiliselt kui ka dünaamiliselt, kuid üldjuhul on eelistatud testide võimalikult dünaamiline ülesehitus, et paranduste tegemisel oleks väga vähesel määral programmeerimist.

Automaattest võimaldab ka hulgaliselt failitöötlust nagu ka eelpool on toodud. On võimalik kasutada testide jaoks erinevaid faile, mis asuvad kindlas failikataloogis. Võimalik on failide otsimine ka suuremast põhikataloogist, mis jaguneb mitmeks alamkataloogiks. Testide raamistik võimaldab ka tulemuste salvestamist failide kujul, mille abil saab samuti vajadusel testidele luua lisa logifaile või salvestada testi käigus kasutatud faile.

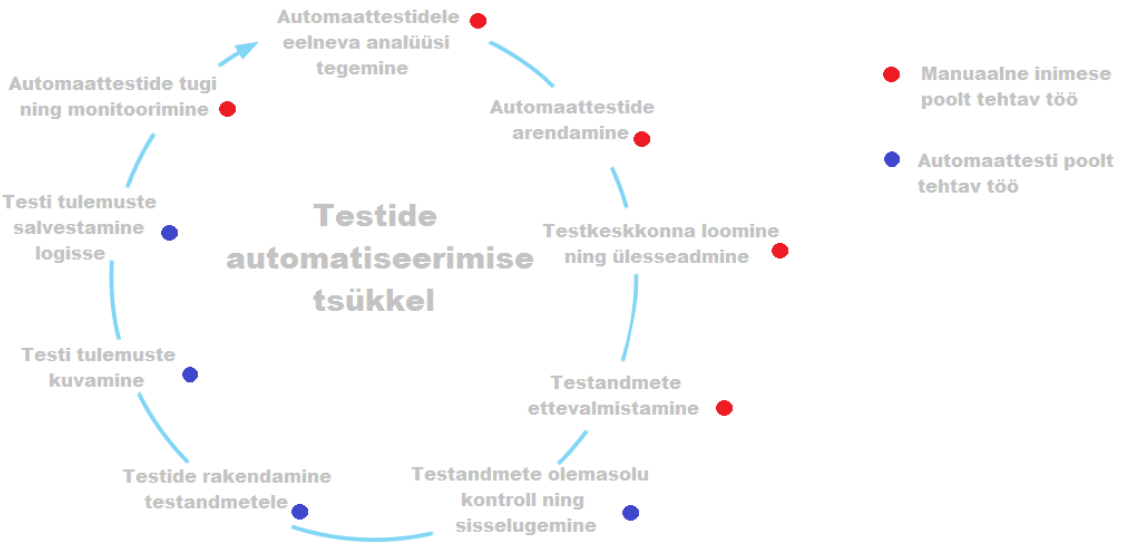
2.8 Ühiktestide valik

Ühiktestide raamistiku valik on tehtud analüüsi tulemusel. Ühiktestide valiku ainsaks eelduseks on Ruby programmeerimiskeel, mis on valitud, sest on kasutusel mitmes erinevas osakonnas, kust on võimalik saada arenduse käigus tuge. Võimalikeks kandidaatideks olid Ruby programmeerimiskeelele ehitatud raamistikud: Test::Unit, RSpec, minitest ning Bacon. Test::Unit kujutab endas väga lihtsat Ruby teeki, mis oli kaasatud Ruby versioonidega 1.8 - 2.2. Test::Unit võimaldab testide loomist ja kontrolli, kuid üldiselt ka sellega piirdub[14]. Praeguseks ajaks on Test::Unit-i asemel kasutusel minitest. Minitest on väike ja üldiselt väga kiire testimise raamistik. Selle abil loodud testid on hästi loetavad ja puhta stiiliga[15]. Minitestis on kasutusel väga sarnane süntaks Ruby-ga, mis tähendab, et Ruby oskamisel on minitest-i kasutamine väga lihtne. Minitesti abil aga ei ole võimalik saavutada kõiki väljundeid, mis on ülesandepüstituses soovitatud. Bacon on RSpeci kloon, mis pakub üldiselt kõiki samu võimalusi mis RSpec ning peegeldab süntaksi poolest oluliselt RSpeci[16]. Ainuke erinevus seisneb selles, et tegemist on suuruselt väga väikse teegiga, mis on 350 LoC.

Ühiktestide valiku otsus on tulnud eelkõige arvestades vajaminevaid tunnuseid. RSpec on võetud automaatestide alusraamistikuks, sest pakub rohkelt lisasid ning samuti on loodud väga lihtsa ülesehituse ja süntaksiga. RSpec on samuti laialt kasutusel ülemaailmselt ning on võimalik leida hulgaliselt näiteid ja tuge foorumitest[17]. Üks valiku põhjuseid oli ettevõtte sisene kasutus, mis näitas, et ka paljud teised arendusega tegelevad osakonnad on võtnud automaattestide loomiseks kasutusele RSpec ühiktestide raamistiku, sest see hõlmab lihtsat süntaksit ning olulisel kohal on ka baasprogrammeerimiskeel Ruby, mis on lihtsasti õpitav objekt-orienteeritud programmeerimiskeel.

3. Automaattestide loomine KE-le

Praktilises osas bakalauruse töös keskendub autor eelkõige Ruby programmeerimiskeelele. Kuna eelnev kokkupuude Ruby programmeerimiskeelega puudub, siis on autor bakalauruse töö käigus omandanud keele iseseisvalt. Automaattestide loomise osakonnas puudub Ruby keele spetsialist ning seetõttu on enamus teadmisi omandatud interneti kaudu. Samuti on olnud vähene kokkupuude ka XML failistruktuuriga ning RSpec raamistikuga. Enamus uusi teadmisi on omandatud lõputöö loomise käigus kaasaarvatud andmebaaside baaskasutuse teadmised ning automaattestide ja nende vajalikkuse mõistmine, nende jooksutamine.

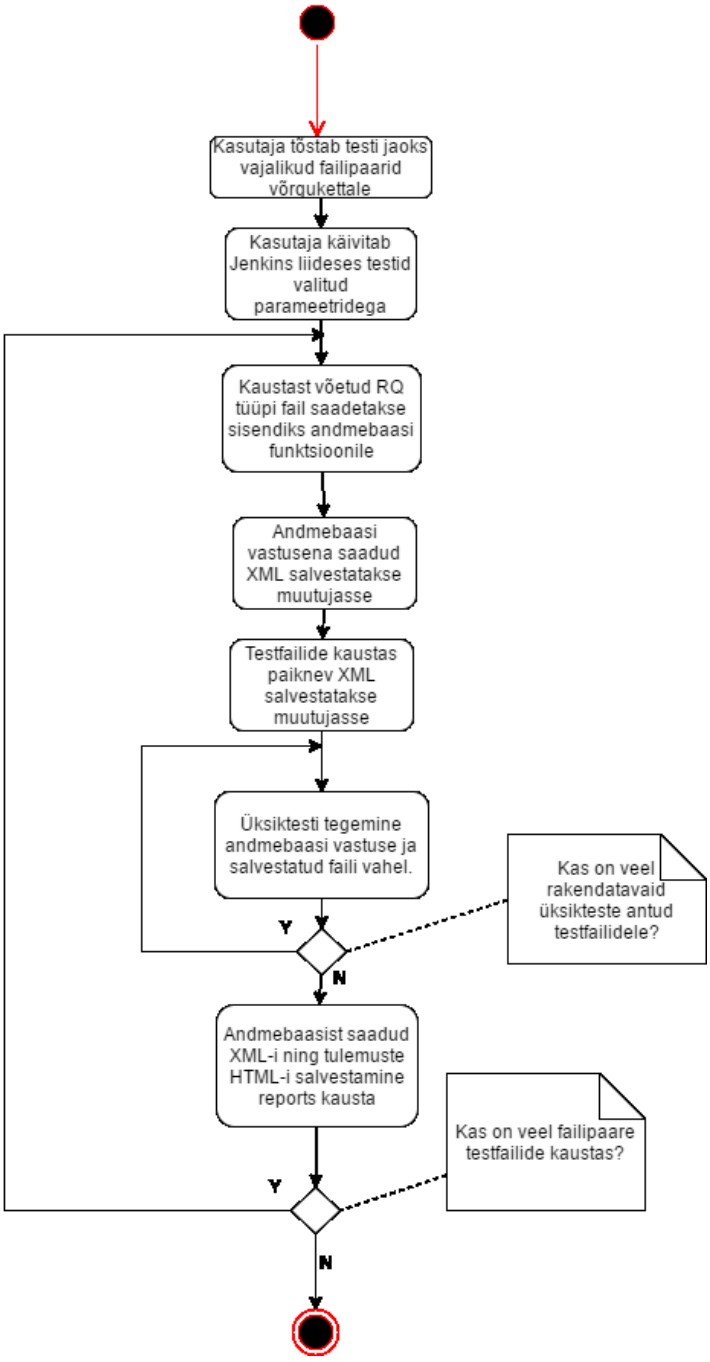


Joonis 5 Automaatiseeritav osa testimises

3.1 Automaattesti töö failide lugemisel ja salvestamisel

KE jaoks loodud automaattest kontrollib töö alustamisel üle failide kausta paigutatud failid. Selle jaoks on olemas koodilõik, mis määrab automaattesti töö enne võrdlusi ning peale võrdlusi(Joonis 6). Selle koodiga toimub failide paaride määramine. Kaustas paiknevad failid käiakse ükshaaval läbi ning loetakse sisse nende nimed. Kuna testide puhul on vajalikud "Request" ja "Response" tüüpi XML-id, siis pannakse need antud kausta samasuguste nimedega. Ainuke erinevus on nimede ees paiknevates tag-ides.

Tag-ideks on "RQ" ja "RP". Kaks failipaari oleks näiteks "RQ_pakkumine_nr1" ja "RP_pakkumine_nr1". Peale paari leidmist algabki testi põhiosa, kus võrreldakse tulemusi. Kui testi põhiosa on lõpetatud, toimub tulemuste salvestamine kausta nimega "rspec_reports". Sinna arhiveeritakse kõik eelnevad testid ning sealt on võimalik näha eelnevaid tulemusi. Testide lõppedes toimub andmebaasist saadud vastuse salvestamine sarnaselt testide tulemuste salvestamisele (Lisa 1). Igal arhiveeritud tulemuste kataloogil on alamkataloog, kuhu salvestataksegi andmebaasist saadud XML failid.



Joonis 6 Testi loomise tegevuste diagramm

3.2 Ruby keele süntaks automaattestimisel

Ruby lühitutvustus on leitav peatüki 1.1 lõpus. Ruby programmeerimiskeeles on kõik elemendid kasutusel objektidena. Igale informatsiooniblokile ning koodile on võimalik kinnitada omadusi või tegevusi. Objekt-orienteeritud programmeerimise puhul kutsutakse omadusi instantsi variaabliteks ning tegevusi meetoditeks. Programmeerides on väga lihtne kindlaid osi koodist eemaldada või lisada. Automaatteste luues on autori jaoks selgeks saanud Ruby keele lihtsus ja mugavus. Ka lihtsale "+" operaatoriga tehtele on võimalik teha asendus "plus" meetodina, et programmeerides oleks võimalikult lihtne uut koodi luua. Selline süntaks lubab automaattestide loomisel genereerida sarnase stiiliga arusaadavat koodi, mis teeb nii testide kirjutaja kui ka modifitseerija jaoks koodi arusaadavamaks.

Sarnaste testide väljakutsumist on autor kasutanud luues meetodibloki, mis võimaldab korduvalt kasutada kindlat üksikut testi koodi vältel, mitmes erinevas kohas. Muutujatele on võimalik külge panna blokistiilis erinevaid funktsioone, mida on kasutatud automaattestimisel näiteks ASCII kodeeringu kontrolliks. Ruby süntaksi abil on loodud ka programmeerimises kasutatav "case" menüü, mille abil toimub antud ettevõttele loodud automaattestide õige testikogumi valik. Testikogumi valikut otsustav tegur on võetav parsides Ruby keele nokogiri teegi abil XML faili ning saades sealt XML faili pealkirja elemendi sisu.

Ruby süntaksis kasutusel oleva koodi wrapper on lõputöö käigus loodud automaattestis kasutusel suhtlemiseks andmebaasiga. Selle abil saab andmebaasist välja kutsuda kindla funktsiooni, mis tagastab soovitud XML faili. Funktsioonile saab sisendi kaasa anda koodi kaudu. Kõik vajalikud andmebaasiga seotud muutujad sisalduvad erinevas klassiblokis, et andmebaasiga oleks võimalik luua ühendus. Andmebaasist XML faili sisse lugemine toimub kursorit kasutades, mis tagastab loetud faili globaalsesse muutujasse. Testide töö kindlustamiseks on loodud ka kontrollfunktsioone, mille abil on võimalik arenduse käigus lihtsamini üles leida tekkinud vead(Joonis 7).

```
def control_existance tag_xpath          #meetodi definitsioon
  if @xmldocRes.xpath(tag_xpath).first #tingimuslause xpathi leidmiseks
    return 1                           #xpathi leidmisel tagastame 1
  else
    return 0                           #xpathi mitte leidmisel tagastame 0
  end
end
```

Joonis 7 Kontrollkoodi näide

3.3 XML failistruktuur

KE jaoks loodud automaattestid keskenduvad eelkõige XML failidele, mis on ka terve raamistiku aluseks. Antud valdkonna puhul on väga oluline, et XML failid oleks korrektse struktuuri ja sisuga, sest need sisaldavad kliendiandmeid, auto andmeid, finantsandmeid ning muid tehingutega seotud andmeid. Seetõttu ongi loodud andmete valideerimiseks automaattestid, mis kontrollivad, et arenduse käigus poleks XML failide sisu ja struktuuriga tekkinud olulisi muudatusi.

Põhiline osa automaattestide tööst on XML elementide sisu kontrollimine, mis tähendab, et oodatud tulemused peavad sisalduma andmebaasist võetud XML-ides. Elemendi kontroll on Ruby kaudu sooritatav XPathi kaudu, mis on päringukeel XML struktuuris olevate elementide leidmiseks ja lugemiseks. XPath on KE automaattestides rakendatav tänu Nokogiri teegile, mille eesmärgiks on teha XML ning HTML failides orienteerumine Ruby keele jaoks lihtsamaks. Nokogiri teek teeb XML-i sõelumise oluliselt lihtsamaks, sest selle abil on võimalik dünaamiliselt lugeda kindlate emaelementide all peituvaid elemente ning korduvate elementide lugemisel on säästetud mitmeid ridu koode, luues dünaamiliselt tsüklis käivat koodi. Nokogiri on eelkõige vajalik just seepärast, et KE poolt kasutatav XML failide struktuur on aastatega muutunud keerukamaks ning ühes XML-is on keskmiselt 600 rida. Kui automaattesti abil võtab ühe failipaari võrdlemine mõned sekundid, siis manuaalse testimise käigus võib see võtta kuni pool tundi.

XML struktuuri testimiseks on loodud automaattestides kasutusel ka XSD valideerimine. XSD kontrollimine ei ole antud töö käigus kirjutatud manuaalselt, kuna tegemist on eeldefineeritud kindla struktuuriga. Kuna tegemist on justkui šablooniga,

mis näitab XML faili sobimist antud vormi, siis on ka testimine lihtsam. XSD valideerimiseks on Nokogiri teegis olemas funktsioon “validate”, mille abil on võimalik välja tuua kõik XSD struktuuris esinenud vead. Selle funktsiooni kasutamiseks on loodud automaattestile XSD kaust, kust on tagatakse ligipääs originaalsele XSD-le ning XSD-d tuleb võrrelda andmebaasist tulnud XML-i struktuuriga(Joonis 8). Eelduseks on korrektselt loodud XSD fail, mis kirjeldab täpselt võimalikku XML struktuuri.

```
def xsd_validate
  it 'Validate xml against xsd' do      #ühe testi loomine rspec süntaksiga
    i = 0                               #esialgne errorite arv 0
    errors = ""                         #määrame tühja errorite massiivi
    @xsdvalid.validate(@xmldocResDB).each do |error| #validate funktsioon,
                                                     #mille  abil  kontrollime
                                                     #XSD vigu
      puts error.message                #prindime välja errori konsooli
      errors << " #{error.message} "    #errori sõnumisisu lisamine massiivi
      if error                          #kui tegemist erroriga, tõstame
        i += 1                          #errorite arvu
      end
    end
    expect(i).to eql_with_msg(0, "Errors: #{errors}    ")  #testi
                    #läbipääsemiseks ootame 0 errorit ning errorite saamise
  end               #puhul prindime need väljundisse
end
```

Joonis 8 XSD valideerimise kood

3.4 RSpec raamistik

RSpec raamistiku abil on bakalaureuse töös loodud automaattesti üksikud testolukorrad, kus koodi keskel valideeritakse tulemust vastavalt oodatud tulemusele. Selle abil on määratakse automaattesti detailsust. Automaattesti loomisel on võimalik teha lihtsustatud versiooni, kus automaattest lõpetab testimise siis, kui on leidnud vea või detailsemat versiooni, kus automaattest käib läbi kõik võimalikud testolukorrad ning genereerib logifaili kõik avastatud vead.

Testid on loodud ükshaaval, kasutades lihtsat funktsiooni põhimõttet. Testi ainsad vajalikud väärtused, mis tuleb kaasa anda on oodatud tulemus ja saadud tulemus. Samuti on RSpeci abil loodud KE automaattestidele testikomplekte, mida saab kasutada teistes testides juba loodud blokkidena (Lisa 2). RSpec pakub mitmeid erinevaid võrdlusvõimalusi vastavalt programmeerija soovidele. Üksikute testide puhul on kasutatud erinevaid võrdlustegureid, nagu näiteks: peavad võrduma, ei tohi võrduda, peab sisaldama, ei tohi sisaldada. Võrdlustegureid on võimalik ise juurde luua ning seda võimalust on antud töös ka kasutatud. Võrdlustegurite puhul on antud töös enim kasutatud egl_with_msg võrdlust, mis on loodud antud automaattesti jaoks ning mille abil on võimalik logifaili kirjutada testi läbikukkumise juurde ka selle põhjus(Joonis 9). Näiteks on võimalik anda kaasa erinenud elemendi XPath, mille abil on võimalik kiirelt ülesse leida erinevused kahe XML faili vahel.

```
def assert_tag_count_d tag_xpath      #meetodi definitsioon(def NIMI MUUTUJA)
  it "Control tag count: <#{tag_xpath.split('/')[-1]}>" do #testcase loomine
                  #kust võtame xpathist ainult viimase xpathi osa
    temp1 = @xmldocRes.xpath(tag_xpath).count
                  #salvestame muutujasse temp1 oodatud elemendi loenduse
    temp2 = @xmldocResDB.xpath(tag_xpath).count
                  #salvestame muutujasse temp2 oodatud elemendi loenduse
    expect(temp2).to eql_with_msg(temp1, "Error in: #{tag_xpath}    ")
                        #ootame, et temp2 võrduks temp1-ga, vastasel juhul
  end                   #kirjutame errori väljundisse
end
```

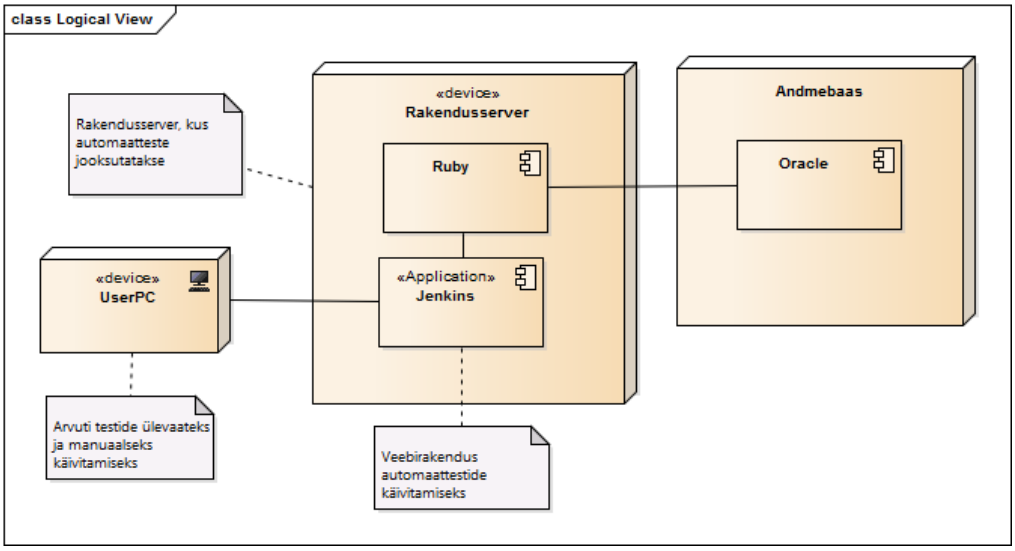
Joonis 9 Elementide arvu esinemise kontrolli kood

3.5 Suhtlus andmebaasiga

Peale vajalike testipaaride leidmist toimub suhtlus andmebaasiga. Selle käigus kasutatakse XML faili “RQ” algusega ära kui tavalist suhtluselementi andmebaaside juures. Kui üldise andmebaaside töö käigus kasutatakse “RQ” XML-i, et saada kätte vastus andmebaasist, siis automaattesti puhul kasutame samat funktsiooni. Selle abil saame vastusena saadud XML-i kasutada ära üksiktestide võrdlemise osas. Andes ette “RQ” XML-i sisendina, saame kasutada PL/SQL funktsiooni, mis on defineeritud andmebaasis (Lisa 3). Samuti on loodud automaattestide puhul kasutatud ka eriolukordi, kus tuleb andmebaasidest saada kätte rohkem kui 1 fail. Selle jaoks on kirjutatud lisa kood, kus kasutame saadud vastuse XML-ist leitud elemente, et luua uus request.

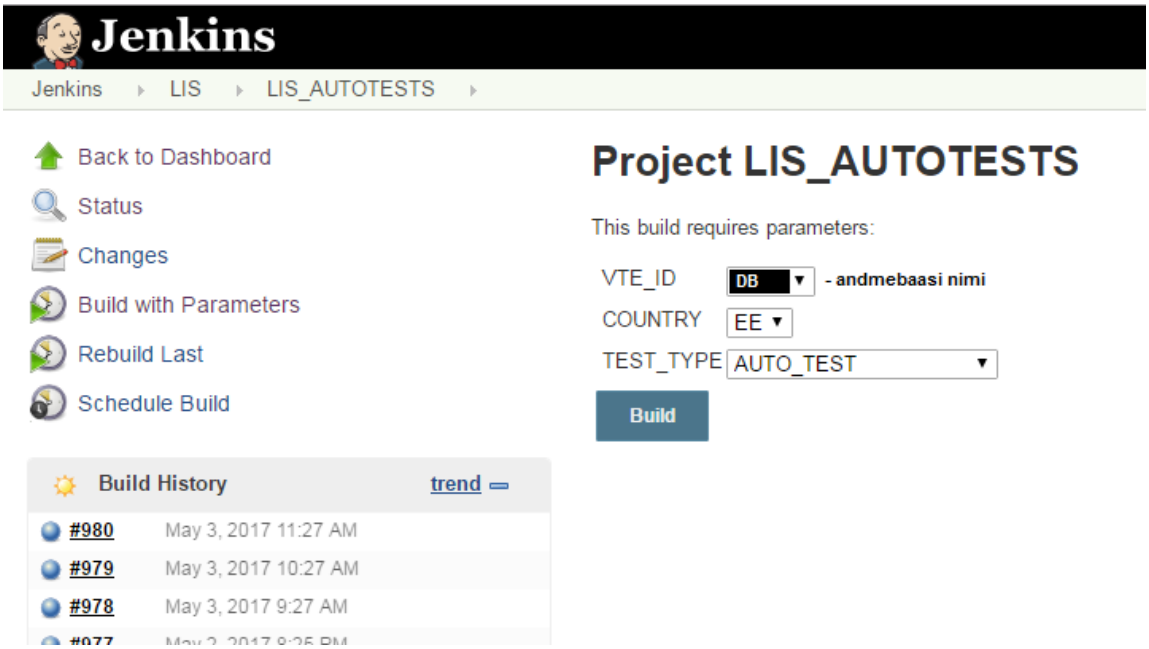
3.6 Testide käivitamine ja tulemused

Testide arendusfaasis kasutas autor testide käivitamiseks virtuaalmasina operatsioonisüsteemi konsooli. Testide kataloogile on võimalik ligi pääseda läbi ühise jagatud ketta, mis defineeritakse virtuaalmasina loomisel. Käivitamiseks tuleb konsooli kaudu navigeerida testide kataloogi ning kasutades kataloogis leiduvad üldist rspec.rb koodi jooksutada kindel testifail. Testifailiks on võimalik valida detailne või lihtsustatud test. Testitavad failid tuleb enne testi jooksutamist panna automaattestide kataloogis leiduvasse "TESTFILES" kausta. Vastavalt failidele toimub ka testiolukordade valimine. Faili kindla elemendi järgi määratakse sellele jaoks kasutatavad testid, mida peab automaattest kontrollima.



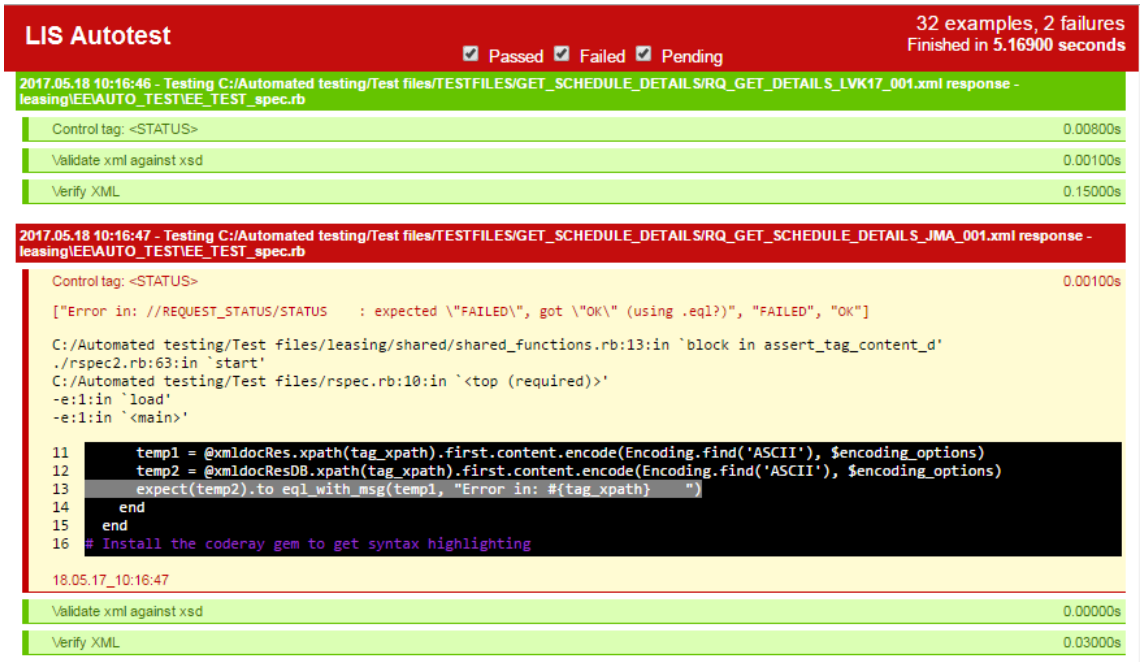
Joonis 10 Testide automaatse käivituse keskkond

Valmis loodud testidekomplekti käivitamiseks on loodud käivituskeskkond läbi Jenkins tarkvara, mille abil on võimalik automatiseerida tarkvara testimist, juurutamist või ülesehitust(Joonis 10). Antud tarkvara on juba osakonnas kasutusel muudel eesmärkidel ning seda on mõistlik kasutada seda ka automaattestide käivitamiseks. Jenkins keskkonnas on seadistatud automaattestid jooksma iga tunni aja tagant ning peale igat taaskäivitust(Joonis 11). Testijad saavad käivitada teste ka manuaalselt, määrates käivitusparameetrid ning tõstes testitavad failid kindlale võrgukettale. See võrguketas on ühenduses masinaga, kus automaattestid jooksevad. Peale automaattestide töö lõppu kuvatakse testijale tulemus, mida tagastab RSpec raamistik html failina. Seda tulemust kuvatakse osakonna seinale.



Joonis 11 Jenkins keskkonna kasutaja vaade

Html väljundis saadud tulemused on defineeritud värvide kaudu, kus roheline tähendab läbimist ning punane läbikukkumist(Joonis 12). Samuti kukutab testi läbi ka väiksem viga mõnes alamtestis ning pikemate testide puhul on juba alguses html väljundi pealkirja kaudu näha kas mõni test on läbi kukkunud. Antud näites on “Control tag <STATUS>” tavaline staatuse välja kontroll, “Validate xml against xsd” XSD struktuuri kontroll ning “Verify XML” on sisuväljade kontroll.



Joonis 12 KE testide XML võrdluse HTML tulemus

3.7 Testide valideerimine

Testide valideerimise jaoks on loodud automaattestis autor enamjaolt kasutanud RSpec teegiga defineeritud võrdlusfunktsioone, mida on lühidalt kirjeldatud ka peatükis 3.3. KE automaattestide puhul peavad olema võrdlused väga spetsiifiliselt määratud. Defineeritud on erinevad meetodid vastavalt testi vajadusele. Igat testi on võimalik käivitada detailse või lihtsa versioonina, millest sõltub testi tulemuste kuvamine. Seda otsustab testide käivitaja. Vastavalt testitüübile on loodud ka detailsed ja lihtsad meetodid. KE automaattestide jaoks on autor loonud eriolukordade jaoks spetsiifilised meetodid, mis tegelevad XML faili kindlate elementide dünaamilise kontrolliga(Lisa 4). Näiteks võib XML-is olla kujutatud korduvate elementidega, kuid erinevate ID-dega muutujad, mida võib ühes grupis esineda mitmeid. Sellist olukorda XML-is kasutatakse tabeli genereerimiseks ning seetõttu on tähtis ka tabeli kindlate elementide valiidsus. Selle jaoks on autor loonud meetodi, mis suudab kahe "for" tsükli abil käia läbi nii oodatud kui saadud XML-i vastavad elemendid. Valmiskirjutatud meetodeid on võimalik testide täiustajal kasutada lihtsa meetodi väljakutsumisega. Meetodi väljakutse sisaldab endas meetodi nimetust ning vastava elemendi XPathi.

KE automaattestis kasutatavad lihtsamad testid kontrollivad XPathi abil kindlat XML elementi, mille puhul tuleb veenduda, et sisu on samasugune mõlemas kontrollitavas elemendis. Valideerimise käigus vigade tekkel tuleb tulemuse HTML-i vigase elemendi XPath, et vea leidmist lihtsustada. Testide valideerimisel on võimalik lisaks “failed” ning “passed” testile kasutada ka “waiting” testi, kui on tegemist mõne suurema testiblokiga, mis ei ole saanud eelnevalt testilt tulemust, mida on vaja võrdluseks. Testi tüüp “waiting” ei ole hetkeseisuga veel kasutusel KE automaattestides.

Kokkuvõte

Töö käigus on välja toodud automaattestide eelised ning puudused võrreldes manuaaltestidega. Antud on ülevaade testija rolli eripära ning selle muutumise kohta tulevikus. Bakalaureuse töö käigus on loodud automaattestid KE valdkonnale.

Seoses automaattestidega on kirjeldatud erinevaid võimalikke automaattestimise tööriistu ning põhjendatud ka Ruby-le orienteeritud testide valikut. KE-le kirjutatud automaattestide loomiseks on kasutatud Ruby programmeerimiskeelt ning RSpec raamistikku. Lisaks on töö käigus kirjeldatud automaattestide kasulikkust ning on välja toodud olukorrad, kus tuleks automaatteste vältida. Loodud automaattestide raamistik on tehtud arvestades osakonnast tulenevaid nõudeid ning kaetud on igapäevase eduka arenduse jaoks vajalikud suhtluse elemendid(XML-id) andmebaaside vahel.

Töö praktilises osas on tutvustatud erinevaid testimise olukordi ning toodud esile arendamise protsessi. Lisaks on automaattestidele toodud praktilises osas juurde ka programmikoodi näiteid ning kirjeldatud üldist testi käitumist. Peale spetsiifiliste testide on antud ülevaade ka raamistiku enda käitumisest ja failide sisselugemisest. Tulemuste analüüs ning võrdlus on esile toodud väljundfaili abil. Töö käigus on loodud testid, mille ülesanded on: XML elementide kontroll, XSD failistruktuuri valideerimine, failide sisselugemine ja kontroll, korduvate elementide arvu kontroll ning dünaamiliste elementide kontroll. Testi abil on võimalik säästa oluliselt aega arendusprotsessis, sest üldiselt kulub manuaalsel testijal ühe XML-i paari kontrollimiseks kuni 30 minutit. Automaattestil kulub ühe XML-i peale sõltuvalt suurusest kuni 8 sekundit ning oluline ajavõit on saadav suurema testide hulga puhul. Loodud testide kuvamine KE osakonna seinal asuvale kuvarile säästab samuti aega regressioonitestide puhul, sest on koheselt nähtav tulemus ning üldine testide valiidsus.

Kasutatud kirjandus

[1] Automaattestimine - mis see on? [WWW] http://asaquality.blogspot.com.ee/2014/05/automaattestimine-mis-see-on.html (12.02.2017)

[2] Software Testing: History, Trends, Perspectives - a Brief Overview [WWW]

[3] Software Test Automation Tools [WWW] http://www.testingtools.com/test-automation/ (19.02.2017)

[4] About Ruby [WWW] https://www.ruby-lang.org/en/about/ (03.04.2017)

[5] Graham, D., Fewster, M. (2012). Experiences of Test Automation: Case Studies of Software Test Automation. London: Addison-Wesley Professional

[6] Why financial institutions need automated testing [WWW] http://www.pwc.com/us/en/financial-services/publications/viewpoints/software-failure-automated-functional-testing.html (24.04.2017)

[7] Kuidas testide automatiseerimisel õnnestuda [WWW] http://people.proekspert.ee/blog/?p=1045 (22.02.2017)

[8] Future of testing and automation [WWW] http://blog.xebia.com/future-of-testing-and-automation-the-role-of-the-tester-in-2020/ (22.02.2017)

[9] How the future of test automation affects you [WWW] https://www.tricentis.com/resource-assets/how-the-future-of-test-automation-affects-you-whitepaper/ (22.02.2017)

[10] Altova XMLSpy XML Editor [WWW] https://www.altova.com/xmlspy.html (15.03.2017)

[11] IntelliJ IDEA [WWW] https://www.jetbrains.com/idea/ (15.03.2017)

[12] Oracle VM VirtualBox [WWW] https://www.virtualbox.org/ (19.03.2017)

[13] Oracle Database 12c PL/SQL [WWW] http://www.oracle.com/technetwork/database/features/plsql/index.html (15.03.2017)

[14] Test::Unit documentation [WWW] https://ruby-doc.org/stdlib-2.1.1/libdoc/test/unit/rdoc/Test/Unit.html (15.03.2017)

[15] Minitest [WWW] https://github.com/seattlerb/minitest (15.03.2017)

[16] Bacon - small RSpec clone [WWW] https://github.com/chneukirchen/bacon (15.03.2017)

[17] RSpec [WWW] http://rspec.info/ (15.03.2017)

Lisa 1 – automaattesti failide sisselugemise kood

```
Dir.glob("#{File.join(File.dirname(__FILE__),'../../../','TESTFILES')}/**/RQ*
.xml").each do |input_test_file|        # iga RQ tüüpi faili jaoks
testitsükkel, lugedes TESTFILES kaustast
  describe "Testing #{File.absolute_path(input_test_file)} response" do
      # Testi käivitamine ühe faili jaoks

    @xmldocRes = Nokogiri::XML(File.open(File.join
                (File.dirname(input_test_file), 'RP' +
                File.basename(input_test_file)[2..-1])))
      # RP failipaari otsimine, võttes esimese faili nimelt ära "RQ" tag ja
lisades ette "RP" tag
    case_select(@xmldocRes.xpath("//REQUEST_TYPE").first.content)
       # Case menüü õige XML tüübi valimiseks
    before :all do # Koodiblokk, mida jooksutatakse enne kõiki teste
      @xmldocReq = Nokogiri::XML(File.open(input_test_file))
      # RQ faili sisselugemine muutujasse
      @xmldocResDB = Nokogiri::XML(database_request(@xmldocReq))
      # Andmebaasilt vastuse saamine, kasutades sisendina RQ faili
      @xsdvalid = Nokogiri::XML::Schema(File.open(File.join
                (File.dirname(__FILE__), '../../../', 'XSD',
                "RP_#{@xmldocRes.xpath("//REQUEST_TYPE").first
                .content}.xsd")))
      # XML-ile vastava XSD sisselugemine XSD kaustast
    end
    after :all do # Peale kõiki teste jooksutatav koodilõik
      File.open($watir_formatter.file_name_with_path('Response' +
      File.basename(input_test_file)[2..-1]), 'w+') { |file|
      file.write(@xmldocResDB) }
      # Andmebaasilt saadud RP tüüpi faili salvestamine logifailidesse
    end
  end
end
```

Lisa 2 – XML väljade sisu kontrolli testikomplekti kood

```
shared_examples 'GET_PARTY_DETAILS' do
  assert_tag_content_d("//REQUEST_RESULT/party-data/party-full-name")
  assert_tag_content_d("//REQUEST_RESULT/party-data/party-address")
  assert_tag_content_d("//REQUEST_RESULT/party-data/city-name")
  assert_tag_content_d("//REQUEST_RESULT/party-data/municipality-code")
  assert_tag_content_d("//REQUEST_RESULT/party-data/zip-code")
  assert_tag_content_d("//REQUEST_RESULT/party-data/phone")
  assert_tag_content_d("//REQUEST_RESULT/party-data/e-mail")
  assert_tag_content_d("//REQUEST_STATUS/NOTE")
end
```

Lisa 3 – andmebaasist vastuse saamise kood

```
def database_request input  # See funktsioon lubab meil ühenduda andmebaasiga
                            ja saada sealt vastus
  cursor = Record::Database.connection.raw_connection.parse("begin
           request_to_db('#{input}', :result_body);
           end;").bind_param(':result_body', nil, String, 1280000)
             # Küsime andmebaasist kindla funktsiooni "request_to_db" abil
              vastust, andes sisendiks eelnevalt muutujale omistatud RQ XML-i
  cursor.exec # Kursori käivitus
  output = cursor[':result_body'] # Kursorist ":result_body"
                                     tagastamine muutujasse output
  cursor.close # Kursori sulgemine
  output       # Output muutuja tagastamine funktsioonist
end
```

Lisa 4 – automaattesti võrdlusfunktsioonide kood

```
$encoding_options = {
    :invalid           => :replace,  # Asendame tundmatud baidijadad
    :undef             => :replace,  # Asendame kõik, mis pole defineeritud
ASCII-s
    :replace           => '',        # Asenduseks kasutame tühikut
    :universal_newline => true       # Ridade murdmine toimub alati
                                  "/n" abil
} # ASCII kodeerimine, et eemaldada tundmatud märgid võrdluse jaoks

def assert_tag_content_d tag_xpath     # Elemendi sisu kontroll
  if @xmldocRes.xpath(tag_xpath).first
    it "Control tag: <#{tag_xpath.split('/')[-1]}>" do
      # Testi pealkirjaks anname elemendi nime
      temp1 = @xmldocRes.xpath(tag_xpath).first.content.encode
            (Encoding.find('ASCII'), $encoding_options)
      temp2 = @xmldocResDB.xpath(tag_xpath).first.content.encode
            (Encoding.find('ASCII'), $encoding_options)
      # Loeme sisse muutujad arvestades kodeerimist
      expect(temp2).to eql_with_msg(temp1, "Error in: #{tag_xpath}  ")
      # Errorit saades lisame väljundisse vigase elemendi xpathi

    end
  end
end
```

```
def assert_tag_count_d tag_xpath # Elemendi arvu kontroll
  it "Control tag count: <#{tag_xpath.split('/')[-1]}>" do
    temp1 = @xmldocRes.xpath(tag_xpath).count
    temp2 = @xmldocResDB.xpath(tag_xpath).count
    expect(temp2).to eql_with_msg(temp1, "Error in: #{tag_xpath}    ")
  end
end

def control_existance tag_xpath  # Elemendi olemasolu kontroll
  if @xmldocRes.xpath(tag_xpath).first
    return 1
  else
    return 0
  end
end

def assert_tag_content_same_d tag_xpath# Ühesuguste elementide
                                       sisu kontroll dünaamiline
  it "Control tag: <#{tag_xpath.split('/')[-1]}>" do
    if @xmldocRes.xpath(tag_xpath).first
      i = 0
      @xmldocRes.xpath(tag_xpath).each do
        temp1 = @xmldocRes.css(tag_xpath.split('/')[-1])[i].text
        temp2 = @xmldocResDB.css(tag_xpath.split('/')[-1])[i].text
        expect(temp2).to eql_with_msg(temp1, "Error in entry #{i}:
        #{tag_xpath}    ")
        i += 1
      end
    end
  end
end
```

```
def limit_entry_count(tag_xpath, limit)# Elementide esinemise
                                       limiidi kontroll
  it "Check entry count limit: #{limit} entries allowed" do
    temp1 = @xmldocResDB.xpath(tag_xpath).count
    expect(temp1).to be <= limit
  end
end

def xsd_validate                       # XML valideerimine XSD
                                       vastu
  it 'Validate xml against xsd' do
    i = 0
    errors = ""
    @xsdvalid.validate(@xmldocResDB).each do |error|
      puts error.message
      errors << " #{error.message} "
      if error
        i += 1
      end
    end
    expect(i).to eql_with_msg(0, "Errors: #{errors}    ")
  end
end
```