TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Tarkvaratehnika õppetool

UI-testide parandamine ja loomine, kasutatud tööriistade ja testtsükli ajakulu analüüs AS Eesti Telekomi veebirakenduse näitel

Bakalaureusetöö

Üliõpilane: Rauno Sams
Üliõpilaskood: 120937IAPB
Juhendaja: Jekaterina Tšukrejeva

Tallinn 2015

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.

(*kuupäev*) (*allkiri*)

Annotatsioon

Töö eesmärgiks on analüüsida kasutajaliidese kaudu tarkvara testimist. Töös uuritakse valesti kirjutatud testide parandamist ning uute testide loomist. Samuti uuritakse UI-testitsükli pikkusega seonduvaid probleeme ning kuidas neid saaks leevendada. Testide parandamisel käsitletakse mitut sorti probleeme, mis võivad põhjustada testi ebaõnnestumist juhul, kui tarkvara ise on korras. Samuti uuritakse juhtu, kus testidest avaldub mingil moel tarkvara enda viga. Testide loomisel leitakse kasutusjuhud ühele tarkvara osale ning seejärel kirjeldatakse neile testide loomist. Kasutatavate tööriistade puuduste analüüsis tuuakse välja töö käigus esinenud puudused ning soovitused antud puuduste likvideerimiseks. Testitsükli pikkuse analüüsimisel uuritakse testide parandamise ja loomise mõju kõigi testide jooksutamisele kuluvale ajale. Samuti uuritakse variante selle ajakulu vähendamiseks nii testi kui testitsükli tasemel.

Lisaks eeltoodud eesmärgile on käesoleval tööl ka teine eesmärk – olla õppematerjaliks ning näiteks tulevastele või algajatele UI-testijatele. Töös on suur rõhk testide parandamisel, kuna oma olemuselt on kasutajaliidese automaattestid ebastabiilsed ning reageerivad ka näiliselt tühistele muudatustele. Töös käsitletud kasutajaliidese testide loomise osa on heaks näiteks, kuidas leida iseseisvat tarkvara tükki, mida testida. Samuti leitakse kasutusjuhud, mida peaks testima. Lisaks on kasutajaliidese testimise suur probleem ajakulu, mille optimeerimist peaks iga UI-testija oskama.

Loodud töö tulemusi on mitu. Esiteks sai töö käigus kaetud suur osa AS Eesti Telekomi veebirakendusest UI-testidega. See tagab, et lõppkasutaja kogemus üle terve veebi jääb soovitud kujule, kui taustal toimuvad arendused. Teiseks tulemuseks on ajakulu analüüs. See näitab selgelt, et antud töös kirjutatud UI-testid vähendavad keskmist aega testi kohta, mis kulub testtsükli jooksutamisele. Kolmandaks sai loodud see töö kui õppematerjal või näidis kasutajaliidese testijatele. See töö hõlmab osi UI-testimisest, mida on reaalselt IT firmas vaja olnud. Seega on see väga kasulik abivahend algajatele UI-testijatele.

Lõputöö aluseks on praktikakohal Eesti Telekomis veebiarenduse grupis tehtud töö UI-testidega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 62 leheküljel, 8 peatükki, 13 joonist, 2 tabelit.

Abstract

The aim of this thesis is to provide an overall analysis of user interface testing. The thesis will cover topics like how to find fixes to broken UI-tests and how to create new UI-tests for your software. Moreover, the thesis will include an analysis of how a UI-testcycle's duration changes and what causes it, suggestions to fixes will be provided. The topic of fixing tests will cover multiple problems that may occur in tests when the software itself is fine. In addition, the case where the process of testing reveals a fault in software will also be covered. In the topic of creating tests use cases for a piece of software will be found and analysed. Tests will be created for those use cases. Analysis of defects and missing functionality in testing tools will describe common problems that occured during the creation of this thesis, suggestions for solutions will be provided. The analysis of test cycle duration will look into how fixing and creating new tests affects the length of the whole test cycle.

In addition to the previously mentioned purpose of this thesis there is a secondary objective – to be used as educational material for aspiring or entry-level UI-testers. There is a big emphasis on the fixing of UI-tests as these types of tests are highly unstable and react to seemingly minuscule changes in software. The section of the thesis on creating new user interface tests is meant as an example of how to find an independent part of software to test. Moreover, use cases that need to be tested will be found. A big problem with UI-tests is their duration, thus optimizing their duration is something every UI-tester should be able to handle.

Multiple results were obtained from this thesis. Firstly, a large part of the Eesti Telekom web application was covered with UI-tests. This assures, that the experience of the end user throughout the website remains the same independent from developments done behind the scenes. Second important result is the analysis of the time spent on running a UI-test cycle. It is clearly shown that the UI-tests that were created as a part of this thesis lower the average time taken per test in the cycle. Thirdly, this thesis functions as an educational material or an example to user interface testers. This thesis covers the topics of UI-testing that were actually needed when working on UI-tests in an IT company. Thus it is a highly useful tool for beginner UI-testers.

The thesis is based on work with UI-tests done as a trainee at Eesti Telekom.

The thesis is in Estonian and contains 62 pages of text, 8 chapters, 13 figures, 2 tables.

Lühendite ja mõistete sõnastik

ET	***AS Eesti Telekom*** Eesti Telekom lühendatult.
UI	***User Interface*** Kasutajaliides, mille kaudu kasutaja rakendusega suhtleb, tavaliselt graafiline.
Bamboo	***Bamboo*** Atlassiani pidevintegratsiooni tarkvara.
API	***Application programming interface*** Rakendusliides. Selle kaudu toimub suhtlus rakenduste vahel.
Liferay	***Liferay*** Ettevõtetele mõeldud portaalitarkvara, mis põhineb *portlet*-tehnoloogial.
Portlet	***Portlet*** Tarkvara iseseisev osa, mida saab lisada portaali mis tahes lehele.
Git	***Git*** Versioonihaldustarkvara, millega saavad arendajad ühiselt arendusi teha.
Gradle	***Gradle*** Automatiseerimistarkvara, millega saab kirjeldada projekti konfiguratsiooni ning luua automatiseeritud ülesandeid.
Apache Tomcat	***Apache Tomcat*** Avatud lähtekoodiga veebiserver.
.war	***Web application archive*** Arhiivifail, mida kasutatakse veebirakenduse jagamiseks, näiteks veebiserverile.

PostgreSQL	***PostgreSQL*** Objekt-relatsiooniline andmebaasihaldussüsteem
URL	***Uniform resource locator*** Internetiaadress, mille abil saab viidata internetis olevatele dokumentidele ja failidele.
Liquibase	***Liquibase*** Rakendus andmebaasimuudatuste rakendamiseks ning jälgimiseks.
Log4j	***Log4j*** Logimise teek Java rakendustele.
Slf4j	***Slf4j*** Logimise teek Java rakendustele.
Selenium	***Selenium*** UI-testimise tarkvara, millele on saadaval API ka keeles Java.
WebDriver	***WebDriver*** Lüli veebibrauseri ning koodi vahel, mis vahendab informatsiooni ja käske nende vahel. Osa Seleniumi tarkvarast.
jQuery	***jQuery*** JavaScripti laialt kasutatav teek, mis suurendab JavaScripti võimekust.
Selenide	***Selenide*** Teek, mis lihtsustab Seleniumiga testide loomist ning lisab sellele funktsionaalsust.
TestNG	***TestNG*** Testimisraamistik, mille eesmärgiks on lihtsustada paljusid automaattestimisega seotud tegevusi.
Annotatsioon	***Annotation*** Java kontekstis spetsiaalne @-prefiksiga märksõna, millega määratakse muutuja või meetodi omadusi.

Kompileerimine, ehitus	***Build*** Tarkvaraarenduse kontekstis tarkvara lähtekoodi ning muude andmete kompileerimine jooksutatavaks tarkvaraks.
Teesklus-, Liba-,	***Mock*** Testimise kontekstis suhtluse teesklemine välise tarkvara või serveri ja testitava tarkvara vahel.
MockServers	***MockServers*** Java teek väliste API-dega suhtlemise *mock*'imiseks.
Vahemälu	***Cache*** Failid kõvakettal, kust laetakse tihti kasutatavaid andmeid, et neid ei peaks välisest serverist iga kord alla laadima.
Silumis-, silumine	***Debug*** Tarkvaras esineva vea esile kutsumine, vea tuvastamine, parandamine.
Vea aruanne	***Bug report*** Aruanne, milles testija või muu vea esile kutsuja kirjeldab tegevusi, mis vea välja kutsusid ning võimalusel kirjeldab ka vea põhjust.
Ühiktest	***Unit test*** Tarkvara mingi võimalikult väikese funktsionaalse osa testimiseks loodud kood.
Stack trace	***Stack trace*** Tarkvaravea tekke järgne info sellest, kus koodis antud viga esines.
Jada	***Array*** Andmestruktuur, kus andmed on jadana ning neile saab viidata nende järjekorranumbri järgi.
JSON	***JavaScript Object Notation*** Andmete edastamise formaat.

Pakendaja ***Wrapper***

Teek, mis vahendab sobiva liidese kaudu mingile olemasolevale teegile päringuid ja käske.

Rippmenüü ***Dropdown menu***

Menüü, millele vajutades avaneb nimekiri valitavate väärtustega.

Valik-element ***Select element***

HTML-i implementatsioon rippmenüüst.

Sisend-element ***Input element***

HTML-i kontekstis andmete sisestamiseks mõeldud element, üldjuhul kasutusel vormides.

Reageeriv ***Responsive***

Märksõna, millega iseloomustatakse tarkvara, mis suudab vastavalt kuva suurusele kasutajaliidest mugavamaks muuta.

Vorm-element ***Form element***

HTML-i kontekstis veebivorm andmete saatmiseks serverile, sisaldab sisend-elemente.

Puhas kood ***Clean code***

Põhimõtete kogum, mida järgides peaks arendaja kood olema kergesti mõistetav ning hallatav teiste arendajate poolt.

Modulaarne ***Modular***

Märksõna iseloomustamaks midagi, mis on iseseisev või mida saab kokku panna iseseisvatest osadest. Näiteks Liferay leht on modulaarne, kuna sellele saab iseseisvaid *portlet*'e lisada ja eemaldada.

Esikomponent ***Front-end***

Kasutajale nähtav osa ning tarkvara osa, mis tegeleb andmete hankimisega kasutajale.

Äärmuslik juht ***Corner-case***

Juht, mis võib olla harva esinev või mida saab väga äärmuslike

	sisenditega tarkvaras esile kutsuda.
Voog	***Flow*** Tarkvara mingi protsess, mida kasutajad läbivad tarkvara kasutamisel.
href	***Hypertext Reference*** HTML-i lingielemendi atribuut, mis määrab aadressi või tegevuse, mis käivitada elemendi aktiveerimisel klikiga.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Tarkvara testimine on väga vajalik tarkvara kvaliteedi tagamiseks. Kui tarkvaral on ka kasutajaliides, on kindlasti oluline katta see kasutajaliidese ehk UI-testidega. Probleemiks nii koodi kui kasutajaliidese testimisel on see, et sageli ei kontrollita, kas test tõesti toimib – testitav rakendus võib olla täiesti funktsionaalne, kuid test ise on kirjutatud vigaselt või hooletult lihtsalt firma nõude täitmiseks. Tihti jäetakse testid üldse kirjutamata, sest mõne arendaja suhtumine võib olla, et testide jaoks on palgatud testijad. Sellest tulenevalt võib rakenduse testide kogumik olla lünklik ja vigane ning aja möödudes kaob kõigil arendajatel ja testijatel tahtmine oma aega kulutada nende probleemide likvideerimiseks.

Tarkvarale testide kirjutamise kasulikkus ei ilmne alles siis, kui mingi tarkvara osa tulevikus vigaseks muutub. Testide kirjutamise käigus leiab suure tõenäosusega mõne kasutusjuhu, mis olemasoleva koodiga on vigane. Seega on testid mitte ainult vigade leidmiseks tulevaste arendustega vaid ka vigade ennetamiseks käesoleva arenduste kogumiga. Arendaja ei oska aga sellist testide kasulikkust mõista enne, kui ta on korra sellisel meetodil mõne vea tarkvaras leidnud.

1.1 Taust ja probleem

Töö eesmärgiks on selle lugejale edasi anda tarkvara testimise vajalikkus UI-testimise näitel. Töös kirjeldatakse, kuidas parandada teste ning kuidas luua uusi teste. Samuti analüüsitakse probleeme kasutatud tööriistadega ning uuritakse ka testtsüklile kujuvat aega.

Töö on kindlasti kasulik algajatele testijatele ja arendajatele. Kuigi töö keskendub kasutajaliidese testimisele, võib sellest leida ideid nii arendajale oma koodi testimiseks kui ka noortele testijatele nii manuaalsel kui automaattestimisel.

Töö praktiline osa on tehtud ET-s veebiarenduse grupis, mis tegeleb ET veebi (www.telekom.ee) arendamisega.

Töö viidi läbi ajavahemikus mai 2015 – oktoober 2015. Esimene etapp oli ET veebile kirjutatud UI-testide ebaõnnestumise põhjuste leidmine ning antud testide parandamine. See etapp lõppes 10. augustil, kui pidevintegratsiooni keskkonnas Bamboo õnnestus kõigi 29 UI-

testi jooksutamine – esimene õnnestunud testitsükkel. Teine etapp peale 10. augustit oli UI-testide loomine, et kaetud oleks võimalikult suur osa ET veebirakendusest. Samas jätkus ka uuesti vigaseks muutuvate testide parandamine.

1.2 Ülesande püstitus

Töö põhilisteks ülesanneteks on:

1. Leida põhjused testide ebaõnnestumisele – kas tegu on testi vea või tarkvara veaga?
2. Analüüsida testimata osi. Luua analüüsi põhjal testiklassid tarkvara osadele, mis on testimata.
3. Kirjeldada puudusi testimisel kasutatud teekides ja klassides ning pakkuda välja lahendusi.
4. Hinnata testitsükli pikkuse muutumist sõltuvalt testide hulgast ning leida viise kaasneva ajakulu vähendamiseks.

1.3 Metoodika

Töös kirjeldatakse meetodeid, mida reaalselt kasutati ülesannete 1 ja 2 saavutamiseks. Ülesannete 3 ja 4 olemused ilmnesid ülesannete 1 ja 2 täitmisel. Eesti Telekomi UI-testid käivad regressioontestide alla – seega kasutatakse meetodeid, millega isoleeritakse veebirakendus välistest API-dest ja andmebaasidest. ET veeb ning selle UI-testid on loodud keeles Java.

1.4 Ülevaade tööst

Esmalt tutvustatakse testitava rakenduse üldist struktuuri. Kirjeldatakse lühidalt erinevaid klasse ning teeke, millele on antud rakendus üles ehitatud. Samuti tutvustatakse UI-testimise keskkonda üldiselt. Seejärel analüüsitakse testitavale rakendusele juba varasemalt loodud UI-teste – mõned neist on kirjutatud vigaselt ning nende jooksutamise tulemus on negatiivne. Tutvustatakse võimalusi selliste testide parandamiseks. Uute UI-testide loomiseks analüüsitakse teatud funktsionaalsuse kasutusjuhtusid ning luuakse nende alusel UI-testid.

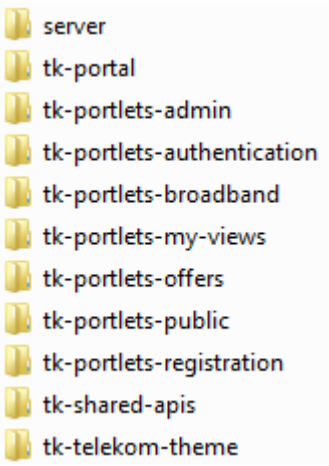
Seejärel kirjeldatakse leitud puudusi UI-testimisel kasutatud tööriistades. Lõpuks uuritakse testitsükli jooksutamise ajakulu – mis on selle põhjused, kuidas seda vähendada?

2. Eesti Telekomi veebirakenduse üldstruktuur

ET veeb on üles ehitatud tarkvaral Liferay, versioon 6.2. Liferay kasutab *portlet*-tehnoloogiat – veebilehele saab läbi administraatori paneeli lisada *portlet*'e ehk väikeseid iseseisvaid tarkvara osi. ET lehel on näiteks eraldi *portlet* selleks, et kontrollida telefonide musta nimekirja. Selle võib lisada mis tahes lehele ning sõltuvalt *portlet*'i struktuurist võib seda lisada kas 1 või mitu ühele lehele. ET vajadustest lähtuvalt on selle veebis ka *portlet*'e, mis on igal lehel olemas, näiteks autentimise ehk sisse logimise *portlet*. Järgnevates peatükkides tutvustatakse testitava rakenduse üldist struktuuri, raamistikke, teeke ning ka rakenduse testimiseks kasutatavate raamistike ja teekide kogumikku.

2.1 Veebirakenduse struktuur

ET veebirakendus jookseb portaalitarkvaral Liferay, mille keskne osa on *portlet*-tehnoloogia. Suur osa Liferay vaikefunktsionaalsusest on muudetud või eemaldatud, et rakendus toimiks just ettevõtte vajadusi arvestades. Koodi tasemel on veebirakendus jaotatud mooduliteks funktsionaalsuse põhjal. Iga moodul on eraldi Git-i repositooriumis.



Joonis 1 Rakenduse lähtekoodi kaustastruktuur

Rakendus kasutab automatiseerimise tööriista Gradle. See võimaldab lihtsustada suurel hulgal tegevusi. Liferay projekti käivitamine eeldab, et kõikidest moodulitest on ehitatud .war failid ning need on edasi antud Tomcat serverile. Mooduli .war faili ehitamiseks ning serverile edasi andmiseks on näiteks käsk

```
gradlew deploy
```

Käsk kompileerib mooduli klassid, loob neist .war faili ning kopeerib selle serveri kausta.

Joonis 2 Avalike *portlet*'ide mooduli *deploy*

Serveri käivitamise eelduseks on PostgreSQL andmebaasi algsete andmete kandmine. Nendeks andmeteks on näiteks konfiguratsiooniseaded – API-de URL-id, erinevad seadistusparameetrid. Samuti on vaja kanda andmebaasi tekstiread – kõik rakenduses kuvatav tekst on kantud andmebaasi koos tõlgetega. Selleks on kasutusel teek Liquibase. See võimaldab pidada järge andmebaasi seisu ja muudatuste üle ning lihtsustab muudatuste andmebaasi sisse viimist. Selle käivitamiseks on kasutusel Gradle käsk

```
gradlew update
```

Käsk laseb peale uusimad teksti- ja konfiguratsiooniväärtused. Seda saab kasutada ka juba olemasolevate väärtuste uuendamiseks, kui projekti on lisandunud Liquibase muudatusi.

Rakenduses on kasutusel väga põhjalik logimissüsteem teekidega log4j ja slf4j. Antud logi abil on võimalik peaaegu kõigi veebirakenduses esinevate vigade põhjuste leidmine. Iga tund luuakse uus logifail ning eelnevalt loodud logi nimetatakse ümber kajastamaks ajavahemikku, mil antud logisse kirjutati.

2.2 Testimissüsteemi struktuur

Kasutajaliidese testimisel on kasutusel mitmed Java teegid. Põhiline teek UI-testimiseks on Selenium, mis võimaldab koodiga juhtida veebibrauserit. Selle tuumaks on WebDriver, mis on liides vahendamaks koodiga antud käske veebibrauserile. ET veebi UI-testides kasutatakse Mozilla Firefox brauserit ning selle WebDriver implementatsiooni FirefoxDriver.

Seleniumi kasutamise lihtsustamiseks on kasutusel Selenide, mis täiendab ja lihtsustab Seleniumi kasutamist. Näiteks võimaldab see kasutada meetodit $. Selenide meetod $ toimib sarnaselt Javascript teegi jQuery omale, võimaldades kasutada CSS selektorit või XPath selektorit, et valida kindlat elementi HTML-dokumendis. Andes meetodile $ ette String-tüüpi parameetri, käsitleb Selenide vaikimisi seda kui CSS selektorit [1]. Selenide võimaldab ka luua kuvatõmmiseid ning salvestada HTML-lähtekoodi hetkest, mil mõni test ebaõnnestus.

Kasutusel on ka teek TestNG, mille funktsionaalsuset on kasutusel testitulemuste raporti genereerimine ning selle annotatsioonid, mis töötavad sarnaselt JUnit annotatsioonidele - @Test, @BeforeClass, @BeforeMethod, … .

Kuna UI-teste jooksutatakse Mozilla Firefox veebibrauseris, on vajalik ka selle eelnev olemasolu, kuna Selenium ega Selenide seda ise ei paigalda. ET sisemise Wiki leht UI-testimise kohta ütleb järgnevat: „Browseriks sobib nt Firefox versioon 33 ja 34. Uuemate Firefoxidega ei pruugi testid käima minna (tuleb error: Unable to connect to host 127.0.0.1 on port 7055)“. Testide loomiseks ja parandamiseks kasutatud arvutis oli kasutusel Mozilla Firefox 40.0.2 ja Waterfox 39.0 ning nendega probleeme ei olnud.

Tähtsaim osa UI-regressioontestide juures on rakenduse välistest serveritest eraldamine. Et rakenduse funktsionaalsus säiliks, on kasutusel teek MockServer. See võimaldab käivitada seadmes serveri, mis vastab päringutele koodis ette antud viisil.

Testide jooksutamiseks on kasutusel pidevintegratsiooni rakendus Bamboo. Selles rakenduses on konfigureeritud iga 3 tunni tagant jooksev automaatne *build project*, mis laeb alla rakenduse uusima versiooni, paneb selle tööle ning jooksutab moodulhaaval kõiki UI-teste. Bamboo rakendus on aga selle jooksutamise juht – tegeliku serveri käivitamise ning testide jooksutamise teevad selleks üles seatud Bamboo agendid. Need on virtuaalmasinad, mis iga 3 tunni tagant saavad Bamboo rakenduselt käsu jooksutada UI-testide *build project*’i. Spetsiaalselt UI-testide jooksutamiseks on kasutusel 2 virtuaalmasinat – zorro4 ja zorro5. Varasemalt oli kasutusel vaid zorro4, kuid testide hulga suurenedes tekkis vajadus ajakulu vähendamiseks konfigureerida zorro5 ümber, et see suudaks jooksutada UI-teste.

UI-testide jooksutamiseks on ka projektis loodud mõningad abi-klassid, mis teevad igas testis või suures osas testides kasutatavad tööd ära. Seega puudub vajadus neid funktsionaalsusi realiseerida igas testiklassis eraldi. Järgmises peatükis kirjeldatakse spetsiaalselt rakenduse UI-testide jaoks loodud abistavad klassid ning milleks neid kasutatakse.

2.3 Veebirakenduse UI-testide jaoks loodud abistavad klassid

Antud nimekiri on klassidest, mis on loodud spetsiaalselt antud rakenduse UI-testide kirjutamise lihtsustamiseks. Nimekirjas olevatest klassidest on ainult üks loodud antud diplomitöö praktilise osa käigus, muud klassid on varasemalt loodud firma töötajate ja partnerite poolt.

- MockServers – klass sisaldab infot selle kohta, millised serverid tuleb *mock*ida. Kuna väliste serverite URL-id on salvestatud rakenduse kohalikku andmebaasi, siis salvestab MockServers testile eelnevad URL-id, käivitab kohalikud *mockserver*id ja kirjutab üle andmebaasis olevad URL-id kohalike serverite URL-idega. Peale testi peatatakse kohalikud *mockserver*id ja taastatakse URL-id andmebaasis.

- PostgreDBcommunication – kõik rakenduses kuvatavad tekstid on salvestatud rakenduse kohalikus andmebaasis. Et kontrollida, kas mingi tekst on testis kuvatud, on korrektne vajalik tekst hankida andmebaasist ning otsida seda lehelt. Tihti kirjutatakse tekst otse testi sisse, mis võib teksti muutumisel põhjustada vigu testis. PostgreDBcommuncation võimaldab hankida tekste võtme järgi.

- ReadGradleProperties – Gradle seadistus on salvestatud arvuti kasutaja kodukausta alamkausta .gradle nimega gradle.properties. ReadGradleProperties võimaldab hankida antud failist kohaliku andmebaasi nagu sisse logimiseks kasutatav nimi ning parool.

- SuiteManagement – seda klassi *extend*ivad kõik UI-teste sisaldavad klassid. See sisaldab meetodeid, mis on märgistatud @BeforeMethod ja @AfterMethod annotatsioonidega ehk see klass sisaldab tegevusi, mida tuleb teha enne või pärast iga testi. Nendeks tegevusteks on *mockserver*ite käivitamine, brauseri avamine, brauseri sulgemine ja *mockserver*ite peatamine. Antud klass ei lähe ühelegi URL-ile, vaid avab lihtsalt brauseri. Kindlale URL-ile liikumine on iga testi vastutus.

- Login – antud klass tegeleb sisse logimisega ja selle jaoks vastavate *mock*-vastuste seadistamisega. Klass sisaldab meetodit, mis Selenide abil avab brauseris rakenduse URL-i ning teeb läbi kasutaja sisse logimise protsessi.

- PortletControls – administraatoritel on võimalik Liferay *portlet*'e seadistada. Iga *portlet*'i üleval vasakus nurgas on menüünupp, mille abil saab seadistada valitud *portlet*'i. PortletControls klass teeb sellega seotud tegevusi nagu näiteks *portlet*'i eelistuste menüü avamine.

- SiteAdministrationMenu – veel üks klass administraatorina tegutsemiseks UI-testis. Administraatorina sisse logituna on rakenduse ülemises ääres administraatori paneel, antud klass teeb vastavaid tegevusi, et selle kaudu avataks sisuhalduse lehekülg.

- WebContentAddMenu – *portlet*'ide lehele lisamise menüü avamiseks kasutusel olev klass.

- Cache – *portlet*'id hoiavad tihti infot vähemälus ehk *cache*'s. Et testis läbi viidud muudatused ilmuksid *portlet*'is, kasutatakse seda klassi vahemälu tühjendamiseks.

- Configuration – klass konfiguratsiooniseadistuste leidmiseks ning muutmiseks. Kasutatakse *mockserver*ite aadresside kirjutamiseks andmebaasi.

- Layout – kasutatakse etteantud *portlet*'i etteantud URL-ile lisamiseks. Kui kindlat URL-i ette ei anta, luuakse lehe URL nii:

```
String friendlyURL = "/uitest-" + System.currentTimeMillis();
```

- ResetBaseUrls – klass, mis sai loodud antud töö praktilise osa käigus. Klass taastab andmebaasis originaalsed API-de URL-id. Vajadus klassi järgi tekkis, kui UI-testide *debug*imisel kasutati testi brauseriakna lahti hoidmiseks lõpmatuid tsükleid. Seetõttu ei jooksutatud brauseri sulgemisel meetodit andmebaasis algsete API URL-ide taastamiseks.

3. Kasutajaliidese testide parandamine

Testide parandamise tähtsaim osa on teha kindaks, kas vigane on test või tarkvara, mida testitakse. Selle välja selgitamine kindlustab, et parandusi ei hakata tegema valel pool ning tulemusena hoitakse kokku väärtuslikku aega.

Et teha kindlaks, kas vigane on test, tuleks rakendust käsitsi testida. Läbides samm-haaval testimeetodi tegevused, saab selgust, kas tarkvara tegelikult töötab.

Kui ilmneb, et test pole vigane, siis tuleb uurida rakendust. Kuna korrektses regressioontestis ei toimu suhtlust ühegi välise serveriga, saab välistada probleemid ühendusega. Siis tuleb leida viga testis peituvate vihjete abil.

3.1 Teoreetilised alused

Automaattestide ebaõnnestumise analüüsi puhul on tähtsaim teha kindlaks, kas testitav tarkvara ise on vigane või mitte. Selleks tuleb kasutada manuaalset testimist antud tarkvarale, et veenduda selle veatus töötamises. Oma artiklis automaattestimise tööriistade kohta kirjutab Jennifer Lent, et kõige edukamad automatiseerimise projektid on need, mis ei ole täielikult autopiloodil [2]. Antud artikli kontekstis tähendab see, et automaattestimine ja manuaalne testimine on mõlemad vajalikud tegevused ning täielikult automaatse testimise peale minek ei ole tark. See idee on rakendatav ka automaattestide haldamisel. Kui pole olemas isikut, kes ebaõnnestunud automaattesti üle vaatab ning kasutusjuhu manuaalselt üle testib, võib tekkida ekslik arvamus, et vigane on tarkvara.

Automaattestide haldajale on tähtsaks tööriistaks pidevintegratsiooni keskkond. Üheks selliseks on Atlassian Bamboo. Pidevintegratsiooni tööriistad jooksutavad automaatteste regulaarselt kõige uuemal tarkvara koodil. Automaattestid tuleb teha korda võimalikult kiiresti, kuna uute ebaõnnestumiste korral probleemid kuhjuvad. Atlassiani veebileht väidab, et kui ebaõnnestuma hakkab vaid üks test, siis on võimalik arendajate tähelepanu kergemini võita. Paljude testide ebaõnnestumise korral aga arendajate efektiivsus probleemile reageerimisel väheneb [3]. Seetõttu ongi kasulik kasutada pidevintegratsiooni tööriistu, kuna see süsteem annab arendajatele märku juba esimese vea esinemisel. See kehtib nii tarkvara

vigade korral kui ka testide vigade korral – automaattestide kirjutaja peab siiski oskama kirjutada ja analüüsida koodi, seega on ka tema näol tegu arendajaga.

Automaattestide ebaõnnestumisel on väga tähtis tööriist logi. Automaattesti korral on logisid kahte tüüpi – tarkvara logi, mis näitab tarkvaras toimuvat ning testi logi, mis näitab testi ebaõnnestumise kohta. Artiklis „4 Reasons to love Your Log Data“ on kõige esimese punktina välja toodud logide kasulikkus tarkvara probleemide tuvastamisel [4]. Peatükkides 3.2 ja 3.3 käsitletavates teemades ilmneb see kasulikkus samuti – peaaegu kõik neis peatükkides analüüsitud probleemid on tuvastatud tuginedes logides sisalduvale infole. Seetõttu peab olema logimissüsteem võimalikult mahukas, et probleemi asukoht võimalikult täpselt kindlaks teha.

Kes peab aga tegelema vigade likvideerimisega? Esmapilgul paistab vastus lihtne – automaatteste parandagu testija, tarkvaravigu parandagu arendaja. Nagu eelnevalt mainitud, on ka automaattestide looja ning haldaja arendaja, kuna automatiseerimine nõuab programmerimisoskust. Siiski on reaalsus see, et arendajatel on käsil palju tööd ning vigane tarkvara osa on loodud kuude või isegi aastate eest ning võimalik, et hoopis mõne muu, asutusest lahkunud arendaja poolt. Kui automaattestija järgib kvaliteetse *bug report*'i põhimõtteid, siis kindlasti kasutab ta oma arendaja oskusi, et analüüsida ka koodi ning leida sealt konkreetne veakoht. Artiklis „How to Write a Quality Bug Report“ kirjutatakse, et tähtsaimad punktid *bug report*'is on pealkiri, vea esile kutsumise sammud, oodatud ja tegelikud tulemused ning muud lisad, mis aitavad arendajal viga üles leida [5]. Põhjaliku analüüsi tulemusena on automaattestija-arendaja suuteline leidma koodis täpse veakoha, seega võib jätta ka vea parandamise vastutuse tema kanda. Samamoodi peaks arendaja parandama automaattesti, mille tema arendus katki tegi – tema on oma loodud arendusega kõige paremini kursis ning peaks olema suuteline automaattesti ümber töötada nii, et see arenduse järgselt toimiks.

3.2 Vigaselt loodud või vigaseks muutunud testid

Unit-testide parandamine on üldjuhul selle arendaja ülesanne, kelle arenduse tagajärjel antud test vigaseks muutus. Need testid jooksevad kiiresti ja ainult koodi tasemel. Kasutajaliidese kaudu rakenduse testimine on aga pikk protsess. Näiteks veebirakenduse UI-testimisel peab kood käivitama brauseri, avama õige lehekülje ning läbima seal kasutusjuhu nagu seda teeks

rakenduse tavakasutaja. Seetõttu jätavad arendajad tihti UI-testid puutumata. Järgnevalt analüüsitakse sagedasi veebirakenduse UI-testide ebaõnnestumise põhjuseid ning parandust.

3.2.1 Vale või puudulik *mock*-vastus

Mahukad veebirakendused vajavad kindlasti API-sid, kuna rakendus ja andmebaasid on üldjuhul eraldi serverites. UI-testide korral oleks mõistlik suhtlus väliste serveritega keelata. Selleks on olemas teek MockServers, millega saab kinni püüda välisele serverile mõeldud päringu ning vastata sellele ise just sellise vastusega, nagu vaja. Suure rakenduse korral on aga API-sid ja nendele tehtavaid päringuid palju ning sellest tulenevalt võib puuduolev *mock*-vastus terve testi läbi kukutada.

Esimene koht, kust sellise vea analüüsimist alustada, on rakenduse logid. Korraliku logimise korral on iga vea *stack trace* logis olemas koos kellaajaga, mil viga esines. Sealt saab juba kätte URL-i, kuhu päring tehakse ning kui ilmneb HTTP protokolli veakood 404, on tegu puuduliku *mock*-vastusega.

```
WARN |rest.BaseUrlRestTemplate|GET request for
"http://localhost:1340/rest/Customer/api/offerings/customers/1234567"
resulted in 404 (Not Found); invoking error handler|
ERROR |service.ApiServiceImpl|Unable to get offers from api |
org.springframework.web.client.HttpClientErrorException: 404 Not Found
    at
org.springframework.web.client.DefaultResponseErrorHandler.handleError(Defa
ultResponseErrorHandler.java:91)
    at
org.springframework.web.client.RestTemplate.handleResponseError(RestTemplat
e.java:615)
    at
org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:573
)
...
```

Antud juhul on logist koheselt näha, millise URL-iga tekib probleeme. See ongi puuduolevaks *mock*'iks. Järgnevalt tuleb tutvuda reaalse vastusega, et oleks võimalik luua *mockresponse* vastavalt testi vajadustele.

Pakkumiste *portlet*'ile, mis antud viga annab, on küll UI-testid varasemalt loodud koos korrektsete *mock*'idega, kuid väljaspool neid kindlaid teste oli *mock*-imine puudulik. Seega kui mõnes testis, mis testis mingit muud tarkvara osa, oli vaja sisse logida ning kliendi esilehele kasvõi hetkeks jõuda, prooviti kuvada ka pakkumiste *portlet*'i. Väljaspool pakkumistele mõeldud UI-testi *mock*-vastused sellele *portlet*'ile puudusid ning logidesse

ilmusid tihti *stack trace*'id, mis ei näidanud tegelikule veale tarkvaras, vaid puudulikule *mock*'imisele.

Seega, kuna viga ilmnes väljaspool seda tarkvara osa, mida testitakse, võib *mock*'i vastus olla võimalikult minimaalne, et lihtsalt vältida logide risustamist mitte midagi ütlevate *stack trace*'idega. Asendades eelnevalt leitud URL-is localhost:1340 reaalse API domeeniga ning kasutaja ID 1234567 reaalse kasutaja ID-ga, saame teha reaalse päringu API-le, kust saame korrektse süntaksiga vastuse. Ilmnes, et antud kasutajale pole ühtegi kehtivat pakkumist. API vastuseks oli vaid tühi JSON-*array* ehk märgid **[ ]**. Seega võib ka *mock*'i vastuseks määrata tühja JSON-*array*. Seda muidugi juhul, kui vastuse sisu pole meile oluline. Selline vastuse analüüs on hea, kui arendaja või testija ei ole tuttav API sisemise tööga ega sellega, kuidas rakendus API-delt andmeid pärib ja neid töötleb. Keerulisemate API-vastuste puhul oleks mõistlik enne *mock*-vastuse kasutamist koodis selles andmed asendada nii, et need sobiks testis kasutatava kasutaja andmetega.

Edasi tuleb Login.java klassis sisse tuua äsja leitud *mock*-vastus logist leitud *mock*-serveri URL-ile. Selleks piisab järgnevast koodist:

```
MockServers.crmApiMockServer
        .when( HttpRequest.request(
"/rest/Customer/api/offerings/customers/" + customerCode ) )
        .respond( MockServers.json200Response().withBody( "[]" ) );
```

Meetod json200Response() paneb vastuse päisesse info selle kohta, et tegu on JSON-formaadis andmetega.

Kontrollimaks, kas antud *mock* parandas probleemi testides, mis ei testi pakkumiste funktsionaalsust, tuleb käivitada taas mõni sellistest testidest ning otsida logist pakkumistega seotud read:

```
DEBUG|rest.BaseUrlRestTemplate|GET request for
"http://localhost:1340/rest/Customer/api/offerings/customers/1234567"
resulted in 200 (OK)
```

HTTP-protokolli vastuse kood 200 näitab, et päring andis vastuse ning logis ei esine enam selle kohta vigu.

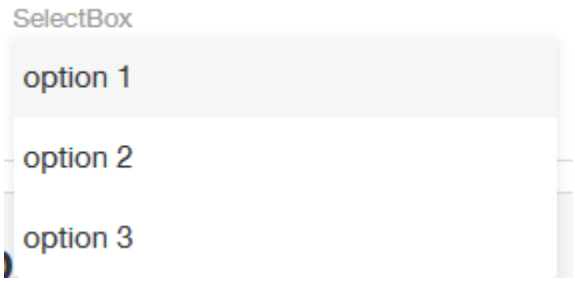
3.2.2 UI-testimise teekide piirangutest ja rakenduse omapäradest tingitud vead

3.2.2.1 Nähtamatu HTML-elemendiga tegevused

UI-testimise teegi Selenium ja selle *wrapper*'i Selenide kasutamine on üldjuhul lihtne. Antakse ette element, millel mingit tegevust teha ning tegevus, mida tehakse. Siiski on mõningad juhud, mil selline lihtne lähenemine ei toimi. See on tingitud sellest, et veebirakenduste struktuur võib olla vägagi erinev ning igat võimalikku juhtu ei ole seetõttu võimalik teeki sisse kirjutada.

AS ET veebirakenduses on kasutusel emafirma TeliaSonera stiilis kujundus. Üheks selle eripäraks on selle *dropdown*-kastid. Lahendus pole küll ainulaadne, kuid UI-testimisel võib selline lahendus tekitada probleeme. Kasutajale kuvatav element on *text*-tüüpi *input*-element. Kasutajale nähtamatu on aga *select*-element ehk tüüpiline HTML-i *dropdown*-i element. Kui Selenide proovib otse antud *select*-elemendis määrata *dropdown*-valikut, ebaõnnestub test.

Joonis 3 ET kujundusega *dropdown*-menüü suletud kujul



Joonis 4 ET kujundusega *dropdown*-menüü avatud kujul

Portlet'is nimega News List, mis on kasutusel kasutajale uudiste kuvamiseks, on seadistusmenüüs olemas just selline *dropdown*-element. Sellega valitakse, kas kuvada era- või äriklientidele mõeldud uudiseid. Antud *portlet* on kaetud ka UI-testidega, üheks neist *portlet*'i seadistamine. Valik antud *dropdown*-is tehakse koodiga

```
$( By.name( "clientType" ) ).selectOptionByValue( "ARI" );
```

Kuna element clientType on nähtamatu, siis test ebaõnnestub viidates faktile, et element on nähtamatu.

Tüüpiline ET *dropdown*-element põhineb järgneval HTML-koodil:

```
<div class="select-wrapper">
```

```
    <input class="select-dropdown" readonly="true" type="text">
    <select class="initialized">
        <option>option 1</option>
        <option>option 2</option>
        <option>option 3</option>
    </select>
</div>
```

Nagu näha, siis *select*-element on koodis olemas. Test ebaõnnestub aga seetõttu, kuna see on kasutajale nähtamatu. Selenium ja sellest tulenevalt Selenide saavad teha tegevusi vaid elementidega, millega inimkasutajagi. Nähtamatus *dropdown*-menüüs valikut seega teha ei saa.

Üheks lahenduseks probleemile oleks vajutada elemendile, mis kuvab *dropdown*-menüü ehk eeltoodud HTML-näites elemendile *input*. Seejärel tuleks Selenide'l vajutada soovitud *dropdown*-menüüs kuvatavale valikule. Tegelikud valikud antud menüüs on aga HTML-dokumendis hoopis teises elemendis

```
<ul id="select-options-7eff3738-86f6-ba4b-5cc4-b506ad7a3ba2"
class="dropdown-content select-dropdown">
```

Selle elemendi ID on mingi genereeritud väärtus, seega testi sisse kirjutamiseks oleks see halb variant, sest see on muutuv. Puhtalt klassi *dropdown-content* järgi elemendile viitamine oleks samuti halb, kuna dokumendis võib esineda mitu sellist elementi. Näiteks siis, kui seadistuses on mitu *dropdown*-elementi.

Realiseeritud sai teistsugune parandus. Nimelt on võimalik Selenium/Selenide teekides jooksutada brauseris JavaScript koodi. Seega piisas eeltoodud koodirea asendamisest reaga

```
((JavascriptExecutor)driver).executeScript(
"document.getElementsByName('clientType')[0].value = 'ARI';" );
```

UI-testi kontekstis on selline elemendile viitamine aktsepteeritav, kuna me teame, et selliseid elemente on lehel vaid 1, kuna enne testi jooksutamist lisatakse lehele vaid 1 selline *portlet* ning ainult selle ainsa *portlet*'i seadistusmenüü avatakse.

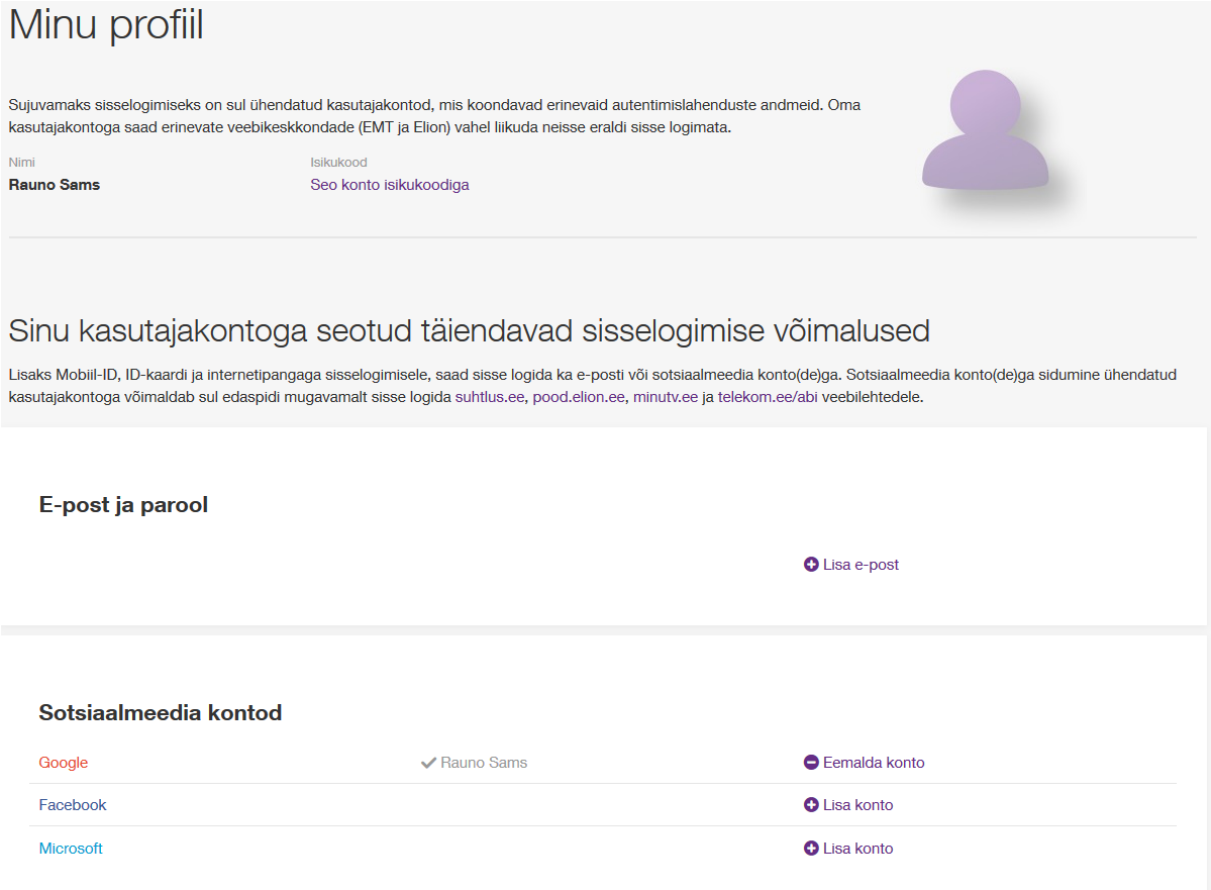
Sellist probleemi töö aluseks olevas rakenduses muudes testides esinenud pole, kuid JavaScript põhise lahendusega saaks lahendada ka paljud teised HTML-elemendiga seotud probleemid, mis tulenevad just rakenduse kujundusest või struktuurist.

3.2.2.2 Brauseri vahemällu talletatud vanad andmed

Seleniumi Java API ei võimalda UI-testis kustutada brauseri vahemälu ehk *cache*'t ning selle võimaldamine pole Seleniumi meeskonnal plaanis [2]. Paljud brauserid salvestavad harvamuutuvad andmed nagu pildid ning CSS-failide sisu vahemällu, et kiirendada korduvaid veebilehe laadimisi. Samuti võib veebirakendus ise dikteerida mingil määral, mis andmeid tuleks säilitada. Sellest tulenevalt võib aga UI-testides esineda olukordi, kus kuvatakse vanu andmeid ning seetõttu test ebaõnnestub.

Antud probleemi analüüs põhineb ET veebirakenduse *portlet*'idel Minu Profiil (*profile*) ning Minu Autentimise Meetodid (*authenticationmethods*). Järgnevad *portlet*'id said UI-testidega kaetud küll antud diplomitöö tegemise käigus, kuid kuna eelpool mainitud probleem on osaliselt põhjustatud Seleniumi Java API puudulikkusest, siis vajaks see ka peatüki 3.1.2 raames mainimist.

Antud *portlet*'id on nähtavad sisse logitud kasutajatele. ET veebilehel asuvad need aadressil https://www.telekom.ee/et/minu-profiil. Kindla kasutajaga seotud andmed nagu kasutatavad e-posti aadressid ning sotsiaalmeedia kontod laetakse kasutaja sisse logimisel ning need salvestatakse vahemällu. UI-testides lisatakse ja eemaldatakse sidumisi nii sotsiaalmeedia kui e-posti kontodega ning kontrollitakse, kas rakendus tuvastab muudatusi korrektselt. Probleem ilmnes selles, et UI-testide raames muudatusi ei kuvatud. Sel juhul oleks kasulik võimalus tühjendada brauseri vahemälu, kuid Seleniumi API seda ei võimalda.



Joonis 5 *profile* ja *authenticationmethods portlet*'id

Antud UI-testides sai kasutusel võetud lihtsaim võimalik lahendus, mille suurim puudus on, et see pikendab UI-testide jooksutamise kestust ligikaudu minuti või paari jagu. Tänapäeva populaarseimad brauserid nagu Google Chrome ja Mozilla Firefox võimaldavad kasutada erinevaid profiile, et erinevad brauseri kasutajad saaks kasutada oma brauserit just nii, nagu vaja. Et kindlasti tühjendada brauseri vahemälu, tuleb see taaskäivitada, kuna Selenium loob igal käivitamisel uue profiili. Selle kõrvalnähuks on see, et kustuvad ka brauseri küpsised, mistõttu on vajalik korduv sisse logimine. Selline protsess põhjustabki ajakulu suurenemise antud *portlet*'ide testide jooksutamisel.

3.2.3 Vale viide HTML-elemendile

Keerulise *front-end* ülesehitusega veebirakenduse puhul on üks raskemaid asju UI-testimisel leida viide elemendile, millega tahetakse tegevusi teha. Kuna HTML-i standard nõuab, et iga atribuudi *id* väärtus dokumendi (.html faili) raames on unikaalne, siis oleks lihtsaim viis kasutada antud väärtust elemendile viitamiseks. Seda toetab ka Selenium. Siiski ilmneb

probleem – tihti on ära jäetud *id* väärtustamine. Elemendid nagu nupud ja tekstiväljad, mis on tähtsad mingi *flow* läbi viimiseks lehel, peaks omama *id* väärtust. Siiski on tihti veebirakendused üles ehitatud nii, et *id* ei ole rakenduse *front-end* toimimiseks hädavajalik. Seega on ka UI-testid tihti kirjutatud ebamugavate ja halvasti loetavate viidetega elementidele, kuna otsene viitamine ei ole võimalik. Nii võib aga mõne väiksemagi kujunduse muudatuse korral esineda testi jooksutamises viga, kuna viide ei vasta enam tegelikkusele.

Digitark *portlet* on ET veebi *portlet*, mille põhiline eesmärk on kasutajale kuvada viimaseid postitusi, mis on postitatud tehnoloogiablogisse Digitark (http://www.digitark.ee). Andmed saab *portlet* Digitarga RSS-voost ning need kuvatakse välja nagu näha Joonisel 6.

Joonis 6 Digitark *portlet*

UI-testid antud *portlet*'ile eksisteerisid juba varasemast ajast, need loodi millalgi aprillis 2015. Mingil hetkel aga tehti *portlet*'i kujundusele uuendus, mille tagajärjel hakkasid selle *portlet*'i UI-testid keskkonnas Bamboo ebaõnnestuma. Kontrollides *portlet*'i ennast ilmnes, et see

töötab korrektselt. Seega tuli üle vaadata testide logid ning vajadusel ka rakenduse logid. Testi logidest ilmnes, et tegu on ühega kahest variandist – kas HTML-elementi, millele viidatakse ei eksisteeri, või on sellele viidatud valesti. Nagu eelnevast *portlet*'i kontrollist ilmnes, siis kuvatakse kõik korrektselt, seega ei tohiks olla tegu puuduva HTML-elemendiga. Testi logi andis vea põhjuseks järgneva:

```
Element not found {By.cssSelector: .digitark-block h2}
Expected: visible
```

Logi annab ka rea testis, kus viga esineb. Antud juhul on tegu järgnevaga:

```
$$( ".digitark-block h2" ).get( 0 ).shouldHave( Condition.text(...) );
```

See rida kontrollib, kas Digitarga postituse pealkiri kuvatakse korrektselt. Testis kasutatakse 2 postitust, seega kontrollitakse vastavalt kahel korral sama elementi. Seega tuleb parandus teha kahes kohas. Vaadates viimaseid *commit*'e testitavale *portlet*'ile, siis selgub, et postituste pealkirjad pole enam elemendis *h2*, vaid elemendis *h4*. Sellised elemendid muudavad teksti suurust nende sees ning neid kasutatakse üldjuhul pealkirjadena.

Muudatus, mille järel test ebaõnnestuma hakkas, on leitud. Kuna antud *portlet*'is ei saa elementidele viidata *id* kaudu, on ainsaks lahenduseks viidet kohandada uue kujundusega sobivaks. Selleks tuleb muuta eelnevalt näidatud rida testis järgnevaks:

```
$$( ".digitark-block h4" ).get( 0 ).shouldHave( Condition.text(...) );
```

Samuti tuleb muudatus teha ka teise uudise pealkirja kontrollis, kuid see muudatus on täpselt sama ning vastav rida on sellest järgmine.

Kuigi selline testi parandamine tundub kiire ning lihtne, siis ideaalis ei tohiks seda viga üldse tekkida. Hea tava oleks seega anda kõigile tähtsatele HTML-dokumendi osadele *id* atribuudi väärtused, et nendele saaks viidata näiteks UI-testides või mujal koodis. Vastasel juhul võib väiksemgi muudatus HTML-dokumendis esile kutsuda vea UI-testides, mida on küll lihtne parandada, kuid võib anda mõtlemisainet selle üle, kas antud tarkvara on ikka kvaliteetne.

3.2.4 Äriliste vajaduste muutustest tingitud vead

Aja jooksul võivad tarkvara tellija vajadused selle funktsionaalsuse muutuda. See tähendab, et olemasolevat tarkvara tuleb kohandada vastavalt tellija uutele nõuetele. Äriliste vajaduste muutus on üks sagedasemaid põhjuseid, miks UI-testid ebaõnnestuma hakkavad, kuna

muutunud funktsionaalsuse testimiseks pole UI-testid veel kohandatud. Seega esineb UI-testide ebaõnnestumine, kuna testid proovivad testida funktsionaalsust, mida sellel kujul enam ei eksisteeri.

3.2.4.1 Uudiste voo *portlet* kuvab lisaks uudistele ka teadaandeid

Uudiste voo portlet ehk *newsfeed portlet* kuvas esialgu vaid uudis-tüüpi artikleid ehk lühidalt öeldes uudiseid. Lisaks on ET veebis kasutusel ka teadaande-tüüpi artiklid ehk teadaanded. Mingil hetkel tekkis aga vajadus, et uudiste voog kuvaks ka teadaandeid, mitte ainult uudiseid. Antud *portlet*'i testivad UI-testid lõid süsteemis kaks uudist ning kaks teadaannet. Tulenevalt *id* atribuudi puudumisest uudiste voo uudiste nimekirjas viidati nende kasutajaliidese kuvale järgnevalt:

```
$( ".home-news-list li:nth-child(1) a" ).shouldHave( text( secondNewsTitle ) );
$( ".home-news-list li:nth-child(2) a" ).shouldHave( text( firstNewsTitle ) );
```

Kuna uuemad uudised kuvatakse eespool, siis teisena loodud uudis kuvatakse eespool ning seega otsitakse seda viitega nth-child(1). Teadaanded loodi antud UI-testis seetõttu, et olla kindel, et neid ei kuvataks koos uudistega.

Uute äriliste vajaduste implementeerimisega hakkasid antud *portlet*'i testivad UI-testid ebaõnnestuma. Nimelt ei olnud viitega li:nth-child(1) elemendi tekstiks mitte teisena loodud uudise pealkiri, vaid teisena loodud teate pealkiri, mida nüüd samuti selles *portlet*'is kuvati. Testide parandamine oli aga lihtne, kuna korrektne kuvamine oli siiski testitav – nimelt loodi uudised ja teated vaheldumisi: esimene uudis, esimene teade, teine uudis, teine teade. Seega sai asendada eelpool toodud koodiread testis järgnevalt:

```
$( ".home-news-list li:nth-child(1) a" ).shouldHave( text( secondAnnouncementTitle
) );
$( ".home-news-list li:nth-child(2) a" ).shouldHave( text( secondNewsTitle ) );
$( ".home-news-list li:nth-child(3) a" ).shouldHave( text( firstAnnouncementTitle )
);
$( ".home-news-list li:nth-child(4) a" ).shouldHave( text( firstNewsTitle ) );
```

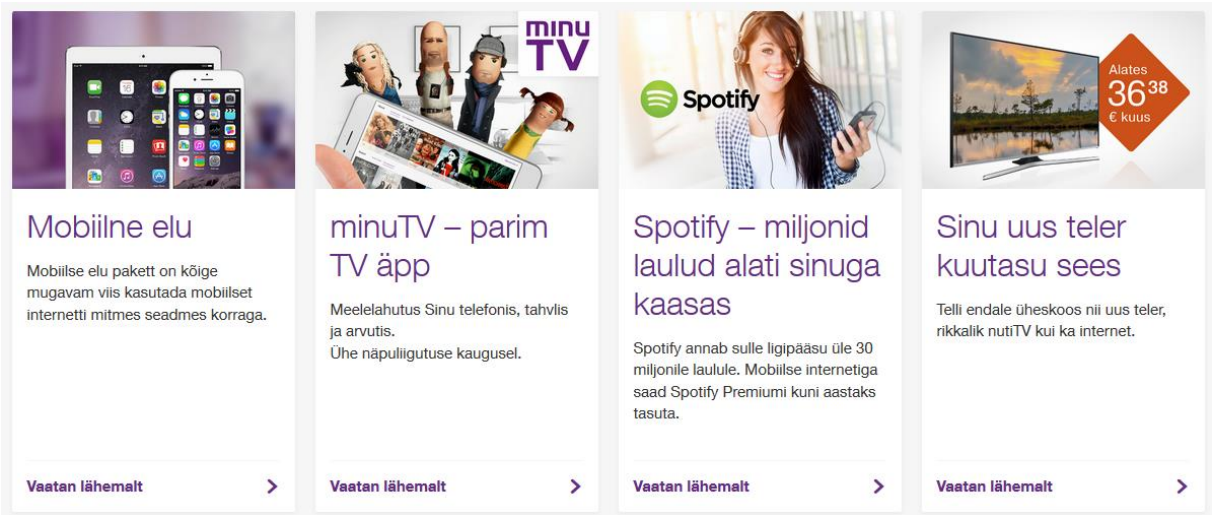
Uuendatud test kontrollib, et teisena loodud teade kuvatakse kõige esimesena (kuna see on kõige uuem) ning esimene teade kuvatakse kahe uudise vahel - just samas järjekorras, nagu neid loodi.

Kuigi üldjuhul oleks hea kasutada *id* atribuuti, siis antud *portlet*'i korral oleks see halb variant – uudiseid ja teateid võib olla 0 või neid võib olla rohkem, kui *portlet* kuvab. Seega oleks *id*

väärtustamine keeruline. Üks variant oleks küll neid nimetada oma kuvamise järjekorranumbri järgi *portlet*'is, kuid siis tekiks täpselt sama probleem, nagu praeguse viitamise korral, kuna järjestus muutuks.

3.2.4.2 Pakkumiste UI-testid ei satu õigele URL-ile

Pakkumiste *portlet*'i eesmärgiks on kuvada kasutajatele pakkumisi. Mõningaid näiteid antud *portlet*'i sisust saab näha joonisel 7. *Portlet*'is pakkumisel "Vaatan lähemalt" vajutades viiakse kasutaja eelnevalt seadistatud URL-ile.



Joonis 7 Pakkumiste *portlet*

ET veebirakenduse eesmärk on koondada kokku Elioni ja EMT klientide alguspunkt ET veebis. See tähendab, et aadressid www.elion.ee ja www.emt.ee hakkasid peale www.telekom.ee veebirakenduse üles panemist suunama aadressile www.telekom.ee. Selle näol on tegemist taas äriliste vajaduste muutumisega. Antud muudatus tekitas aga probleeme pakkumiste *portlet*'i UI-testide töös.

Pakkumiste *portlet*'i test loob vastavalt vajadusele kindlate parameetritega pakkumise ning seejärel kontrollitakse, kas pakkumine kuvatakse kliendile korrektselt ja korrektsete parameetritega. Üheks seadistatavaks parameetriks on URL, kuhu link "Vaatan lähemalt" kasutajad viib. Pakkumiste *portlet*'i testide looja oli aga testi kirjutanud nii, et selleks URL-iks määrataks www.elion.ee. UI-testi ebaõnnestumise põhjuseks oligi fakt, et oli loodud suunamine www.elion.ee-st www.telekom.ee-sse. UI-test kontrollis aga, et peale nupule "Vaatan lähemalt" vajutamist jõuaks brauser lehele www.elion.ee. Viga ilmnes testi logist, mis tõi väga elementaarselt välja vea põhjuse – oodatud URL ja tegelik URL erinesid.

Selge on see, et testi parandamine on teoorias lihtne – asendada vana URL. Tekkis aga probleem – mis URL-ile pakkumine seadistada ning kas üldse muuta pakkumise seadistust või muuta URL-i kontrolli, millele brauser jõudma peab? Et testida võimalikult paljusid juhtusid, sai muudetud URL-i kontroll peale "Vaatan lähemalt" vajutust. P*ortlet* on seadistatud minema URL-ile www.elion.ee, kuid test peaks kontrollima jõudmist URL-ile www.telekom.ee, sest siis saab veenduda ka selles, et seadistatud suunamised toimivad korrektselt.

Kuigi vea tuvastamine ja parandamine olid teoreetiliselt imelihtsad, siis antud probleemi puhul oli tähtis analüüs. Kõige kasulikum variant oli lisaks *portlet*'i toimimisele kontrollida ka suunamise toimimist. Seega toimus igas pakkumiste *portlet*'i testis vaid lihtne muudatus, asendati rida

```
Assert.assertTrue( cUrl.contains( "www.elion.ee" ) );
```

Selle asemele pandi

```
Assert.assertTrue( cUrl.contains( "www.telekom.ee" ) );
```

3.2.5 Pidevintegratsiooni süsteemis esinevad vead

Võib juhtuda, et pidevintegratsiooni agent, mis jooksutab UI-teste teatud intervalli tagant, annab teada, et test X ebaõnnestub. Arendaja-testija esimeseks ideeks on siis jooksutada antud testi ise arendusmasinas. Siis esineb aga, et agendi leitud viga ei esine arendaja arvutis.

Kui test ebaõnnestub vaid pidevintegratsiooni agendi seadmes, siis tuleb üle vaadata, et selle virtuaalmasina konfiguratsioon on võimalikult lähedane arendamiseks kasutataval masinal. Järgnevalt lahendatavaks probleemiks on uudiste listi *portlet*'i seadistamise testi ebaõnnestumine ainult pidevintegratsiooni virtuaalmasinas. Enne korrektse info sisestamist seadistuse vormis saadetakse vorm tühjalt ära, et kontrollida, kas sel juhul kuvatakse veateade. Virtuaalmasinas aga miskipärast veateadet ei ilmunud.

Uudiste listi *portlet* kuvab nimekirja portaali postitatud uudistest ja teadaannetest klientidele. Selle seadistusmenüüs on võimalik määrata URL-id, kus kuvatakse vastavalt uudiseid ja teadaandeid ning samuti valida, kas kuvatakse äri- või eraklientide uudiseid ja teadaandeid.

Probleemi lahenduseks oli taas põhjalik logide analüüsimine. Nimelt testis tehti läbi järgnev

```
$( "form p button[type='submit']" ).shouldHave( Condition.text( "Salvesta"
) );
$( "form p button[type='submit']" ).click();
$( "#newslist-edit-alert" ).shouldHave( Condition.text( "Vigane sisend." )
);
```

Esimesel real kontrolliti, et seadistuse vormi ära saatmise nupul oleks tekst „Salvesta“. Seejärel klikiti sellele nupule ning kolmandal real kontrollitakse, et ilmus veateade „Vigane sisend.“. Testi logis oli kirjas, et toimus ebaõnnestumine kolmandal real, kus otsiti veateadet. Testi kulgu jälgides ilmnes, et teisel real tehtavat klikki ei teostata. Siiski peaks kliki ebaõnnestumine viitama sellele reale, kus klikki tehakse. Võib järeldada, et klikk tehti, kuid selle järel vormi saatmist ei toimunud. Kindel oli ka, et nupp ise tegelikult töötab, kuna seda sai proovitud manuaalselt mitmel korral.

Testi kulgu virtuaalmasinas jälgides tekkis idee, et kuigi Selenium peaks saama klikkida igale „nähtavale“ ehk mitte nähtamatuks tehtud elemendile, siis antud juhul oli veebirakenduse *footer* nupul ees virtuaalmasina kuva väikese resolutsiooni tõttu. Seega sai muudetud virtuaalmasina kuva resolutsioon – väikeselt 1024x768 resolutsioonilt 1920x1080 resolutsioonile. See parandas testis esineva vea. Kuna automatiseeritud UI-testide ülesandeks antud rakenduses pole *responsive*-kujunduse testimine ega väikese resolutsiooni testimine (mida võib teha testija käsitsi), siis sai tehtud otsus, et virtuaalmasin jääb sellisele resolutsioonile, et vältida tulevikus tekkivaid sarnaseid situatsioone.

Hiljem sai tehtud ka koodis muudatus, kuna Seleniumi API-ga tutvudes ilmnes, et HTML-vormi saab ära saata vormis oleval mis tahes elemendil (või *form*-elemendil endal) submit() meetodit kasutades [7]. Nii muutus

```
$( "form p button[type='submit']" ).click();
```

rida, ning selle asemele sai tehtud

```
$( "form p button[type='submit']" ).submit();
```

Sellega saab vormi ära saata sõltumata sellest, kas vormi saatmise nupul on ees mõni muu element või mitte, kuid samas kontrollitakse, et element on olemas ning „nähtav“. Edaspidi võeti käsile submit() kasutamine kõigis testides, kus oli vaja vorme serverile saata.

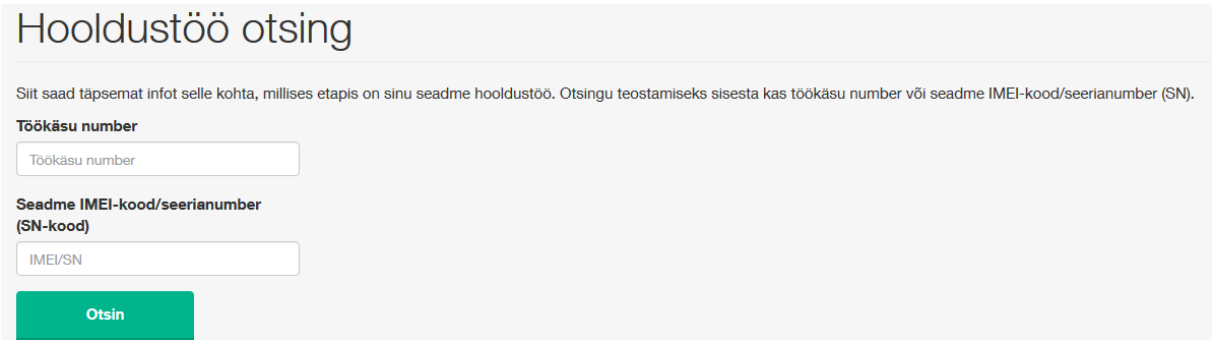
3.3 Tarkvara vigade leidmine kasutajaliidese testide abil

Tarkvara vigade võimalikult kiire tuvastamine on üks põhjuseid, milleks luuakse automaatteste. Antud töö praktilise osa käigus ei õnnestunud kordagi sellist momenti tabada, kus tarkvara viga põhjustas UI-testide ebaõnnestumise. Selleks on mitu põhjust – üks neist on see, et ühtegi viga tegelikult ei esinenudki. Teine variant on, et vead tuvastati enne UI-testimiseni jõudmist, näiteks kordades kiiremini jooksvates *unit*-testides. Kuna UI-testid ei testi *corner-case* juhtumeid, vaid põhilisi protsesse, siis on teine variant tõenäolisem.

Kuna tarkvara vigu UI-testid välja ei toonud, siis võiks arvata, et peatükis 3.3 ei ole millestki kirjutada. Siiski ilmnes tarkvara vigu testimise ühes protsessis, milleks on testide loomine. Nimelt peab automaattesti loomisele eelnevalt tutvuma testitava protsessi või kasutusjuhuga ning proovima seda käsitsi läbida. Nii tekib arusaam sellest, kuidas üles ehitada testi, milliseid *mock*'e on vaja luua ning täpsemalt milliseid protsesse tuleb automatiseerida. Sellise katsetamise käigus võib tihti esile kutsuda vigu, mis on jäänud märkamata just seetõttu, et tarkvara osal puuduvad testid. Järgnevalt kirjeldatakse mõnda viga, mis said tuvastatud just testide loomise käigus ning mis said ka paranduse antud töö praktilise osa raames.

3.3.1 Seadme remondi *portlet* kuvab valet infot

Seadme remondi *portlet* ehk hooldustöö *portlet* on mõeldud töökäsu numbri, seadme IMEI või seadme seerianumbri järgi otsides selle remondi seisu jälgimiseks. Enne antud töö aluseks oleva praktikatöö läbi viimist sellel *portlet*'il UI-testid puudusid. Seega oli üheks praktilise osa ülesandeks ka see *portlet* UI-testidega katta.



Joonis 8 Hooldustöö otsingu *portlet*

Hooldustöö otsingu vormi saatmise järel tagastatakse tabel, mis koosneb kolmest veerust – kuupäev, seadme nimetus ning töökäsu staatus. UI-testide loomisele eelnevalt oli plaan võimalikult palju funktsionaalsust ära testida ilma *corner-case* juhtumitesse süvenemata.

Seega sai otsustatud, et *mock*'i loomisel antakse *mock*-hooldustööle igast võimalikust staatusest vähemalt 1, seega on kõikide staatuste tekstide kuvamine testis kontrollitud. Probleem ilmnes aga siis, kui tabeli veerus „Seadme nimetus" ei suutnud arendusjärgus UI-test leida korrektset seadme nime.

Testi käiku jälgides ilmnes, et seadme nime asemel kuvati antud veerus mingit aadressi – tegu oli firma esinduse aadressiga, millega antud hooldustöö seotud oli. Ilmselgelt oli tegu tarkvara veaga, kuna *mock* sisaldas korrektset infot just sellisel kujul, nagu seda API tagastaks. Nagu varasemalt mainitud, siis on ka automaattestija arendaja, seega selline pealtnäha lihtne viga ei tohiks käia talle üle jõu. Nii sai selle töö praktilise osa käigus tehtud parandus leitud veale.

Vea põhjuse analüüsimisel on vaja selgeks teha, kus muutuvad andmed vigaseks. API tagastas infot korrektselt, seega tuli vaadata kas Java koodi või .jsp-faili sisu, mis kuvab Java koodis töödeldud infot. Esimesena sai üle vaadatud .jsp-fail, kus üldjuhul on sarnased vead sagedasemad. Selles failis aga ilmnes, et infot loetakse õigest muutujast. Seega sai probleem olla vaid Java-koodis.

Antud *portlet*'i tööd juhtivast kontrollerist saigi viga leitud. Nimelt väärtustati jsp-failile edastatav info järgnevalt:

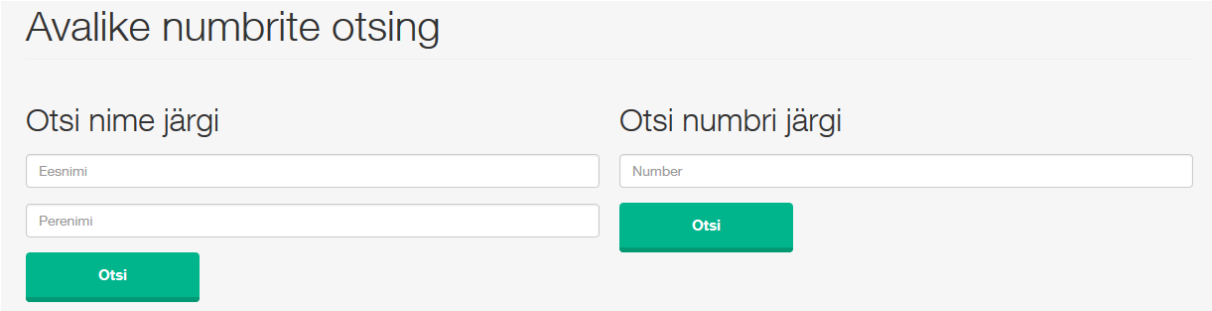
```
devicesRepairTask.setRepairTaskOffice(resp.getTaskDealerOfficeName());
devicesRepairTask.setRepairTaskDevice(resp.getTaskDealerOfficeName());
```

Nagu näha, siis väärtustatakse hooldustöö kontori muutuja kui ka seadme muutuja sama API vastuse väärtusega. Selline viga võib tekkida, kui koodi kopeeritakse ja kleebitakse, kuid kleebitud koodi kohandamine unustatakse ära. Selle vea parandamine oli lihtne, objektil *resp* tuli välja kutsuda õiget meetodit:

```
devicesRepairTask.setRepairTaskDevice(resp.getDeviceName());
```

3.3.2 Avalike numbrite otsing ei kuva korrektset veateadet

Avalike numbrite *portlet* toimib sarnaselt telefoniraamatule. Seal saab otsida kas osalise või terve nime järgi, mis võib anda 1 või rohkem tulemust ning numbri järgi, mis annab täpse numbri korral vaid 1 vaste. Vaste antakse juhul, kui numbri omanik on jätnud oma numbri avalikuks. Sellest on tingitud ka *portlet*'i nimi.



Joonis 9 Avalike numbrite otsingu *portlet*

Antud *portlet*'ile UI-testide kasutusjuhtusid otsides ilmnes aga loogikaviga koodis. Nimelt lisaks edukale kasutusjuhule sai otsitud kasutusjuhtusid, kus *portlet* annab veateate. *Portlet*'i jsp-failis on üldjuhul sees kood, mis kontrollib, kas *portlet*'i kontroller tagastas mõne vea ning seejärel tehakse sobivad tegevused, üldjuhul kindla veateate kuvamine. Probleem ilmnes siis, kui ebaõnnestus esile kutsuda viga liiga suure otsingutulemuste kogumi kohta. Nimelt peaks veateade ilmnema, kui vasteid on üle 50. *Portlet* andis aga üldist viga, mis ei olnud kasutajasõbralik.

Vea põhjus tuli *helper*-klassist, millega suhtleb *portlet*'i kontroller. Sealne meetod setError(), mis valmistab kogumi veateadetest, sisaldas loogikaviga:

```
if(errorCode == PublicNumbersResponse.ErrorCode.TOO_MANY_RESULTS){
    request.setAttribute( "tooManyResults", Boolean.TRUE );
}
if(errorCode == PublicNumbersResponse.ErrorCode.NO_RESULTS){
    request.setAttribute( "noResults", Boolean.TRUE );
}
else{
    throw new IllegalStateException("Unknown error code.");
}
```

Nagu näha, siis kui esineb liiga palju tulemusi, pannakse vastav muutuja tõeseks. Kuna see on kirjutatud eraldiseisva *if*-blokina, siis järgnev *if-else*-blokk annab kindlasti *IllegalStateException*'i, kuna samaaegselt ei saa olla tõesed *tooManyResults* ja *noResults*.

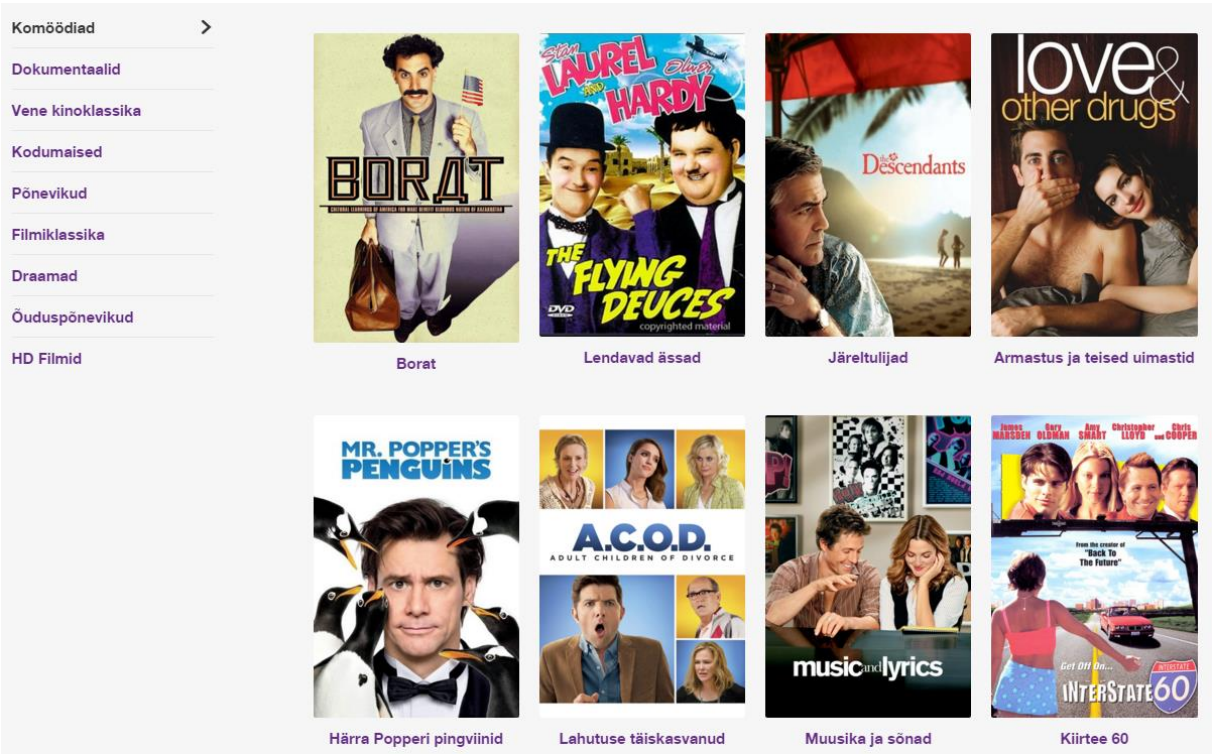
Vea parandus oli taas lihtne. Meetodis, kus kutsutakse setError() meetodit välja, kontrollitakse eelnevalt, kas viga esineb. Seega ei ole antud meetodis eraldi seda kontrolli vaja lisada. Nii võib loogika läbi mõelda järgnevalt – kui ei esine *tooManyResults* viga, siis tuleb kontrollida, kas esineb *noResults* viga, kui ei esine ka seda, kuid viga kindlasti esineb, tuleb öelda, et saadi tundmatu viga. Seega tuleks *if*-blokkide loogika üles ehitada järgnevalt: *if, else if, else*:

```
if(errorCode == PublicNumbersResponse.ErrorCode.TOO_MANY_RESULTS) {
    request.setAttribute( "tooManyResults", Boolean.TRUE );
```

```
}
else if(errorCode == PublicNumbersResponse.ErrorCode.NO_RESULTS) {
    request.setAttribute( "noResults", Boolean.TRUE );
}
else {
    throw new IllegalStateException("Unknown error code.");
}
```

3.3.3 Videokataloogis kategooria vahetus ei toimi

Videolaenutuse kataloogi *portlet* kuvab kategooriate kaupa kõiki filme, mida ET pakub oma TV-teenuse videolaenutuses. Kuna see oli varasemalt UI-testidega katmata, siis käesoleva töö käigus sai ette võetud ka esialgu testitavate juhtude leidmine. Selle käigus ilmnes aga viga – kategooriale vajutus ei teinud midagi. Visuaalselt anti kasutajale küll märku, et kategooria peaks olema vahetatud, kuid mingisugust päringut serverile ei saadetud.



Joonis 10 Videolaenutuse kataloogi *portlet*

CSS ja JS arendajate abiga sai leitud vea põhjus. Kategooriat kajastav element oli *a*-element ehk link. Suurem osa brauseritest nõuavad, et lingielemendil oleks *href*-atribuut. Kuna kategooria vahetamine toimis teiste skriptide kaudu, siis *href* peaks tegema mitte midagi. Seega oli selleks antud sellele väärtus

```
href="javascript:void(0);"
```

Mingil põhjusel takistas see muu JavaScripti jooksutamist. Kuna tegemist on mahuka kogumiga kujundusest ning skriptidest, siis täpne vea põhjus ei olnud 100% kindel, kuid kindlasti oleks *href* väärtuse muutmine mingiks mitte midagi tegevaks väärtuseks vea parandanud. Seega saigi lihtne viga taas lihtsa koodimuudatusega parandatud. *Portlet*'i jsp-failis tuli eelnevalt välja toodud *href* väärtus muuta järgnevalt:

```
href="#"
```

Esmapilgul on see väike parandus, kuid vea paranduseks täiesti piisav. Valmis tootes oleks see viga tekitanud suuri probleeme, kuna *portlet* on kättesaadav kõigile rakenduse kasutajatele, mitte ainult registreeritud klientidele.

4. Testide loomine

Testide loomisel tuleks lähtuda sellest, et igale eraldiseisvale osale tuleb luua eraldi testid. *Portlet*-tehnoloogia teeb UI-testide loomise seega lihtsamaks, kuna iga *portlet* on eraldiseisev osa. Teste saab luua ühele kindlale *portlet*'ile kartmata, et teised *portlet*'id testi tulemusi mõjutavad.

4.1 Teoreetilised alused

Järgnevates peatükkides leitakse testimist vajavad kasutusjuhud ning luuakse neile testid. Selline testide loomine on kasutusjuhtude põhine testimine. Kasutusjuht on kirjeldus mingist süsteemis tehtavast tegevusest. Kasutusjuhus kirjeldatakse kasutaja tehtavad tegevused et mingi kindla eesmärgini jõuda, samuti on selles kirjeldatud eel- ja järeltingimused [8]. Tarkvara testimine on protsess, mille käigus uuritakse tarkvara osa, et tuvastada erinevusi sisestatud sisendis ning oodatavas väljundis [9]. Seega on kasutusjuhtude põhisel testimisel eeltingimused osa sisendist ning järeltingimused osa väljundist. Kui täidetud eeltingimustega tarkvara testida, siis testi tulemuseks peavad olema oodatud järeltingimused.

Nagu sai mainitud peatükis 3.1, võib ka automaattestide kirjutajat pidada arendajaks. Seega saab head automaattestijat hinnata selle järgi, kas ta järgib *clean code* põhimõtteid. Robert C. Martin kirjeldab oma *clean code* teemalises raamatus järgnevaid nõudeid [10]:

- Muutujate ja meetodite nimed peavad näitama nende kasutuseesmärki.
- Kommentaarid peaks olema vaid asjadeks, mida koodis edasi öelda ei saa. Kood peaks seega olema loetav ilma kommentaarideta.
- Failid ei tohiks olla liiga pikad, ~200 rida on sobilik. Samuti peaks failis olema eespool tähtsamad osad.
- Meetodid peaksid olema lühikesed ning täitma vaid üht eesmärki.

Antud töös loodud automaattestides on proovitud järgida neid põhimõtteid, kuid just viimase välja toodud nõudega on ilmnenud erandeid. Nimelt ilmnes töö käigus vajadus luua kogu *portlet*'i testimine ühe meetodina, kuna *portlet* kirjutab informatsiooni rakenduse andmebaasi,

seega modulaarne lahendus tekitab probleeme eeltingimuste rahuldamisega – andmed, mida on vaja hiljem kontrollida tuleb kõigepealt andmebaasi lisada enne testi. Seega on tulemuseks üks pikk meetod, mis teeb läbi kõik vajalikud funktsionaalsuse testid ning taastab seejärel andmebaasis algse olukorra. Eraldi meetoditena oleks töömaht liialt suur, kuna eeltingimuste paika panemiseks tuleks kasutada testitavat tarkvara või välja töötada meede eeltingimuste muul viisil seadistamiseks.

4.2 Testimist vajavate kasutusjuhtude leidmine

Igasuguse testimise keskne idee on testitavale tarkvara osale ette anda kindel sisend teades seda, milline peab olema väljund. Seejärel kontrollitakse, kas testitav osa tõepoolest andis oodatud väljundi. *Unit*-testimise korral on sisendiks mingid meetodi argumendid, millele on meetodi väljund teada. UI-testimine toimub aga *front-end*'i tasemel. See tähendab, et nii sisend ja väljund on midagi, mida lõppkasutaja sisestab või näeb ning meetodite väljundit otseselt ei kontrollita. Sisendiks võib olla näiteks HTML-vormi sisestatud andmed, väljundiks aga veateade tekstiga, mis viitab sisendandmete valele formaadile. Seega saab UI-teste luua reaalsete kasutusjuhtude põhjal.

Üheks kasutusjuhuks on kindlasti protsess, mille läbimiseks testitav tarkvara osa loodi. Näiteks peatükis 3.1.2.2 mainitud Minu Autentimise Meetodid *portlet*'i peamisteks ülesanneteks on näidata kasutajaga seotud sotsiaalmeedia ja e-maili kontosid ning võimaldada neid lisada ja eemaldada. Järgnevad kasutusjuhud on toodud Minu Autentimise Meetodid *portlet*'i näitel, kuid sellist analüüsi saab kohandada mis tahes muule *portlet*'ile või tarkvara osale, millel on olemas kasutusjuhud.

Teisesteks kasutusjuhtudeks on *corner-case* juhud. Nende UI-testimise vajadus sõltub täiesti firmasisesest ideoloogiast. Ühelt poolt võib kui tahes äärmuslikud kasutusjuhud *frontend*-tasemel läbi testida. Teisalt oleks UI-testid aga põhiliste *flow*'de testimiseks. Äärmuslikud juhud jääksid siis *unit*-testide alla. *Unit*-testimine on äärmuste testimiseks palju paindlikum. Järgnevalt testitakse vaid põhilisi *flow*'sid.

4.2.1 Minu Autentimise Meetodid *portlet*'i kasutusjuhud

Minu Autentimise Meetodid *portlet*'is on võimalik lisada enda kontole e-maili aadress. Reaalne isikuga sidumine toimub aga teises serveris. Seal määrab kasutaja ka parooli, mida ta soovib. Väline server saadab kasutajale valideerimismeili, selles oleval lingil klikkides seos

aktiveeritakse ja e-mail ja parool on sisse logimiseks aktiveeritud. Siiski vajab testimist see, kas peale eelnevalt kirjeldatud tegevusi välises serveris kuvatakse Minu Autentimise Meetodid *portlet*'is korrektne kasutajainfo.

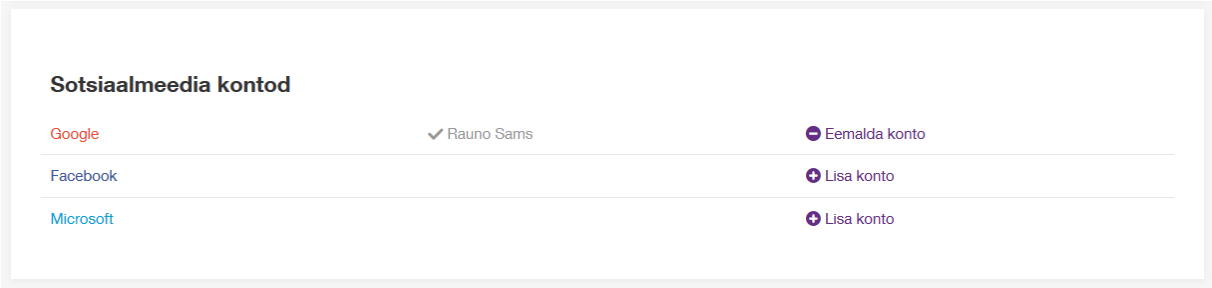
Joonis 11 Kasutajaga on seotud 1 e-posti aadress

Esimene lisatud e-posti aadress loetakse esmaseks. See tähendab, et kontoga on võimalik siduda rohkem kui 1 e-posti aadress, kuid peale esimest lisatud aadressid salvestatakse kui teisesed aadressid. Seega tuleks luua test ka juhule, kui kasutajal on juba olemas 1 e-mail, kuid soovitakse lisada veel. Eelduseks on, et kasutajal on juba 1 e-mailiga sidumine olemas. See on aga *mockserver*'i vastuse seadistamise abil eelnevalt seadistatav.

Joonis 12 Kasutajaga on seotud 2 e-posti aadressi

Järgmisteks kasutusjuhtudeks antud *portlet*'is on kasutajakonto erinevate sotsiaalmeedia kontodega sidumine. ET veeb võimaldab kasutajal oma konto siduda Google+, Facebook ja Microsoft Live kontodega. Kõigi kolme sidumise korral on protsess sama – sidumist alustatakse veebirakenduses, seejärel suunatakse kasutaja valitud sotsiaalmeedia serverile, kus ta lubab oma sotsiaalmeedia kontoga sidumise. Kui sealne luba on käes, antakse edasi päring ET vastavale serverile, mis loob sotsiaalmeedia konto ja veebirakenduse kasutajakonto vahelise seose. Veebirakenduses kuvatakse seejärel selle isiku nime, kelle sotsiaalmeedia kontoga sidumine tehti. Kasutusjuhud on kõigi sotsiaalmeedia kontodega sidumise korral

sarnased, seega testis peab kontrollima vaid, et seos on loodud korrektse nimega ning korrektse kontoga.



Joonis 13 Konto, mis on seotud ainult Google+ kontoga

Nagu joonistelt 6, 7 ja 8 on näha, siis olemasoleva e-maili või sotsiaalmeedia konto eemaldamine on samuti võimalik, seega tuleb ka need kasutusjuhud testida ning kontrollida, kas eemaldamise protsessi järgselt seos tõesti on kustutatud.

4.3 Testide loomine leitud kasutusjuhtudele

Peatükis 4.2 kirjeldatud kasutusjuhtude testimiseks tuli esiteks uurida, mis API edastab kasutaja kasutajaga seotud kontode infot *portlet*'ile. Selle leidmine oli rakenduse logi abil lihtne. Nimelt saadab API need andmed järgneval kujul:

```
userdetails.attribute.name=[nimi]
userdetails.attribute.value=[väärtus]
userdetails.attribute.name=[nimi]
userdetails.attribute.value=[väärtus]
userdetails.attribute.name=[nimi]
userdetails.attribute.value=[väärtus]
```

Edasine väärtuste lugemine toimub vastavalt sellele, millise atribuudiga oli tegu. Täpse süntaksi võib leida koodist. Näiteks teiseste e-mailide korral on atribuudi nimeks *other-mail* ning kõik sellele järgnevad *value*'d kuni järgmise *name*'ni loetakse teiseste e-mailide alla. Selle lahenduse tunneb koodist ära, kui peale nime *other-mail* leidmist tehakse *for*- või *while*-tsükliga kõigi järgnevate väärtusete lugemine. Seega kui kasutajal on 4 teisest e-maili, sisaldaks APi vastus järgnevat:

```
userdetails.attribute.name=other-mail
userdetails.attribute.value=[väärtus]
userdetails.attribute.value=[väärtus]
userdetails.attribute.value=[väärtus]
userdetails.attribute.value=[väärtus]
```

Peatükis 4.2 kirjeldatud kasutusjuhtudest sai luua 9 testmeetodit. Nendeks on:

1. addEmail() – konto sidumine (esmase) e-mailiga.
2. addGooglePlus() – konto sidumine Google+ kontoga.
3. addFacebook() – konto sidumine Facebook kontoga.
4. addMicrosoft() – konto sidumine Microsoft kontoga.
5. addAdditionalEmails() – konto sidumine teiseste e-mailidega (eeltingimusega, et esmane on olemas).
6. removeEmail() – kontolt e-mailiga sidumise eemaldamine.
7. removeGoogle() – kontolt Google+ kontoga sidumise eemaldamine.
8. removeFacebook() – kontolt Facebook kontoga sidumise eemaldamine.
9. removeMicrosoft() – kontolt Microsoft kontoga sidumise eemaldamine.

Kuna kasutusel on *mock*'imine ehk keelatud on ühendamine väliste API-dega, siis sai loodud meetod *mock*-vastuse kokku panemiseks vastavalt testmeetodi nõuetele:

```
public void isLoginMethodAdded( boolean hasEmail, boolean hasFacebook,
boolean hasGoogle, boolean hasLive, boolean hasExtraEmails )
```

Antud meetodi ülesandeks on manipuleerida *mock*-vastuse sisuga vastavalt sellele, mis seisus peaks kasutajakonto info teatud hetkel olema. Näiteks alustatakse meetodit addEmail() sellega, et pannakse paika ilma ühegi sidumiseta konto:

```
isLoginMethodAdded( false, false, false, false, false );
```

Test vajutab nupule, mis viib lehele, kus toimub e-mailiga sidumise protsess. Kuna protsess toimub teises serveris, siis võib brauseri taaskäivitada. Enne brauseri taaskäivitamist aga tuleb seadistada järeltingimused ehk siis API vastus tuleb muuta selliseks, et kajastuks e-maili lisamine:

```
isLoginMethodAdded( true, false, false, false, false );
```

Sellele järgneb taas sisse logimine ning nüüd kontrollitakse, et kuvatakse *mock*-vastuses ette antud e-mail ning nupp sidumise eemaldamiseks.

Testis addAdditionalEmails() antakse eeltingimusena ette, et esmane e-mail on juba kontoga seotud. Testi järeltingimustena antakse ette, et on olemas nii esmane e-mail kui ka teisesed e-mailid. Kuigi test teeb teisese e-maili lisamist vaid ühe korra, lisatakse regressiooni mõttes API *mock*-vastusesse 2 e-maili. Sellega saab veenduda, et rakendus tuvastab kõik teisesed e-mailid, mitte ainult esimese peale *userdetails.attribute.name=other-mail* rida.

Sotsiaalmeedia kontodega sidumise testimise põhimõte on vägagi sarnane esmase e-mailiga sidumisele. Põhiline erinevus on vaid HTML-elemendil, millele protsessi alustamiseks tuleb vajutada ning HTML-elementidel, mille olemasolu ja sisu tuleb kontrollida, et veenduda järeltingimuste täitmises.

Kontodega sidumise eemaldamine on samuti väga sarnane lisamise protsessidega, kuid tagurpidine. Seega on järeltingimustes vaja veenduda, et mingi element või tekst (näiteks e-mail) ei oleks lehel kuvatud. Üldiselt on lisamise ja eemaldamise protsessid väga sarnased, toimub ainult taustal *mock*-vastuste muutmine.

Clean code põhimõtete järgimine on antud testide puhul osaliselt küsitav. Kuigi kõik sotsiaalmeedia kontodega sidumiste ja eemaldamiste testid täidavad vaid üht ülesannet (sidumise lisamine/eemaldamine ja järeltingimuste kontroll), siis on meetodite sisud peaaegu identsed. Seda saab põhjendada sellega, et iga sotsiaalmeedia konto sidumist näidatakse sellele ette nähtud real kasutajaliideses ning seal võib kuvatud olla konto omaniku nimest erinev nimi. Seega igas meetodis tehakse küll samu tegevusi ning kood on suuremalt jaolt korduv, kuid järeltingimused sisaldavad nüansse, mis toetavad valikut luua iga sidumise ja eemaldamise jaoks eraldi meetod.

Loodud koodi näidetega saab lähemalt tutvuda veebis – aadress on leitav töö lõpust, peatükis Lisa.

5. Testimise tööriistade puudused

Pikemalt mingite tööriistadega kokku puutudes ilmnevad probleemid ja puudused, mis vajaksid parandust. Kui parandamine pole võimalik, tuleb teha lisatööd, et puudusest ümber minna ning mingil moel vajalik funktsionaalsus implementeerida. Järgnevates peatükkides on kirjeldatud probleemid, mis esinesid diplomitöö tegemisel nii rakenduses eelnevalt loodud tööriistades kui ka välistes testimist abistavates teekides.

5.1 Rakenduse spetsiifilised abivahendid

5.1.1 Login.java

Login.java klass on põhiline sisselogimise funktsionaalsuse testimise klass. Seda ei loodud antud diplomitöö käigus, vaid eksisteeris varasemast ajast. Küll sai vastavalt vajadusele ka diplomitöö käigus lisatud *mock*-vastuseid ning muud funktsionaalsust antud klassi. Klass võimaldab rakendusse sisse logimist ilma väliste API-de poole pöördumata. Lisaks sisse logimisele sisaldab see põhilisi *mock*-vastuseid, mis tagavad põhilise sisse logitud kasutaja funktsionaalsuse.

Kuna antud klassil lasub niivõrd suur hulk ülesandeid, siis on klassi suurus koodiridade arvult suur. See on probleemne, kuna see on vastuolus *clean code* põhimõtetega. Sellest tulenevalt on kood halvasti hallatav. *Mock*-vastused on osaliselt jagatud eraldi meetoditesse, samas on põhiline meetod setMockResponses() samuti *mock*-vastuste sisu täis. Võib öelda, et antud klass läheb väga suurelt vastuollu *clean code* põhimõtetega. Selle põhjustatud kahju on vägagi tuntav – vigade leidmine on raskendatud, samuti on keeruline lisada uusi *mock*-vastuseid.

Kuna antud klass on loodud spetsiifiliselt käesoleva rakenduse jaoks, siis selle muutmine on võimalik – lähtekood on projektis ning vabalt muudetav. Mida peaks aga selle klassi parandamiseks tegema? Esiteks tuleks selgitada, kus hoida siiski *mock*-vastuseid. Paljud *mock*-vastused on kirjeldatud ka klassides ja testides, mis neid otseselt kasutavad. Samas on mõningad vastused, mida vajab vaid 1 test, kirjeldatud Login.java klassis.

Teiseks tuleks *mock*-vastuste sisu paremini struktureerida. Üheks variandiks oleks nende salvestamine failidesse ning neist lugemine. Sellega oleks tagatud modulaarsem struktuur, kuna saaks luua meetodi, mis loeb faili, mille nimi ette antakse ning tagastatakse sisu. Sellisel

juhul tuleks vältida juhtu, kus sama meetod paneb paika ka *mock*-vastuse, kuna siis ei ole meetodil vaid 1 ülesanne, nagu soovitavad *clean code* juhised.

Viimaseks tuleks ühtlustada klassi meetodite kasutust teiste meetodite poolt. Töö kirjutamise hetkel on probleemne fakt, et meetodit setMockResponses() meetodit on Login.java mitu. See on küll mugav, kui ei soovi mõnd argumenti ette anda, kuid tegelikult on need tekkinud lihtsalt seetõttu, et muudatuste korral ei ole suvatsetud muuta meetodit, mis kasutab setMockResponses() vanemat signatuuri. Meetod peaks suutma vastu võtta kas erinevaid argumentide alamhulki või vastu võtma vaid üht kindlat signatuuri.

5.1.2 ResetBaseUrls.java

Klass ResetBaseUrls.java sai loodud käesoleva töö praktilise osa käigus. Vajadus sellele tekkis sellest, et UI-testide parandamisel oli tihti vaja jätta brauserit käima lõputult – näiteks *while(true)* tsüklis. Arendajal on sellises situatsioonis võimalik teha käsitsi oma tempos läbi samme, mida testki. Sellises seisus brauseri sulgemine aga ei taasta algset seisu andmebaasis, kus hoitakse API-de URL-e. Seega jääks *mock*-serverite URL-id andmebaasi, ning rakenduse kasutamine väljaspool testimist oleks võimatu.

Klassi põhiline probleem on, et sellesse on *final*-muutujatena sisse kirjutatud tegelikult API-de URL-id. See võib tekitada aga probleeme, kui mõni API kolib teisele domeenile. Miks sai töö käigus tehtud selline lahendus? Selle probleemi lahendamiseks ei olnud antud hetkel muud võimalust – õigeid URL-e ei salvestatud kuhugi mujale, kui muutujatesse, mis olid *mock*-servereid käivitavas klassis. Tegelikult on sellele probleemile ka lahendus, kuid see oleks veidi aega nõudvam ning vajaks suuremat arutamist projektiga seotud inimestega. Lahenduseks oleks andmebaasitabeli loomine, kuhu kirjutatakse testi eel õiged URL-id ning vajadusel saaks neid taastada just sealt nii *mock*-servereid käsitlev klass kui ka ResetBaseUrls.java.

Antud klassi lähtekood on kättesaadav veebist – aadress on leitav peatükist Lisa töö lõpus.

5.2 Välised teegid ja tööriistad

5.2.1 Brauseri *cache* tühjendamine Seleniumis või Selenides

Brauserid salvestavad vahemällu ehk *cache*'i andmeid, mida kasutatakse tihti ning mis ei pruugi tihti muutuda – tavaliselt on nendeks näiteks pildid, JS kood ja CSS kujundus. UI-

testimisel on aga probleemiks, kui muudatusi ei kuvata, kuna vanad andmed on salvestatud *cache*'i.

Selenium tekitab igal brauseri käivitamisel küll uue profiili ning sellega koos uue tühja *cache* ja *cookie*'de kogumi, kuid kui tühja *cache*'t on vaja ühe testi raames, siis peab brauseri taaskäivitama. Selenium ei toeta *cache* tühjendamist ühe WebDriver'i instantsi raames. Kuigi kasutajad on antud funktsionaalsust soovinud, on Seleniumi arendajad öelnud, et tühja *cache* saavutamiseks tuleks luua uus brauseri sessioon [11].

Kuna Seleniumi Java API on avatud lähtekoodiga, saaks sellele kirjutada vajaliku funktsionaalsuse juurde. Sellega kaasneb aga paar probleemi. Esiteks pole Selenium kasutatav ainult Java keeles. Seega peaks täiendama kogu Seleniumi teeki, kuna kõigil erinevatel API-del peaks olema sama funktsionaalsus. Teine probleem on see, et kuigi antud funktsionaalsuse jaoks on varemgi avaldatud soovi, on Seleniumi projektiga seotud isikud keeldunud selle lisamisest ning pakkunud välja lahenduse, mis on juba raamistikus olemas. Seega antud funktsionaalsuse loomine vaid enda tarbeks nõuab rohkem tööd, kui sellega võidetud aeg testides.

Probleemi saab lahendada paaril erineval viisil. Esiteks on projekti arendajate soovitus teha brauserile *restart*. Uuel käivitamisel luuakse täiesti uus profiil ilma *cookie*'de ega *cache*'ta. Minu Autentimise Meetodid testides on kasutatud just sellist lahendust. Selle puuduseks on vaid see, et seetõttu tuleb uues brauseri sessioonis ka sisse logida uuesti, mis suurendab vähesel määral testile kuluvat aega. Teiseks lahenduseks oleks keelata brauseril *cache* talletamine. See lahendus ei ole hea UI-testimise poolel, kuna UI-testid peavad testima reaalset kasutajakogemust. Juht, et kasutajad on keelanud *cache* talletamise on väga ebatõenäoline.

Lõpetuseks peaks mainima, et testitav rakendus ei talleta reaalselt kasutades Minu Autentimise Meetodid *portlet*'is kasutatavat kasutajainfot *cache*'s, kuid mingil põhjusel toimub see, kui toimub UI-testimine kohalikult jooksvas rakenduse serveris.

6. Kasutajaliidese testide ajakulu

Võrreldes ühiktestidega kaasneb kasutajaliidese testidega märkimisväärne ajakulu, kuna info sisestamine ja lugemine toimub kasutajaliideses. Koodi tasemel öeldakse vaid ette tegevused, mida kasutajaliideses teha tuleb. Selle probleemi leevendamiseks tuleb leida viise, kuidas UI-testide ajakulu minimeerida. Seda tuleks teha ühe testi haaval, analüüsides selle tegevust ning leides, kus kulub testis kõige kauem aega. Tihti võib aga ka see tegevus kujuneda liiga aeganõudvaks ning ilmneb, et mõistlikum oleks ajakulu vähendamise protsess vahele jätta.

6.1 Analüüs ajakulu muutusele

Järgnevad andmed tulevad pidevintegratsiooni keskkonnast Bamboo, kus jooksutatakse ET veebi UI-teste iga 3 tunni tagant. Andmete põhjal saab leida keskmise ajakulu ühele testile ning selle muutuse testide hulga suurenedes. See on aga ligikaudne hinnang, kuna testid võivad võtta erineva aja. Järgnevas tabelis on välja toodud ainult 100% edukad testtsüklid, kuna testi ebaõnnestumisel on kogu tsükli kestus lühem.

Tabel 1 Testtsükli pikkuse sõltuvus testide arvust

Testide arv tsüklis	Testtsükli pikkus	Ühe testi keskmine kestus
29 testi	39 minutit	80 sekundit
35 testi	47 minutit	80 sekundit
40 testi	50 minutit	75 sekundit
42 testi	51 minutit	73 sekundit
51 testi	59 minutit	69 sekundit
54 testi	62 minutit	69 sekundit
59 testi	64 minutit	65 sekundit

Tabelist on näha, et kuigi testide arv peaaegu kahekordistus, siis ajakulu kasvas vaid ~1.6 korda. Samuti langes keskmine ajakulu ühe testi kohta. Selle põhjuseks on arvatavasti see, et antud diplomitöö käigus kirjutatud testid olid kiiremad, kui varasemalt loodud testid. Samuti oleks keskmine ajakulu väiksem, kui testtsükli pikkuseks näidatud aeg Bamboo poolt ei sisaldaks testserveri käivitamist ning ette valmistamist testide jooksutamiseks. Sellega oleks tsükli pikkus umbes 10 minutit väiksem, kuid serveri igakordne üles seadmine on tähtis samm, kuna sellega on tagatud, et ükski osa tarkvarast ei jookse vana koodi peal, mis ehk toimib korrektselt ning võimalik viga tarkvaras jääb avastamata.

Bamboo keskkond näitab ka kõige aeganõudvamate testide pingerida. Väga pika testi puhul võib arvata, et test on liiga suur. See tähendab, et ei testida vaid üht töövoogu rakenduses, vaid tehakse erinevaid tegevusi ühes testis. Kuigi ka ET veebirakenduse kasutajaliidese testides selliseid teste esineb, siis sellel on üks põhjus – Liferay rakendusel on oma andmebaas, millesse ka testid kirjutavad, kuna see pole väline server. Selles algse olukorra taastamiseks luuakse ühe testi raames testandmed läbi kasutajaliidese (kuna muu liides selleks puudub). Erinevaid tegevusi testitakse just nende andmetega ning lõpuks need andmed eemaldatakse kasutajaliideses. Üksikute testide korral peaks igale testile eraldi tegema andmed ning testi lõpus need eemaldama.

Tabel 2 Pikima kestusega testid testtsüklis

Testmeetodi nimi	Testmeetodi keskmine kestus üle kõigi testtsüklite
adminCanConfigurePersonalWebContents	13 minutit
newsAndAnnouncementsAreVisible	6 minutit
checkOffer	6 minutit
commonConfigAdminPortlet	5 minutit
twoFeedItemsAreVisible	3 minutit
adminCanConfigurePortlet	2 minutit

newslistPortlet	2 minutit
textAdminPortlet	1 minut
staticAdminPortlet	1 minut
addEmail	1 minut

Esikohal olev test adminCanConfigurePersonalWebContents ei ole tegelikult kestuselt nii pikk, kuid seda jooksutatakse esimese testina peale serveri käivitamist. Selle testi käigus sisenetakse administraatori vaatesse, mille esmakordne laadimine võttis Bamboo agentmasinas aega mitu minutit. Selle ajakulu eemaldamise järel on siiski tegu lihtsalt pika testiga, kuna lõpptulemuse saavutamine vajab palju eelseadistamist.

Testid newsAndAnnouncementsAreVisible, checkOffer ja twoFeedItemsAreVisible testivad küll üht kindlat funktsionaalsust, kuid ajakulu põhjus peitub nende koodis – need sisaldavad mitmes kohas kontrolli, et teatud elementi lehel ei eksisteeri. See võtab aga kauem aega, kui elemendi eksisteerimise kontroll. Sellest lähemalt aga peatükis 6.2.

Testid, mis lõpevad sufiksiga „*portlet*“ testivad kogu mingi *portlet*'i funktsionaalsust, kuna nende algseadistamist saab teha vaid nende kasutajaliideses. Nendeks testideks on commonConfigAdminPortlet, adminCanConfigurePortlet, newslistPortlet, textAdminPortlet ja staticAdminPortlet. Need testid saaks küll tükeldada vastavalt ühiktestimise põhimõtetele väiksemateks, kuid selle funktsionaalsuse jaoks toe loomine on küllaltki ajakulukas. Samuti võib väiksemate testide summaarne ajakulu kujuneda pikemaks.

6.2 Võimalikud lahendused

Ajakulu vähendamiseks on meetodeid mitmeid. Järgnev nimekiri kirjeldab mõningaid võtteid, mis peaks testi käiku kiirendama. Seejärel analüüsitakse välja toodud lahenduste positiivseid ning negatiivseid mõjusid.

1. Elemendile viitamisel kasutada selle ID-d.

2. Viia miinimumini kontrollid, kus tehakse kindlaks, et elementi ei leidu.

3. Koodi korduse likvideerimine.

4. Mitme Bamboo agendi kasutamine.

5. Brauseri sessiooni säilitamine.

6. Testide tükeldamine vastavalt ühiktestimise põhimõtetele.

Punkt 1 annaks ajavõidu üksiku testmeetodi tasemel, kuna elemendi otsimine mingi identifikaatori järgi on üks põhilisi tegevusi kasutajaliidese testi läbiviimisel. Selleks tuleks tähtsamad elemendid HTML-dokumendis tähistada atribuudiga *id*. Identifikaatoratribuudi järgi otsimine on kiirem, kuna HTML standard näeb ette, et see peab olema unikaalne dokumendi piires. Selle miinuseks on aga, et kõik rakenduses kasutatavad HTML-sisu sisaldavad failid tuleb üle kontrollida ning analüüsida, kas need sisaldavad olulisi elemente, millele atribuudiga *id* nimi anda.

Punkt 2 tuleneb Seleniumi põhilisest olemusest. Nimelt saab ja tuleks Seleniumi testile konfigureerida üldine ooteaeg, mille test ootab, et oodata mingi elemendi ilmumist või mitte ilmumist. Elemendi eksisteerimise korral loetakse veaolukorraks juht, kus seda ei eksisteeri. Sel juhul oodatakse ära terve ooteaeg enne, kui test ebaõnnestunuks loetakse.Elemendi mitte eksisteerimise kontrolli korral oodatakse ära terve ooteaeg enne, kui test veendub, et seda tõesti ei eksisteeri. Seega tuleks kontrollida seda juhtu ainult siis, kui see on ainus märk sellest, et rakendus toimib korrektselt ning mingil muul moel seda kontrollida pole võimalik.

Koodi kordust ei tohiks küll nagunii eksisteerida, kui kood on loodud *clean code* juhiste järgi, kuid mõnel juhul on see vajalik või lihtsam olnud. Näiteks ET veebirakenduse puhul on mitmes kohas korratud *mock*-vastuste loomist, kuna lihtsam oleks seda korrata enda testis kui kohandada olemasolevat, et seda saaks kasutada mitu meetodit. Kuigi selliste korduste likvideerimine annaks väga väikese ajakulu, kaasneks boonusena puhtam ja loetavam kood. See on punkti 3 mõte.

Mitme agendi kasutamine on üldiselt suurepärane viis vähendada ajakulu kasutajaliidese testide jooksutamisel – ajakulu saaks jagada kahega, kolmega või rohkemagagi. Piiriks oleks vabade agendilitsentside arv, mida tuleb Bamboolt osta. Mitme agendi korral saaks jagada testid erinevate masinate vahel vastavalt nende keskmisele ajakulule. Seda tegevust saab nimetada balansseerimiseks, kuna tasakaalustatakse erinevate agentide ajakulu. Paralleelselt jooksvate testtsüklite jooksutamise kestus on aeglaseima osatsükli pikkus. ET veebirakenduse

korral on probleemiks aga viis, kuidas teste jooksutatakse ning rakenduse modulaarsus. Vastava mooduli all on kõik selle koodiga seotud UI-testid ning neid jooksutatakse Gradle'ga kõik koos. Seega ühe meetodi kaupa balansseerimine on välistatud. Sellest tulenevalt tuleb balansseerida moodulite tasandil. See tekitab aga probleemi – kõik moodulid ei sisalda samat arvu teste. Seega on balansseerimine miski, mis vajab sellise rakenduse ülesehituse korral suuremat analüüsi. Mitme tsükli järel on võimalik hinnata mooduli keskmist ajakulu ning leida võimalikult ühtlane paralleeljooksutamise konfiguratsioon.

Brauseri sessiooni säilitamine tähendab, et *WebDriver* ei sulgu peale testi lõppu, vaid seda kasutatakse jooksvalt ka järgnevate testide korral (ühe klassi piires). Kuigi sellega hoitakse kokku aega, kuna pidev brauseri sulgemine ning avamine tekitab iga testi korral mõnesekundilise ajakulu. Miinuseks antud lahendusel on aga, et siis võivad säilida brauseris *cache* või küpsised, mis segavad järgneva testi tulemuse „puhtust".

Testide tükeldamine on taas miski, mida ei oleks *clean code* ja ühiktestimise põhimõtete järgi vaja teha, kui neid põhimõtteid on tõepoolest järgitud. Kahjuks aga esineb juhtusid (ka diplomitöö raames tehtud töös), kus suuremad multifunktsionaalsed testid on vajalikud. Testi tükeldamine annab ajavõidu siis, kui tegelikult saaks väiksemate testidega läbi ajada. Näiteks kui testmeetodis suletakse ning avatakse brauserit uuesti, et hakata testima uut osa rakendusest, siis selle koha peal peaks tegema testmeetodi pooleks.

7. Kokkuvõte

Põhilisteks eesmärkideks töös oli UI-testide parandamine ning lisamine reaalselt kasutatavale tarkvarale. Selle eesmärgi täitmise käigus toimus kaks protsessi, nende analüüsimine sai omakorda töö eesmärkideks. Esimeseks protsessiks oli see, et ilmnema hakkasid puudused kasutatavates tööriistades. Need puudused said analüüsitud ning samuti pakuti välja võimalusi, kuidas neid lahendada. Teine protsess oli testtsükli ajakulu muutus. Testide parandamise ja lisamisega ajakulu aina kasvas. Analüüs oli vajalik, et näha, kas antud töös kirjeldatud meetodid testide loomisel mõjutasid ajakulu paremini, kui varasemalt loodud testid. Analüüsist ilmnes, et töö käigus loodud testid olid tõesti efektiivsemad, kui varasemalt loodud testid, kui võtta mõõtühikuks keskmine ajakulu ühe testi kohta.

Lisaks sellele, et sai loodud varasemast efektiivsemaid UI-teste, parandati ka varasemad testid, kuna need olid kas kirjutatud lohakalt või aja jooksul unarusse jäänud. Analüüsipeatükkide käigus leitud probleemid ja välja pakutud lahendused on kindlasti tulemus, mida saaks kasutada tulevaste tööde kirjutamisel algmaterjalina. Samuti on ajakulu analüüsi arvulised tulemused heaks mõõtühikuks sellest, et antud töös loodud testid on tõepoolest efektiivsemad ajakulu mõistes, kui eelnevalt olemas olnud UI-testid.

Esiteks ilmnes tööst, et võrreldes *unit*-testidega on kasutajaliidese testid väga ebastabiilsed ning võivad lakata töötamast ka siis, kui tarkvara tegelikult toimib. Teiseks tähtsaks järelduseks on, et kasutajaliidese testide loomine on väga sarnane *unit*-testide loomisel. Erineb ainult tarkvara osa, mida testitakse, kuna testimine käib erinevatel tasanditel. Seega saab rakendada *unit*-testimise põhimõtteid UI-testimisel ning vastupidi. Kolmandaks järelduseks on, et ajakulu on kasutajaliidese testide puhul suureks probleemiks ning UI-testijale on väga tähtis omadus osata teste optimeerida nii, et ajakulu oleks minimaalne.

Antud tööst võib leida mitmel eri teemal edasiarendusi. Kuigi töö käigus sai parandatud olemasolevad UI-teste, siis nende optimiseerimist ajakulu mõttes ei tehtud. Selleteemaline põhjalik analüüs ning teooria rakendamine koodis on üks võimalik edasine suund. Peatükis, mis analüüsis tööriistade puudusi, toodi välja küll puudused ja võimalikud lahendused, kuid lahenduste realiseerimine ei olnud osa töö skoobist. Seega oleks üheks heaks edasimineku suunaks nimetatud puuduste reaalne likvideerimine antud tööriistades.

1. Kas eesmärk saavutati?

Töö praktilisteks eesmärkideks oli testide parandamine ning uute loomine. Töö käigus sai kirjeldatud probleeme ning lahendusi, mis reaalselt koodis sisse viidi. Lahenduste abil parandati kõik testid, mis eelnevalt ebaõnnestusid. Probleemide ja lahenduste kirjeldus peaks kindlasti andma lugejale reaalse tarkvara näitel ülevaate, kuidas alustada samasuguse töö läbiviimist. Kuna töö on läbi viidud algaja testija käe läbi, siis on tegu kasuliku näidismaterjaliga teistele algajatele UI-testijatele, mis oli samuti üheks töö eesmärgiks. Analüüsipeatükkidest ilmnesid järeldused ning tulemused, mis annavad võimaluse tulevasteks edasiarendusteks, mis on samuti iga sellise töö eesmärgiks.

2. Põhitulemuste loetelu

Ülesannete püsititusest tulenevalt on põhilised tulemused, mis antud tööga saavutati, järgmised:

1. Parandatud said kõik töö alguses ebaõnnestuvad UI-testid. Analüüsi põhjal viidi sisse parandused vastavalt kas testis või tarkvaras.

2. Eesti Telekomi veebirakenduse UI-testidega kaetus suurenes uute testide kirjutamisega palju – peaaegu kõik kasutatavad *portlet*'id veebirakenduses said UI-testidega kaetud.

3. Töö käigus ilmnenud puudused kasutatud tööriistades said loetletud.

4. Testtsükli pikkuse muutumine sõltuvalt testide hulgast sai analüüsitud – analüüsi tulemustest selgus, et töö raames kirjutatud testid on kiiremad, kui varasemalt eksisteerinud testid, kuna keskmine ajakulu testi kohta vähenes.

3. Kas eesmärgid saavutati?

Nagu eelmise küsimuse vastusest näha on, siis töö põhilised ülesanded said täidetud. Kuigi testidega ei kaetud kogu Eesti Telekomi veebirakendust, siis sellel on pigem ärilised põhjused. Muud eesmärgid said täidetud täielikult ning nende tulemused on rakenduse testtsükli ajakulus silmnähtavad. Antud töö põhjal on võimalik analüüsida tarkvara testide ebaõnnestumise põhjusi mis tahes teisel tarkvaral. Samuti on selle töö üldiseid ideid testide

loomisel võimalik rakendada mis tahes tarkvarale. Seega võib öelda, et antud tööd on võimalik kasutada õppevahendina, seega on ka see sissejuhatuses esile toodud eesmärk täidetud.

Antud töös loodud teste ning testide parandusi oleks saanud teha korrektsemalt. Näiteks tööriistade puuduste analüüsis mainitud Login.java ja ResetBaseUrls.java klassides olevatele probleemidele sai töö käigus mõningal määral kaasa aidatud. See tulenes sellest, et nende täiustamine oleks olnud liialt suure ajakuluga ning see on ressurss, mida tuleb firmas väärtustada. Tulevikus saaks kindlasti edasi minna antud peatükis välja toodud puuduste likvideerimisega.

8. Summary

The main objectives of this thesis were to give an overview of methods and theory used to fix and create UI-tests. The analysis topics in the thesis were meant to result in possible future thesis topics or possible future developments by the author. Finally, the entire thesis was meant to be as a guide or educational material for beginner UI-testers or other automated test creators.

The biggest problem covered in this thesis was the fixing of UI-tests. First it was necessary to determine which is failing – the software or the test. Then a proper solution was to be found and finally it had to be applied in code. Secondly, parts of the software not covered with automated user interface tests were to be covered with UI-tests. That involved finding use cases per portlet and creating automated tests using clean code theory and practises already in use at the company the software belonged to. Finally, analysis of the issues that surfaced during the creation of this thesis was conducted.

The main result of this thesis software wise was the fixing of all UI-tests failing at the time and creating UI-tests for almost all portlets used in the software. In addition, the analysis of issues with testing tools and the analysis of the time it takes to run a full UI-test cycle provided future topics to cover by either the author of this thesis or the readers. Finally, as the full process of fixing tests and creating tests was described in the thesis it functions well as a guide for aspiring or beginner testers.

Kasutatud kirjandus

[1] „Selenide 2.4 API documentation,“ [Võrgumaterjal]. Available: http://selenide.org/javadoc/2.4/com/codeborne/selenide/Selenide.html.

[2] J. Lent, „Automated testing tools: Four reasons why projects fail,“ August 2013. [Võrgumaterjal]. Available: http://searchsoftwarequality.techtarget.com/opinion/Automated-testing-tools-Four-reasons-why-projects-fail. [Kasutatud 8 October 2015].

[3] „Automated Software Testing,“ Atlassian, [Võrgumaterjal]. Available: https://www.atlassian.com/software-testing/?tab=automated-software-testing. [Kasutatud 8 October 2015].

[4] D. Chernicoff, „4 Reasons to Love Your Log Data,“ 21 April 2014. [Võrgumaterjal]. Available: https://blog.logentries.com/2014/04/love-your-log-data/. [Kasutatud 8 October 2015].

[5] L. Dargis ja A. Weintrob, „How to Write a Quality Bug Report,“ 2 July 2014. [Võrgumaterjal]. Available: http://university.utest.com/writing-quality-bug-reports-and-utest-etiquette/. [Kasutatud 8 October 2015].

[6] „Selenium's old code repository issue discussions,“ [Võrgumaterjal]. Available: https://code.google.com/p/selenium/issues/detail?id=40.

[7] „Selenium WebElement documentation,“ [Võrgumaterjal]. Available: http://selenium.googlecode.com/svn@7093/trunk/docs/api/java/org/openqa/selenium/WebElement.html. [Kasutatud 14 October 2015].

[8] „What is use case testing in software testing?,“ [Võrgumaterjal]. Available: http://istqbexamcertification.com/what-is-use-case-testing-in-software-testing/. [Kasutatud 14 October 2015].

[9] R. Zafar, „What is software testing? What are the different types of testing?,“ 20 March 2012. [Võrgumaterjal]. Available: http://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty. [Kasutatud 14 October 2015].

[10] „Clean Code - A Handbook of Agile Software Craftmanship,“ [Võrgumaterjal]. Available: http://it-ebooks.info/book/4263/. [Kasutatud 14 October 2015].

[11] „Selenium Google Code repository issue tracker discussions,“ [Võrgumaterjal]. Available: https://code.google.com/p/selenium/issues/detail?id=40#c16. [Kasutatud 9 November 2015].

Lisa

Bakalaureusetöö käigus loodud kood on kättesaadav veebis aadressil:

https://www.dropbox.com/sh/3ees4r0vqk8reb3/AABiRSRy46NBaIXVTROx8Vzpa?dl=0

Lähtkoodis on tehtud muudatusi, et mitte avalikustada firmasisest informatsiooni. Eemaldatud info kajastub koodis kolme punktina (...).