TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Kasutajaliidese automaatne regressioontestimine agiilse tarkvaraarenduse põhimõtetel

Bakalaureusetöö

Üliõpilane: Lauri Varendi
Üliõpilaskood: 094066IABB
Juhendaja: Dotsent Enn Õunapuu

Tallinn
2016

Autorideklaratsioon

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

……. ……

(*kuupäev)* *(allkiri)*

Annotatsioon

Käesoleva bakalaureusetöö eesmärkideks on tutvustada tarkvara projekti alustamisel tehtavaid vajaminevaid samme, et tarkvara testimine oleks maksimaalselt efektiivne. Põhjalikumalt keskendutakse kasutajaliidese testimisele, sest see on tarkvara osa, mida lõppkasutajad näevad ja mille korrektse toimimise järgi nad enda hinnanguid tarkvarale annavad.

Tarkvara arendatakse üha enam agiilsete põhimõtete järgi, mis tähendab, et tarkvara testides tuleb samuti kasutada agiilseid meetodeid, sest traditsioonilised meetodid ei ei anna parimat võimalikku tulemust. Käesolevas töös kirjeldatakse agiilseid tarkvaraarendus meetodeid, mida tuleks jälgida nii rakendust kui ka teste programmeerides.

Töö käigus loodud testide najal näidatakse, kuidas seada üles keskkond automaatsete testide kirjutamiseks projekti alustades ning kuidas kirjutada ja hallata automaatseid regressioonteste kasutajaliidesele, et tagada projekti alustamisest peale maksimaalne kvaliteet.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 3. peatükki, 4 joonist.

Abstract

As more and more software is being developed using agile methods, software testing should be capapble of keeping up with development aswell. That is why it is necessary to automate as many of the testing processes as quickly as possible to insure best quality for the software.

This research paper explains how to setup automated regression testing framework using agile methods. The main focus is writing tests for user interface, because that is the part that the end users use to give their opinions on the software based on the correct behaviour of the interface.

Writing, running, maintaining and fixing automated user interface tests should consider certain best practicies to be efficient and useful. These practicies are introduced in this paper.

The tests created in this resarch paper introduce how to set up the framwork to write and maintain automated tests in the start of the software project. Also, how to write automated user interface regression tests to insure the best quality for the software.

The thesis is in Estonian and contains 35 pages of text, 3 chapters, 4 figures.

Lühendite ja mõistete sõnastik

80/20 reegel ***80/20 rule***

80/20 reegel, nimetatakse ka Pareto reegliks, tähendab, et tüüpiliselt 80% tulemustest sõltub 20% tegevustest [11].

Agiilne manifest ***Agile manifesto***

Agiilne manifest on moto, mis loodi tarkvaraarendajate poolt, kes soovisid alternatiivset varianti traditsioonilisele rohke dokumentatsiooni kirjutamisele orienteeritud tarkvaraarendusele. Agiilne manifest loetleb neli põhilist arendamise põhimõtet: Isikud ja koostöö on tähtsamad kui protsessid ja vahendid, töötav tarkvara on tähtsam kui põhjalik dokumentatsioon, klientidega koostöö on tähtsam kui lepingu läbirääkimised ja muutustele reageerimine on tähtsam kui esialgse plaani jälgimine [1].

Agiilne tarkvaraarendus ***Agile software development***

Agiilne tarkvaraarendus on metoodika, mis näeb ette vajalikkust paindlikkule ja pidevale arengule projekti arendamise ja väljastamise protsessis. Agiilse tarkvaraarenduse eesmärgiks on ehitada ja väljastada väikseid klientide poolt heakskiidetud rakenduse osasid niipea, kui need on valminud, kogu projekti arendustsükli vältel, samal ajal hoides koodi lihtsana ja rakendust pidevalt testides [16].

Backlog ***Backlog***

Backlog on prioriseeritud list arendamist vajavate tarkvaraosade kirjeldustest, mida pole veel arendama hakatud.

Bugi ***Bug***

Bugi on tarkvara arenduse käigus tekkinud soovimatu tulemus või viga, mis takistab töö tegemist või mille põhjusel ei ole tegevuse tagajärjel võimalik saada soovitud tulemust.

Build ***Build***

Build või *build version* tähendab konkreetse rakenduse versiooni, kirjutatud koodi, paketiks kokkupanemist. Koostatud paketti saab seejärel

	kindlasse keskkonda üles seada, et rakendust kasutada.
Deploy	***Deploy*** *Deploy* tähendab valitud arendusversiooni rakenduskeskkonda kasutamiseks üleslaadimist.
Dünaamiline süsteemiarendamise meetod	***Dynamic systems development method (DSDM)*** Dünaamiline süsteemiarendusmeetod on agiilne projektijuhtimise ja -väljalaskmise raamistik, mille filosoofiaks on, et iga projekt tuleb reastada kindlalt defineeritud strateegilistele eesmärkidele ja fokuseerima ärile reaalselt kasulike vajaduste varajasele üleandmisele.
Kanban	***Kanban*** *Kanban* on agiilne arendusmeetod, mis keskendub toote loomisel järjepidevale väljaandmisele samal ajal ülekoormamata arendusmeeskonda. Nagu *Scrumgi*, on *kanban* loodud aitamaks meeskonnal paremini koostööd teha. *Kanbani* puhul visualiseeritakse töökorraldus, et saada projekti seostest parem ülevaade, limiteeritakse käimasoleva töö hulka, et meeskond ei pühenduks liiga paljule ja kohandatakse töökorraldus nii, et ühe osa lõpetamisel võetakse *backlogist* järgmine kõrgeima prioriteediga töö [23].
Koskmudel	***Waterfall*** Koskmudel (ka klassikaline mudel) on esimene kirjeldatud tarkvarasüsteemi elutsükli mudel, mis lähtus tavalistest toomisprotsessidest ehituses, mehaanikas vms. Mudeli kirjeldas Winston W. Royce 1970. Aastal. Koskmudel on kõige vanem ja kõige rohkem kritiseeritud protsessimudel. Põhiidee kohaselt jagatakse tegevused nii, et iga tegevus toimub jadamisi eraldi etapina. Protsess on jagatud järgnevateks etappideks: nõuete määratlemine, süsteemi ja tarkvara kavandamine, teostus ja moodulite testimine, integratsioon ja süsteemi testimine, kasutamine ja hooldus [7].
Pidev deployment	***Continuous deployment (CD)*** Pidev *deployment* on agiilse tarkvara arendamise praktika, mille järgi seatakse pidevalt vajaminevasse keskkonda üles uusim arendatud versioon.

Pidev integratsioon ***Continuous integration (CI)***

Pidev integratsioon on agiilse tarkvara arendamise praktika, mille järgi peavad arendajad kirjutatud töötava koodi mitu korda päevas ühtsesse repositooriumisse lisama. Tänu sellele avastatakse koodi kirjutamisel tekkinud vead kiiresti.

Scrum ***Scrum***

Agiilne tarkvaraarenduse raamistik, milles keskendutakse kasvavate ja muutuvate nõuete ning probleemide lahendamisega meeskondlikult ja pideva tagasisidega, et pidevalt välja anda kõrge kvaliteediga toodet [19].

Tarkvara väljalase ***Software release***

Tarkvara väljalase on tarkvararakenduse viimase versiooni väljajagamine. Tarkvara väljalase võib olla avalik või privaatne ning kujutab endast tarkvararakenduse esialgset või täiendatud versiooni [17].

Testilabor ***Test lab***

Testilabor on testimise keskkond, mis peab tagama stabiilsuse, et läbi viia kontrollitud taasesitatud teste.

Testipõhine arendus ***Test driven development (TDD)***

Testipõhine arendus on lähenemine arendusele selliselt, et kõigepealt kirjutatakse arendatavale osale test ning alles seejärel just piisavalt rakenduskoodi, et testitingimusi täita [3].

Vabatarkvara ***Open source software***

Vabatarkvara on tarkvara, mille lähtekood on avalikult tasuta kättesaadav ning mida saab igaüks kasutada või täiendada enda vajadustele vastavalt.

Ühiktestid ***Unit tests***

Rakenduse koodi testimiseks rajatavad testid. Ühiktestid kirjutatakse meetodite tasemel ja kindlatele väikestele koodi osadele. Ühikteste kirjutatakse enne uue koodi kirjutamist ja jooksutatakse pidevalt arenduse käigus.

XP ***XP, Extreme programming***

Tarkvaraarenduse raamistik, mis põhineb lihtsusel, suhtlusel, tagasisidel ja austusel. Raamistiku järgi antakse meeskonnale pidevalt piisavalt tagasisidet, et kõik liikmed saaksid ülevaate, mis seisus arendus on ning

selle läbi tagada, et igale unikaalsele situatsioonile leitakse kiire lahendus [10].

Sisukord

Jooniste loetelu

1. Sissejuhatus

Tarkvaraarendus on tänapäeval väga aktuaalne teema. Ettevõtted tahavad, et nende toode või teenus oleks inimestele lihtsalt kättesaadav ning kuna elutempo on kiire, siis kliendid soovivad sageli tulemusi võimalikult varakult näha, mistõttu üha rohkem kasutatakse tarkvara arendamisel agiilseid põhimõtteid. See annab tarkvara loojatele võimaluse töö valmimist jooksvalt demonstreerida ning vajalikku tagasisidet saada. Agiilselt arendades muutuvad versioonid kiiresti, mistõttu on vaja tarkvara pidevalt testida veendumaks, et juurdearendatav osa ei lõhuks olemasolevat ja ei tekiks probleeme. Arenduse mahu kasvades suureneb ka testimise vajadus ning seetõttu tuleb lisaks manuaalsele testimisele kasutada ka automaattestimist, et tagada tarkvara kvaliteet.

1.1 Taust ja probleem

Töö autor on töötanud erinevates ettevõtetes mitme tarkvaraprojekti loomises. Tarkvara arendamine on toiminud agiilsete meetodite alusel ja testimine on suures osas manuaalne. Nõuded projektile muutuvad pidevalt ning arendusi soovitakse välja lasta pidevalt 3-4 nädala tagant.

Projekti alustades ei panda tihti piisavalt rõhku testsüsteemi ülesseadmisele. Suure funktsionaalsuse ja pidevalt areneva rakenduse testimine manuaalselt muutub väga raskeks, sest lisaks uutele arendustele on vajalik kindlaks teha, et ei ole tekkinud regressioonvigu juba valmis süsteemis. Automaattestimine võetakse kasutusele hetkel, kui manuaalselt ei ole enam võimalik kõike testida. See hetk on aga liiga hilja alustamiseks, sest automaattestide kirjutamine võtab aega, kuna on vajalik katta kõik olemasolev rakendus. See omakorda tähendab, et kannatab uute arenduste testimine, sest testija peab samaaegselt tegelema ka automaattestide loomisega varem arendatud osadele. Protsessi parandamiseks tuleks kirjutada automaatteste samal ajal või vahetult pärast mõne tarkvaraosa arendamist.

1.2 Ülesande püstitus

Antud bakalaureusetöö eesmärgiks on anda ülevaade, kuidas kaasata automaattestimist uut tarkvararakendust luues ning praktiliselt näidata, kuidas luua kasutajaliidesele automaatseid regressioonteste.

1.3 Metoodika

Töö eesmärkide saavutamiseks analüüsitakse agiilses arenduskeskkonnas automaattestide keskkonna valiku ja ülesseadmise põhimõtteid. Analüüsitakse kasutajaliidese automaattestimise kirjutamist ja haldamist. Valitakse välja sobivad vahendid rakenduse automaattestide kirjutamiseks. Automatiseeritakse kasutajaliidese regressioontestid.

1.4 Ülevaade tööst

Bakalaureusetöö on jaotatud kolmeks peatükiks. Esimeses peatükis tutvustatakse tarkvaraprojekti loomise alustamisega seotud tegevusi. Antakse ülevaade agiilsetest arendusmetoodikatest ning kirjeldatakse põhjalikumalt testimise rolli agiilses tarkvara arendustsüklis. Peatüki lõpus selgitatakse rakenduse testkeskkonna valimise ja ülesseadmise protsessi.

Teises peatükis seletatakse lahti erinevad kasutajaliidese vormid ning tuuakse välja hea kasutajaliidese omadused. Kirjeldatakse, kuidas arendada kasutajaliidesele automaatteste ning kuidas neid hallata.

Kolmandas peatükis kirjeldatakse töö praktilise osana koostatud automaatseid regressioonteste kasutajaliidesele, nende kirjutamise vahendeid ning koodinäiteid. Kirjeldatakse põhimõtteid, mille järgi regressioontestid planeeriti ja koostati.

Töö kokkuvõttes tuuakse välja tekkinud probleemid ja saadud õppetunnid, millele saab tarkvara arendades ja testides lähtuda, et tagada edaspidi parem rakenduse kvaliteet.

2. Tarkvararakenduse arendamine

Ettevõtte arengu ja jätkusuutlikkuse tagamiseks on üheks eelduseks kaasaegse tehnoloogiaga kaasaskäimine. Seepärast arendatakse pidevalt uusi tarkvararakendusi, et kauba või teenuse pakkumine oleks lihtsustatud. Rakendus luuakse ärivajadusi arvestades ning selle eesmärgid vastavad ettevõtte eesmärkidele konkreetse projekti jaoks. Tänapäeval on tarkvara loomiseks palju vahendeid, mistõttu nende valimine sõltub kindla projekti vajadustest ja rakenduse arendamisega tegelevate inimeste kogemustest ja oskustest.

Kuna käesoleva bakalaureusetöö raames vaadatakse agiilselt arendatavate rakenduste testimist, siis järgnevas peatükis selgitatakse agiilseid arendusmeetodeid. Seejärel tuuakse välja, milliste põhimõtete alusel valida keskkond ja vahendid tarkvara agiilseks testimiseks. Peatüki lõpus antakse ülevaade, kuidas luua häid automatiseeritud regressioonteste.

2.1 Agiilsed arendusmeetodid

Esmased agiilsete arendusmeetodite kasutuselevõtjad olid väikesed iseseisvad meeskonnad, kes töötasid väikeste iseseisvate projektidega. Nad tõestasid, tarkvara loojate meeleheaks, et agiilne mudel töötab. Mida aeg edasi, seda rohkem ja suuremad ettevõtted on hakanud kasutama agiilseid võtteid ning otsivad lahendusi, et neid kasutada kogu organisatsiooni tegevuses [15].

Traditsioonilised projekti juhtimismeetodid nagu koskmudel, ehitavad kindlates etappides. Selline tootearendus on rajatud tarkvara ühekordsele, suurele, kõrge riskiga väljalaskele. Kui projekt liigub ühest etapist teise, siis muutub vana etapi juurde tagasiminek vaevaliselt, sest meeskond surub alati edasi järgmise poole. Traditsiooniliselt projekti juhtides luuakse tihti kriitilisi piire, kust projekt ei saa enne edasi liikuda, kui blokeerivad probleemid on lahendatud. Lisaks ei saa lõppkasutajad tootega tutvuda enne selle täielikku valmimist, mistõttu ei ole võimalik neilt saada tagasisidet disaini ega funktsionaalsuse osas [15].

Agiilne projekti juhtimismeetod seevastu läheneb arendusele iteratiivselt koos regulaarse tagasisidega tehtud tööle. Iteratsioonid annavad meeskonnale võimaluse töö ümber suunata projekti osadele, mis on hetkel tähtsaimad. See tagab pidevad võimalused arendada, välja

anda, õppida ja kohaneda. Turu muudatused ei häiri arendamist, sest meeskond on valmis ja võimeline kohanema kiiresti uute nõudmistega.

Agiilsel tarkvaraarendusel on mitu põhimõtet, mis kehtivad erinevatele agiilsetele metoodikatele. Need omadused teevad selle fundamentaalselt erinevaks traditsioonilisest koskmudelist ning neid tuleks järgida, et arendus toimiks ilma probleemideta. Nendeks põhimõteteks on:

1. Aktiivne kasutajate kaasamine on hädavajalik
2. Meeskonnal peab olema otsustusõigus lahenduste osas
3. Nõuded muutuvad, kuid ajakava jääb samaks
4. Tähtsad ärinõuded tuleb kirjeldada lihtsalt ja visuaalselt
5. Arendada tuleb väikeselt ja kasvavalt ning kasutada iteratsioone
6. Tuleb fokuseerida tihedatele tarkvara väljalasetele
7. Arendatav tarkvaraosa tuleb lõpetada enne, kui alustada järgmist
8. Rakenda 80/20 reeglit
9. Testimine peab olema integreeritud läbi kogu projekti arengutsükli, testida tuleb varakult ja tihedalt
10. Kõigi osapoolte pidev koostöö on vajalik

On erinevaid meetodeid, mida peetakse agiilseteks, sest nad järgivad eeltoodud põhimõtteid ja üldist agiilset manifesti. Enimkasutatavad on dünaamiline süsteemiarendamise meetod, mida peetakse originaalseks agiilseks arendusmeetodiks, sest see eksisteeris enne agiilse mõiste kasutuselevõttu. *Scrum*, mis keskendub projekti jaotamiseks väiksemateks tööülesanneteks meeskondlikus arenduskeskkonnas ning mis on kõige laialdasemalt kasutuselevõetud agiilne meetod, sest seda on lihtne implementeerida ja see lahendab arendusmeeskondi vaevanud projektijuhtimise probleeme. *XP*, mis keskendub rohkem tarkvaratehnikale ja läheneb analüüsile, arendamisele ja testimisele uudselt [24].

Agiilseid meetodeid on teisigi, näiteks *kanban*, mis võeti esmalt kasutusele Toyotas, kuid tänu agiilsete põhimõtete kasutuselevõttu tarkvaraarenduses, hakati kasutama ka tarkvaraprojektide juures. *Kanban* sarnaneb *Scrumile*, kuid seal ei kasutata kindlaid ajalisi sprinte vaid ülesandeid võetakse vastavalt vajadusele *backlogist* ning välja antakse tarkvaraosad, mis on valmis arendatud.

Uue tarkvararakenduse loomisel on ettevõttel ja arendusmeeskonnal palju võimalusi, kuidas toodet arendama hakata. Meetodi valikul tuleb kindlaks teha projekti eesmärgid, kuid lõplik otsus oleneb projektijuhi ja meeskonna varasematest kogemustest ja eelistustest. Agiilselt arendades on eeliseks, et meetodeid võib ühildada, võttes kasutusele erinevaid tehnikaid erinevatest meetoditest, näiteks sprinte *Scrumist* ja paarisprogrammeerimist *XP-st*.

Eeltoodud põhimõtted kehtivad ka konkreetsete arendusvahendite valimisel. Arenduskeele, andmebaasi, versioonikontrolli ja muu vajaliku valimisel sõltutakse meeskonna oskustest ja teadmistest, et tarkvara arendamisega saaks alustada võimalikult kiiresti ilma, et arendajad peaksid liigselt eeltööd tegema ja uusi tehnikaid õppima.

Ainuke viis olla tõeliselt agiilne, on automatiseerida nii palju rutiinseid protsesse kui võimalik. Testide automatiseerimine on üks kriitilisemaid ülesandeid, sest agiilselt arendades muutub rakenduse versioon tihti ning seetõttu tuleb sama osa testida korduvalt. Manuaalselt seda teha muutub pidevalt raskemaks, sest rakenduse maht kasvab pidevalt ja sellega koos ka testimist vajav hulk. Automatiseeritud testid annavad võimaluse pidevalt üle kontrollida, et rakendus toimib endiselt nii nagu oodatud. *Build* keskkonna automatiseerimine annab võimaluse valmis versiooni kokkupanekuks ja klientidele demonstreerimiseks siis kui vajadus tekib. Automatiseerimine tuleb kasuks, kui protsessi on vaja teha rohkem kui ühe korra. Mida rohkem automatiseerida, seda rohkem saab keskenduda uute komponentide arendamisele [6].

2.2 Tarkvara testimine

Tarkvara testimine on meetod identifitseerimaks vigu tootes, enne kui see kasutajatele suunatakse. See sisaldab tarkvara vigu koodis ja töötamises kui ka võimalikku vahet selles, mida kood tegema peaks ja mida ta tegelikult teeb. Enne testimisega alustamist tuleb selgeks teha peamised testimise põhimõtted, milleks on järgnevad: testimine võib näidata, et probleemid eksisteerivad, aga mitte, et probleeme ei eksisteeri; testimine katab piiratud arvu näitesituatsioone ja ei ole kunagi täielik; mida varasemas tarkvaraarenduse etapis viga

avastatakse, seda odavam on seda parandada; vead koonduvad spetsiaalsete alade ümber (näiteks süsteemi keerukus või meeskonna kogenematus); komplekt teste, mis käivitatakse korduvalt, omavad üha vähem efekti; testi tüüp, disain ja fookus erineb vastavalt sellele, millist tarkvara testitakse; test, mis ei paljasta ühtegi viga ei ole tõestus, et tarkvara on veatu. Mida enam nende põhimõtetega arvestatakse, seda efektiivsem on testimine ja kvaliteetsem tarkvara [5].

2.3 Testimine agiilses tarkvaraarenduses

Koskmudeli järgi projekti arendades lahutatakse arendus ja testimine kaheks eraldi sammuks. Arendajad kodeerivad tarkvara funktsiooni ning seejärel annavad selle testijatele kvaliteedikontrolliks, kes seejärel kirjutavad ja täidavad detailse testplaani. Lisaks dokumenteerivad testijad regressiooniga tekkinud vigu, mis on avaldunud varem valmis tehtud funktsioonides uue töö lisamisega. Meeskonnad, kes koskmudelit kasutavad, avastavad tihti, et projekti mahu kasvades suureneb testimist vajav hulk eksponentsiaalselt, mistõttu on testijatel raske arendajatega sammu pidada. See viib olukorrani, kus projektiomanikul on valik, kas tarkvaraversiooni välja andmine edasi lükata või mitte nii põhjalikult testida. Tihti valitakse sellises situatsioonis teine variant. Selle tõttu kasvab tehniline võlg ja projekti kasvades suureneb vigade hulk, mida on üha raskem parandada, sest arendajad on nende avastamise hetkeks tööga edasi liikunud ning erinevate koodibaaside vahel navigeerimine on keerukas ja ajamahukas [14].

Agiilsete põhimõtete järgi ei ole testimine eraldiseisev samm, vaid arenduse pidev osa. Arendajad ja testijad teevad pidevalt koostööd ja tagasiside andmine toimub igapäevaselt. Lisaks ühendatakse arendus ja testimine ka aja ja protsesside tasemel. Kõige selle tulemusel väheneb vajadus spetsiaalsete testimeeskondade järele, samal ajal endiselt austades mõlema rolli vajadust ja tähtsust arendusprotsessis. Võib öelda, et agiilses süsteemis julgustatakse arendajaid mõtlema rohkem testimisele, pidevalt enda koodi kontrollides ning testijaid mõtlema arendamisele, automatiseerides oma teste. Sedasi luuakse toote väljaandmise meeskond, kus kõigil liikmetel on ühiseks sihiks kvaliteetse rakenduse arendamine juba esimesel katsel [8].

Selle eelduseks on, et testimisega tegelevad kõik meeskonnaliikmed. Kuigi meeskonda on vaja kindlate testimisoskustega inimesi manuaalseks testimiseks, siis automaattestimise tasemel annavad suure panuse arendajad ühiktestidega. Nimelt kasutatakse agiilses arenduses

testipõhist arendust, kus enne funktsionaalse koodi kirjutamist kirjutatakse ühiktestid. Kuna arendusele suudab üldjuhul kompetentseima testi kirjutada selle tegija, siis on ühiktestide loomine arendajate ülesanne.

Kuigi ühiktestid on hea viis koodi testimiseks, siis ei ole nad regressiooni testimiseks eriti efektiivsed. Agiilselt arendades versioonid muutuvad tihti, mistõttu tuleb regressiooni pidevalt testida. Selleks tuleb testida rakenduse tasemel kasutajaliidest. Et mitte kulutada iga versioonivahetuse korral liigselt aega regressiooni testides, on kasulik luua automatiseeritud kasutajaliidese regressioonteste. Neid kirjutavad üldjuhul testijad, sest pikas perspektiivis vähendab see nende töö mahtu [4].

Pidevalt automaatseid teste kirjutades väheneb aeg, millal saab arendada rakenduse funktsionaalsust. Seetõttu muutub arendus aeglasemaks ja eriti märgatav on see projekti alguses. Siiski annab pidev testide kirjutamine suurema kindluse, et projekti arenedes ei teki palju probleeme ja rakendus on kvaliteetne. Lisaks avastatakse tekkinud bugid kiiresti ja arendaja saab kohest tagasisidet enda töö kohta, mis muudab paranduste tegemise kergemaks, sest arendajal on tehtud töö värskelt meeles. Seetõttu õigustab agiilne testimine aeglasemat arendustempot, sest tagab sellega arenduse pideva ja jätkusuutliku kvaliteedi.

Agiilne testimine on kasulik ka põhjusel, et üldjuhul puudub rakendusel täielik analüüs. Traditsiooniliste meetodite järgi arendades saaks uued nõuded lisada alles järgmises arengutsüklis ning puuduliku analüüsi tulemusel võib projekti väljaandmine edasi lükkuda. Agiilselt testides on võimalik seevastu arenduse käigus avastatud uued nõuded jooksvalt sisse kirjutada, sest nende testimine toimuks koheselt pärast arendust [2].

2.4 Agiilse testikeskkonna valiku ja ülesseadmise põhimõtted

Projekti alustades on vaja üles seada keskkond, mille alla kuuluvad osad nagu tööala, riistvara ning arendustarkvara ja -vahendid. Lisaks on vaja üles seada testikeskkond. Seda peab tegema nullist, kui testikeskkond puudub täielikult või kohandama olemasolevat keskkonda selliselt, et see toetaks testimisvajadusi. Testikeskkonna organiseerimiseks on soovitatav järgida järgnevaid põhimõtteid [2].

1. Kasutada arendamiseks vabatarkvara. On olemas palju häid vabatarkvara vahendeid, näiteks xUnit C# või .NET raamistikele ja JUnit Java raamistikele. Need vahendid on

suunatud arendajatele, sest nad on mõeldud ühiktestide kirjutamiseks. Lisaks on neid vahendeid reeglina lihtne üles seada ja kasutama õppida.

2. Kasutada nii vabatarkvara kui ka kaubanduslikke tarkvara vahendeid testimiseks. Kuna testijad tegelevad tihti keerulisemate testiülesannetega nagu integratsioonitestimine üle mitme süsteemi, turvalisuse testimine, kasutatavuse testimine, regressioontestimine jne, siis on neil vaja vahendeid, mis oleks piisavalt arenenud, et kõige sellega hakkama saada.

3. Ühine süsteem bugide jälgimiseks. Agiilses arenduses annavad testijad tihedalt tagasisidet uute bugide kohta. Et tagasiside oleks hea ei piisa ainult sõnadest, vaid märkused tuleb kirja panna ja visuaalselt edastada ja raporteerida. Kui väikeste meeskondade puhul võib kasutada lihtsaid meetodeid näiteks märkmeid, siis parema ülevaate ja keerulisemate süsteemide puhul on mõttekas kasutada tarkvaralist projektihaldamise süsteemi, kuhu pannakse kirja nii ärinõuetest tulenevad ülesanded kasutajalugudena kui ka tekkinud bugid. Tänu sellele on tervel meeskonnal ülevaade valmis tehtud ja eelseisvast tööst ning tekkinud probleemidest.

4. Kasutada testimisriistvara. Nii arendajatel kui testijatel on kindlasti vaja riistvara, mille abil rakendust testida.

5. Kasutada virtualiseerimis- ja testilabori haldamise vahendeid. Testimisriistvara on kallis ja seda ei ole kunagi liiga palju. Virtualiseerimisvahendid, mis võimaldavad laadida testkeskkonna riistvara peale lihtsalt ja testilabori haldusvahendid, mis võimaldavad jälgida riistvara konfiguratsiooni, on kriitilised, tagamaks testimise õnnestumist, eriti kui testijad töötavad koos mitme arendusmeeskonnaga.

6. Kasutada pideva integratsiooni ja pideva *deployment* vahendeid. Ei ole küll eraldi testimiseks vajalik, kuid need vahendid on väga tähtsad agiilses tarkvaraarenduses, sest kasutatakse praktikaid nagu testipõhine arendus ja üldine regressioontestimine ning eksisteerib vajadus töötava *buildi deploymine* testkeskkonda, näitekeskkonda või toote lõppkeskkonda. Selliseid vahendeid leiab palju vabavarana.

Järgides eeltoodud põhimõtteid on rakenduse arendamise algusest tagatud, et suudetakse alustada testimisega nii varakult kui võimalik, mis omakorda tagab kvaliteetse rakenduse. Tekkinud vead avastatakse kiirelt ning seetõttu on neid odavam lahendada.

3. Kasutajaliides

Kasutajaliides on enamasti ainuke osa tarkvararakendusest, mida lõppkasutaja näeb ja kasutada saab. Hästi disainitud ja vigadeta töötav rakenduse kasutajaliides on ettevõtte maine hoidmiseks ja funktsionaalsuse tagamiseks tähtis komponent. Järgnevas peatükis kirjeldatakse kasutajaliidese rolli ja nõudeid tarkvararakenduses. Seejärel tuuakse välja erinevaid kasutajaliidese tüübid. Peatüki lõpus selgitatakse kasutajaliidese automaattestimist ning kirjutatud testide haldamist.

3.1 Kasutajaliidese roll ja nõuded tarkvararakenduses

Arvutisüsteemidel on erinevaid tüüpi liideseid. Tehnilised liidesed võivad ühendada erinevaid arvutisüsteeme omavahel või füüsiliste protsessidega. Riistvaraliidesed ühendavad arvuti erinevaid seadmeid omavahel ja vooluallikaga. Rakendusliides on operatsioonisüsteemi või rakendusprogrammiga määratud reeglistik, mille alusel rakendusprogramm kasutab operatsioonisüsteemi või teise rakendusprogrammi teenuseid. Kõigi nende liidestega suhtleb kasutaja kaudselt, tehes seda kasutajaliidese abil. Arvutisüsteemi kasutajaliides võimaldab kasutajal suhelda nii operatsioonisüsteemi kui rakendusprogrammidega, olles ise samal ajal samuti programm ehk tarkvarasüsteem [12].

Kasutajaliides on vahend, mis võimaldab inimesel suhelda masinaga ehk kõik, mida näeme, kuuleme, tunneme. Teised süsteemi osad on harilikult varjatud. Arvutisüsteemi kasutajaliidese välisteks komponentideks loetakse ka ekraani, klaviatuuri, hiirt ja muid vahendeid, millelt signaale vastu võetakse. Kasutajaliidese all võib mõista ka sisend- ja väljundseadmeid: ühelt poolt on vaja vahendeid, mis võimaldavad arvutile käske edastada, ning teiselt poolt vahendeid, mis võimaldavad arvuti poolt teostatud operatsioonide tulemust tajuda [12].

Tarkvarasüsteemis on kasutajaliides vahend rakendusega suhtlemiseks. Suure osa tavakasutajate jaoks ongi kasutajaliides niiöelda kogu rakendus, seepärast tuleks lähtuda kasutajaliidese arendamisel kindlatest põhimõtetest. Kasutajaliides peab olema selge ja järjepidav, lihtne kasutada ja aru saada ning atraktiivne. Lisaks tuleb olulist osa rõhutada ja ebaoluline hüljata. Kasutajaliidese eesmärkideks peaks olema ka kiire töökäskude sooritus,

emotsionaalse sideme tekitamine, usaldusväärsus ja eksklusiivsus. Kõike eelnevat tuleb arvesse võtta nii kasutajaliidese disainimisel, arendamisel kui testimisel [22].

Eelnevate põhimõtete ja eesmärkide saavutamiseks seatakse kasutajaliidesele nõuded. Nõuded jagunevad funktsionaalseteks ja mittefunktsionaalseteks. Funktsionaalsed nõuded hõlmavad nõudeid riistvarale ja seadmetele ning defineerivad, mida süsteem peab tegema, näiteks kasutajaliidese puhul reageerima klahvivajutusele. Mittefunktsionaalsed nõuded defineerivad piirangud ja kohustused süsteemile ja selle arendusele ning jagunevad järgnevalt [9]:

1) Tehnoloogilised nõuded - missuguses tarkvarakeskkonnas arendatakse ja missugustega peab ühilduma ning millised on limiteerivad tehnoloogilised tegurid.

2) Andmete nõuded - hõlmavad vajalike andmete tüübi, suuruse, püsivuse, täpsuse ja andmehulkade väärtuse. Andmed peavad olema kuvatavad, sisestatavad, muudetavad, kaasajastatud ja täpsed.

3) Sotsiaalse keskkonna nõuded - defineerivad koostööd ja koordineerimist. Näiteks andmete jagamine ja kasutamine sünkroonselt mitme töötaja vahel ning avaliku või privaatse ligipääsu vajadust.

4) Organisatoorsed nõuded - kui efektiivne või stabiilne on kommunikatsioon, kasutajatoe kättesaadavus, koolitusvahendite olemasolu jne.

5) Kasutaja nõuded - tulenevad kavandatud kasutajarühma omadustest. Näiteks võimed ja oskused, arvuti kasutamise tase (algaja, ekspert, juhuslik või sagedane kasutaja). See informatsioon on kasutajaliidese kujundamiseks oluline, kuna algaja ilmselt vajab sammhaaval juhtnööre, dialooge ja piiratud interaktiivsust koos selge informatsiooniga. Eksperdile aga oleks vajalik paindlik suhtlemine süsteemiga ning rohkem volitusi funktsioonide juhtimiseks. Sage kasutaja eelistab vastupidiselt juhukülastajale ilmselt kiirklahvide kasutamist selle asemel, et pikki käske sisse trükkida või menüüsüsteemides navigeerida. Süsteemi omaduste kogumit „tüüpkasutaja” jaoks nimetatakse kasutajaprofiiliks ning mistahes seade võib omada mitut erinevat kasutajaprofiili.

6) Kasutatavuse nõuded - hõlmavad kasutatavuse eesmärgid ja nendega seotud meetmed konkreetse tarkvaratoote jaoks. Kasutatavuse nõuded on seotud kõigi teiste nõuetega, mis vajavad samuti defineerimist. Kasutaja nõuded ja nõuded kasutatavusele ei ole samatähenduslikud mõisted. Kasutatavus koosneb kuuest tegurist:

a) funktsionaalsus: süsteem saab hakkama kasutajale vajalike ülesannetega

b) õppimise lihtsus: kui lihtne on süsteemi kasutamise selgeks õppimine erinevatele kasutajagruppidele

c) kasulikkus: kui suure kasuteguriga on süsteem sagedase kasutaja jaoks

d) meeldejätmise lihtsus: kui lihtne on süsteemi kasutamise põhimõtteid meelde jätta juhukasutajal

e) subjektiivne rahulolu: kui rahul on kasutaja süsteemiga

f) arusaadavus: kui lihtne on aru saada, mida süsteem teeb

Kasutajasõbralikkuseks võib nimetada kombinatsiooni teguritest 2-6. Selleks, et hinnata kasutatavust, peavad need näitajad olema mõõdetavad. Kui on teada, kuidas erinevaid tegureid mõõta, on võimalik koostada nõuded, mis on oma olemuselt eesmärgid, mida soovitakse saavutada.

3.2 Erinevad kasutajaliidese tüübid

Tarkvaraarenduses on erinevaid tüüpe kasutajaliideseid. Järgnevalt selgitatakse enimkasutatavaid.

Käsurea kasutajaliides on tekstipõhine kasutajaliides, mida kasutatakse tarkvara või operatsioonisüsteemiga töötamiseks. Käsureaga suhtleb kasutaja ekraanile üksikuid käske sisestades, millele tuleb vastuseks samuti üks käsule vastav vastus. Käsurea kasutajaliides on vanem ja ei ole eriti kasutajasõbralik. Siiski kasutatakse ka tänapäeval käsurida päris palju erinevates süsteemides ning mõni programmeerimiskeel, näiteks Python, pakub kasutamiseks enda käsurida [20].

Graafiline kasutajaliides on rakendus, mis võimaldab kasutajal suhelda arvutiga kasutades sümboleid, visuaalseid metafoore ja osutusseadmeid. Tänapäeval on see enimlevinud kasutajaliidese tüübiks, kuna lihtsustab programmide kasutamist ning on meeldivam ja naturaalsem vaadata. Graafilise kasutajaliidese komponendid on kuvaril liikuv kursor, osutusseade (harilikult hiir), klaviatuur, ikoonid, aknad ja rippmenüüd. Tuntumad graafilise kasutajaliidesega operatsioonisüsteemid on Windows ja Macintosh [13].

Veebipõhine kasutajaliides kasutab Interneti vahendusel kuvatavaid veebilehti, mida kasutaja saab vaadata veebibrauserite abil. Üldjuhul on veebiliidesed avalikult kasutatavad. Uuemad veebirakendused kasutavad vahendeid nagu Java, AJAX, .NET raamistik, mis võimaldavad reaalaja kontrolli kasutajaliidese üle, mille tulemusel saab andmeid uuendada jooksvalt.

3.3 Kasutajaliidese automaattestimine

Kasutajaliidese testimine on protsess testimaks, kas rakendus töötab korrektselt. Kasutajaliidest võib testida manuaalselt või tarkvaraprogramme kasutades automaatselt.

Kasutajaliidese automaattestimine on manuaalselt tehtavate testiülesaanete automatiseerimine. Kuna manuaalne testimine on ajamahukas ja olenevalt testijast võib olla ka mitte efektiivne, siis automatiseeritakse kasutajaliidese teste täpsemate, tõhusamate ja usaldusväärsemate testitulemuste saavutamiseks. Pikema aja vältel on automaatne kasutajaliidese testimine kuluefektiivsem kui manuaalne. Siiski tuleks üldjuhul automaattestid implementeerida pärast ühe rakenduse osa funktsionaalsuse valmimist. Luues teste samal ajal funktsionaalsust arendades, võib osutuda liiga keerukaks ja tekitada rohkem probleeme. Seetõttu ei ole võimalik manuaalset testimist täielikult asendada. Manuaalselt tuleb testida ka erijuhud, sest neid ei ole efektiivne automatiseerida [21].

Automaatsed kasutajaliidese testid võivad joosta samal ajal mitme platvormi ja brauseri peal. Negatiivse poole pealt ei avasta nad juhtumeid, kui kasutajaliides on moonutatud, sest niipea, kui kõik liidese elemendid on leitud, siis test õnnestub. Testide automatiseerimine muutub tähtsamaks, kui mitu inimest töötavad sama koodiga. Ilma heade automaattestideta võivad keerulisemad kooditükid muutuda ühe inimese pärusmaaks.

Üks suurimaid kasutegureid automaatsete kasutajaliidese testide puhul on regressioonivigade avastamine, kuna nad on loodud avastama olukordi, kus uus lisatud kood lõhub või muudab olemasolevat funktsionaalsust. Kasutajaliidese automaattestidega ei avastata tihti bugisid, vaid regressioonivigu, seetõttu nimetatakse neid tihti just regressioontestideks.

3.4 Kasutajaliidese automaattestide kirjutamise head praktikad

Kasutajaliidese testid peavad olema kvaliteetsed, et nad oleks kasulikud ja pakuks rakenduse arendamisel lisaväärtust. Halvasti kirjutatud testid lisavad probleeme või ei kontrolli

vajalikku, mistõttu võivad and valesid tulemusi. Seetõttu tuleks hea testi kirjutamiseks järgida järgnevaid põhimõtteid:

1. Kasutada *Page Object* mudelit. Kasutajaliidese automaatteste kirjutades on vajalik viidata kasutajaliidese elementidele, millega kasutaja suhelda saab. *Page Object* modelleerib need objektideks, mida saab teste kirjutades kasutada. See vähendab dubleeritud koodi, tänu millele tuleb kasutajaliidese muutumise korral teste parandada ainult ühes kohas. Hea tava järgi defineeritakse kõik vajaminevad elemendid lehel ühes kohas ning teste kirjutades kasutatakse neid meetodeid elementidega suhtlemisel. Tänu sellele on testid loetavamad. Reeglina peaks *Page Object* mudeli järgi olema võimalik teha kõike tegevusi, mida kasutaja teha saab. Lisaks tuleks defineerida ainult need elemendid, mida teste kirjutades vaja läheb, et vähendada testide loomisele kulutatavat aega.

2. Test tuleb kirjutada kohe pärast arendust. Selliselt on vajalikud nõuded ja tegevused värskelt meeles ning vigade tekkimisel avastatakse need kiiremini. Mida hiljem testid kirjutatakse, seda rohkem aega selleks kulub, sest vajalike ülesannetega tuleb ennast uuesti kurssi viia. Mõnel juhul võib teste kirjutada ka enne arendust või arenduse käigus. Sellisel juhul peab olemas olema prototüüp, et oleks ülevaade kasutajaliidese elementidest ja ülesannetest. Selliselt saab automaatteste jooksutada kohe pärast arenduse valmimist. See siiski ei ole tavaline praktika, sest agiilses arenduses on muudatused sagedased ning võib tekkida suurem ajakulu, et töötav test valmis kirjutada.

3. Testandmed mõjutavad automaattestide stabiilsust. Parim lahendus oleks, kui testidel välised sõltuvused puuduvad, kuid alati ei ole see võimalik. Testandmeid saab testide käivitamiseks andmebaasi lisada või neid andmebaasist otsida. Selle tarbeks on kasulik, kui eksisteerib spetsiaalne andmebaasiversioon, mille peal automaattestid jooksevad. Siis on võimalik enne testimist andmebaas ja andmed taastada. Üks võimalus on ka testandmed testi sisse kirjutada, kuid see ei ole üldjuhul hea praktika, sest sellisel juhul testitakse ainult ühtede andmetega.

4. Testide kirjutamisel tuleb kasutada testiraamistiku funktsionaalsust. Kasutajaliidesel on tihti protsessid, mis tekitavad laadimisel või täitmisel ooteaega. Testis ei tohiks selle koha peal olla lahenduseks test mingiks ajaks ootama panna, vaid tuleks kasutada

vahendeid ootamaks, millal vajalikud elemendid laetud on. See tagab testide parema stabiilsuse, mis suurendab testide õnnestunud läbimist.

5. Test tuleb kirjutada lihtsalt ja arusaadavalt. Siin tuleb abiks varem mainitud *Page Object* mudel, kuid konkreetse testi kirjutamisel tuleb siiski arvestada heade koodikirjutamise praktikatega. Testi lugejal peab olema lihtne aru saada testi eesmärgist ja läbitavatest sammudest.

6. Test peaks kontrollima ühte funktsionaalsust. Kasutajaliidest testides tuleks väljade testimisel arvestada kahe erineva stsenaariumiga: korrektsed sisendid ja ebakorrektsed sisendid. Esimesel juhul tuleb kontrollida sisestamise õnnestumist ja teisel juhul valideerimisvigade kuvamist. Rohkemat korraga testides muutuvad testid liiga pikaks ja raskesti hallatavaks. Hästi kirjutatud test on lühike ja seetõttu on kergem vigast testitulemust analüüsida või testi muuta.

7. Testide kirjutamisel tuleb vältida duplikatsiooni. See, mis on juba ühiktestidega kaetud, ei tuleks teistkordselt kasutajaliidese testidega üle käia, sest kasutajaliidese testid on niigi ajamahukad.

3.5 Kasutajaliidese automaattestide haldamine

Kasutajaliidese automaattestide kirjutamine on ajamahukas tegevus. Lisaks sellele jooksevad nad erinevalt ühiktestidest pikka aega, isegi tunde, mis tuleneb sellest, et testid peavad reaalselt töötavat rakendust kontrollima. Seetõttu käivitatakse kasutajaliidese testid reeglina korra päevas või parema variandina konfigureeritakse käivituma öösel. Selle tulemusel ei avastata ja analüüsita tekkinud vigu koheselt ja arendaja ei pruugi nendest teadlik olla. Tavaliselt kontrollib testide eest vastutav isik korra päevas automaattestide tulemusi ja nende põhjal kas parandab vigased testid või esitab vearaporti. Tavaliselt kontrollib testija ebaõnnestunud testid uuesti üle, et veenduda korrektsetes tulemustes. Kui probleem tõesti eksisteerib, siis lisatakse see vearaportisse. Kuna reeglina kontrollitakse kasutajaliidese testide tulemusi korra päevas, siis nende haldamine ei ole väga ajamahukas tegevus, kui testid on hästi koostatud. Seetõttu on testijal aega tegeleda teiste tähtsate ülesannetega olles samal ajal kindel, et rakendus töötab ja regressioonivigu ei ole tekkinud. Pidev tähelepanu on siiski vajalik, sest uue funktsionaalsuse lisamisel tuleks kirjutada sellele automaattest.

4. Kasutajaliidese automaattestide kirjutamine

Järgnevas peatükis kirjeldatakse bakalaureusetöö raames loodud kasutajaliidese automaattestimise süsteemi. Tuuakse välja testide kirjutamiseks valitud vahendid. Seejärel kirjeldatakse testide paiknemist rakenduse koodis ning üksikute kasutajaliidese testide kirjutamist ja testide käivitamist ning tulemusi.

4.1 Valitud vahendid

Kasutajaliidese testide kirjutamiseks on valitud *Selenide*[18] raamistik, mis töötab *Selenium WebDriver* raamistikul ning mis on spetsiaalselt loodud kasutajliidese testimiseks. *Selenide* on vabatarkvara, millega saab kontrollida veebibrauseri tegevusi. *Selenide* on lihtne kasutada, sest kasutamiseks on vajalik vaid projekti lisada vastav *selenide.jar* fail.

Selenide abil on kergelt võimalik kasutada *Page Object* mudelit kasutades selektoreid, millega saab valida veebilehe elemente. Veebilehe elementideks on erinevad tekstiväljad, tabelid, nupud, lingid, listid, raadionupud jne. Valitud elementidega saab teha erinevaid toiminguid nagu kontrollida olekut, oodata mõne tingimuse täitumist. Selektoreid ühendades on elemente võimalik valida ka teiste elementide seest, näiteks erinevaid ridu tabelitest. Elemente on võimalik valida erinevate atribuutide alusel näiteks teksti, tiitli või väärtuse järgi.

Selenide kasutab *Condition* klassi, millega saab teste ootama panna, et mingid tingimused lehel oleks täidetud. See tuleb eriti kasuks, kui on vajalik testida dünaamiliselt muutuvaid veebilehti. Tingimusi kasutatakse ka kontrollimaks, kas veebilehel on korrektsed elemendid ja et need elemendid oleksid vajalikus staatustes. Tingimusteks võivad olla mõne elemendi nähtavus, kasutamise võimalus, tekstiväärtuse sobivus jne. Kuna C*ondition* klass on abstraktne, siis kasutaja saab ise uusi tingimusi luua kui selleks vajadus tekib.

Selenide-ga saab lihtsalt kasutada HTML konstrukte, mis koosnevad mitmest elemendist nagu tabelid, listid, raadionuppude kogumid jt. Selleks kasutatakse *ElementsCollection*-it, mille abil saab teostada käsklusi kõigile sama klassi kuuluvatele elementidele korraga.

Testide ebaõnnestumise korral saab *Selenide* abil teha brauserist ekraanitõmmise, mis salvestatakse koos HTML sisuga soovitud asukohta, millele tänu on testide ebaõnnestumise põhjuste leidmine mugavam.

Selenide on kasutatav enamike populaarsemate brauseritega, mis muudab testimise paindlikumaks ja annab arendajale rohkem võimalusi. Samuti on *Selenide* testid sobilikud jooksma pideva integratsiooni serveritel, tänu millele saab neid konfigureerida automaatse käivituse jaoks kindlal testserveril.

Selenide teste käivitatakse sarnaselt ühiktestidega. Töös kirjutatud testid käivitatakse JUnit raamistiku abil.

4.2 Testide kirjutamise ülevaade

Käesoleva bakalaureusetöö käigus on loodud kasutajaliidese testid veebirakendusele. Veebirakenduse kasutajaliideseks on veebiliides, mille kaudu kasutaja rakendusega suhtleb, lisab isiku- ja tooteandmeid läbi veebilehtedel olevate vormide. Kasutajaliidese testid kirjutatakse paralleelselt koos arendatud veebilehtedega. See tagab kiirema tagasiside tehtud töö kvaliteedi osas. Kasutajaliidese testide kirjutamisel pannakse põhirõhk rakenduse enda testimisele, sest seotud süsteemide testimiseks ei ole kasutajaliidese regressioontestid otstarbekad. Automatiseerides teste samaaegselt rakenduse arendusega, annab võimaluse luua testlood juba automaattesti kujul. Sedasi ei ole vaja kirjutada eraldi testlugusid mujale dokumentidesse ning vajaliku leiab juba koodist. See eeldab aga, et testid on kirjutatud kergesti loetavalt.

Kasutajaliidese testid asuvad rakenduse koodiga samas repositooriumis eraldi failides. Tänu sellele on arendajal hea ülevaade kogu süsteemist ja ressursse kulub vähem. Testifailid jagunevad *Page Object* mudeli järgi kaheks. Objektifailides (Joonis 1, lk 27) defineeritakse ja leitakse veebilehel olevad objektid, mida testide kirjutamiseks vaja läheb. Testifailides (Joonis 2, lk 28, Joonis 3, lk 30) kirjutatakse objekte kasutades konkreetsed testid, mida rakenduse testimiseks jooksutatakse. Iga testimist vajava veebilehe jaoks luuakse eraldi objektifail ja testifail. Ühte testifaili kirjutatakse kõik testid, mida failiga seotud veebilehel testida soovitakse, mistõttu võib olla erineva keerukuse ja mahuga veebilehtede testifailidel erinev arv teste.

```java
package ui;

//Kaasatud failid, kus asuvad või on viidatud kasutatavad meetodid
import static com.codeborne.selenide.Selenide.*;
import ui.UiTester;
import org.openqa.selenium.By;

//Objektifailis defineeritakse kõik vajaminevad tegevused, mida testi läbimiseks tehakse.
public class ApplicationForm {

    private UiTester uiTester = UiTester.get();

    public ApplicationForm openForm() {
        open(uiTester.getProjectBaseUrl() + "/applications/new");
        return this;
    }

    public ApplicationForm setPersonalData(String firstName, String lastName) {
        $("#firstName").setValue(firstName);
        $("#lastName").setValue(lastName);
        return this;
    }

    public ApplicationForm setFinancialData(String incomeAmount, String liabilities) {
        $("input#incomeAmount").setValue(incomeAmount);
        $("input#liabilityPayment").setValue(liabilities);
        return this;
    }

    public ApplicationForm setContactData(String email, String phone) {
        $("#applicationEmail").setValue(email);
        $("#applicationPhone").setValue(phone);
        return this;
    }



    public ApplicationForm submitApplication() {
        $(By.id("submit")).click();
        return this;
    }

    public String getApplicationStatus() {
        return $(By.name("status")).getText();
    }

    public String getApplicationCreditDecision() {
        return $(By.name("creditDecision")).getText();
    }

    public String getRejectionReason() {
        return $(By.name("rejectReason")).getText();
    }
}
```

Joonis 1. Objektifaili näidis. Allikas: näidisrakendus.

```java
package ui;

//Kaasatud failid, kus asuvad või on viidatud kasutatavad meetodid
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.Selenide.*;
import ui.SignInActions;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;

public class ApplicationTest {

    //Sisselogimine kindla kasutajaga. Rakendust saab kasutada ainult sisselogitud kasutaja.
    @Before
    public void signInBeforeTestsRun() {
        new SignInActions().signInAsDefaultUser();
    }

    //Test kontrollib taotluse sisestamist sobivate andmetega. Sobivate andmete korral peab
    //taotlusele määratama sobiv staatus.
    @Test
    public void testApprovedApplication() {
        ApplicationForm form = new ApplicationForm();
        form.openForm();
        form.setPersonalData("Test", "Taotlus");
        form.setFinancialData("1200", "0");
        form.setContactData("test@email.com", "55566677");
        form.submitApplication();
        form.getApplicationStatus().shouldHave(exactTextCaseSensitive("Taotlus sisestatud"));
        form.getApplicationCreditDecision().shouldHave(exactTextCaseSensitive("Heaks kiidetud"));
    }



    //Test kontrollib taotluse sisestamist ebasobivate andmetega. Taotlusele tuleb tagasi lükata
    //ja selgitada otsuse põhjust.
    @Test
    public void testRejectedApplication() {
        ApplicationForm form = new ApplicationForm();
        form.openForm();
        form.setPersonalData("Ebaõnnestunud", "Taotlus");
        form.setFinancialData("200", "800");
        form.setContactData("test@email.com", "55566678");
        form.submitApplication();
        form.getApplicationStatus().shouldHave(exactTextCaseSensitive("Taotlus sisestatud"));
        form.getApplicationCreditDecision().shouldHave(exactTextCaseSensitive("Tagasi lükatud"));
        form.getRejectionReason().shouldHave(exactTextCaseSensitive("Kohustus suurem kui sissetulek"));
    }
}
```

Joonis 2. Testfaili näidis 1. Allikas: näidisrakendus.

```java
package ui;

//Kaasatud failid, kus asuvad või on viidatud kasutatavad meetodid
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.WebDriverRunner.*;
import ui.SignInActions;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class ApplicationDetailsTest {

    private static String applId;

    //Enne testide käivitamist luuakse uus taotlus ja leitakse selle ID,
    //mille abil teste jooksutatakse
    @BeforeClass
    public static void preconditionsBeforeTestsRun() {
        new SignInActions().signInAsDefaultUser();
        ApplicationForm newApplicationform = new ApplicationForm();
        newApplicationform.openForm();
        newApplicationform.setPersonalData("UI", "Test");
        newApplicationform.setFinancialData("700", "50");
        newApplicationform.setContactData("testmail@mail.ee", "56565656");
        newApplicationform.submitApplication();
        Matcher matcher = Pattern.compile("\\d+").matcher(url());
        matcher.find();
        applId = String.valueOf(matcher.group());
    }

    @Before
    public void signInBeforeTestRun() {
        new SignInActions().signInAsDefaultUser();
    }


    //Testitakse taotluse positiivse otsuse saamist ja otsuse kinnitamist.
    //Positiivse otsuse saamiseks lisatakse sobivad andmed.
    @Test
    public void testConfirmPositiveDecision() {
        ApplicationDetailsForm detailsForm = new ApplicationDetailsForm();
        detailsForm.openForm(applId);
        detailsForm.setApplicantNameAndCitizenships("UI", "Test", "ESTONIA");
        detailsForm.setApplicantContactData("test@email.com", "56756756", "ESTONIA", "testitänav", "testilinn");
        detailsForm.setIncome("1000");
        detailsForm.setLiabilities("100", "Kõik kohustused kokku");
        detailsForm.setCreditDecisionData("testikommentaar", "200", "25", "Heaks kiidetud");
        //Taotluse esitamine
        detailsForm.presentForDecision();
        detailsForm.getCancelDecisionButton().should(appear);
        detailsForm.getDiscardApplicationButton().should(disappear);
        detailsForm.getConfirmDecisionButton().should(disappear);
        detailsForm.getPresentForDecisionButton().should(disappear);
        detailsForm.getPutApplicationOnHoldButton().should(disappear);
        detailsForm.getFormTitle().shouldHave(exactTextCaseSensitive("Esitatud kinnitamiseks taotlus " + applId));
        //Positiivse otsuse kinnitamine
        detailsForm.confirmDecision();
        detailsForm.getApplicationStatusFromHeader().shouldHave(exactTextCaseSensitive("Otsustatud"));
        detailsForm.getApplicationDecisionStatus().shouldHave(exactTextCaseSensitive("Heaks kiidetud"));
        detailsForm.getCancelDecisionButton().should(appear);
    }

    //Taotluse otsimine staatuse abil.
    private String searchApplicationByStatus(String status) {
        ApplicationsListForm listForm = new ApplicationsListForm();
        listForm.openForm();
        listForm.setApplicationStatus(status);
        listForm.searchApplication();
        String applId = listForm.getApplIdFromList(1).getText();
        return applId;
    }
```

```
    //Testitakse taotlust, mis saab negatiivse otsuse. Negatiivse otsuse saamiseks
    //lisatakse sobivad andmed.
    @Test
    public void testNegativeApplicationDecision() {
        String applId = searchApplicationByStatus("Uus");
        ApplicationDetailsForm detailsForm = new ApplicationDetailsForm();
        detailsForm.openForm(applId);
        detailsForm.setIncome("100");
        detailsForm.setLiabilities("1000", "Kõik kohustused kokku");
        detailsForm.setCreditDecisionData("test", "300", "2500", "Tagasi lükatud");
        detailsForm.presentForDecision();
        detailsForm.getFormMessage().shouldHave(exactTextCaseSensitive("Tagasilükatavat taotlust ei ole vaja kinnitamiseks esitada."));
        detailsForm.decide();
        detailsForm.getPutApplicationOnHoldButton().should(disappear);
        detailsForm.getDecideButton().should(disappear);
        detailsForm.getPresentForDecisionButton().should(disappear);
        detailsForm.getDiscardApplicationButton().should(disappear);
        detailsForm.getApplicationStatusFromHeader().shouldHave(exactTextCaseSensitive("Otsustatud"));
        detailsForm.getApplicationDecisionStatus().shouldHave(exactTextCaseSensitive("Tagasi lükatud"));
    }
}
```

Joonis 3. Testifaili näidis 2. Allikas: näidisrakendus.

Testid on enamasti kirjutatud järgimaks ärireegleid ja neist tulenevaid kasutuslugusid. Enamik toimingutest, mida kasutaja teha saab, on mõistlik kasutajaliidese testidega katta. Siiski üldjuhul ei kirjutata teste kogu kasutusloo raames, vaid eraldi osadena iga veebilehe jaoks, sest vastasel juhul muutuksid testid liiga pikaks ja raskesti hallatavaks ning vigade tekkimine muutuks sagedasemaks ja vigade põhjuste väljaselgitamine keerulisemaks.

Testid lisatakse ühte testtsüklisse (Joonis 4, lk 31) ning käivitatakse korraga. Tsükli lõppemisel saadakse raport, mis sisaldab nii õnnestunud kui ebaõnnestunud testide tulemusi koos ekraanitõmmistega, mille abil arendaja teeb järeldusi rakenduse töötamisest või parandab vigased testid.

```java
package ui;

import ui.SignInTest;
import ui.ApplicationDetailsTest;
import ui.ApplicationsListTest;
import ui.AddContractTest;
import ui.ApplicationCommentTest;
import ui.ApplicationContractListTest;
import ui.ApplicationTest;
import ui.ContractDetailsTest;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

//Testtsüklis jooksutatavad testifailid
@RunWith(Suite.class)
@SuiteClasses({
        SignInTest.class,
        ApplicationTest.class,
        ApplicationContractListTest.class,
        AddContractTest.class,
        ApplicationCommentTest.class,
        ContractDetailsTest.class,
        ApplicationsListTest.class,
        ApplicationDetailsTest.class,
})
public class UiTestsSuite {

}
```

Joonis 4. Testtsüklis jooksutatavad testid. Allikas: näidisrakendus.

Kokkuvõte

Veebirakenduste puhul on kasutajaliidese testimine väga oluline, sest läbi selle toimub rakendusega suhtlemine. Et tagada pidevalt areneva rakenduse jätkusuutlik kvaliteet kogu elutsükli vältel, on vaja kasutajaliidese regressioontestid automatiseerida. Agiilselt arendades muutub rakendus tihti ning automaattestide olemasolul avastatakse tekkinud regressioonvead kiiresti ning arendajad saavad tööst pideva tagasiside. Bakalaureusetöös on kirjeldatud meetodid, kuidas luua toimiv süsteem kasutajaliidese automaatseks testimiseks, et tagada rakenduse kvaliteet arenduse algusest kuni lõpuni.

Kasutajaliidese automaatne testimine on pidev protsess, sest iga uue arendusega võib tekkida regressioonvigu. Sellepärast on vajalik, et testimine oleks korraldatud läbimõeldult ja efektiivselt. Tähelepanu tuleb pöörata kõigile tegevustele nagu vahendite valimine, korrektsete ja vajalike testide kirjutamine ja haldamine ning testide pidev käivitamine ja testraportite uurimine. Korraliku testsüsteemi loomine annab tulemuseks hästi toimiva rakenduse, milles tekkivad vead avastatakse ja likvideeritakse kiiresti.

Tähtsaim järeldus tööst on, et kasutajaliidese testide kirjutamisel tuleks lähtuda ärinõuetest ja kasutajalugudest. Testida tuleks reaalseid situatsioone, mis rakenduse kasutamisel esinevad. Tehnilisi nõudeid ja sõltuvusi teistest rakendustest pole kasutajaliidese testidega otstarbekas kontrollida, sest need peaksid olema kaetud juba varasemalt teist tüüpi testidega nagu näiteks ühiktestidega.

Teiseks saab välja tuua, et individuaalsed testid peaksid olema kirjutatud võimalikult lühidalt ja minimaalsete ressurssidega, sest kasutajaliidese testid võivad joosta sõltuvalt rakendusest mitu tundi. Liiga pikad ja keerulised testid lisavad testtsükli jooksutamisele palju aega ja ebaõnnestuvad tihemini, mis tähendab, et testide parandamiseks kulub rohkem aega ja osa rakenduse funktsionaalsusest võib jääda tihti kontrollimata.

Kolmandaks tasub välja tuua, et vajalik oleks kasutada sobivaid vahendeid kasutajaliidese testide kirjutamiseks. Parimal juhul on testid kirjutatud samas keeles, mis rakendus, et testid saaksid olla koos ülejäänud koodiga ühes repositooriumis ja arendajad ei peaks juurde õppima teist programmeerimiskeelt või pidevalt vahetama kahe erineva vahel. Suurema mahu ja

keerukusega süsteemide korral oleks ka mõttekas kasutada spetsiaalselt kasutajaliidese testimiseks loodud tarkvara, kuid need on enamjaolt tasulise litsentsiga.

Tagamaks mugavam testtsükli käivitamise ja haldamise protsess, tuleks edasi arenduse korral kasutusele võtta pideva integratsiooni ja virtuaalmasina süsteemid. Nende abil saab testtsükleid lihtsamalt käivitada ja konfigureerida kasutajaliidese automaattestide tarbeks eraldi masinaid, et testid saaksid joosta stabiilsel ja spetsiaalselt testidele mõeldud keskkonnas.

Kasutatud kirjandus

[1] Agile Manifesto, „Manifesto for Agile Software Development,“ 2001. [Võrgumaterjal]. http://www.agilemanifesto.org/. [16 Mai 2016].

[2] Ambler S., „Agile Testing and Quality Strategies: Discipline Over Rhetoric,“ [Võrgumaterjal]. http://www.ambysoft.com/essays/agileTesting.html. [16 Mai 2016].

[3] Ambler S., „Introduction to Test Driven Development (TDD),“ [Võrgumaterjal]. http://www.agiledata.org/essays/tdd.html. [16 Mai 2016].

[4] Black R., „How Agile Methodologies Challenge Testing,“ September 2009. [Võrgumaterjal]. http://www.istqb.org/images/Articles/black_How%20Agile%20Methodologies.pdf. [16 Mai 2016].

[5] Buehring S., „The Fundamentals of Software Testing,“ [Võrgumaterjal]. http://www.articlecity.com/articles/computers_and_internet/article_4752.shtml. [16 Mai 2016].

[6] Collier K. W., Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing, Addison-Wesley Professional, 2011.

[7] Hughey D., „The Traditional Waterfall Approach,“ 2009. [Võrgumaterjal]. http://www.umsl.edu/~hugheyd/is6840/waterfall.html. [16 Mai 2016].

[8] Huston T., „What Is Agile Testing,“ [Võrgumaterjal]. https://smartbear.com/learn/software-testing/what-is-agile-testing/. [16 Mai 2016].

[9] Jakimov T., „Kasutajaliidesed ja Neile Esitatavad Nõuded,“ 11 Detsember 2006. [Võrgumaterjal]. http://www.tud.ttu.ee/im/Vladimir.Viies/materials/OS-systeemid/os-referaadid/OS_referaat.pdf. [16 Mai 2016].

[10] Jeffries R., „What is Extreme Programming?,“ 16 Märts 2011. [Võrgumaterjal]. http://ronjeffries.com/xprog/what-is-extreme-programming/. [16 Mai 2016].

[11] Kiremire A. R., „The application of the pareto principle in software engineering,“ 19 Oktoober 2011. [Võrgumaterjal]. http://www2.latech.edu/~box/ase/papers2011/Ankunda_termpaper.PDF. [16 Mai 2016].

[12] Lauk M., „Kasutajaliides ja selle erinevad tüübid (tekstiline, graafiline, heliline),“ [Võrgumaterjal]. http://www.e-uni.ee/e-kursused/eucip/arendus/413_kasutajaliides_ja_selle_erinevad_tbid_tekstiline_graafiline_heliline.html. [16 Mai 2016].

[13] Levy S., „Graphical user interface (GUI),“ 18 Detsember 2014. [Võrgumaterjal]. http://www.britannica.com/technology/graphical-user-interface. [16 Mai 2016].

[14] Radigan D., „Engineering higher quality through agile testing practices,“ [Võrgumaterjal]. https://www.atlassian.com/agile/testing. [16 Mai 2016].

[15] Radigan D., „Running agile programs (without losing your mind),“ [Võrgumaterjal]. https://www.atlassian.com/agile/program. [16 Mai 2016].

[16] Rouse M., „Agile software development (ASD),“ Veebruar 2007. [Võrgumaterjal]. http://searchsoftwarequality.techtarget.com/definition/agile-software-development. [16 Mai 2016].

[17] Rouse M., „Release,“ Märts 2008. [Võrgumaterjal]. http://searchsoftwarequality.techtarget.com/definition/release. [16 Mai 2016].

[18] Selenide, [Võrgumaterjal]. http://selenide.org/ [16 Mai 2016].

]
[19] Sutherland J., „The Scrum Guide,“ Juuli 2013. [Võrgumaterjal]. http://www.scrumguides.org/scrum-guide.html. [16 Mai 2016].
[20] Techopedia, „Command Line Interface (CLI),“ [Võrgumaterjal]. https://www.techopedia.com/definition/3337/command-line-interface-cli. [16 Mai 2016].
[21] Urbonas R., „Automated User Interface Testing,“ 11 Aprill 2013. [Võrgumaterjal]. https://www.devbridge.com/articles/automated-user-interface-testing/. [16 Mai 2016].
[22] Vanem P., „Kasutajaliides ja kasutajamugavus,“ Oktoober 2012. [Võrgumaterjal]. http://maurus.ttu.ee/sts/wp-content/uploads/2012/11/Ettekanne_2012-10-29.pdf. [16 Mai 2016].
[23] Versionone, „What is Kanban? Kanban Software Tools,“ [Võrgumaterjal]. https://www.versionone.com/what-is-kanban/. [16 Mai 2016].
[24] Waters K., „What Is Agile? (10 Key Principles of Agile),“ 10 Veebruar 2007. [Võrgumaterjal]. http://www.allaboutagile.com/what-is-agile-10-key-principles/. [16 Mai 2016].