TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

IT40LT
Elina Volkova 164069IABB

TARKVARA TESTIMISE KÄIGUS LEITUD VIGADE ANALÜÜS ETTEVÕTTE ANDEVIS AS NÄITEL

Bakalaureusetöö

Juhendaja: Jekaterina Tšukrejeva
Magistrikraad

Tallinn 2019

Autorideklaratsioon

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

Autor: Elina Volkova

20.05.2019

Annotatsioon

Töö eesmärgiks on analüüsida tarkvaraarenduse ettevõtte Andevis AS testimise protsessi, leida peamised probleemid ning pakkuda meetmeid probleemide lahendamiseks ja vigade analüüsimiseks. Töös viiakse läbi ühe projekti näitel vigade analüüsi ja sõltuvalt vea algpõhjusest pakutakse vastava lahenduse.

Töös leitakse, et testimise protsess Microsofti meeskonnas ei ole hästi struktureeritud, mille tõttu leitud vigade arv projekti kestvuse jooksul ainult suureneb, mitte väheneb.

Töö tulemusena pakutud meetmeid on võimalik kasutada reaalsetes projektides, et vähendada tekkivaid vigu ja tõsta Virosofti rakenduse kvaliteedi lõppkasutaja jaoks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 47 leheküljel, 4 peatükki, 9 joonist, 2 tabelit.

Abstract

Analysis of the errors found during software testing on the example of the Andevis AS company

The aim of the research is to analyze the testing process of the software development company Andevis AS, to find the main problems and to propose solutions to solve those problems and analyze any errors found during software testing. In the research, an error analysis of one project is carried out and a corresponding solution is offered based on the root cause of the found defect.

In this research, it was found that the testing process used by the Microsoft team is not well structured, which only increases the quantity of errors found during the project's lifespan.

The thesis consists of four parts. The first part of the research work introduces the different levels of testing, testing types and methods that are used in the Andevis AS company. In the second part, the analysis of defects is explained, and the analysis of the errors found during software testing in one Virosoft project is carried out with the possible solutions offered. The third part of the work describes the testing process that is currently used in the company, highlighting the problems therein. The fourth part of the research presents amendments to the Andevis AS company and describes how the proposed measures can be implemented.

The actions that are offered as a result of this work can be used in real projects to reduce the quantity of errors and increase the quality of the Virosoft application for the end user.

The thesis is in Estonian and contains 47 pages of text, 4 chapters, 9 figures, 2 tables.

Lühendite ja mõistete sõnastik

Andevis AS	Tarkvaraarenduse ettevõtte
Back-end	Arvutisüsteemi või rakenduse osa, mis ei ole otseselt kasutajale kättesaadav. Tavaliselt vastutab andmete salvestamise ja manipuleerimise eest
Bizagi Modeler	Äriprotsesside modelleerimiseks kasutatav tarkvara
Desktop	*Desktop* rakendus on rakendus, mis töötab laua- või sülearvutis eraldi (ei ole vaja kasutada näiteks veebibrauserit)
Front-end	Arvutisüsteemi või rakenduse osa, millega kasutaja vahetult suhtleb
GUI	*Graphical User Interface*. Arvuti graafikakuvamise võimalusi kasutav tarkvaraliides
JMeter	Jmeter on *Apache* projekt, mida saab kasutada koormustestimise vahendina teenuste analüüsimisel ja mõõtmisel, keskendudes veebirakendustele
JIRA	*Jira* on *Atlassiani* poolt välja töötatud kaubanduslik probleemide jälgimise süsteem, mis võimaldab vigade jälgimist ja agiilset projektijuhtimist
Live	Kliendi tootmiskeskkond, kuhu paigaldatakse viimaseid muudatusi
Postman	*Postman* on tööriist, mida kasutatakse päringute saatmiseks ja vastuste vastuvõtmiseks *REST API* kaudu
PractiTest	*PractiTest* on testijuhtimise tarkvara, mis võimaldab jälgida vigu, nõudeid ja saada aruandeid
qTest	*qTest* on organisatsioonide testijuhtimise tarkvara, mis aitab luua tsentraliseeritud testijuhtimissüsteemi, et lihtsustada suhtlemist ja ülesande kiire andmise kvaliteedi tagamise meeskondadele ja arendajatele
Redmine	*Redmine* on tasuta ja avatud lähtekoodiga veebipõhine projektijuhtimise ja probleemide jälgimise vahend, mis võimaldab kasutajal hallata mitu projeti ja nendega seotud allprojekte
Selenium	*Selenium* on tarkvara testimisraamistik veebirakenduste testimiseks
Source code	Lähtekood, mis on kirjutatud inimloetaval

	programmeerimiskeeles, võib olla koos kommentaaridega
TestComplete	*TestComplete* on funktsionaalne automatiseeritud testimiseplatvorm, mis annab testijatele võimaluse luua Microsoft Windows, Web, Android ja iOS rakenduste automatiseeritud testid
TFS	*Team Foundation Server*. Töö organiseerimise keskkond (veade ja ülesannete püstitamine)
Virosoft	Palgaarvestuse tarkvara

Sisukord

1 Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Eesmärk 11
1.3 Metoodika 11
1.4 Ülevaade tööst 12
2 Ülevaade tarkvara testimisest 13
2.1 Testimise tasemed 13
2.2 Testimise tüübid 14
2.3 Testimise läbiviimise meetodid 15
3 Tarkvara testimise käigus leitud vigade ja defektide analüüs 17
3.1 Defektide analüüs 17
3.2 Koodivigade klassifikatsioon 21
3.3 Näitajad vigade ja defektide analüüsimiseks 24
4 Tarkvara testimise protsess Andevis AS ettevõttes 30
4.1 Kasutusel olev testimismeetod ja selle nõrgad küljed 30
5 Muudatusettepanekud Andevis AS ettevõttele 38
5.1 Testimise protsessi muutmine 38
5.2 Leitud vigade analüüsi juurutamine 46
6 Kokkuvõte 50
Kirjanduse loetelu 52

Jooniste loetelu

Joonis 1. TFSi keskkonnas püstitatud ülesanne. 19
Joonis 2. Defect Pareto Chart näidis [14]. 24
Joonis 3. Ühe kuu jooksul leitud defektide arv Virosofti projektis. 25
Joonis 4. Aruande parameetrite aken. 33
Joonis 5. Parandatud aruande vorm. 34
Joonis 6. Kasutusel olev testimise protsess. 39
Joonis 7. Testjuhtumite näidis Excelis. 41
Joonis 8. Parandatud testimise protsessi mudel. 44
Joonis 9. Vigade analüüsimise protsess. 48

Tabelite loetelu

Tabel 1. Parandatud raporteerimise viis. 20
Tabel 2. Tugitundide aruanne ühe kuu eest. 35

1 Sissejuhatus

Andevis AS on palgaarvestuse tarkvara ettevõtte, mis pakub oma klientidele Virosofti rakendust. Microsofti meeskond teenindab peaaegu 20 kliente, kes soovivad probleemivabalt maksta töötajatele palka, arvestada tööaega ning sisestada programmi uusi töötajaid. Ettevõtte missiooniks on luua tööriistad klientide äriprotsesside parendamiseks ja efektiivsuse tõstmiseks. Peaaegu poole klientide Virosofti juurutamisest on möödunud juba üle kolme aastat, kuid iga päev saavad Microsofti meeskonda klienditeeninduse spetsialistid tagasisidet vale või mitte töötava funktsionaalsuse kohta. Suurem osa kliendi poolt leitud vigadest esineb puuduva testimise pärast. Seega käesoleva bakalaureusetöö raames keskendutakse Eesti palgaarvestuse firma Andevis AS testimise protsessi parandamisele, vigade analüüsile ja pakutakse meetmeid, mida juurutatakse Microsoft meeskonda. Meetmete rakendamisel peab parenema testimise protsess, leitud vigade analüüsimine peab aitama säästa aega ja raha pikemaajalises perspektiivis, mis on oluline kvaliteetse tarkvara pakkumisel ning heade kliendisuhete hoidmiseks.

1.1 Taust ja probleem

Ettevõtted testivad oma toodet põhjalikult, kuid programmeerija ei saa alati ette näha kõike situatsioone, kus viga võib esile tulla. Leitud viga ja selle parandamise maksumus kasvab aja möödudes, kuna üks viga tehtud, näiteks nõuete faasis, võib üle kanduda disaini ja ehitamise faasi. Samuti peab testija kontrollima kõike, mis oli tehtud vea avastamiseni, et tagada tarkvara usaldusväärse töö. Kui viga kandub live versioonisse, mida kasutab lõppkasutaja ehk ettevõtte puhul nende klient, siis vea leidmise ja parandamise maksumus on suurem, kui selle avastaks ettevõtte testija. Kliendi poolt leitud defekt, mis analüüsi käigus koostatud nõuete järgi ei peaks esinema, parandab ettevõtte tasuta, kuna tellija ei ole nõus maksma selle eest, mis pidi olema tehtud esialgselt ilma vigadeta. Seega ettevõtte teeb parandusi oma raha eest, mis ei mõju hästi kasumile. Mõned vead, mis jäid avastamata testimise käigus, võivad viia õnnetuseni,

kus saavad kannatada inimesed, näiteks 8 patsienti hukkusid, kuna programm, mis pidi kalkuleerida täpset kiirguse annust, ei saanud seda korralikult arvutada [1].

Hetkel Andevis AS ettevõttes autori silmis ei ole testimise protsess organiseeritud nii hästi, kui võiks. Teatud praktikate ja standardite mitte kasutamine viib suurema arvu vigade avastamiseni. Iga arendustsükli käigus testija kontrollib manuaalselt rakenduse funktsionaalsuse ja nõuetele vastavuse mitu korda. Kuna üks arendus võib kesta kuni kuue kuuni, siis suurem osa testitud funktsionaalsusest ja tehtud töödest ununeb ning on raske meenutada, milliste sammude tagajärjel tekkis viga. Meeskonnas puudub ühine testimise protsessi kirjeldus, mis aeglustab kogu arendustsükli ja ei võimalda mäletada ja säilitada testide tulemusi. Selle probleemi lahendamiseks on vaja muuta testimise protsessi läbipaistvamaks ja arusaadavamaks ning läbi viia leitud vigade põhjal analüüsi, mis aitab mäletada kõike saadud tulemusi ja muudab testimise sügavamaks.

1.2 Eesmärk

Käesoleva töö eesmärgiks on analüüsida testimise protsessi ja pakkuda võimalikke meetmeid erinevate testimisega seotud probleemide lahendamiseks ning vigade analüüsimiseks. Antud töö ja selle tulem on ettevõtte jaoks oluline, et tagada tarkvaraarenduse kvaliteedi pideva arengu. Töö tulemusena tekib dokument, mida saab rakendada Microsofti meeskonna Virosofti projektides, et vähendada tekkivate vigade arvu ja saada aru defektide algpõhjusest. Meetmete rakendamise tulemusena on testimise protsessi lihtsustamine, läbipaistvamaks muutmine, dokumenteerimine ja projektide kvaliteedi tõstmine.

1.3 Metoodika

Metoodikaks on kvalitatiivne analüüs, mis põhineb erinevatel teadusartiklitel, artiklitel ja raamatutel, mida uuriti eesmärgiga leida parimad võimalused testimise probleemide lahendamiseks. Töö autor on töötanud üle poole aasta ettevõttes testijana ning võttis arvesse lahenduste leidmisel igapäevase töökogemuse.

1.4 Ülevaade tööst

Antud töö koosneb neljast osast. Bakalaureusetöös sõnastatakse testimise probleeme, mis esinevad Microsofti meeskonnal.

Töö esimeses osas tutvustatakse erinevaid testimise tasemed, tüübid ja meetodid, mida kasutatakse Andevis AS ettevõttes.

Teises osas tutvustatakse vigade analüüsi ja viiakse läbi ühe projekti raames leitud vigade analüüsi koos võimalike lahenduste pakkumisega.

Töö kolmas osa kirjeldab hetkel ettevõttes kasutusel oleva testimise protsessi koos probleemide väljatoomisega.

Neljandas peatükis esitatakse muudatusettepanekuid ettevõttele ja kirjeldatakse, kuidas on võimalik rakendada pakutuid meetmeid Microsofti meeskonnas.

2 Ülevaade tarkvara testimisest

Iga ettevõtte jaoks, mis töötab IT-valdkonnas on oluline pakkuda oma klientidele usaldusväärset toodet või teenust, kus oleks võimalikult vähe vigu ja parandusi vajavaid defekte. Seega tarkvara testijad peavad tagama tarkvara kvaliteedi ja programmi oskust igas olukorras käituma niimoodi, et see ei rikuks midagi kogu programmis.

Glenford Myersi järgi testimine on programmi käivitamise protsess vea tuvastamise eesmärgiga [2]. Testimise käigus kindlasti ei saa leida kõike vigu, mis võivad esile tulla programmi lõppkasutajal, aga suurema osa situatsioonidest saab analüüsida ja ennustada, millised tegevused võivad ajada programmi katki.

Selles peatükis tutvustatakse testimise teooriat, sealhulgas kirjeldatakse testimise tasemeid, tüüpe ja meetodeid.

2.1 Testimise tasemed

Selles peatükis tutvustatakse lähemalt nelja peamist testimise taseme: ühiktestimine, integratsioonitestimine, süsteemitestimine ja vastuvõtutestimine.

Esimene testimise tase on **ühiktestimine** – programmi väiksemate koodiosade testimine, et tagada nende korrektse töö. Antud tasemel keskendutakse kontrollimisele [3]. Ühikteste tavaliselt kirjutavad programmeerijad, harvemini testijad, kui nad töötavad konkreetse koodi kallal. Ühiktestidega saab kontrollida, et konkreetne testitav funktsioon tagastab õige väärtuse ja ei riku koodi. Samuti aitavad ühiktestid leida vead, kus on erinevad andmetüübid või valed loogilised operaatorid [3]. Kui leitakse viga, siis arendaja saab seda koheselt parandada, sellega kiireneb tarkvara arendamise ja testimise protsess.

Teiseks testimise tasemeks on **integratsioonitestimine**, mille käigus kas kõik või osa komponentidest liidetakse kokku ja testitakse kui ühisosa, et tagada moodulite ühilduvuse üksteisega [3]. Tavaliselt projekti erinevad tiimid vastutavad oma teatud mooduli kirjutamise eest ja arendajad ei tea, kas ühe töörühma poolt kirjutatud

komponenti saab integreerida teiste töörühmade poolt kirjutatud komponentidega, ilma et süsteem annaks veateadet. Integratsioonitestimise eesmärgiks on garanteerida, et parameetrite edasiandmisel ei ole vigu, kui üks moodul kutsub välja teist moodulit. Süsteemi erinevad moodulid integreeritakse planeeritud viisil, kasutades integratsiooniplaani, mis määrab sammud ja järjekorra moodulite kombineerimisel täissüsteemi realiseerimiseks [3].

Kolmandaks tasemeks on **süsteemitestimine**, kus kõik allsüsteemid moodustavad tervikliku süsteemi. Testimisprotsess on seotud vigade leidmisega, mis tulenevad alamsüsteemide ja süsteemikomponentide vahelistest ootamatutest koostoimetest. Süsteemitestimisel veendutakse, et süsteem täidab funktsionaalseid ja mittefunktsionaalseid nõudeid [3].

Vastuvõtutestimine on viimane testimise tase, mida viiakse läbi siis, kui arendusmeeskonna poolt on kogu testimisprotsess läbi viidud ning leitud vead on parandatud. Valmis süsteemi antakse üle lõppkasutajatele ehk klientidele, kes omakorda testivad süsteemi kasutajate vaatenurgast ning otsustavad, kas arendust võib kasutusele võtta või on vaja viia sisse muudatusi ja parandada kasutajate poolt leitud vead. Vastuvõtutestimise eesmärgiks on saada kindlustustunnet süsteemi, selle osa või mittefunktsionaalsete omaduste kohta. Antud testimisel ei ole oluline vigade leidmine, vaid hoopis süsteemi valideerimine [4]. Andevis AS ettevõttes vastuvõtutestimise viivad läbi testijad enne suurema uuenduste paketi kliendi baasi paigaldamise, tagamaks, et tarkvara töötab õigesti ja ei teki vigu, mis võimaldavad rakenduse kasutamist takistada.

2.2 Testimise tüübid

On olemas erinevad testimise tüübid selleks, et määrata toote või teenuse kvaliteedi, mida saab lõppkasutaja projekti valmides.

Testimise tüüpe kasutatakse teatud taseme eesmärkide määramiseks programmi või projekti jaoks. Ettevõtte huvides on kasutada erinevaid tüüpe teste, kuna ühiktestimine või süsteemitestimine ei ole piisav kõikide testimise eesmärkide täitmiseks [5].

Testimise tüüpe on olemas nii palju, et kõike neid rakendada ühe projekti sees on raske ja aeganõudev protsess. Selles peatükis tutvustatakse testimise tüüpe, mida kasutatakse Andevis AS ettevõttes Microsofti meeskonnas.

Regressioonitestimise viiakse läbi pärast programmi funktsionaalsuse täiustamist või paranduse sisseviimist. Selle eesmärgiks on teha kindlaks, kas muudatuste tagajärjel või keskkonna muudatuse tõttu ei läinud katki mõni programmi töötav funktsionaalsus ning kas süsteem endiselt vastab nõuetele. Antud testimine on eriti oluline, kuna muudatused ja vigade parandused tekitavad tavaliselt rohkem vigu kui programmi lähtekood [2], [4].

Veaparanduse testimine on testitüüp, mida viiakse läbi kontrollimaks, et viimase testimise käigus ebaõnnestunud testjuhtumid on edukalt läbitud pärast parandamise [6]. Oluline on tagada, et veaparanduse testi täidetakse täpselt samamoodi nagu varasemal testimisel ehk samade sisendite, andmete ja keskkondade kasutamise abil [7].

Jõudlustestimise läbiviimisel kontrollitakse tarkvara töökiirust ja suutlikkust töötada kindlal koormusel määratud hulga andmete või kasutajatega. Hinnatakse kui kaua kulub ülesande täitmiseks aega ning kui palju korda on vaja jõudlusteste käivitada [4]. Jõudluse näitajad peavad olema kirjutatud nõuete kogunemise etapis, kuna ilma kirjalike spetsifikatsioonide või eesmärkideta ei saa teada, kas rakendus toimib vastuvõetavalt või mitte. Jõudluse näitajateks on tavaliselt reageerimisaeg või läbilaskevõime [2]. Näiteks, 500 palgalipikud peavad olema saadetud 20 minuti jooksul, ilma et programm mingil hetkel ei reageeri enam kasutaja tegevustele.

UI testimine on graafilise kasutajaliidese testimise protsess, mille tulemusel tagatakse kasutajaliidese vastavuse spetsifikatsioonidele. GUI (*Graphical User Interface*) testimine hõlmab endas ekraanide kontrollimist selliste juhtnuppudega nagu menüüd, nupud, ikoonid, tööriistariba, dialoogiaknad jne [8].

2.3 Testimise läbiviimise meetodid

Enne testimise läbiviimist on vaja valida konkreetset strateegiat, et see oleks efektiivne antud projekti raames ja selle abil saaks leida võimalikult palju vigu. Selles peatükis tutvustatakse kolme erinevat testimise strateegiat. Kaks kõige levinumat strateegiat on valge kasti ja musta kasti testimine. On olemas ka nende kahe kombineeritud meetod – halli kasti testimine.

Musta kasti meetodi kasutamisel käsitletakse programmi kui musta kasti ehk testija ei tea, kuidas see peab töötama sisemisel tasemel, ning ei tea ka midagi selle struktuurist [2]. Testimine toimub lõppkasutaja vaatenurgast ning teatud sisendi andmisel peab

programm andma testija poolt oodatava väljundi. Testide koostamise aluseks on programmi spetsifikatsioon, mitte programmi kood. Selle abil saab leida vigu, mis nõuete järgi ei pea esinema. Musta kasti strateegia võimaldab testida tarkvara kasutajaliidese kaudu, mis on kättesadav kasutajale. See tähendab, et testija ei pea oskama lugeda programmi koodi, et saada aru, kas rakendus töötab õigesti või mitte.

Valge kasti testimise meetodi kasutamisel uuritakse programmi sisemist struktuuri ja lähtekoodi. Tavaliselt testimise viib läbi kogenud testija, kes oskab lugeda koodi ja teab kogu tarkvara, või arendaja, kes paneb end testija rolli [2]. Valge kasti meetodi kasutamisel leitakse vigu koodis, mis on peidetud lõppkasutajast, ning sunnitakse testide arendajat hoolikalt rakendamise läbi mõtlema [3]. Reeglina selle strateegiaga katsetatakse ainult üksike programmi komponente. Antud meetod tagab süsteemi stabiilsuse ja jõudluse.

Viimastel aastatel on kasutusel kolmas meetod – **halli kasti testimine** ehk musta kasti ja valge kasti kombinatsioon. See on defineeritud kui tarkvara testimine koodi või rakenduse loogika osalise teadmisega, mis hõlmab juurdepääsu sisemiste andmestruktuuride dokumentatsioonile ja kasutatud algoritmidele. Halli kasti meetod on oluline integratsioonitestimise läbiviimisel kahe mooduli vahel, kui koodi kirjutasid kaks erinevad arendajad, kus ainult kasutajaliidesed on nähtavad testimiseks [9].

3 Tarkvara testimise käigus leitud vigade ja defektide analüüs

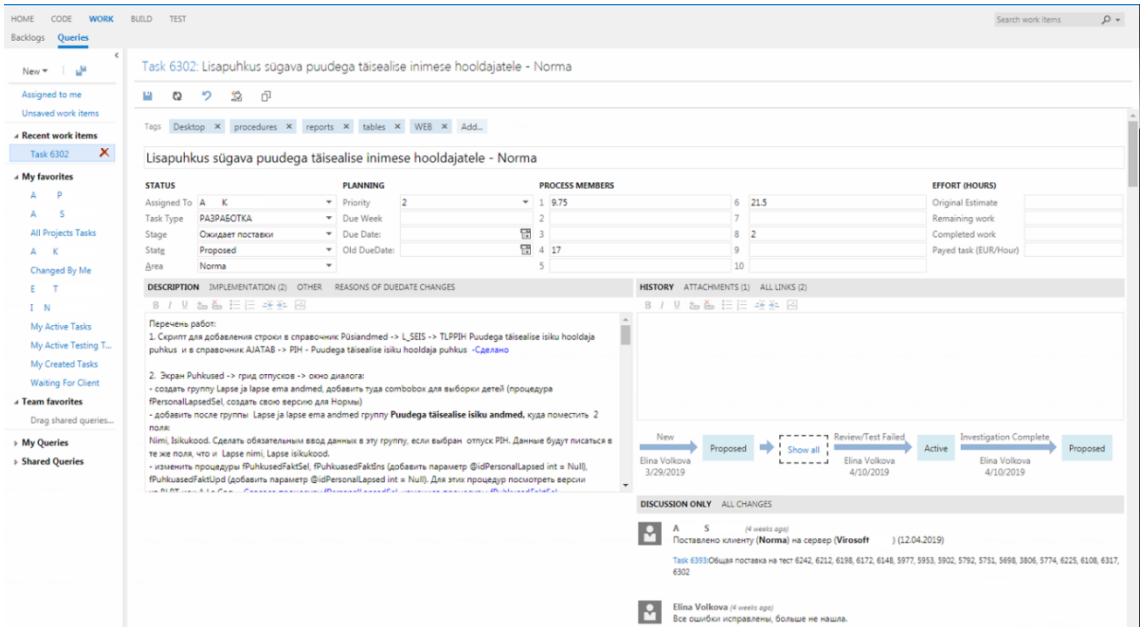
Kliendile toote või teenuse üleandmisel, ettevõtted soovivad pakkuda kvaliteetset tarkvara, et klient jääks rahule ja ei leidnud uusi vigu endapoolsel testimisel. Tihti aga juhtub nii, et kasutaja võib sisestada selliseid andmeid, mis ajavad programmi katki või annavad mitteoodatava tulemuse, mille peale keegi ei osanud arvata, et selline olukord saab esile tulla, kuna see tundus ebaloogiline. Lõppkasutaja leitud vigu mõnedes ettevõtetes otsustavad kohe parandada ilma analüüsi läbiviimiseta, mis aitaks selgitada, kas on vaja kulutada ressurssi probleemi lahendamiseks, kas see on tõesti programmi viga või mitte. Analüüsi käigus saab aru saada, miks antud viga tekkis, kas seda saab korrata mitu korda või see oli ainult ühekordne viga, kuidas seda saab parandada ja millise oodatava tulemuse peaks programm väljastama teatud sisendi andmisel. Selles peatükis kirjeldatakse erinevad tehnikad defektide analüüsimiseks ning näitajad, mida selle juures kasutatakse, ning viiakse läbi analüüsi viimase arenduse põhjal ja selgitatakse, kuidas on vaja teisiti organiseerida testimise protsessi. Viimati on saadud tellimus arendusele „Lisapuhkus sügava puudega täisealise inimese hooldajatele“, mille alusel toimub praktilise realisatsiooni kirjeldamine.

3.1 Defektide analüüs

Tarkvara defekt on tarkvara tootes tingimus, mis ei vasta tarkvara nõuetele, mis olid määratud nõuete faasis, või lõppkasutaja ootustele, kuidas peaks programm käituma [10]. Selleks, et tagada tarkvara kvaliteedi ja säästa raha, on parem ennetada vigu ning vältida neid kui hiljem parandada, kuna see võtab aega ja tööjõudu – ressursse, mida ei pruugi hetkel ettevõttes olemas olema. Defektide analüüs kasutab leitud vigade andmeid pideva kvaliteedi parandamiseks, püüab klassifitseerida defekte kategooriate järgi ja teha kindlaks selle esile kutsutud põhjused, et suunata jõudu protsessi parandamiseks ja täiustamiseks [11]. Pärast tehtud analüüsi tulemusi antakse edasi projektijuhile ja arendajatele, et viia sisse parandusi ning täiustusi. Sellele järgneb defektide ennetamine, mille eesmärgiks on tuvastada vigade põhjust ja vältida nende kordamist tulevikus [11].

Hetkel Andevis AS ettevõttes ei toimu vigade ja defektide analüüsi. Kui tarkvara kasutamise ajal või uue arenduse testimisel ilmub viga, siis kirjeldatakse leitud viga ning lisatakse pilte, kus on näha, milles on probleem. Selleks, et teha korraliku analüüsi, mis aitab selgitada vea põhjuse ning parandada seda, on vaja täiustada testimise protsessi. Esiteks, testijad peavad koostama *test case*'id – artefaktid, mille põhiolemuseks on teatud arvu tingimuste või toimingute teostamine, mis on vajalikud tarkvara funktsionaalsuse testimiseks [12]. Testjuhtumis on kirjas identifikaator, kirjeldus, nõuded, sammud, oodatav tulemus, tegelik tulemus, autor, testimise tulemus (kas on õnnestunud või mitte), lisad (näiteks ekraanipildid, logid). Teiseks, on vaja määrata vea raskustaset, mis näitab kui suurt mõju avaldab leidud viga rakendusele, ning prioriteeti, mis näitab ülesande täitmise või vea kõrvaldamise järjekorda. Kolmandaks, kui eelnevad kaks tingimust on täidetud, on vaja koostada nende põhjal aruannet, kus on lisaks kirjas projekti nimetus, rakenduse versioon, andmebaasi nimi, vea staatus, brauseri nimetus ja versioon (juhul kui toimub WEBi osa testimine) ning kellele on määratud.

Järgmisena näidatakse praeguse ja tulevase vigade raporteerimise viisid. Praegune vigade raporteerimine toimub TFSis (*Team Foundation Server*), kus on eeltäidetud väljad projekti nimetus, staatus (*Active, Closed, Resolved, Proposed*), tegeleja (valitakse töötajate nimekirjast, kes on kantud süsteemi), etapp (Hindamine, Töös, Testimine, Lisaküsimused, Kliendi baasile panemine jne) ning samuti on väli, kuhu iga projekti liige saab kirjutada enda poolt kulutatud aega. Joonisel 1 on esitatud TFSi keskkonnas püstitatud ülesande näidis, kus *Description* on ülesande kirjeldus, *History*'s on näidatud meeskonna liikmete kommentaarid, veergu *Process Members* kirjutatakse töötajate poolt kulutatud aeg ülesandele:



Joonis 1. TFSi keskkonnas püstitatud ülesanne.

Praegune raporteerimise viis TFSis:

1. Väljad Puudega täisealise isiku nimi ja Puudega täisealise isiku isikukood ei ole kohustuslikud.

2. Aruande 6.3 Tõend Sotsiaalkindlustusametile puhkuste perioodid ja tasud ei tule faktiliselt puhatud perioodi järgi.

3. Nupp Arvesta ei tööta.

4. Puhkuse ümberarvestusel veergu Puhkuseliik ilmub nimetus „Täiendav lapsepuhkus“, mitte „Hoolduspuhkus“.

5. Kustutatud puhkus jääb aruandes 6.3 Tõend Sotsiaalkindlustusametile.

6. Ei ole lisatud veerg PIH ekraanile Tööajatabel → Summeeritud töötunnid ning WEBist kutsutavatesse aruandesse.

7. Aruandes WEBist veergu PIH ei ilmu andmeid puhkuse kestvuse kohta (tundides ja päevades).

8. Arvestuse tegemisel tuleb ette viga „cursor is not open“.

Järgnevalt tabelis 1 on esitatud **parandatud raporteerimise viis** ühe eelpool mainitud vea põhjal.

Tabel 1. Parandatud raporteerimise viis.

Test Case ID	PIH_test1
Test Description	Kontrollida puhkusetasu ümberarvestust
Database	vir_norma_test
Severity	Major
Priority	Medium
Requirements	Puhkusetasu ümberarvutamine toimub vastavalt valemile *puhatud päevade arv * päevatasu puudega täisealise isiku hooldaja puhkus*, kus viimase väärtust saab muuta ekraanil *Teatmikud → Üldhäälestus.* Puhkuseliik jääb samaks pärast ümberarvestuse tegemist.
Test Steps	1. Ekraanil *Personal → Puhkused* sisestan hooldajapuhkuse. 2. Ekraanil *Palk → Arvestused → Puhkuste arvestamine* teen puhkuse arvestuse. 3. Ekraanil *Personal → Puhkused* lühendan arvestatud hooldajapuhkuse. 4. Ekraanil *Palk → Arvestused → Puhkuste üldümberarvestamine* arvestan puhkuse.
Expected Results	Puhkusetasu arvutatakse õigesti ilma vigadeta, puhkuseliik jääb samaks.
Actual Results	Puhkusetasu arvutamine on õige, kuid puhkuseliik pärast ümberarvestamise tegemist muutub teiseks.
Pass/Fail	Fail
Build Version	1.0.84.2
Browser Version	IE10

Parandatud raporteerimise viis aitab arendajal kiiremini aru saada, kuidas viga tekkis ning kuidas seda saab uuesti esile kutsuda, kuna kõik informatsioon on struktureeritud ja sammud kirjeldatud. Kui avastatud viga avaldab suurt mõju programmi käitumisele, siis arendajad saavad aru, milliseid tuleb parandada esmajärjekorras, et kõik toimiks korrektselt. Kui mingi nõue on arusaamatu või seda pole võimalik testida, siis testjuhtumi kirjutamine aitab leida probleemseid kohti nõuetes. Testjuhtumite kirjutamine aitab ettevõtte töötajatele hiljem tuletada meelde, mis oli juba testitud ning milliste tulemustega. Kui mõned testid ebaõnnestuvad, siis testijatele on lihtsaim leida ning ülekontrollida just neid esmajärjekorras peale vigade parandamist. Hetkel

laaditakse TFSi ainult neid testimistulemusi, mis ebaõnnestusid. Ajalugu testidest, mis olid täidetud korrektselt ei säili, ning neid on raske tuletada meelde paari päeva pärast. Uue töötaja tööle võtmisel on vähendatud aeg, mis on vajalik toote tutvumisega, kuna igal ajal on võimalik näha ja tuletada meelde, mis oli tehtud kas nädal või aasta tagasi. Järgmises peatükis on esitatud vigade klassifikatsioon ning veapõhjus.

3.2 Koodivigade klassifikatsioon

Tuntuim koodivigade klassifitseerimine ja nende esinemise põhjused tarkvara testimise ekspertide Cem Kaner & Burstein järgi on esitatud selles peatükis [13]. Selles peatükis analüüsitakse, millised kolm koodivead tekivad kõige sagedamini Andevis AS ettevõttes ning kuidas võiks nendest lahti saada või ennetada.

Initsialiseerimise defektid tekivad väljajäetud või ebaõigete muutujate initsialiseerimise lausete tõttu. **Algoritmilised ja töötlemisvead**, mis võivad tekkida näiteks ebaõige algoritmilise disaini rakendamisest või aritmeetiliste toimingute ebaõigest kodeerimisest. **Kontrolli, loogika ja jada defektid** tekivad peamiselt seetõttu, et kasutatakse ebaõiget tingimuslike avalduste rakendamist, näiteks *„if – then – else“, „case“* avaldused. **Vea käsitlemise defektid**, mis tekivad peamiselt sellepärast et pole võimalik õigesti ennustada ebaõigeid sisendandmeid, ning vigade käsitlemise protseduurid on kas täiesti välja jäetud või rakendatud mittetäielikult. **Koodi korduvkasutuse defektide** põhjuseks on selle liideste rakendamisel ja kasutamisel koodipiirangutega mittearvestamine. **Andmedefektid** tekivad selle pärast, et rakendatakse valesti andmestruktuure, nagu valed muutujate tüübid, kursorite ebakõlad ja teised. **Andmevoo defektide** põhjuseks on vale andmejada, näiteks kõigepealt on vaja muutujat initsialiseerida, siis kasutada ning kõige viimasena ignoreerida seda. Kui viga ignoreeritakse enne kasutamist või kasutatakse enne initsialiseerimist, siis tekib andmevoo defekt. **Teenuse, mooduli ja objekti liidese defektid** on need, mille tekkepõhjusteks on teenuste vaheline vale sõnumivorming, vale parameetrite deklaratsioon või valede parameetrite numbrid, mida edastatakse moodulite või objektide vahel. **Välised riist- ja tarkvara liidese defektid** tekivad väliste riistvarakomponentide vaheliste vale kodeerimise, väliste teenustega sõlmitud lepingute või sideprotokollide ebaõige rakendamise tõttu või valede parameetrite edastamise korral süsteemikõnedele. **Kasutatavuse defektid** on need defektid, mis tekivad kasutajaliidese elementide puudumisel või ebakorrektse rakendamise tõttu.

Koodidokumentatsiooni defektid tekivad olukordades, kus koodidokumentatsioon ei peegelda koodi funktsionaalsust või see on liiga salajane, et aidata saada aru koodi funktsionaalsusest. **Konfiguratsiooni ja versiooni defektid**, mis tekivad lähtekoodi komponentide vale versiooni haldamise, valesti ühendatud teekide ja muude süsteemiosade tõttu, mis on hetkel arendamise faasis [13].

Analüüsides erinevates projektides leitud vead, mis tekivad suuremates osades juhtudes nii arenduste kui ka vigade testimisel, siis Virosoftis esineb ainult osa nendest koodivigadest. Kõige suuremaks probleemiks on **konfiguratsiooni ja versiooni defektid**, kuna puudub ühine süsteem, kus oleksid nii lähtekood kui ka kasutajaliidese objektid. Versioonide haldamiseks hetkel kasutatakse Visual SourceSafe tarkvara, kuid tihti juhtub nii, et ühele kliendile pannakse protseduuri versiooni teisest kliendist, kuna nendel on üks loogika, näiteks ületundide arvestuses. Probleem tekib siis kui uus või isegi kogenud programmeerija ekslikult muudab ja rakendab vale protseduuri. Testija ei saa aru kui protseduur on vale, kuna ei säili ajalugu, mille abil võiks teada saada, et peab olema rakendatud teine versioon. Näiteks esialgu oli versioon Lidost, mida muudeti ja rakendati iga kord, kuid hiljem rakendati hoopis Olympicu oma. Ainult viimane arendaja, kes seda muutis võib, aga ei pruugi mäletada, mis oli enne. Hiljem on raske aru saada, milline versioon peab olema, kui muudetus on tehtud kohe kliendi serveril, ning ettevõtte jaoks kajastub see probleem kuludes. Praegu ettevõtte peab töötama selle nimel, et üle viia kõik Virosofti failid ühise süsteemi, näiteks Githubi, kus oleks lihtsam jälgida ja hallata versioone ning lähtekoodi. Selline organisatsioon aitaks vähendada konfiguratsiooni ja versioonide defekte.

Teiseks suureks probleemiks on **koodi korduvkasutuse defektid**. Kuna Andevis AS on palgaarvestuse ettevõtte, siis seadusest tulenevad muudatused on vaja teha kõikidele klientidele ning siin on üks loogika, mis kehtib kõikides Virosofti projektides. On olemas mõned muudatused meie riigis, mis ei kuulu seaduste hulka, mida teeb meeskond klientidele soovi korral ning need sõltuvad konkreetsest ettevõttest ja loogika võib erineda. Näiteks, kui tuli uus kohtutäiturite seadus ja lisapuhkus sügava puudega täisealise isiku hooldamiseks, siis peaaegu pool klientidest soovisid neid muudatusi kasutada oma Virosofti versioonis. Kui üks ülesanne oli valmis ja testitud, siis teiste projektide jaoks programmeerijad tihti kopeerisid koodiosasid ning ei otsinud liiga kaua erinevusi. See omakorda viis selleni, et iga kord tekkisid uued vead, kuna ühe ettevõtte jaoks kirjutatud koodi kasutati teise ettevõtte funktsioonides, kus on kasutusel teine

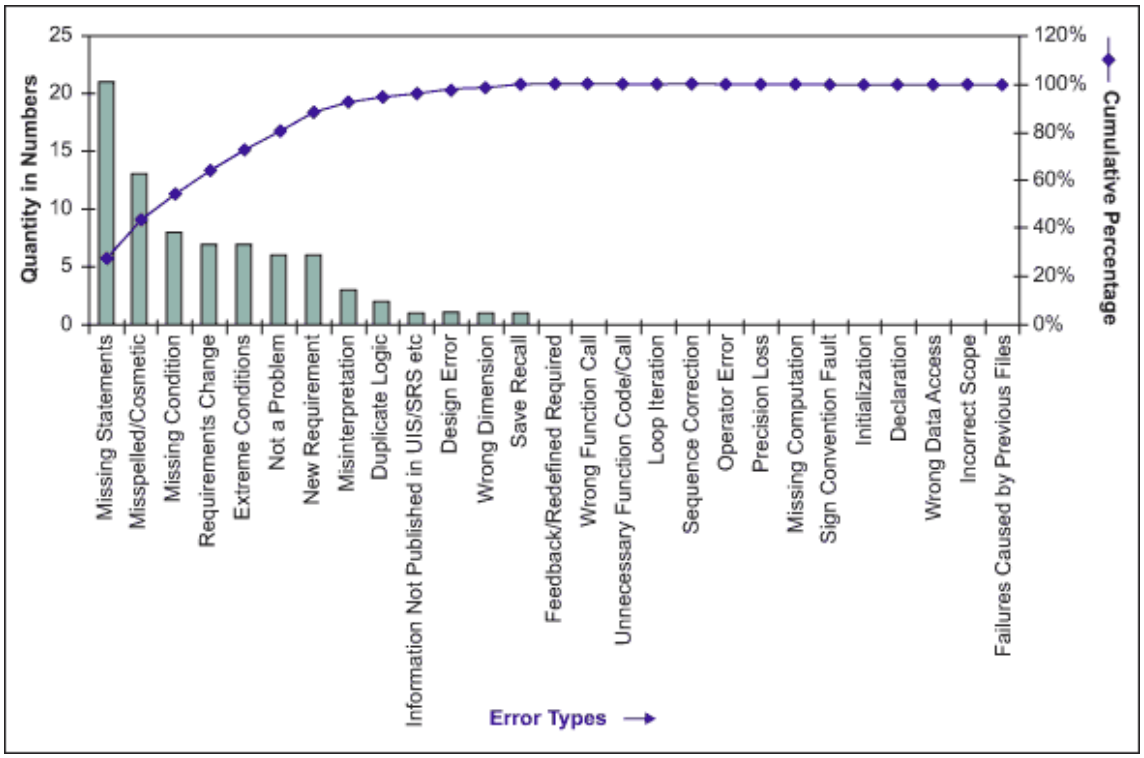
loogika ning koodipiirangutega ei arvestanud. Koodi korduvkasutusega seotud probleemid esinevad sagedamini uutel arendajatel, kuna nad arvavad, et saavad aru koodist, või ei soovi teada saada, kuidas konkreetne koodiosa töötab, vaid kopeerivad ja ei muuda midagi funktsioonides. Antud probleemi lahenduseks on vaja eraldada aega selleks, et arendajad kas ise või koos testijate ja analüütikutega vaataksid üle juba kirjutatud koodi funktsioonides ning nõudeid klientidelt, et leida võimalikud ebakõlad projektide vahel ja probleemsed kohad ning vältida neid. Kuigi see võtaks rohkem aega nii arendamiseks kui ka testimiseks, aitaks antud lahendus hoiduda probleemidest, mis on seotud koodipiirangute ja erinevate koodiosade kopeerimisega.

Kolmandaks probleemiks, mis vajab lahendust, on **andmevoo ja andmedefektid**. Andmevoo peamiseks probleemiks on parameetrite initsialiseerimise unustamine. Uue arenduse käigus lisatakse uue funktsionaalsuse ja ei kontrolli piisavalt kõike väärtusi, mida saab kirjutada väljadesse, ning hiljem esineb probleem palgaarvestusega kliendil, kus tuleb ette viga *„Arithmetic overflow error converting numeric to data type numeric.“* või *„Cannot divide by zero.“*, mis ei lase edasi programmiga töötada. Selliste operatsioonide lahenduseks tuleb kasutada erandite käsitlemist, mis annab vea ning laseb kasutajal edasi töötada programmis. Mõnikord juhtub nii, et funktsiooni lisatakse uusi parameetreid, kuid ei arvesta andmetega, mida võib väljadesse kirjutada. Näiteks klient soovib sisestada nii numbreid kui ka tähed, kuid programmeerija ei pane selle tähele ning valib andmetüübiks *integeri*, mitte *varchari*. Samuti tihti unustatakse kas lisada uue parameetri või eemaldada mittevajaliku tabelist, mille tagajärjel ilmub veateade *„Column name or number of supplied values does not match table definition.“* või *„Procedure or function has too many arguments specified.“*. Selliste probleemide lahendamiseks peavad programmeerijad ise kontrollima kirjutatud koodi ja veenduma, et kõik vajalikud parameetrid on lisatud ning ebavajalikud eemaldatud nii funktsioonidest kui ka tabelitest. Programmeerijad või testijad ise võiksid kirjutada automaatteste, mis kontrolliksid, et kõik tabelite, funktsioonide ja kasutajaliidese objektide parameetrid oleksid kooskõlas. Antud lahendus võtaks rohkem aega esimestel etappidel, kuid hiljem aitaks säästa nii raha kui ka aega, mida saab kulutada uutele arendustele.

3.3 Näitajad vigade ja defektide analüüsimiseks

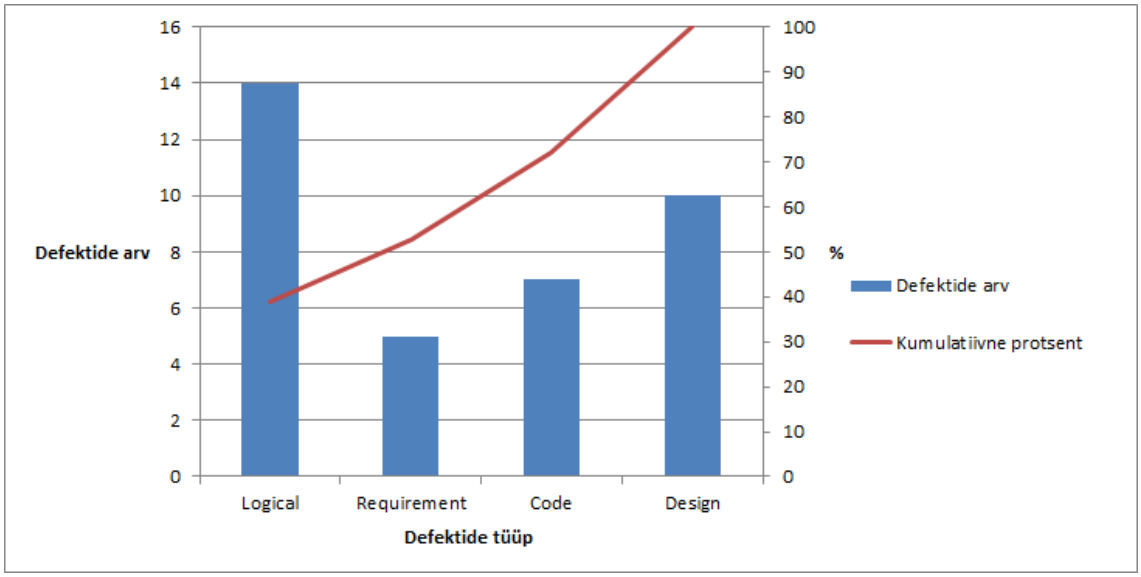
Selleks, et läbi viia analüüsi, mille järel saab ennustada ja vältida vigu tulevikus, on olemas kaks peamist näitajaid – *Defect Pareto Chart* ja *Root Cause Analysis*.

Enne algpõhjuste analüüsi läbiviimist koostatakse *Defect Pareto* diagrammi, kus näidatakse defekti tüüpi, mis esineb kõige sagedamini. Selle abil saab määrata kindlaks, millistele probleemidele tuleks anda suurema prioriteedi ja käsitleda neid esmajärjekorras. X-teljel näidatakse vigade tüüpe ning Y-teljel kumulatiivse protsendi [11]. Allpool joonisel 2 on esitatud näide *Defect Pareto* diagrammist:



Joonis 2. Defect Pareto Chart näidis [14].

Ühe Virosofti projekti defektide analüüsimise jaoks vaatasin üle viimase kuu jooksul leitud vead, klassifitseerisin neid nelja tüübi (loogika, nõuded, kood, disain) järgi ning koostasin *Defect Pareto* diagrammi, mida on näha joonisel 3



Joonis 3. Ühe kuu jooksul leitud defektide arv Virosofti projektis.

Joonisest 3 on näha, et kõige rohkem tekib loogika ja disaini tüübi vigu, vähem nõuete ja koodi tüübi vigu. Neile defektitüüpidele on vaja anda suurema prioriteedi ja parandada neid esmajärjekorras. *Root Cause Analysis* on protsess, mille käigus leitakse tegevusi või protsessi, mis põhjustavad defekte, ning selgitatakse võimalusi selle kõrvaldamiseks või vähendamiseks parandusmeetmete abil [11]. Selles peatükis analüüsitakse eelpool toodud diagrammi järgi leitud vigade põhjusi ja pakutakse meetmeid, mida võib rakendada kõikidele Virosofti projektidele, et parandada tarkvara kvaliteedi. Hiljem kui defektide analüüsi rakendatakse Microsofti meeskonnas, iganädalasel koosolekul on vaja eraldada aega, et arutada leitud vigu, nende põhjusi, kuulata kõike liikmete soovitusi ja valida parima lahenduse, mis aitab vähendada vigade arvu.

Loogika vigade peamised algpõhjused on vähene kogemus, äriloogika mitteteadmine, algoritmi kasutamine, mis ei sobi konkreetsele kliendile. Tavaliselt ettevõttesse tulevad töötada noored arendajad, peamiselt Venemaalt, kellel pole piisavalt kogemust antud valdkonnas. Suurem osa tulnud programmeerijatest on töötanud vabakutselistena ning tegelenud oma projektidega, peamiselt veebilehtede loomisega. Kuna ettevõttes kasutatavate programmeerimiskeelte oskus ei ole piisavalt heal tasemel, siis on vaja investeerida nõuetekohasele koolitusele, mis aitab uutele arendajatele kiiremini viia end kurssi kasutatavate tehnoloogiate ja protsessiga. Enne reaalsete projektidega töötamise ning peale koolituse läbimist peab uus arendaja lahendama eelkoostatud ülesandeid baasversioonis, mis annab nii analüütikutele kui ka programmeerijatele ülevaadet, kuidas uus töötaja lahendab probleeme, milliseid tehnikaid ja koodi kirjutamise stiili

kasutab ning kas tehakse edusamme. Koolitus ja ülesannete lahendamine annavad algteadmisi kasutatavates programmeerimiskeeltes ning tehtavate vigade arv muutub väiksemaks. Kuna ettevõtte tegeleb palgatarkvara arendamisega, siis programmeerija peab teadma nii kasutatavat keelt kui ka äriloogikat, mis kehtib Eestis ja igas projektis eraldi. Äriloogika teadmiseks tuleb tarkvara arhitektil viia läbi kõikidele meeskonnaliikmetele koolituse, kus tutvustatakse, kuidas toimub Eestis palga arvestamine ning selle realiseerimisest rakenduses. Kui kõik meeskonna liikmed omandavad vajalike teadmisi üldisemal tasemel, näiteks kuidas toimub sotsiaalmaksu juurdearvutamine ning milliste tingimuste järgi, on vaja tutvustada eraldi iga kliendi eripärasid, et noored arendajad juhuslikult ei muudaks õigesti töötava algoritmi. Klientide omapäradest peavad rääkima projektijuhid, kes tegelevad analüüsidega ning teavad oma klientide soovidest.

Ebaselged nõuded lähteülesandes, mittetäielik läbivaatamine, süsteemi teadmiste puudumine, disaini punktide vahelejätmine ning oma otsuste tegemine on peamised põhjused, miks tekivad **disaini tüüpi vead**. Testide ebaõnnestumiste peamisteks põhjusteks on punktide vahelejätmine ja oma otsuste tegemine, kui arendus on kliendi poolt kinnitatud. Programmeerijad peavad läbi viima omalt poolt testimise enne ülesande üleandmise testijale. Kui arendaja arvab, et ülesanne on valmis testimiseks, peab ise kontrollima end, käies läbi iga kirjeldatud punkti ja katsetades süsteemi lõppkasutaja seisukohalt. Selle käigus leitakse, et mõni punkt võib olla puudulikult tehtud või üldse tegemata olla. Antud tegevus aitab hoida silma kõikidel punktidel, mis on tehtud ja mis veel pole, vähendab testija poolt leitud vigade arvu ning hoiab kokku aega, mis läheb ülesande edasi-tagasi saatmisele ja kõikide punktide uuesti üle kontrollimisele. Ebaselgete nõuete puhul peavad arendajad küsima rohkem informatsiooni analüütikute ja süsteemi arhitekti käest, et veenduda nõuete õigesti mõistmises. Selleks võib koostada eraldi dokumendi, kuhu kantakse kõik disaini nõuete punktid ning esitatakse sellega seotud küsimusi, analüütik või süsteemi arhitekt vaatab neid üle ja kas dokumenteerib oma vastusi kirjalikult või arutab suuliselt ülesande eest vastutava arendajaga. Kui kõik vastused on saadud, on vaja täiendada disaini nõuete dokumendi, et lähteülesandes oleksid kirjas kõik arutatud nüansid.

Koodivead tekivad peamiselt koodi korduvkasutuse, vale muutuja tüübi valimise, vale, ebavajaliku või puuduva parameetri, ning kontrollimise funktsiooni ebapiisava realiseerimise tõttu. Näiteks, viimati leiti viga, kus ei avanenud maikuu vahetustöötajate

planeeritud kuugraafik, kuna mõningatel töötajatel oli tehtud töölepingu lõpetamine viimasel päeval (31.05.2019) ja ilmus viga *„Date must be beginning of the month 01 June 2019“*. Antud veateade tuli ette, kuna koodi kirjutamisel ei arvestanud viimasel päeval vallandamistega ning ei teostanud vajaliku kontrolli. Lahenduseks võib pakkuda arendamise käigus koodi läbivaatamine ja muudatuste sisseviimine. Leitud defektsed koodiplokid, mis ei täita nõudeid, ei tööta nii nagu vaja on ning mis ei ole vale, kuid mida saab parandada (näiteks loetavamaks muutmine, puhta koodi kirjutamine), võib programmeerija kohe parandada ja testida enne ülesande edasiandmist. Läbivaatuseid saab teostada arenduse varajases faasis, mis aitab leida nõuetega seotud vigu, ning leitud vea parandamine on odavam. Koodi ülevaatamine on kasulik programmeerijate koolitamiseks ja noorte arendajate abistamiseks programmeerimiskeelte ja -meetodite õppimisel. Koodi korduvkasutuse vigade vähendamiseks on vaja eraldada ressursse funktsioonide ülevaatamiseks, kontrollimiseks nõuetega kooskõlastust ja muudatuste sisseviimiseks. Kopeeritava koodi ülevaatamine aitab leida võimalikud ebakõlad, mis võivad kutsuda esile viga. Vale muutuja tüübi valimise ennetamiseks peab programmeerija kontrollima kas automaattestidega või manuaalselt, et valitud tüüp rahuldab kõike esitatud nõudeid (välja suurus, kas kasutatakse ainult numbreid, tähte või mõlemaid jne). Vale, ebavajaliku või puuduva parameetri saab samuti kontrollida automaattestiga, et funktsioonides ja tabelites kasutatavate parameetrite arv ning tüüp oleks sama. Kui antud test õnnestub, siis on vaja kontrollida, et parameetrid funktsioonides ja kasutajaliidese objektides oleksid kooskõlas.

Kõige vähem tekib **nõuete tüüpi vigu**. Sellise tulemuse saamine on seotud põhjaliku analüüsiga, mida viiakse enne arenduse töösse andmist. Peamisteks probleemideks on nõuete ebamäärasus, vale või ebaselge nõude kirjeldus või puuduvad nõuded. Probleemide lahendamiseks on vaja arutada analüütikute ja projektijuhtidega iga nõuet, esitada küsimusi aru saamaks punktide kirjeldust. Kui nõude osutub puuduvaks või ebaselgeks, siis on vaja võtta ühendust kliendiga, et täpsustada iga nüanssi ning kliendi vajadusi, mis aitab täiendada tehnilise kirjelduse ja saada aru loogikast. Enne kliendile hinnangu saatmist on vaja arutada süsteemi arhitektiga, kas on üldse võimalik teatud funktsiooni realiseerida Virosofti projektis. Selguse ja ühise arusaama saamiseks on vaja korraldada kohtumisi, kus on kõik meeskonnaliikmed ja ühe projekti kliendid ning iga liige saab esitada klientidele küsimusi, kui tundub et midagi on arusaamatu või ebaselge.

Acceptance criteria ehk vastuvõetavuse kriteeriumid on tingimused, millele tarkvara toode peab vastama, et oleks kasutaja või kliendi poolt aktsepteeritud [15]. Vastuvõetavuse kriteeriume on vaja kirjutada enne arenduse alustamist lähteülesandesse, mis tagavad et kõik realiseeritud funktsionaalsused töötavad nii nagu on kliendile vaja. Testimise käigus saavad testijad käia üle kõik vastuvõetavuse kriteeriume ja kontrollida, kas süsteem toimib ootusepäraselt. Kui mõni kriteerium ei ole vastuvõetav, siis ei saa arenduse kliendi baasile panna. Enne kliendi serverile muudatuste panemist programmeerija või süsteemi arhitekt kontrollib iseseisvalt kõike kirjapandud kriteeriume, mis lihtsustab lõpliku kontrolli. Kirjutatud kriteeriumid aitavad vältida ebamäärasust kliendi nõuetes ja väärkasutamist, kuna erinevad inimesed võivad läheneda püstitatud probleemile oma vaatenurgast. Selgelt püstitatud kriteeriumid kehtestavad ühe lahenduse, mida on vaja rakendada funktsionaalsuse lisamiseks või muutmiseks. Vastuvõetavuse kriteeriumi kirjutamiseks võib kasutada stsenaariumi šablooni *Given/When/Then*, mis on kirjutatud kasutusloo põhjal [16]. Näiteks: Personalitöötajana soovin sisestada töötajale lisapuhkuse sügava puudega täisealise isiku hooldamiseks, et raamatupidaja saaks välja maksta puhkusetasu. Stsenaariumi vastuvõetavuse kriteeriumi jaoks saab kirjutada järgmiselt. ***Acceptance Criteria:*** Puhkuste sisestamisel on kohustuslikud täisealise isiku andmed – isikukood ja nimi. ***Scenario:*** Arvestades, et olen personalitöötaja ja olen ekraanil *Puhkused*, kui ma vajutan nupule *Lisa uus* ja täidan väljad puhkuseliik, alguse ja lõppu kuupäev, täisealise isiku isikukood ja nimi, siis süsteem salvestab puhkuse.

Kui valmib suur uuenduste pakett, kus ühe projekti jaoks on kuu jooksul tehtud erinevad ülesanded, siis on vaja koostada **testimise ringi**, kuhu on koondatud kõik testitud ülesanded. Enne kliendi serverile panemist, testija kontrollib uuesti kõik tehtud muudatused ühel päeval. Omalt poolt programmeerijad peavad teostama seoste otsimist, mis annab teada, kas ühes funktsioonis tehtud muudatus viib vigadeni teistes kohtades. Näiteks, kas WEBis tehtud funktsiooni muudatus ühe ülesande sees võib kutsuda esile viga rakenduses. Antud lahendus tagab, et kogu süsteemi testimisel ja seoste otsimise käigus on võimalik leida vead, kus ühes rakenduse osas tehtud muudatus on mõjutanud teise rakenduse osa.

Ülaltoodud lahenduste rakendamine projektides ja vigade analüüsimine võtab rohkem aega testimiseks, kuid see aitab parandada kogu testimise protsessi, kuna seda analüüsitakse sügavamalt. Tulemusena meeskonnaliikmed saavad teada, mis oli vea

algpõhjuseks, kust viga tuli, kes vastutab tekkinud vea eest ning kuidas on paremini parandada ja ennetada tulevikus vigade tekkemist. Lõpptulemusena saab Andevis AS pakkuda oma olemasolevatele ja tulevastele klientidele kvaliteetsemat tarkvara.

4 Tarkvara testimise protsess Andevis AS ettevõttes

Andevis AS on tarkvaraarenduse ettevõtte, mille esialgseks fookuseks oli pangasüsteemide loomine. Hiljem aga otsustas ettevõtte keskenduda personali, tööajaarvestuse ja palga programmide arendamisele. Oma klientidele pakub ettevõtte Virosofti rakenduse, mille kasutavad Eestis tuntud suured ja keskmised ettevõtted. Kasutatakse agiilset tarkvaraarendust. Virosofti back-end rakenduse osa Microsofti tiimis on kirjutatud SQLi abil ning front-end osa on kirjutatud C# keeles. Ettevõtte peamiseks tugevuseks turul on see, et igale kliendile pakutakse erilahendust vastavalt vajadustele ja eelarvele, võrreldes teiste palga- ja tööajaarvestuse ettevõtetega, näiteks Meriti, Neptoni ja Taaviga. Seoses sellega kasvab uute arenduste järjekord ning suureneb testijate koormus, kuna enne lahenduse üleandmist kliendile on vaja läbi viia põhjaliku testimise, veendumaks et kõik kliendi soovid on täidetud ja töötavad laitmatult.

Töö autor on selles ettevõttes töötanud juba üle poole aasta tarkvara testijana ning arvab, et testimise protsess ei ole hetkel organiseeritud nii hästi kui võiks. Uute testijate töölevõtmisel ei anna ettevõtte edasi teadmisi, kuidas on organiseeritud testimisprotsess ning millele tuleb pöörata tähelepanu rakendusega töötamisel. Seega uued testijad mõnikord leiavad nende arvates vigu, kuid suuremates juhtudes see on õige programmi käitumine ning arendaja kasutab umbes tunni oma ajast, et saada aru, kas see on ikka viga või mitte, ja kirjutada selgitust. Seoses sellega kannatab loodud rakenduse ja uute arenduste kvaliteet, kuna puudub kindel testplaan iga arenduse või parandatud vea kohta ning osa vigadest jääb avastamata.

4.1 Kasutusel olev testimismeetod ja selle nõrgad küljed

Ettevõttes tänapäeval kasutatakse ainult manuaalset testimist ja musta kasti strateegiat, kuna testimine toimub läbi kasutajaliidese (WEBi või desktopi rakenduse osa). Enne uue arenduse üleandmise kliendile arendustsükli käigus viiakse ettevõttes vastuvõtutestimist, kus testija simuleerib lõppkasutaja käitumist programmi

kasutamisel. Selle käigus tagatakse, et programm töötab vigadeta ja täidab kõik püstitatud eesmärgid. Ettevõtte testijad kasutavad samuti veaparanduse testimise kontrollimaks, et kõik mainitud vead on parandatud ning uusi ei teki. Kuna rakendus on suur ning on kirjutatud üle tuhande protseduuri, siis veaparanduse testimise viiakse läbi iga ülesande korral ehk üks testija proovib leida nii uusi vigu, kontrollida, et varem leitud vead oleksid parandatud ning veenduda, et varem töötav funktsionaalsus ja varem parandatud viga ei tekiks uuesti.

Manuaalne testimine on tarkvara käsitsi testimise protsess defektide leidmiseks [17]. Testija käsitleb tarkvara lõppkasutaja vaatenurgast ja veendub, et kõik funktsioonid ning oodatavad rakenduse omadused töötavad nii, nagu on nõutud. Selles protsessis testijad täidavad erinevad testjuhtumid ja kirjutavad vigade raporti ilma automatiseerimise tarkvarata. Käsitsi tarkvara testimine on küll kõige primitiivsem testimise tüüp, kuid see on kõige kasutatavam meetod ettevõtete seas, kuna see tagab suurema leitud vigade arvu, kui kasutusel oleks ainult automatiseeritud testimine. Kõik uued rakendused või veebilehed tuleb enne testimise automatiseerimist ja kliendile üleandmist käsitsi kontrollida.

Iga arenduse, mida klient saatis analüüsimiseks ja teostamiseks, või vea, mida leidis lõppkasutaja, kohta koostatakse vastava ülesande TFSis. Seega kui on vaja kontrollida ja korrata tekkinud olukorda, tarkvara testija näeb ainult kliendi poolt saadud kirjeldust, kuid koodi ise testija ei vaata. Kui leitakse arenduse käigus tehtud viga või kordub kliendi poolt saadud viga, siis ülesande suunatakse arendaja poole, kes kontrollib koodi ja vajadusel viib sisse parandusi. Antud tööprotsess suurendab ressursside kasutamist ja nõuab rohkem aega.

Järgmisena on toodud peamised probleemid, mis on seotud manuaalse testimisega Andevis AS ettevõttes ja millele peavad nii projektijuhid, arendajad kui ka testijad tähelepanu pöörama, et rakenduse kvaliteet ei langeks.

1. **Kogu testimisprotsess ja saadud tulemused sõltuvad testija kujutlus- ja analüüsivõimest.** Arenduse korral projektijuhid viivad läbi analüüsi ja annavad kinnitatud dokumendi süsteemi arhitektile, et tema koostaks vastava ülesande. Kuna suurem osa kirjeldusest on kirjutatud tehnilises keeles, kasutatakse funktsioonide nimetusi, tabelite ja vaadete loetelu, siis testija vaatab sealt ainult

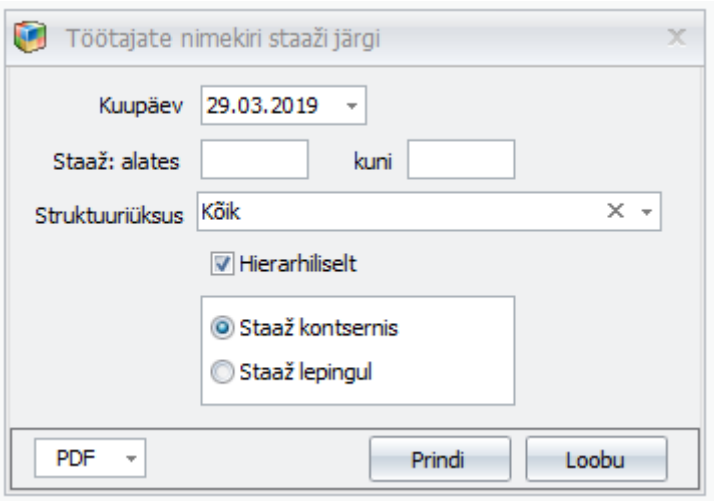
relevantse informatsiooni. Näiteks, millistel ekraanidel peab olema lisatud uus väli, millistes aruannetes peab see kajastuma. Kuna testplaani ei projektijuht ega süsteemi arhitekt ei koosta, siis testija peab ise välja mõtlema erinevad situatsioonid, millal võib tekkida viga või programm ei tööta õigesti. Seega kogu testimisprotsess sõltub ainult testija kujutlus- ja analüüsivõimest. Mõnede programmi eripärade ja seaduste mittetundmise tõttu tähtsad kohad jäävad märkamata ning testimata, mis kindlasti mõjub rakenduse kvaliteedile ja ettevõtte mainele klientide seas.

2. **Lähtekoodi mittevaatamine testimise käigus.** Kui valmib täiesti uus arendus, mida tellis klient, siis testijad kontrollivad peamiselt seda, mida klient saatis oma sooviks, näiteks uus töötundide arvestamise kord, et see töötaks nii nagu peab. Samuti kui arenduseks on, näiteks uute veergude lisamine aruandesse või vanade kustutamine, siis testijad ei kontrolli kui palju inimesi või millistest ekraanidest andmeid peab aruande funktsioon väljastama, kuna arenduse kirjelduses TFSis ei olnud seda kirjas. Seoses sellega tekivad probleemid, mis vajavad kiiret ja tasuta lahendamist, kuna see on Andevise viga – arendaja valesti kirjutas funktsiooni ning testija ei leidnud õigeaegselt viga, sest lähtekoodi manuaalse testimise käigus ei vaadata.
3. **Täieliku rakenduse testimine.** Uute arenduste või parandatud vigade testimine toimub peamiselt ainult selles programmi osas, kuhu viidi sisse muudatusi. See tähendab, et kogu programmi või kõikide kohtade, kus võib esineda parandatud väli, testimine ei toimu. Seoses sellega võib tekkida olukord, kus mõni ekraani avamine rakenduses annab veateadet ja ei lase kasutajat edasi, kuna puudub kas uus sisse viidud parameeter või vana parameeter ei ole esitatud. Kogu rakenduse testimine või kõikide kohtade leidmine, kus esineb uus väli või puudub vana, võtab palju aega kui teha seda ainult manuaalselt.
4. **Testimine on ajaliselt piiratud.** Veel üheks probleemiks on ajapiirang, mis ei lase läbi viia korraliku testimise. Kui saabub probleem, mis vajab kohe parandamist ja kliendi serverile lahenduse paigaldamist paari tunni jooksul, kuna kliendid peavad palka töötajatele maksma, siis projektijuhtidest tuleb informatsioon piiratud ajast testimiseks. Testijad omakorda keskenduvad ainult püstitatud ülesande täitmisele, et saada valmis õigeks ajaks, mitte testide katvusele ja kvaliteedile. Piiratud aja jooksul peab testija kontrollima, et enne leitud viga ei esineks ja samuti ei tekiks uusi defekte, ning vajadusel

dokumenteerima saadud tulemusi, et projektijuht annaks informatsiooni edasi kliendile.

5. **Piiratud piirkonna ja parameetrite testimine**. Kui saabub mingi konkreetne probleem, näiteks aruandes valesti kuvatakse tööstaaž kontsernis või ettevõttes, siis testijad pööravad testimisel tähelepanu ainult sellele probleemile. Viimati sai Microsofti meeskond kliendilt teada, et ühes aruandes valesti kuvatakse tööstaaži ettevõttes ja tööstaaži arvestamise kuupäeva. Arendaja poolt olid sisse viidud vajalikud parandused ning kuupäevad ja tööstaaž oli nüüd õige. Kui muudatused olid pandud kliendi serverile, siis süsteemi arhitekt avastas, et nüüd ei tööta staažiaastate filtreerimine ehk ei saa väljastada inimesi, kelle tööstaaž on vahemikus 0 – 5 aastat. Selgus, et arendaja paranduste sisseviimisel rikkus koodiosa, mis vastutas filtreerimise eest ja enne muudatuste sisseviimist oli töökorras. See on nii arendaja viga, kes millegipärast otsustas muuta koodi vales kohas, kui ka testija möödalaskmine, kes ei kontrollinud kõike protseduurisse edastatavaid parameetreid.

Joonisel 4 on esitatud aruande parameetrite aken:



Joonis 4. Aruande parameetrite aken.

Joonisel 5 on näidatud parandatud aruande vorm:

Kontsern
Töötajate nimekiri staaži järgi lepingul
Seisuga : 04.04.2019
Struktuuriüksus : Kõik

Lk. 1/67 04.04.2019 10:30

Töötaja nimi	Isikukood	Ametikoht	Tööstaaž lepingul: alates	Tööstaaž lepingul	ISCO
BLRT GRUPP AS Juhatus		**10101**			
B A	3	nõukogu liige	01.07.2018	00a 09k 08p	
I V	4	juhatuse esimees	12.01.2015	04a 02k 24p	
L V	3	nõukogu liige	01.03.2018	01a 01k 05p	
R A	4	nõukogu liige	03.12.2014	04a 04k 04p	
Kokku :		**4**			

Joonis 5. Parandatud aruande vorm.

Konfidentsiaalsuse huvides töötajate nimedest ja isikukoodidest on jäänud ainult esimesed tähed ja numbrid.

6. **Testimise läbiviimine lõppfaasis.** Ettevõttes toimub uue arenduse testimine ainult siis, kui kõik vajalikud protseduurid ja funktsioonid on arendajate poolt loodud või parandatud. Suuremas osas juhtudes vigade leidmise ja parandamise protsess võib kesta mitu päeva, kuna ühes kohas vea parandatakse ja teises – tekib uus. Üks arendus võib kesta kuni kahe kuu ning vea leidmise hetkest kuni arendaja saab hakata parandusi sisse viima võib mööduda üle nädala. Nii suur ajavahemik aeglustab kogu protsessi ja tihti on raske meelde tuletada, kuidas leitud viga tekkis, milles seisnes lähteülesanne ja miks oli tehtud mingi otsus programmeerimise käigus. Kui testimine algaks arendustsükli alguses, siis saaksid testijad näha kvaliteediprobleeme ja anda kohe tagasisidet uute funktsioonide kohta, mis omakorda kiirendaks kogu protsessi, vähendaks viivitusi ja lisakulusid, mis tulenevad vigade hilisest avastamisest.
7. **Dokumentatsiooni puudumine testide tulemustest.** Microsofti meeskonnas pannakse TFSis kirja ainult need tulemused, mis ebaõnnestusid testimise käigus. Kusjuures kirjeldatakse ainult leitud viga ning ei lisa kirjeldusele sammud, mis aitavad arendajal korrata situatsiooni. Kuna arendus võib kesta üle ühe kuu, siis aja möödudes on raske mäletada, milliseid situatsioone testiti ning millised tegevused ei viinud vigade tekkemiseni. Kui meeskond kasutaks testjuhtumeid, siis säiliks informatsioon testitud funktsionaalsusest, kus oleksid esitatud sammud, mida testija tegi konkreetse tulemuse saamiseks ja milliseid andmeid kasutas. Testjuhtumite dokumenteerimine aitab säilitada informatsiooni nii ebaõnnestunute kui ka õnnestunute testide tulemuste kohta.
8. **Ühelt testijalt ülesande üleandmine teisele testijale.** Microsofti meeskond teenindab üle 20 ettevõtet, kellel on erinevad soovid, vajadused ja seega on ka

algoritmid seadistatud erinevalt. Kui kliendilt tuleb soov uue arenduse kohta või leitakse viga, mis mõjub andmete täpsusele või ei lase kasutajat edasi, siis projektijuhid ja klienditoe spetsialistid püstitavad ülesande parandamiseks ning testimiseks. Üks testija viib end kurssi probleemiga ja hakkab mõtlema erinevate variantide peale, kuidas saab viga tekitada ning kuidas rakendus hakkab reageerima. Kui päeva lõpus ülesanne ei ole veel tehtud, siis see jääb plaanis järgmisteks päevadeks. Tihti juhtub nii, et üks testija on hõivatum kui teine, näiteks töötajal A on 5 ülesannet plaanis ja töötajal B hoopis 20. Kui selline olukord juhtub, Microsofti osakonna juhataja otsustab anda töötaja B mõned ülesanded, mis võivad olla juba pooleli testitud, töötajale A, kes peab end viima kurssi kogu tehtud töö ja saadud tulemustega. Kogu protsess muutub ajaliselt kulukamaks, sest on vaja läbi viia nii testimise, kui ka kogu ülesande eelanalüüsi, mis võib võtta mitu tundi testija tööst.

Üleval mainitud probleemid on ettevõtte peamised murekohad, mida on vaja parandada, et pakkuda klientidele parimat tarkvara, kus ei tekiks üllatavaid vigu. Tabelis 2 on toodud ühe kuu tugitundide aruanne, kus kajastatakse klientide pöördumisi kuu jooksul ja kulutatud aeg probleemide lahendamisele.

Tabel 2. Tugitundide aruanne ühe kuu eest.

Ettevõtte	**Tasuta tunnid**	**Kulutatud aeg**	**Aeg arvestamiseks**	**Aeg maksmiseks**	**Pöördumiste arv kuu jooksul**
Maxima Eesti OÜ	18	73t 20m	11t 45m	0t 0m	12
A. Le Coq AS	3	4t	30m	0t 0m	2
Atria Eesti AS	0	1t 45m	0t 0m	0t 0m	1
Enics Eesti AS	2	1t 40m	25m	0t 0m	2
Orkla AS	12	21t 45m	0t 0m	0t 0m	1
Meriton Hotels AS	4	4t	3t 30m	0t 0m	4
Norma AS	0	7t 45m	0t 0m	0t 0m	2
Tere AS	4	12t 30m	4t	0t 0m	4
Trafotek AS	2	24t 20m	2t	0t 0m	5
Olympic Entertainment Group AS	6	27t 25m	6t	0t 0m	8

Ettevõtte	**Tasuta tunnid**	**Kulutatud aeg**	**Aeg arvestamiseks**	**Aeg maksmiseks**	**Pöördumiste arv kuu jooksul**
Hilton Tallinn Park	6	14t 5m	3t 15m	0t 0m	5
Logistika Pluss OÜ	0	4t 30m	0t 0m	0t 0m	3
AS Kunda Nordic Tsement	4	3t 45m	0t 0m	0t 0m	2
Lido Eesti OÜ	0	9t 15m	0t 0m	0t 0m	9
BLRT GRUPP AS	1,5	10t 45m	2t 10m	40m	12
Onninen AS	0	6t 10m	0t 0m	0t 0m	4
Cramo Estonia AS	0	8t 15m	0t 0m	0t 0m	6
Ensto Ensek AS	4	11t 30m	1t 30m	0t 0m	4
Sanatoorium Tervis AS	0	2t 30m	0t 0m	0t 0m	2
Kokku		249t 15m	38t 35m	1t 40m	88

Tabelis 2 on näidatud Microsofti osakonda klientide tasuta tundide jääki, mille kogus sõltub sõlmitud lepingu tingimustest. Aeg arvestamiseks on osa kulutatud ajast, mis läheb kliendile kas maksmiseks (kui tasuta tunnid puuduvad) või mida arvestatakse maha tasuta tundidest. Kui ettevõttel on tasuta tunnid, kuid aeg arvestamiseks ületab selle kogust, siis selle vahe läheb maksmiseks (veerg *Aeg maksmiseks*).

Tabelist 2 on näha, kui palju tasuta tööd (tundides) teeb ettevõtte ebapiisava testimise pärast. Kokku oli kuus kulutatud 249 tundi ja 15 minutit probleemide lahendamisele, siin on nii arendajate, analüütikute kui ka testijate aeg. Sellest 38 tundi ja 35 minutit lähevad arvestamiseks ning ainult 1 tund 40 minutit esitatakse kliendile eraldi arvena ja saadakse selle eest raha vastavalt lepingu hinnakirjale. Kui võtta arvesse väärtused veergudes *Aeg arvestamiseks* ja *Aeg maksmiseks* ning lahutada neid veerust *Kulutatud aeg*, siis ettevõtte antud kuus tegi 209 tundi tasuta tööd.

Kui testimise protsess oleks organiseeritud paremal viisil, siis ettevõte pakuks kvaliteetsemat tarkvara, saaks rohkem raha oma tööde eest ning tõstaks oma mainet klientide seas. Protsessi organiseerimiseks on vaja pöörata tähelepanu aruannete koostamisele, vigade analüüsile ning tegevustele, mis aitavad kõrvaldada vigade allikaid ning ennetada neid.

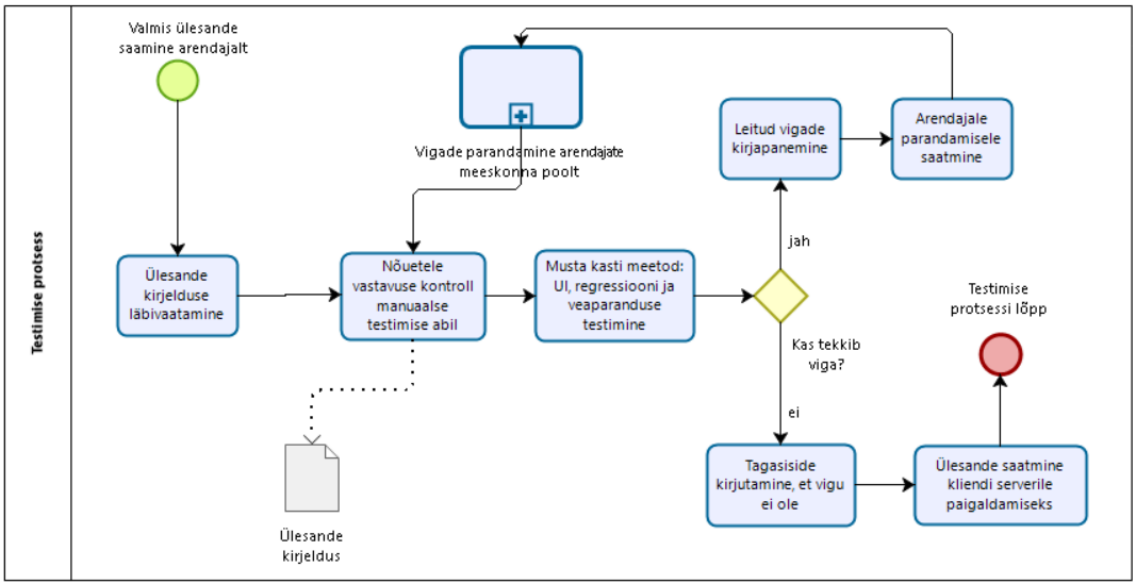
Järgmises peatükis pakutakse mitmeid lahenduse meetmeid, mida juurutatakse Andevis AS ettevõttesse Microsofti meeskonda.

5 Muudatusettepanekud Andevis AS ettevõttele

Tulenevalt eelnevas peatükis välja toodud probleemidest selles peatükis kirjeldatakse, kuidas on vaja testimise protsessi muuta ning kuidas on vaja juurutada ja läbi viia leitud vigade analüüsi.

5.1 Testimise protsessi muutmine

Microsofti meeskonnas, kuhu töö autor kuulub, toimub testimise protsess järgnevalt: valmis ülesande saamine arendajalt, kus on realiseeritud kõik kirjelduses olev funktsionaalsus, ülesande kirjelduse läbivaatamine ja arusaamine, kontrollimine nõuete vastavustele manuaalse testimise ja musta kasti meetodi kasutamise abil, nõuetega vastuolus olevate vigade leidmine, vigade esitamine kirjeldusena *„1. Arvestuse tegemisel programm arvutas valesti tulumaksu“* ning tagasi arendajale parandamisele saatmine. Kui vead on arendajate meeskonna poolt parandatud, siis korduvad eelmises lauses nimetatud tegevused seni kuni kõik leitud vead on parandatud, kasutusel on veaparanduse testimine ja regressioonitestimine. Juhul kui muudatusi tehti nii graafilises kasutajaliideses kui ka funktsioonides viiakse läbi UI testimise. Jõudlustestimine toimub ainult juhul kui kliendi poolt on tellitud arendus, kus on olulised ajalised näitajad, näiteks palgalipikute saatmine ning üle tuhande palgalipikud peavad jõudma e-postile ühe tunni jooksul. Kui ülesande testimise käigus ei leita ühtegi vea, siis kirjutatakse, et vigu ei leitud ning võib kliendi serverile panna. Joonisel 6 on näidatud hetkel kasutatav testimise protsessi voog.



Joonis 6. Kasutusel olev testimise protsess.

Joonisel 6 esitatud protsessi probleemsed kohad on järgmised tegevused: tagasiside kirjutamine, et vigu ei ole; leitud vigade kirjapanemine; ainult musta kasti meetodi kasutamine ning testimise läbiviimine lõppfaasis, mida näitab algtegevus *Valmis ülesande saamine arendajalt*. Leitud vigade ja tagasiside kirjutamise parandamiseks on vaja muuta meeskonnas kasutatavat praktikat testide tulemuste kirjapanemiseks, kus pannakse kirja ainult vigu. Selle probleemi lahendamiseks lisatakse testjuhtumite kirjutamist ja Exceli faili pidamist, kuhu kirjutatakse kõiki testjuhtumeid ja saadud tulemusi. Lahendamaks probleemi, mis on seotud testimisega lõppfaasis, hakatakse läbi viima testimise kohe peale nõuete ja vastuvõetavuse kriteeriumite kirjapanemist, mis kiirendab protsessi ja lahendab probleemi, mis on seotud piiratud ajaga testimiseks. Ainult musta kasti meetodi kasutamine on probleemiks, kuna ei vaadata programmi koodi ning ei saa teada, kas funktsioon on realiseeritud õigesti. Selleks peale musta kasti meetodi lisatakse valge kasti testimise meetodit. Antud meetod lahendab probleemi, mis on seotud piiratud piirkonna ja parameetrite testimisega, sest hakatakse vaatama tehnilise realisatsiooni, leidma vigu kirjutatud funktsioonides ning läbi viima seoste leidmise protsessi.

Protsessi muutmiseks ja probleemide, mis olid esitatud neljandas peatükis, lahendamiseks on vaja kasutusele võtta uue testimise protsessi mudelit. Järgmistes lõikudes on esitatud peamised meetmed protsessi parandamiseks.

Esiteks, iga ülesande jaoks on vaja lisada vastuvõetavuse kriteeriume, mis põhinevad kliendi tellimusel ja aitavad põhjalikumalt testida kogu süsteemi tööd. Peale iga

arenduse punkti, mis on esitatud lähteülesandes TFSis, süsteemi arhitekt koos analüütikutega peavad kirjeldama vastuvõetavuse kriteeriume, mille alusel testijad hindavad, kas arendus on valmis tellijale üleandmiseks või mitte. Kui arendus ei ole kliendi poolt tellitud, vaid muudatusi viiakse sisse ettevõtte poolt kõikidele klientidele, siis vastuvõetavuse kriteeriume peab kirjutama ainult süsteemi arhitekt, kes on kursis Eesti Vabariigi seaduse ning programmi õige käitumisega erinevates olukordades. TFSis lisab süsteemi arhitekt iga ülesande punkti järele vastuvõetavuse kriteeriumi järgmisel kujul:

1. Punkti realiseerimise kohta tehniline kirjeldus arendajatele

    a. Acceptance Criteria 1: *nõuetega kooskõlas vastuvõetavuse kriteeriumi kirjeldus*

    b. Acceptance Criteria 2: *nõuetega kooskõlas vastuvõetavuse kriteeriumi kirjeldus*

Vastuvõetavuse kriteeriumite järgi toimub enne realiseeritud funktsionaalsuse tööbaasile panemist arenduse ja rakenduse kontroll.

Teiseks, testimine peab toimuma nii varakult kui on võimalik. Kui kliendi poolt või süsteemi arhitekti poolt (juhul kui on ettevõttepoolne arendus) on kõik nõuded saadud, siis võib alustada testjuhtumite kirjutamist, mis aitavad leida puudusi nõuetes. Kui on leitud puudused või kvaliteediprobleemid, testijad annavad koheselt tagasisidet uutest funktsioonidest arhitektile või projektijuhile. Tulemusena on väiksem arv ebaselgeid ja puuduvaid nõudeid, mida võib avastada arenduse lõpus ning parandamiseks on vaja muuta kogu lisatud funktsionaalsust. See kiirendab kogu arendustsükli protsessi, viivitusi ja lisakulusid on vähem, kuna vead parandatakse jooksvalt. Mida varem avastatakse viga, seda odavam on selle parandamine. Samuti lahendab antud lähendamine piisava aja puudumise testimise jaoks, mis avaldab suurt mõju tarkvara kvaliteedile. Kui nõuded ja vastuvõetavuse kriteeriumid on kirja pandud, alustab testija *Test Case*'ide kirjutamist, mis aitavad leida puudusi nõuetes või realiseeritavuse võimatuse. Kui leitakse puudusi nõuetes, saadetakse ülesande tagasi arendajale ja projektijuhile ülevaatamiseks koos kirjeldusega. Kui puudusi nõuetes ei leitud, siis alustatakse realiseeritud funktsionaalsuse testimisega, kui on tehtud valmis vähemalt 1 punkt arendusest. Testjuhtumite kirjutamiseks kasutavad meeskonna testijad esialgu

Exceli programmi iga projekti jaoks. Joonisel 7 on esitatud Exceli näidis, mida hakatakse kasutama testjuhtumite kirjutamiseks, väljad *Actual Result* ja *Pass/Fail* täidetakse peale funktsionaalsuse realiseerimise ja testimise läbiviimise.

Requirement Number	Test Case ID	Test Case Description	Test Steps	Test Data	Expected Result	Actual Result	Pass/Fail
1	PIH_test1	Kontrollida puhkusetasu ümberarvestamist	1. Ekraanil *Personal → Puhkused* sisestada hooldajapuhkust.	PIH 2.03.2019-10.03.2019	Puhkusetasu arvutatakse õigesti ilma vigadeta, puhkuseliik jääb samaks.		
			2. Ekraanil *Palk → Arvestused → Puhkuste arvestamine* arvestada puhkust.				
			3. Ekraanil *Personal → Puhkused* lühendada arvestatud hooldajapuhkust.	PIH 2.03.2019-7.03.2019			
			4. Ekraanil *Palk → Arvestused → Puhkuste üldümberarvestamine* arvestada puhkust.				

Joonis 7. Testjuhtumite näidis Excelis.

Kolmandaks, on vaja dokumenteerida kõike testitulemusi. Isegi kui testimise käigus olid saadud positiivsed tulemused, mis ei viinud vigadeni, on vaja neid kirja panna, kuna hiljem aitab dokumenteeritud informatsioon testida kogu süsteemi. Tulemuste kirjapanemiseks ja säilitamiseks võib luua iga projekti jaoks ühise dokumenti, Exceli faili või kasutada veebipõhist tööriistu, näiteks *qTest*, *PractiTest*, mis aitavad hallata teste ja annavad kasutajale ülevaadet. Tööriistu saab integreerida ka teiste vahenditega, mis kasutatakse igapäevatöö organiseerimiseks, näiteks *Jira* platvormi, *Redmine*'i ja teistega. Testjuhtumite tulemustega dokumendi põhjal on arendajatel lihtsamini korrata viga ja parandada seda. Kliendile arenduse üleandmisel võivad projektijuhid saata dokumendi, kus on kirjas kõikide testide tulemused. Selle põhjal võib tellija kontrollida arenduse täitmist ja ootustele vastavust. Kuna TFSist üleminek Jira keskkonnale on aeganõudev protsess ja ei saa kõike faile üle viia ühe päevaga, siis meeskonnas hakatakse kasutama esialgu Exceli tabelit iga projekti jaoks, kus on mitu lehekülge iga arenduse kohta. Exceli tabelisse kirjutatakse testide tulemusi loodud testjuhtumite väljadesse *Actual Result* ja *Pass/Fail*, kusjuures pannakse kirja nii ebaõnnestunute kui ka õnnestunute testide tulemusi. Saadud tulemused hiljem kasutatakse analüüsides ja testimise ringi tegemisel.

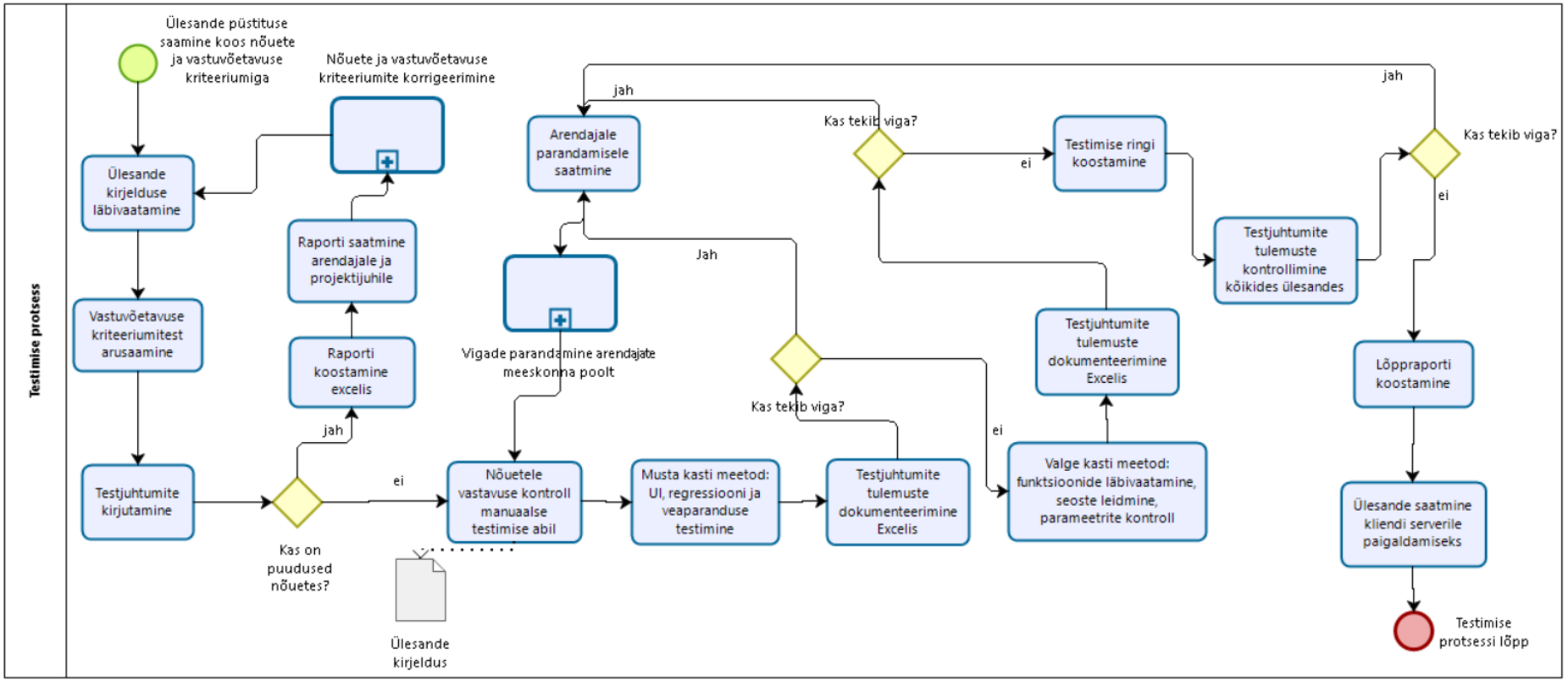
Neljandaks, kasutatakse tarkvara testimise valge kasti meetodi abil. Peale musta kasti meetodi testimist, testija uurib vajalikke funktsioone, kuhu olid viidud sisse muudatused, määrab, millised sisendid peavad olema õiged ja millised valed (ehk selle sisendi andmisel programm peab andma veateadet), ning kontrollib väljundeid oodatud tulemuste suhtes. Oodatud tulemusi määratakse koodi uurimisel ning analüütikute või projektijuhtidega suhtlemisel. Testija kirjutab vastavat lauset, mida on vaja andmebaasis jooksutada, või testi kasutajaliidese objektide testimise jaoks. Oodatud tulemuste kontrollimiseks kirjutatakse parameetritesse lubatud ja lubamata väärtusi ning vaadatakse, kuidas käitub funktsioon, kas tekib viga või mitte. Samuti sellise meetodi

testimisel viiakse koodi läbivaatusi ja seoste leidmise protsessi, mis tagab, et ühes rakenduse osas tehtud muudatus ei riku kogu süsteemi või ei mõjuta olemasoleva algoritmi. Kõik testimise tulemusi dokumenteeritakse Exceli abil, kus on kirjas kontrollitud funktsiooni nimetus, koodirida või koodiosa, kus on leitud viga, kasutatud parameetrid, oodatud tulemus ja saadud tulemus.

Testide automatiseerimine aitab kiirendada protsessi. Kui rakendus on testitud manuaalselt, kuid on leitud vead, mis peavad olema lahendatud, siis võib kirjutada automaattesti, mis kontrollib, et teatava sisendi andmisel süsteem annab õiget väljundit. On võimalik kirjutada teste, mis imiteerivad lõppkasutaja tegevusi. Testija jaoks on see ajavõit, mida võib suunata teiste ülesannete testimiseks, kuna ei ole vaja kogu manuaalse testimise protsessi uuesti läbi käima. Automatiseeritud testide jaoks saab kasutada vastavaid tööriistu, näiteks *Selenium, Postman, TestComplete*, *JMeter* ja teised. Automatiseeritud testimisse investeerimine võib olla kulukas, kuid aja jooksul tasub see end ära. Automaattestimise platvormi valiku ei jõudnud käesoleva bakalaureusetöö raames Virosofti projektijuhid teha, kuna on vaja konsulteerida testimises kompetentsemate spetsialistidega ning arutada juhatuse liikmega üle kui palju litsentse on vaja osta.

Enne kliendile tellimuse üleandmist on vaja käia läbi kõik testitud ülesanded, mida plaanitakse serverile panema. Selline testimise ring, kus on kõik konkreetsel perioodil tehtud arendused, mis ei ole veel serverile pandud, aitab leida vigu, kus ühe ülesande lahendus mõjutab teises ülesandes realiseeritud funktsionaalsuse. Projektijuhid peavad arvestama ajaga, mida hakatakse kulutama lõpptestimisele kõikide ülesande jaoks (mille kogus võib olla kas 5 või 20 ülesannet ühes ringis), ning eraldama selleks aega ja esitama kliendile täpsema (testimise ringi läbiviimise jaoks) hinnangu. Kui on kontrollitud iga tehtud ülesanne eraldi, siis luuakse suurema ülesande, kus on kirjutatud kõigi ülesannete numbrid, mida plaanitakse kliendi serverile panna. Testija jaoks eraldatakse minimaalselt neljandik tööpäeva ajast (2 tundi), sõltuvalt koondatud ülesande mahust ja keerukusest. Testimise läbiviimiseks kasutatakse projekti Exceli tabelit, kus on kirjutatud kõik testjuhtumid, ning käiakse neid üle ühe päeva jooksul. Kui kõik ülesanded on üle kontrollitud ja vigu ei leitud, siis saadetakse koondülesande kliendi serverile paigaldamiseks. Juhul kui on leitud viga, siis testimise protsess algab otsast peale kuni kõik vead on arendajate poolt parandatud.

Võttes arvesse eelpool kirjeldatud lahendusi on valminud parandatud testimise protsessi mudel, mis on esitatud järgmisel leheküljel joonisel 8. Protsesside modelleerimiseks kasutati Bizagi Modeler tarkvara. Parandatud testimise protsess on täiuslikum, põhjalikum ning kasutab valge kasti meetodi testimise läbiviimiseks, mis suurendab vea leidmise tõenäosust koodi läbivaatamisel.



Joonis 8. Parandatud testimise protsessi mudel.

Joonisest 8 on näha, kuidas muudeti protsessi ning milliseid tegevusi lisati. Protsessi lisati testjuhtumite kirjutamise, tulemuste dokumenteerimise Excelis, valge kasti meetodi, testimise ringi koostamise ning lõppraporti koostamise. Järgmisena on kirjeldatud parandatud testimise protsess.

Parandatud testimise protsess algab kohe peale ülesande püstituse saamist, kus on kirjas kõik nõuded ja vastuvõetavuse kriteeriumid süsteemi arhitektilt ning projektijuhilt. Edasi testija loeb ülesande kirjelduse, saab aru vastuvõetavuse kriteeriumitest ning hakkab esitatud informatsiooni põhjal kirjutama testjuhtumeid.

Kui leitakse nõuetes puudusi, näiteks ei ole võimalik testida või realiseerida funktsionaalsust, siis koostatakse raportit Excelis, mida saadetakse arendajatele ja projektijuhile. Sellele järgneb nõuete ja vastuvõetavuse kriteeriumite korrigeerimine vastavalt testijalt saadud informatsioonile. Kui antud alamprotsess lõpeb, siis algab testimise protsess otsast peale.

Kui nõuetes ei leia puudusi, siis alustatakse nõuete vastavuse kontrollimist manuaalse testimise abil. Kasutatakse musta kasti meetodit, kus on kasutusel UI, regressiooni ja veaparanduse testimise tüübid. Kõik saadud tulemused, kaasa arvatud õnnestunud testid, dokumenteeritakse Excelis kohe peale testjuhtumite täitmist. Kui leitakse viga, siis testija kohe suunab ülesande arendajale, kes omalt poolt parandab vigu ja saadab ülesande testijale tagasi manuaalse testimise läbiviimiseks.

Kui peale musta kasti meetodi testimist vigu ei teki, siis viiakse läbi valge kasti meetodi testimise, mille käigus vaadatakse läbi muudetud või lisatud funktsioone, otsitakse seoseid teiste funktsioonidega ja kontrollitakse parameetreid. Vea leidmisel saadetakse ülesande tagasi arendajale parandamiseks.

Kui valge kasti meetodi testimise käigus vigu ei teki, siis kõikide arenduste kohta teatud perioodi eest koostatakse testimise ringi, kus on kirjapandud kõik plaaneritud ülesanded kliendi serverile paigaldamiseks. Testija kontrollib kõiki testjuhtumeid, mis olid täidetud igas ülesandes, mille võtab Exceli failist. Juhul kui tekib viga, siis ülesande saadetakse tagasi arendajale parandamiseks. Vigade puudumisel koostatakse lõppraporti, kus esitatakse kõiki testjuhtumite tulemusi, ning ülesannet saadetakse kliendi serverile paigaldamiseks. Sellele järgneb testimise protsessi lõpp.

Viimaseks ettepanekuks on üleminek teisele versioonihalduse platvormile, näiteks Githubile, ja teisele ülesande raporteerimise platvormile, näiteks Jira või Redmine keskkonnale. TFS oli loodud 2000-ndatel aastatel ning ei ole enam Microsofti meeskonna jaoks aktuaalne, kuna ei peegelda nii hästi töövoogu ning selle ei saa integreerida teiste platvormidega. Üleminek Githubile aitab paremini hallata funktsioonide ja rakenduste versioone, mis lahendab probleemi versioonihaldusega, kui ühe kliendi jaoks mõeldud algoritm sobib ka teistele klientidele ning uue personaliseeritud funktsiooni loomine ei võimalda säilitada ajalugu eelnevatest versioonidest. Hetkel töötab ettevõttes renditöötaja, kes kasutab oma projektides Githubi süsteemi ja teab kuidas on paremini ülekanda kõike funktsioone, klasse ja meetodeid.

Microsofti osakonna juhataja poolt saadud informatsiooni järgi rakendatakse parandatud testimise protsessi projektides. Selleks eraldatakse arendajatele aega testijate koolitamiseks koodi lugemiseks, muudatuste sisseviimiseks ja vastavate programmide kasutamiseks (Microsoft SQL Server, Visual Studio, Visual Source Safe).

Järgmises peatükis kirjeldatakse vigade analüüsi juurutamise protsessi.

5.2 Leitud vigade analüüsi juurutamine

Vigade analüüsi juurutamise jaoks koostatakse iga projekti jaoks Exceli faili nimega „[*Projekti nimi*] vigade analüüs“. Ühele lehele hakatakse lisama ebaõnnestunud testjuhtumid ja määrama tüüpe vastavalt sellele, kus viga tekkis, näiteks loogika, disaini, koodi tüübid. Lehe nimetuseks on „Ebaõnnestunud testjuhtumid“. Teisel lehel hakatakse koguma informatsiooni vigade analüüsimiseks – leitud ühte tüübi vigade arv, protsent kõikidest vigadest, kumulatiivne protsent. Lehe nimetuseks on „Andmed algpõhjuste analüüsi jaoks“. Protsendi vigade koguarvust leitakse Exceli valemiga: ROUND([Ühte tüüpi vigade arv/Kogu vigadearv]*100,1). Kumulatiivse protsendi arvutamiseks kasutatakse valemit: Eelmise tüübi vigade protsent koguarvust + Jooksva tüübi vigade protsent koguarvust, näiteks nõuete vead = loogika vigade protsent koguarvust + nõuete vigade protsent koguarvust ehk 38.9 + 13.9 = 52.8 (nõuete vigade kumulatiivne protsent).

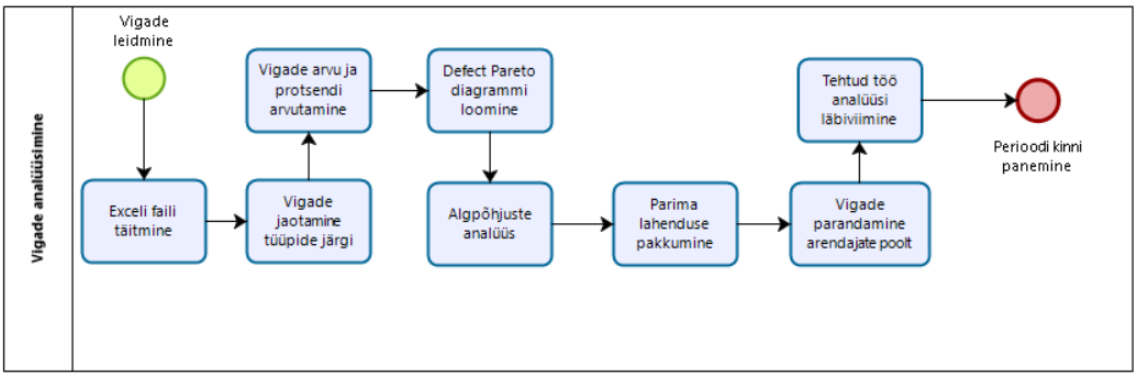
Kui juurutuseelsed tegevused on tehtud (kõik vajalikud failid loodud kõikidele liikmetele kättesaadavas kaustas), siis viiakse läbi kõikidele meeskonnaliikmetele koolituse. Koolitusel tutvustatakse defektide analüüsi, selgitakse mis see on, kirjeldatakse Defect Pareto diagrammi, Root Cause analüüsi, tuuakse positiivseid ja negatiivseid külge ning näidatakse kuidas on vaja Excelit kasutada diagrammide loomiseks. Koolitusel tutvustatakse näidise põhjal vigade analüüsi, kus valitakse ühes projektis leitud vead, selgitakse nende algpõhjusi ning pakutakse lahendusi. Selline lähenemine näitab praktiliselt, kuidas on võimalik analüüsi läbi viia. Koolituse lõpus saavad töötajad teada, mis on defektide analüüs ning kuidas tuleb samm-sammult vigu analüüsida, et vähendada tekkivate vigade arvu projektide lõikes.

Dokumendi, mille loomist kirjeldati peatüki esimeses lõigus, perioodiks valitakse pool aastat, kuna projektide arv on suur ja ühe kuu jooksul tegeleb Microsofti meeskond mitmete projektidega korraga. Informatsiooni leitud vigadest võtavad testijad eelmises peatükis kirjeldatud testjuhtumite failist ning kopeerivad neid esimesele Exceli lehele. Leitud vead kohe kirjutatakse faili sisse ja jaotatakse tüüpide järgi. Koostatud Exceli faili põhjal testijad hakkavad läbi viima defektide analüüsi hiljemalt ühe tööpäeva jooksul. Kui informatsiooni on kogutud piisavalt, siis testija loob kolmandal lehel Defect Pareto diagrammi, kus näidatakse arvuliselt, kui palju on ühte tüüpe defekte.

Saadud diagrammi põhjal viiakse läbi põhjalikuma analüüsi, mille abil saab tuvastada ühiseid mustreid vigade tekkimisel. Esimesena analüüsitakse neid vigade tüüpe, mille arv on kõige suurem. Tulemusena luuakse tabelit, kus on kolm veergu: vea kirjeldus, tekkepõhjus ja tegevuskava probleemi parandamiseks. Tabelis kajastatakse kõike leitud vigu, olenemata selle raskustaset. Peale lahenduse pakkumist edastatakse faili arendajale vigade parandamiseks. Iganädalaste koosolekute lõpus eraldatakse aega, et arutada nädala jooksul leitud vigu, millise lahenduse pakkus testija analüüsi läbiviimisel ning kas see aitas. Kui lahenduse probleemile ei ole leitud, siis arutatakse hiljem koos arendajatega selgitamaks võimalike variante. Rakendatud lähenemisviis veale aitab pikaajalises perspektiivis vähendada vigade arvu ja toob esile probleeme, mis oleks pidanud ära hoidma ettevõttes kasutatav poliitika. Iga uue perioodi alguses, näiteks iga kuue kuu lõpus, on tarvis läbi viia tehtud töö analüüsi ja saada aru, kui palju vigu ühe projekti sees oli leitud ning kui palju oli parandatud.

Kui parandatud defekte perioodi jooksul on rohkem kui leitud, siis on see hea näitaja, mis tähendab, et vigade analüüs aitas vähendada tekkivate vigade arvu. Kui aga leitakse rohkem vigu kui jõutakse ära parandada, siis meeskonna jaoks tähendab selline näitaja, et peab sisse viima muudatusi arendusprotsessi. Meeskonnas võib pidada eraldi statistikat, kus on loetletud kõik vead, mis on jaotatud vea raskustasemete järgi. Antud näitaja aitab leida, kui suur arv probleemidest tõkestavad rakendusega töötamist ning kuidas on võimalik sellise tüüpe probleeme lahendada.

Joonis 9 esitab samm-sammult vigade analüüsimise protsessi:



Joonis 9. Vigade analüüsimise protsess.

Joonisel 9 esitatud protsessi hakatakse kasutama Andevis AS ettevõttes Microsofti meeskonnas. Peale algsündmuse *Vigade leidmine* hakkab testija täitma Exceli faili esimese lehe, kus on esitatud kõik ebaõnnestunud testjuhtumid, ja jaotama vigu tüüpide järgi. Kui vead on jaotatud tüüpide järgi, siis teisel Exceli lehel arvutatakse ühte tüüpi kuuluvate vigade arvu ja protsenti vigade koguarvust, sealhulgas ka kumulatiivse protsenti.

Peale informatsiooni kogumist luuakse kolmandal lehel Defect Pareto diagrammi, mis esitab visuaalset, millise tüüpide vigu on leitud kõige rohkem. Siis viiakse läbi algpõhjuste analüüsi ja leitakse parima lahenduse, mille tulemusena tekib tabel koos vea kirjelduse, tekkepõhjuse ja tegevuskavaga probleemi lahendamiseks. Kui vigade analüüs on läbi viidud, siis toimub vigade parandamine arendajate poolt loodud tabeli põhjal.

Perioodi lõpus viiakse läbi tehtud töö analüüsi, kus vaadatakse kui palju vigu oli leitud ning kui palju neist olid parandatud, peale seda perioodi pannakse kinni. Tehtud töö analüüsi põhjal saadakse aru, kas juurutatud vigade analüüs aitab vähendada tekkivate

vigade arvu või mitte. Vastavalt analüüsi tulemustele viiakse sisse muudatusi arenduse protsessi.

Eelpool toodud muudatusettepanekud on vaja rakendada Microsofti meeskonnas. Projektijuhid, testijad ja arendajad peavad töötama koos ning leidma tekkivate probleemidele lahendusi, mis pikaajalises perspektiivis parandab kogu arendustsükli, pakub klientidele kvaliteetset ja põhjalikumalt analüüsitud tarkvara.

Projektijuhtide poolt eelpool toodud lahendused juurutatakse Microsofti meeskonda ning jälgitakse, kas pakutud meetmed toovad positiivse tulemuse või mitte.

6 Kokkuvõte

Käesoleva töö eesmärgiks oli analüüsida testimise protsessi ja pakkuda võimalike meetmeid erinevate testimisega seotud probleemide lahendamiseks ning vigade analüüsimiseks. Töös leiti lahendused, mis olid seotud ebapiisava testimise protsessi kasutamise, vigade dokumenteerimise ja vigade analüüsiga.

Antud töös tutvustati lähemalt testimise teooriat, toodi välja ja kirjeldati meetodeid, tasemeid ja tüüpe, mida kasutatakse ettevõttes.

Bakalaureusetöös toodi välja Andevis AS Microsofti meeskonna testimise protsessi kirjelduse, peamisi probleemi, mis tekivad ebapiisava manuaalse testimise tõttu, ning statistikat tugitundide kohta, mis peegeldab tasuta parandamiste ajalise kulu. Töös toodi välja, et hetkel kasutatakse ettevõttes ainult musta kasti testimist, kus tarkvara testija kontrollib süsteemi vastavuse nõuetele läbi kasutajaliidese ja ei vaata koodi.

Lähtuvalt leitud probleemidest ja vigade analüüsist ühe projekti näitel, koostati Defect Pareto diagrammi, kus avastati, et suurem vigade arv tekib loogikas ja disainis. Vigade põhjalikumalt analüüsides toodi välja võimalikke algpõhjusi ja pakuti meetmeid nende lahendamiseks.

Testimise protsessi peamisteks puudusteks on lähtekoodi mittevaatamine testimise käigus, testimise protsessi ja saadud tulemuste sõltuvus testija analüüsi- ja kujutlusvõimest, täieliku rakenduse testimise võimatus, ajaline piirang testimise läbiviimise jaoks, piiratud piirkonna ja parameetrite testimine, testimise läbiviimine lõppfaasis ja ühelt testijalt ülesande üleandmine teisele testijale. Samuti vigade haldamine ei ole struktureeritud ja testitud funktsioonide ning sammude kirjapanemist ettevõttes ei kasutata, mis raskendab ülevaadet juba testitud vigadest.

Viimases osas esitati muudatusettepanekuid Andevis AS ettevõttele, kus kirjeldati hetkel kasutatava ja parandatud testimise protsessi ning vigade analüüsi juurutamise. Meetmete rakendamine Microsofti meeskonnas aitab paremini struktureerida testimise

protsessi, jälgida testitud testjuhtumeid ja tulemusi ning läbi viia analüüsi, mis pikaajalises perspektiivis vähendab leitud vigade arvu.

Töö tulemusena tekis dokument, mida saab rakendada Virosofti projektides, et vähendada tekkivate vigade arvu ja saada aru leitud defektide algpõhjusest. Töö autor on veendunud, et meetmete rakendamise tulemusena ja pideva protsessi kontrollimise järel on testimise protsess lihtsustatud, läbipaistvam ning projektide üldine kvaliteeditase on tõusnud. Antud töös kirjeldatud vigade analüüsimise tehnikat saab kasutada, et selgitada välja tekkivate vigade põhjusi ja vähendada leitavate vigade arvu.

Vigade analüüsimist plaanitakse kasutusele võtta Andevis AS ettevõttes lähimal ajal ning antud töö tulemusi ja näiteid kasutatakse testimise protsessi ümber organiseerimiseks ning olemasolevate probleemide lahendamiseks. Edasiarendusena tuleks uurida vigade ennetamise meetmeid, et tulemus oleks parem.

Kirjanduse loetelu

[1] S. N. Shah, „Two of history's worst software bugs reported to be in medical software," 9. november 2005. [Võrgumaterjal]. https://www.healthcareguy.com/2005/11/09/two-of-historys-worst-software-bugs-reported-to-be-in-medical-software/. [Kasutatud 25. veebruar 2019].

[2] G. J. Myers, C. Sandler ja T. Badgett, The Art of Software Testing, Third Edition, New York: John Wiley & Sons, Inc., 2012.

[3] B. B. Agarwal, S. P. Tayal ja M. Gupta, Software Engineering and Testing: An Introduction, Jones and Bartlett Publishers, LLC, 2010.

[4] D. Graham, E. Van Veenendaal, E. Evans ja R. Black, Foundations of Software Testing, ISTQB Certification, Revised Edition, Cengage Learning EMEA, 2008.

[5] R. Silva, P. Perera, I. Perera ja K. Samarasinghe, „Effective use of test types for software development," 2017. [Võrgumaterjal]. https://ieeexplore.ieee.org/document/8257795. [Kasutatud 10. märts 2019].

[6] „Difference Between Retesting and Regression Testing," [Võrgumaterjal]. https://www.guru99.com/re-testing-vs-regression-testing.html. [Kasutatud 25. aprill 2019].

[7] „What is Confirmation testing in software?," [Võrgumaterjal]. http://tryqa.com/what-is-confirmation-testing-in-software/. [Kasutatud 25. aprill 2019].

[8] „GUI Testing Tutorial: User Interface (UI) TestCases with Examples," [Võrgumaterjal]. https://www.guru99.com/gui-testing.html. [Kasutatud 26. aprill 2019].

[9] I. Jovanović, „Software Testing Methods and Techniques," jaanuar 2009. [Võrgumaterjal]. http://tir.ipsitransactions.org/2009/January/Paper%2006.pdf. [Kasutatud 21. aprill 2019].

[10] „Defect," [Võrgumaterjal]. http://softwaretestingfundamentals.com/defect/. [Kasutatud 10. märts 2019].

[11] S. Kumaresh ja R. Baskaran, „Defect Analysis and Prevention for Software Process Quality Improvement," Oktoober 2010. [Võrgumaterjal]. https://www.researchgate.net/publication/49586396_Defect_Analysis_and_Prevention_for_Software_Process_Quality_Improvement. [Kasutatud 3. märts 2019].

[12] „Test Case," [Võrgumaterjal]. https://qalight.com.ua/baza-znaniy/test-case/. [Kasutatud 30. aprill 2019].

[13] „Analysis of defects found during software testing and action plan to prevent them," [Võrgumaterjal]. https://www.softwaretestinggenius.com/analysis-of-defects-found-

during-software-testing-and-action-plan-to-prevent-them/. [Kasutatud 10. märts 2019].

[14] ST3PP, „KPI – Identifying Areas in Process that Requiring Tuning,“ 12 märts 2014. [Võrgumaterjal]. https://www.st3pp.com/kpi_pareto/. [Kasutatud 12. märts 2019].

[15] S. Povilaitis, „Acceptance Criteria,“ 9. september 2014. [Võrgumaterjal]. https://www.leadingagile.com/2014/09/acceptance-criteria/. [Kasutatud 10. mai 2019].

[16] AdvancedAnalyst, „Приемочные тесты (Acceptance Tests) в виде сценариев для пользовательских историй (User Stories),“ 12. aprill 2017. [Võrgumaterjal]. https://advancedanalyst.com/2017/04/12/acceptance-criteria/. [Kasutatud 10. mai 2019].

[17] „Automation Testing Vs Manual Testing,“ 16 märts 2019. [Võrgumaterjal]. https://www.softwaretestingmaterial.com/automation-testing-vs-manual-testing/. [Kasutatud 23. märts 2019].