TARTU ÜLIKOOL

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Infotehnoloogia eriala

Polina Morozova

Süstematiseeritud testimise protsessi läbiviimine AS Webmedia näitel

Magistritöö (30 EAP)

Juhendajad: dotsent Helle Hein

Artur Assor, AS Webmedia

Autor: ……..… "….." mai 2011

Juhendaja: ……..…. "….." mai 2011

Juhendaja: ……..…. "….." mai 2011

Lubada kaitsmisele

Professor ..... ……..……. "….." mai 2011

TARTU 2011

Sisukord

1. Sissejuhatus

Tarkvaraarenduse maailmas on testimine hetkel üks kiiremini arenevaid valdkondi. Testimisest kujuneb välja omaette distsipliin, mis vajab kindlaid oskusi, teadmisi ja mõtlemisviisi. Võrreldes muu maailmaga, on testimise valdkond Eestis veel noor ja arenev. Seoses sellega võib esile tõsta mitut suurt probleemi.

Esiteks, tarkvara kvaliteet muutub iga päevaga järjest olulisemaks, kuna tarkvara ise muutub iga päevaga keerulisemaks nii tehnilisest aspektist kui ka äriloogika poolest. Järjest olulisemaks muutub ka tarkvara korrektne töö, sest sellest sõltuvad sageli firmasisesed tegevused, raha ja isegi inimelud (meditsiiniline tarkvara, lennundus). Järelikult, kvaliteetse tarkvara tootmiseks on vaja välja töötada vastavad testid ja standardid.

Teiseks, Eestis on raskusi kompetentse tööjõuga. Nagu näitab praktika, hakkavad testijana töötama inimesed (suur osa nendest on tudengid), kellel puuduvad teadmised testimise teooriast ja praktilised kogemused. Selle põhjusi võib leida infotehnoloogia õppekavadest, mis ei sisalda palju testimisega seotud aineid. Näiteks Tartu ülikooli infotehnoloogia ja informaatika bakalaureuseõppe õppekavade maht on 180 EAP ja testimisele on pühendatud ainult üks 3 EAP aine [1,2]. Samas võib väita, et situatsioon on paranenud võrreldes olukorraga, mis oli viis aastat tagasi.

Kolmandaks, teatud probleeme ja takistusi tekitavad tarkvara tellijad ise. Tellijatel ei pruugi olla IT-alast haridust ja ettekujutust tarkvara arenduse protsessist. Praktika näitab, et iga tarkvara tellija ei oma täielikku arusaamist testimisest ja tarkvara arendusest üldiselt. Seoses sellega peavad tarkvara arendajad pakkuma vastavat teenust (koolitusi, seminare, jne), et tarkavara arendamise protsess kulgeks sujuvalt ja tõhusalt.

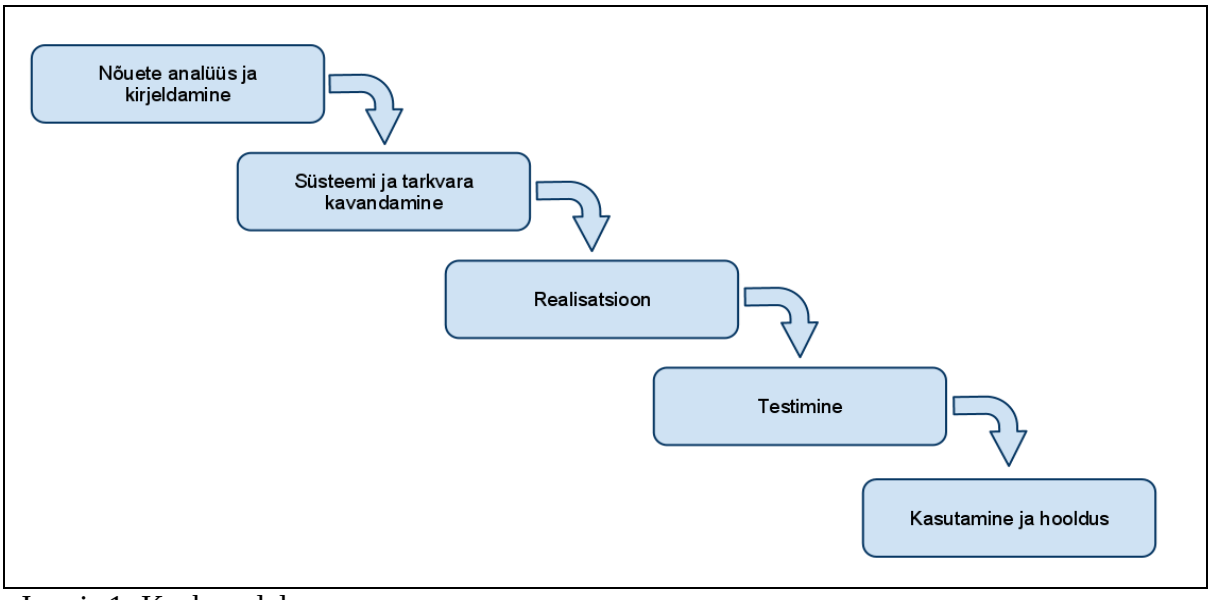
Neljandaks, kõige olulisem probleem Eestis testimise valdkonnas on seotud eestikeelsete materjalide puudusega. Kättesaadavate materjalide hulk koosneb praegu artiklitest Wikipedias [3] ja paarist testimise teemal kirjutatud lõputööst ja loengumaterjalist [4,5]. Need tööd on pühendatud testimisele konkreetsel testimise perioodil ja ei ole otseselt suunatud testijatele.

Samas tänapäeval arendusettevõtted organiseerivad oma tööprotsessi erinevalt. Suurtes firmades on tavaliselt aktiivne tööjõu liikuvus.

Võttes arvesse eespool nimetatud argumente, on käesoleva töö eesmärgiks anda süstematiseeritud ülevaade protsessidest ja printsiipidest, mida tavaliselt nimetatakse "testimiseks". Praktiline osa seisneb AS Webmedia jaoks testimise juhendi koostamisest. Töö raames puudutatakse ainult tarkvara arendamise protsessi ning välistatakse tootmisprotsess. Täpsemalt, magistritöös käsitletakse AS Webmedias tehtavat tarkvaraarendust. Tööd hakatakse kasutama Webmedias uute testijate väljaõpetamiseks.

Käesolev töö kirjeldab kogu arendusprotsessis toimuvat testimist, kuid ei puuduta testimise tehnikaid. Samas on selle töö eesmärgiks olla maksimaalselt lakooniline, et anda kogu vajalik informatsioon projekti testimise edukaks läbiviimiseks.

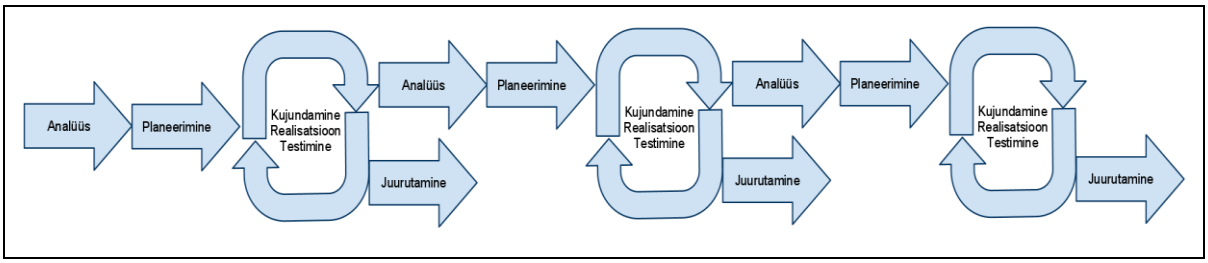
Mõned aastad tagasi oli arenduspotsessi klassikaliseks võtteks koskmudel e. lineaarne mudel (*waterfall model)*. Selline mudel kirjeldab konkreetset elutsüklilist lähenemist projektijuhtimisele, kus kogu arendusprotsess on eraldud faasideks ja ühe faasi töö tehakse korraga ära [6]. Joonisel 1 on esitatud koskmudeli elutsükkel: protsessi faaside läbimine on näidatud nooltega:



Joonis 1. Koskmudel.

Sellise mudeli puuduseks on raskendatud muudatuste sisseviimine, kuna faasid läbitakse järjest ja on väga raske eelmiste faaside juurde tagasi minna. Praktikas esineb aga tihti vigu, mille tõttu on vaja tagasi minna eelmisesse faasi. Sellest vajadusest lähtudes püüavad kaasaegsed tarkvaraarendamise- ja testimise protsessid järjest rohkem kasutada agiilset arendusmetoodikat [7]. Agiilne arendusmetoodika on kogemusel põhinev metodoloogia

tarkvarasüsteemide efektiivseks modelleerimiseks ja dokumentatsiooni loomiseks [8]. Joonisel 2 on esitatud agiilse tarkvaraarenduse elutsükkel:



Joonis 2. Agiilne arendusmetoodika.

AS Webmedia testijate eesmärgiks on luua selline testimise protsess, mis kulgeks pidevalt ilma tööseisakuteta. Erinevate arendusfaaside ja etappide käigud võiksid koosneda erinevatest meetoditest ja võimaldaksid organiseerida testimist tõhusalt kogu projekti elutsükli jooksul. Testimise eesmärgiks on kontrollida testitavat süsteemi võimalikult detailselt, efektiivselt ja produktiivselt ning teha seda testimiseks ettenähtud aja jooksul. Testimisprotsessi ülesehitamisel kasutatakse erinevaid testimise metodoloogiaid (*context-driven testing* [9], *ISTQB (International Software Testing Qualifications Board)* [10], *TMM*i (*Test Maturity Model integrated*) [11], *Tmap (Test Management Approach)* [12]), kuid erinevates projektikeskkondades võetakse kasutusele ainult vastavasse konteksti sobilikud meetodid ja lähenemised. Kasutatavad meetodid ei ole unikaalsed, kuid nende komplektid on unikaalsed iga projektis.

Antud töö peaks pakkuma huvi nii uutele testijatele kui ka tarkvara loomisega seotud ettevõtete klientidele.

2. Tarkvara testimine

Testimist on väga raske defineerida ja erinevad allikad teevad seda erinevalt. Kõige universaalsem, kuid keeruline selgitus on antud sõnastikus [13]: „testimine on protsess, mis koosneb nii staatilistest kui ka dünaamilistest elutsükli tegevustest. Testimine puudutab korraga nii tarkvara planeerimist, ettevalmistamist, hindamist kui ka nendega seotud töid. Testimine aitab teha kindlaks, kas valmistatav tarkvara vastab kindlaksmääratud nõuetele (sealhulgas ettenähtud otstarbele) ning avastab defekte.“ Cem Kaner [14] defineerib testimist lihtsamalt: „tarkvara testimine on läbiviidav uurimine, mille eesmärgiks on saada informatsiooni toote kvaliteedist või teenuse käitumisest testide all.“ Glenford Myers oma raamatus „The Art of Software Testing“ [15] esitab väga lihtsa ja arusaadava definitsiooni: „tarkvara testimine on rakenduse käitamise protsess kavatsusega tuvastada vead.“

Testija professionaalses elus hakkab testimise sisu kasvama ja muutuma vastavalt omandatud kogemustele ja teadmistele.

2.1 Testimise vajadus

Praktilises elus tegelevad inimesed testimisega iga päev. Testimine võib olla väga lihtne ja triviaalne – näiteks enne supi söömist proovitakse, kas supi temperatuur on paras söömiseks. Testimine võib olla ka mahukas ja aeganõudev - hea arvuti ostmiseks uuritakse arvutite turgu ja kui kohalikul turul sobivat ei leitud, siis uuritakse välismaa turgusid ning saadud informatsiooni põhjal tehakse valik.

Uue toote või teenuse puhul on testimine väga oluline nii tootjale (arendajale) kui ka tarbijale. Tootjal (arendajal) võimaldab testimine raha kokku hoida. Testimine võib näidata toote turvalisust ja seeläbi kaitsta tootjat lisakuludest peale toote tarnimist. Tarbija jaoks tähendab läbi testitud toode tavaliselt toote kvaliteeti ja ohutust.

Paraku kõik tooted turul ei ole turvalised ja tootja poolt hoolikalt läbi testitud, sest testimise protsess on keeruline, kallis ja ajamahukas. Selline olukord ei ole ainult tarkvara tootmises, vaid ka teistes valdkondades.

Miks on vaja testida, kui see on nii kallis ja ajamahukas protsess? Põhjus on väga lihtne – kõik inimesed teevad vigu. Vead ja nende tagajärjed võivad olla erinevad, mõnikord isegi katastroofiliste tagajärgedega. Mõned näited:

- Brauseri krahh (*browser crash)*. Brauser läheb ootamatult katki ja sulgub. Enamasti tekitab see vaid ebamugavusi ja kahju ei ole suur.
- *Ariane 5 Flight 501* [16]. See tarkvara viga on üks kuulsamaid. Seoses veaga tarkvara projekteerimisel (kehv kaitse täisarvu ületäitumise vastu), muutis kosmoserakett oma lennutrajektoori 37 sekundit peale käivitamist ja hävitas iseennast automatiseeritud süsteemiga sel hetkel, kui tugevad aerodünaamilised jõud põhjustasid sõiduki südamiku lagunemise. Õnneks oli tegemist mehitamata lennuga ja keegi ei saanud viga, kuid kahju oli rohkem kui 370 miljonit dollarit.
- *National Cancer Institute, Panama City* [17]. Mitmetes õnnetusjuhtumites arvutas raviplaneerimise tarkvara (loodud ameerika firma *Multidata Systems International* poolt) patsientidele kiiritusravi vale doosi. Vähemalt kaheksa patsienti suri ja veel 20 patsiendil tekkisid seoses kiirguse üledoosiga märkimisväärsed terviseprobleemid. Arste, kes olid kohustatud üle kontrollima arvuti arvutusi, süüdistati mõrvas.

Ülaltoodud näited tõestavad, et vead tarkvaras põhjustavad erineval tasemel kahju – rahast ja ettevõtte maine kaotamisest kuni inimeste surmani. Seetõttu võib väita, et testida on palju odavam kui mitte testida.

2.2 Testija

Kõik inimesed teevad vigu. Aga inimpsühholoogia on selline, et inimesed on pimedad oma vigade suhtes. Inimese aju ei lase meil piisavalt kriitiliselt ja erapooletult oma tööd kontrollida. Selle probleemi lahendamiseks peavad tehtud tööd kontrollima teised inimesed, eelistatavalt spetsiaalsete oskustega töötajad – testijad [18].

Testijal peavad olema teatud tüüpi iseloomujooned. Testija peab olema uudishimulik, kohusetundlik, rahulik, sihikindel ja järjepidev. Testijal peab olema hästi paindlik mõtlemisviis – ta peab oskama jälgida olukorda ja teha otsuseid lähtudes muutuvast olukorrast. Testija ei tohi midagi oletada ja mitte millessegi uskuda – kõike tuleb kontrollida. Samas on ka ülioluline õppimisvõime [19].

Testimisega tegeleva inimese lähenemine testitava objekti suhtes peab olema süstematiseeritud ja isegi teatud määral negatiivne [20].

Testija on inimene, kes kogub informatsiooni ja esitab selleks palju küsimusi:

- Kuidas peab rakendus töötama?
- Mis on oluline antud toote puhul?
- Kes hakkab seda kasutama? Kuidas? Miks?
- Miks rakendus ei tööta nagu oodatud?
- Mida on võimalik teha, et toodet parandada ja selle kvaliteeti tõsta?
- …

Küsimused esitakse mitte ainult projekti meeskonnale, vaid alati ka endale:

- Millised on minu eesmärgid?
- Millist tulemust ma tahan saavutada?
- Milliseid ressursse on vaja eesmärkide saavutamiseks?
- Milliseid vigu ma tegin ja mida ma edasi saan paremini teha?
- Kuidas on võimalik enda tööd parandada ja mida on vaja selleks teha?
- …

Testija analüüsib saadud informatsiooni ja teeb järeldusi. Rakendades oma oskusi, teeb testija oma tööd järjest paremini ja omandab väärtuslikke kogemusi. Nii nagu paljude teiste elukutsete korral, sõltub testija töö väga palju kogemustest. Mida rohkem on testijal kogemusi, seda paremini oskab ta oma tööd teha, suurendades nii produktiivsust kui ka efektiivsust.

Siin võib tekkida küsimus: „Kuidas testija erineb arendajast, kes testib oma koodi?“ Põhiline erinevus seisneb selles, et arendaja testib mõttega, et tema kood töötab, (arendaja teab, või siis arvab, et teab, kuidas tema poolt arendatud funktsionaalsus toimima peab), aga testija lähenemine on erapooletu (testija ei tea, kuidas arendaja tõlgendas analüüsi dokumenti) – ta ei oleta midagi – ta kontrollib kõike üle. Erinevad lähenemised annavad erinevaid tulemusi. Kuna testimine on testija põhitöö, siis erineb tema suhtumine kontrollitavasse objekti ja testimisse võrreldes teiste projektiliikmetega (arendajad ja analüütikud), kes suhtuvad testimisse kui lisakohustusse. Näiteks, kui arendaja testib funktsionaalsust, siis ta kontrollib, et see töötaks tema arusaamise järgi, testija samal juhul vaatab, kas loodud funktsionaalsus töötab korrektselt spetsifikatsiooni järgi ja kuidas see

töötab – kas konkreetne funktsionaalsus on kooskõlas rakenduse üldloogikaga, kas seda on mugav kasutada. Lisaks üritab testija leida puudujääke (ka triviaalseid ja ilmseid) nii arendaja kui ka analüütiku töös.

Webmedia testija igapäevane töö seisneb komponentide ja valmisrakenduse funktsionaalsete ja mittefunktsionaalsete nõuete testimises (erinevate valdkondade piires) vastavalt spetsifikatsioonile ja analüüsidokumendile (või vastavalt vajadusele). Testija tööks on samuti testimise protsessi planeerimine ja pidev jälgimine.

Testija peab tegelema mitte ainult päris testimisega, vaid ka teiste tegevustega, sest testimisega kaasnevad ka paljud teised lisatoimingud. Näiteks, paljude nooremate testijate jaoks on suureks üllatuseks tihe suhtlemine. Tavaliselt on vaja suhelda mitte ainult teiste testijatega, vaid ka arendajatega, analüütikutega, projektijuhiga ning kliendiga. Samas vastutatakse ka toote kvaliteedi eest. Seega testija peab olema ka hea suhtleja.

Testija kohustuste hulka kuulub ka erinevate dokumentide ja artefaktide koostamine:

- projekti kvaliteedi tagamise (*quality assurance* (*QA*)) strateegia defineerimine ja täiendamine projekti elutsükli jooksul;
- testplaanide koostamine;
- testjuhtumite koostamine;
- tarne raporti koostamine;
- kasutusjuhendite koostamine.

Üks väga oluline osa testijate töös on kliendisuhte hoidmine ja klienditoe teostamine. Testijad peavad tihti koordineerima kliendi testimise protsessi ja kliendilt sissetulevate vigade haldust. Kliendisuhtluse koordineerimisega tegelevad tavaliselt kogemustega testijad, kes on tihti ka projekti tootejuhid või projekti vastutavad testijad. Projekti ülejäänud testijate tööjaotus toimib vastavalt testplaanidele.

Projektijuhid teevad otsuseid, analüütikud kirjutavad spetsifikatsioone ja arendajad kirjutavad koodi; nende töö tulemuseks on artefaktid. Testija tegeleb just selliste artefaktidega. Et saada parimaid sisendeid oma töö jaoks (kõige aktuaalsemad, "*up-to-date*" artefaktid), peab testija jälgima kogu projekti rutiinset tööd. Näiteks peab testija jälgima, et kõik muudatused spetsifikatsioonides oleksid kirja pandud.

Veel üks oluline asjaolu, mida testija peab silmas pidama, on audit. Kui projekti peavad tulevikus hindama audiitorid, siis tuleb kontrollida, et kõik dokumendid (testjuhtumid, testimise aruanded, *QA* strateegia ja testplaanid) oleksid alati korras ja kirjutatud ettenähtud reeglite järgi. Sellest täpsemalt saab lugeda loengumaterjalist „Tarkvara kvaliteet ja standardid (IDX5721, IDX5722)“ [5].

3. Arendusprotsessi kirjeldus

Arendusprotsessi spetsiifikat ja protsessi tundmata on raske testimise protsessist aru saada. Webmedias kasutatakse agiilset, inkrementaalset ja iteratiivset arendusprotsessi. Inkrementaalne protsess on etappideks eraldamise ja planeerimise strateegia, kus süsteemi erinevad osad arendatakse järjest eraldi etappide kaupa ja pärast lõpetamist pannakse kokku. Iteratiivne protsess on planeerimise ümbertöötamise strateegia, kus eraldatakse aeg süsteemi osade ülevaatamiseks ja parandamiseks [21].

Järgnev protsessi kirjeldus ei puuduta müügitöid. Lisas 1 on esitatud arendusprotsessi diagramm.

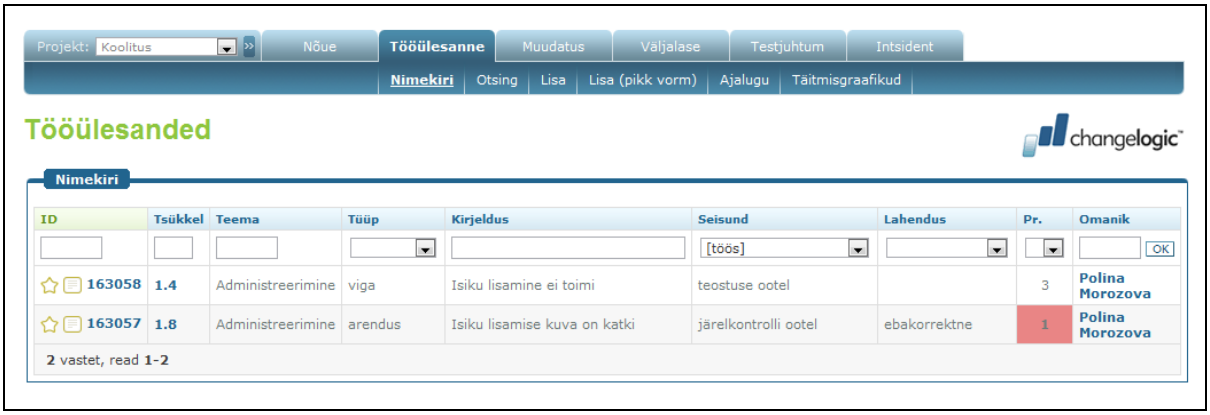
Protsess algab kliendi tulekuga või hanke avaldamisega. Hanke korral on protsess pikem ja algab pakkumusdokumendi koostamisega teenusepakkuja poolt. Juba pakkumuse koostamise etapil hakkavad töötama arhitekt, analüütik ja testija. Peale pakkumuse esitamist on kaks võimalikku sündmuste arengut: kui hanget ei võideta, siis protsess lõpeb, vastasel juhul sõlmitakse leping.

Kui leping on sõlmitud, alustavad analüütikud kliendiga suhtlemist ja koostavad projekti üldise ülevaate (*helicopter view*). Esimene ülevaade koostatakse eelanalüüsi alusel pakkumuse dokumendi kirjutamisel. Peale nõuete fikseerimist alustavad analüütikud tehniliste spetsifikatsioonide kirjutamist. Valminud spetsifikatsioonid testitakse eeskätt staatiliselt, pöörates tähelepanu tehnilistele aspektidele, loogilisele korrektsusele ja vormistamisele. Enne komponentide programmeerimist tuleb spetsifikatsioonid tellijaga kooskõlastada, seda vähemalt prototüüpi tasemel. Rakenduse mittefunktsionaalses prototüübis näidatakse, kuidas rakendus ja selle komponendid (nimekirjad, menüüd, nupud, lingid) hakkavad välja nägema ning kuidas protsessi läbitakse. Prototüübid koostakse spetsiaalsete vahendite abil ja tavaliselt need kujutavad endast staatilisi HTML lehti.

Projekti alguses paneb arhitekt paika rakenduse arhitektuuri ja valib kasutatavad vahendid (nt. millist andmebaasi ja selle versiooni hakatakse kasutama, mitu serverit on vaja, milliseid teeke kasutada, milline on põhiline programmeerimiskeel, kas on vaja rakendust jagada mooduliteks). Muutuva arhitektuuriga projektides osaleb arhitekt kogu projekti jooksul, teistel juhtudel lahkub ta pärast oma töö tegemist.

Suhtlemise protsess osapoolte vahel toimub enamasti järgmise skeemi alusel. Esmalt analüütikud suhtlevad kliendiga ja saadud informatsiooni põhjal kirjutatakse spetsifikatsioonid ning koostatakse prototüübid. Hiljem saadetakse need kliendile valideerimiseks; negatiivse tagasiside saamisel tehakse parandused ja saadetakse kliendile tagasi. Positiivse tagasiside juhul luuakse tööülesanded ja saadetakse teostamisele. Arendaja teostab ülesande ja saadab selle järelkontrolli. Testija kontrollib teostatud ülesanded ja saadab parandamisele tagasi või paneb kinni (lõpetab) ülesande. Kui avastati viga, siis see parandatakse ja saadetakse tagasi testijatele.

Nagu oli juba eespool juttu, valminud spetsifikatsioonide põhjal koostakse tööülesanded arendajate jaoks. Tööülesannete arv on reeglina suur ja nende haldamine ei ole primitiivne töö. Selleks kasutatakse veahaldustarkvara (nt. Bugzilla [22], Mantis Bug Tracker [23]), mis valitakse projekti vajadustest lähtudes. Webmedia kasutab enda loodud tarkvara - Changelogic [24]. Antud süsteem võimaldab luua ja hallata tööülesandeid, muudatusi, versioone ja testjuhtumeid, vaadata tegevuste ajalugu ja hallata testkeskkondi. Joonisel 3 on toodud Changelogic tööülesannete halduse menüü näidis:



Joonis 3. Changelogic tööülesannete halduse menüü.

Kuid Changelogic ei ole ainult veahalduse tarkvara, vaid ka versioonihalduse ja konfiguratsiooni haldamise (*configuration management*) tarkvara.

3.1 Versioonihaldus

Iga projekti raames, isegi väikese projekti puhul, tekib palju dokumente. Siia kuuluvad arhitektuuri- ja disaini lahendusega seotud dokumendid, spetsifikatsioonid, testimist puudutavad dokumendid ning ka rakenduse kood. Versioonihaldussüsteeme saab kasutada nii koodi kui ka kõikide ülalmainitud dokumentide haldamiseks.

Rakenduse kood kujutab endast suurt hulka tekstifaile, mis on ühendatud ühte suurde struktuuri ja on kirjutatud programmeerimiskeele reeglite järgi. Tihti sama kood (fail) vastutab suure funktsionaalsuse, mitme komponendi ühise loogika või kogu rakenduse turvalisuse eest ja versioonihalduse süsteemi mitte kasutades võib aja möödudes tekkida suur segadus. See suurendab ülesannete lahendamise keerukust ja tekitab rohkem defekte. Versioonihalduse süsteemid aitavad organiseerida koodi nii, et on võimalik niisugust olukorda vältida. Kõige olulisem aspekt versioonihalduse süsteemi kasutamiseks koodi jaoks on see, et kui koodi muudavad samaaegselt mitu inimest, siis versioonihalduse süsteemid võimaldavad koodi korras hoida. Veel üks suur pluss seisneb selles, et koodi kõik versioonid on kättesaadavad ja probleemi tekkimisel saab alati koodi töötavat seisu tagasiulatuvalt taastada.

Changelogic on seotud Apache Subversion (*SVN*) [25] versioonihaldussüsteemiga. Viimane hoiab rakenduse koodi puukujulises struktuuris, kus on olemas põhi- (*trunk*) ja lisaharud (*branch*). *Changelogic* peidab suurema osa interaktsioone alamsüsteemiga (*subversion*): kui rakenduse versioon on juba valmis, siis arendaja loob endale "muudatuse" (*change*), mis tekitab versioonihalduse struktuuris uue haru. See tähendab, et tehakse koopia rakenduse koodist, mille arendaja kopeerib enda masinasse. Versioon ja põhiharu jäävad samaks. Lõpetades arenduse, "muudatus" integreeritakse põhiharusse. See tähendab, et versiooni ja arendaja koodi koopiat võrreldakse ja kui muudatused olid erinevates koodiosades, siis arendaja muudatused lihtsalt kirjutatakse põhiharusse ja tekib uus versioon. Kui arendamise jooksul versioonis midagi muutus, (nt. integreeriti teine "muudatus") ja need muudatused ei ole automaatselt ühilduvad arendaja poolt tehtud muudatustega, siis tekib konfliktne olukord. Sellisel juhul "muudatuse" koodi ei kirjutata põhiharusse seni, kuni arendaja on vaadanud konflikti üle ja käsitsi kinnitanud, et konflikt on lahendatud ja tema kood on turvaline versiooni paigaldamiseks. Sellega tekib uus versioon Changelogic'us. Protsessi korratakse niikaua kuni projekt on valmis.

Koodi versioonihaldus on kasulik ka testijate jaoks - selle abil on alati võimalik teada saada, millises versioonis või muudatuses teostati konkreetne tööülesanne ja millisest versioonist tuleb rakenduse kood testimiskeskkonda paigaldada.

3.2 Projekti elutsükkel

Projekti on võimalik eraldada kaheks suureks osaks:

- *Pre-live* - periood enne rakenduse kasutusele võtmist, tavaliselt on ajavahemik rangelt piiratud.
- *Post-live* - periood peale rakenduse kasutusele võtmist. Tavaliselt selle perioodi pikkus ei ole nii rangelt piiratud.

Erinevates perioodides toimub rakenduse versioonide tarnimine erinevatesse keskkondadesse: *live*-keskkond on keskkond, mis on kliendil kasutusel ja kliendi testkeskkond on keskkond, kus klient testib ja valideerib teostatud funktsionaalsust. *Pre-live* perioodis toimub uute versioonide tarnimine kliendi testkeskkonda *live*-keskonna asemel. Sellel perioodil on kõige pingelisem aeg projekti elutsüklis, sest toimub rakenduse versiooni ettevalmistus toodangusse (*live*-keskonda) paigaldamiseks - rakendus antakse üle kliendile kasutamiseks. Toodangusse jõudmisega projektid reeglina ei lõpe (muidugi sellised projektid ka eksisteerivad), vaid jätkatakse arendusega ja täiendustega edasi.

Esimese versiooni paigaldamine *live*-keskonda tähendab *pre-live* perioodi lõppu. *Post-live* perioodi erinevus pre-*live* perioodist seisneb selles, et tarnimine toimub *live*-keskkonda, aga samuti kliendi testkeskkonda, kus tellija aktsepteerib tarnitud versiooni ja nõustub selle toodangusse paigaldamisega.

Post-live perioodis osutatakse ka rakendusele tuge (*support*) reaalajas. Seda tehakse kriitiliste vigade parandamiseks. Klienditoega tegelevad testijad, vajaduse korral arendajaid kaasates. Reeglina aidatakse klienti mitte ainult siis, kui probleem on põhjustatud arendusmeeskonna poolt, vaid ka juhul, kui klient ise tegi vigu. Eesmärgiks on tagada kliendi rahuolu.

On väga oluline analüüsida kõiki *live*-rakendusest tulnud vigu ja leida nende põhjused, et tulevikus vältida samade probleemide tekkimist.

Peale arenduse perioodi algab rakenduse hoolduse staadium. Selle perioodi jooksul parandatakse leitud vead ja teostatakse teisi töid vastavalt lepingule.

4. Testimise protsess ja sellega kaasnevad tegevused

Testijate kaasamiseks projekti eksisteerib kaks lähenemist – sõltumatu (*independent*) ning integreeritud (*integrated*) testimine. Sõltumatute testijate meeskonnad ei kuulu arendusmeeskonda - tihti nad asuvad teises kontoris, linnas või riigis. Testijad ei osale arendusprotsessis, neil ei ole mingit ülevaadet või kontrolli protsessi üle - kindlal hetkel projekti elutsüklis lihtsalt saadetakse arendatava süsteemi uus versioon testimisele. Testimine teostatakse selleks ettenähtud tsükli jooksul ja selle lõpul saadetakse tulemused (avastatud vead, probleemid) tagasi arendusmeeskonnale. Integreeritud testimise korral töötavad testijad arendajatega koos - ühes meeskonnas [26]. Integreeritud testimise korral on testimine arendusprotsessi lahutamatu osa. Webmedias tegeldakse integreeritud testimisega.

Sõltumatu testimise plussid:

- sõltumatud testijad on erapooletud ja näevad teist tüüpi vigu;
- sõltumatu testija saab kontrollida oletusi, mida tehti süsteemi analüüsi ja realisatsiooni käigus.

Puudusteks on järgmised asjaolud:

- isolatsioon arendusmeeskonnast;
- sõltumatud testijad võivad olla pudelikaelaks (*bottleneck*) nagu viimane kontrollpunkt;
- protsessi aeglus;
- puudub korralik suhtlus testijate ja arendajate vahel;
- süsteemi arhitektuuri ja arenduse meetodite mittetundmine;
- arendajad võivad kaotada vastutustunde kvaliteedi eest.

Integreeritud testimise plussideks on:

- lihtsam suhtlus;
- vähem dubleerimist;
- teadmiste jagamine;
- madalam testimise hind;
- terves meeskonnas (nii arendajatel kui testijatel) on ühine eesmärk - saada arendatav süsteem kvaliteetselt, õigeaegselt ja eelarve piires valmis.

Integreeritud testimise puudusteks on:

- eelarvamuslikkus;

- käitumise sõltuvus motivatsioonist.

Vastupidiselt üldlevinud arvamusele alustatakse testimisega palju varem kui mingi koodi osa on valmis programmeeritud. Testimise alustamine langeb kokku projekti algusega. Testimisega peab alustama hetkel, kui on teada esialgsed nõuded. Lepingu tingimuste ja pakkumuse dokumendid peavad olema ka testijate poolt üle vaadatud:

- kas testimise mahud on hinnatud adekvaatselt;
- kas testimisele vajalikud ressursid on selgelt defineeritud;
- kas testimise tegevused (üldjoontes) ja eesmärgid (üldjoontes) on defineeritud korrektselt;
- kas tähtajad on reaalsed.

Järgmise sammuna koostatakse projekti kvaliteedi tagamise strateegia (*QA* strateegia), mis peab olema kooskõlas ettevõtte *QA* poliitikaga ja strateegiaga (kasutatavate meetodite, vahendite ja protsesside suhtes). Enne strateegia koostamist defineeritakse eesmärgid ja sellest lähtudes valitakse meetodid, vahendid ja protsessid, arvestades konkreetse projekti spetsiifikat. Dokumendid, mis käsitlevad kogu testimise protsessi antud projekti jaoks algusest lõpuni:

- Testimise planeerimine. Kirjeldatakse ja koostatakse master- ja iteratsiooni testplaanid, vaadatakse läbi ja hinnatakse võimalikud riskid ja sellega kaasnevad kahjumid.

- Testimise protsessi jälgimine ja juhtimine. Defineeritakse nõuded ja vajalikud aktsepteerimise/tagasilükkamise protseduurid. Valitakse monitooringu vahendid, protseduurid, vastutajad ja eesmärgid. Lisaks kvaliteedi hetkeseisu jälgimine ja protsessi progressi jälgimine - kus me oleme, palju on tehtud, palju on teha jäänud, kas oleme graafikus jne.

- Testimise tehnikad ja disain. Kirjeldatakse testimise meetodid ja eraldi käsitletakse arendaja poolt teostatavad testimise tegevused. Määratakse meeskonna rollid ja kohustused.

- Testimisekeskkonnad. Pannakse kirja kõik testimise jaoks kasutatavad keskkonnad. Sellest täpsemalt peatükis „Testimiskeskkonnad“.

Kolmanda sammuna koostakse master testplaan, kus on planeeritud üldjoontes projekti testimine kogu elutsükli vältel.

Webmedia arendusprotsess kasutab iteratiivset ja inkrementaalset agiilset arendusmudelit. Kuna tarkvara arendus on jagatud iteratsioonideks, siis testimise protsess kordab sama mustrit. Iteratsiooni testimist on võimalik jagada kolmeks osaks: ettevalmistamine (planeerimine, läbimõtlemine), testide käivitamine ja lõpetamine. Sellisel kujul korratakse protsessi iga iteratsiooni puhul. Lisas 2 on esitatud testimine protsessi diagramm projekti jooksul.

4.1.1 Iteratsiooni testimise etapid

Ettevalmistamise etapp

Ettevalmistamise etapil mõeldakse läbi kogu iteratsiooni testimise protsess ja valmistatakse ette kõik vajalikud dokumendid ja plaanid, kuhu kuuluvad:

- iteratsiooni testplaan;
- uut funktsionaalsust katvad testjuhtumid;
- sobivate testimise metoodikate valikud.

Korralikult tehtud ettevalmistus lihtsustab ja muudab testimist efektiivsemaks ning võimaldab kokku hoida aega. Eduka ettevalmistuse eelduseks on arendatava tööülesande funktsionaalsusest arusaamine, mida on vaja tagada iga iteratsiooni alguses. Analüütikud või projektijuht peavad enne arendust tutvustama meeskonnale tulevasi töid ja seletama püstitatud eesmärke.

Testide käivitamine

Testide käivitamise etapil tegeldakse järelkontrollimisega, uuriva testimisega (*exploratory testing*) ja vigade otsimisega (*bug hunting*). Samal etapil, protsessi lõpus, käivitatakse ka regressioontestid.

Järelkontroll – testimise meetod, mille eesmärk ei ole vigade otsimine, vaid ainult kinnitatakse, et vajalik arendus/parandus on teostatud. Järelkontrollimist on võimalik rakendada kahe erineva eesmärgi saavutamiseks:

- kinnitada, et arendus on teostatud vastavalt nõuetele;

- kontrollida, kas parandus on teostatud või mitte.

Juhul, kui järelkontrollimist kasutatakse uue arenduse korrektse teostamise kinnitamiseks, nimetatakse seda „verifitseerimiseks“. Kui järelkontrollimist kasutatakse selle kindlaks tegemiseks, kas raporteeritud probleem sai parandatud või mitte, siis „kinnitustestimiseks“ (*confirmation testing*).

Regressioontestimine kujutab endast kogu rakenduse või selle kõige olulisemate protsesside läbi testimist. Regressioontestimist teostatakse nii käsitsi kui ka automaattestimise abil. Eesmärgiks on teha kindlaks, kas uus funktsionaalsus ja vana funktsionaalsuse täiendused ei rikkunud rakenduse osi, mis jäid selles iteratsioonis muutumatuks. Selleks koostatakse ja täiendatakse jooksvalt rakenduse protsesside nimekirja kogu projekti eluea jooksul. Joonisel 4 on esitatud näide regressioontestimise nimekirjast.

Protsess	Olulisus	Automaattest	Testjuhtum	Tulemus
Isikute otsing	1	ee.webmedia.project.isik.IsikKontekst.test		OK
Isiku lisamine	1		Testjuhtum 111111	NOK
Töökogemuse lisamine	1	ee.webmedia.project.tookogemus.TookogemuseLisamine*	Testjuhtum 111112	
Töökogemuse muutmine	1	ee.webmedia.project.tookogemus.TookogemuseMuutmine*	Testjuhtum 111113	OK
Töökogemuse kustutamine	1	ee.webmedia.project.tookogemus.TookogemuseKustutamine*		

Joonis 4. Rakenduse protsesside nimekiri regressioontestimiseks.

Iga protsessi jaoks määratakse protsessi olulisus, testjuhtum ja automaattest, juhul kui see on olemas. Regressioontestimise jooksul määratakse ka testide tulemus, mis hakkab näitama protsessi seisu. Seda dokumenti saadetakse ka kliendile tarne raporti osana koos tehtud tööde nimekirjaga. Sama dokumenti on võimalik kasutada rakenduse kaardistamiseks projektist parema ülevaate saamise eesmärgil, näiteks uue meeskonna-liikme projekti sisse elamisel.

Testimise lõpetamine

Testimise lõpetamine koosneb tavaliselt tegevuste ja tulemuste analüüsist, tulemuste raporteerimisest ja artefaktide arhiveerimisest.

Selle etapi raames vaadatakse üle kõik tööülesanded - kõik prioriteetsed ja olulised ülesanded peavad olema kinni pandud. Juhul, kui mõnda ülesannet ei ole võimalik teostada või kontrollida, siis tuleb sellest klienti kiiresti teavitada.

Eelmisel etapil saadud tulemused analüüsitakse ja tehakse vastavad järeldused. Analüüsi eesmärk on saadud kogemuse põhjal teha vajalikud parandused protsessi nii vara kui võimalik ja vältida samasuguste vigade kordamist. Kuid alati tuleb silmas pidada, et pidevat protsessi parandust ei saa alati tagada: tähtajad, ressursi puudus vms võivad olla takistuseks. Seoses sellega kogu projekti jooksul iga kuue kuu tagant uuendatakse testimise strateegiat ja teisi dokumente.

Projekti lõpus kogutakse kõik dokumendid, testjuhtumid ja automaattestid kokku ja arhiveeritakse. Neid andmeid on võimalik kasutada järgnevates projektides või eeskujuna teistele projektidele või noorematele testijatele.

4.2 Testimise protsessi planeerimine

Testimise protsess on keeruline ja kestab tavaliselt kaua. Protsessi keerulisus on põhjustatud erinevate osapoolte suurest arvust (tuleb teha palju kooskõlastusi) ja sisendite erinevusest. Testimine sisaldab palju erinevaid tegevusi ja seepärast on vaja testimist väga hoolikalt planeerida.

Arenduse protsessis arendatakse funktsionaalsust paralleelselt erinevate arendajate poolt. Testida suvalises järjekorras ei ole efektiivne, see võib põhjustada palju dubleerimist. Samas ei saa välistada, et peale edukat testimist mõne teise tööülesande teostamise raames (mis on seotud juba testitud funktsionaalsusega) tehakse juba testitud funktsionaalsus katki. Samal ajal ei saa tagada seda, et ajapuuduse korral just olulised protsessid saavad kontrollitud (vajadusel parandatud ja parandused uuesti üle testitud). Ühesõnaga planeerima peab selleks, et etteantud aja jooksul saaks süsteem võimalikult täiuslikult testitud.

4.2.1 Testplaan

Testplaan on testimise planeerimise vahend. See on dokument, mille abil kirjeldatakse ja planeeritakse testimise protsessi - tegevused, tähtajad, ressursid. Dokumendis loetletakse arenduses oleva projekti funktsionaalsed ja mittefunktsionaalsed nõuded, kasutatavad meetodid, testimise alustamise ja lõpetamise kriteeriumid ning vajalikud ressursid.

4.2.2 Testplaani koostamise vajadus

Testplaane on vaja koosta selleks, et paremini organiseerida testimise protsessi. Testimise protsess peab katma projekti terve elutsükli ja koostamise hetkeks peavad olema planeeritud teatud testimise tegevused.

Testimise protsessi planeerimine tõstab testimise produktiivsust ja efektiivsust. Korraliku planeerimise abil (riske hinnates ja selle alusel tegevustele prioriteete andes) saab testimine sooritatud parimal viisil selleks ettenähtud ajaga. Testplaan annab võimaluse saada hea ettekujutus loodavast rakendusest ja vajaminevatest ressurssidest.

Vajalikeks ressurssideks on:

- meeskond (mitu testijat on vaja?);

- oskused (kas testijatele on vaja lisakoolitusi?);

- tarkvara (kas on vaja lisa tarkvara?);

- riistvara (kas on vaja lisa riistvara?);

- aeg (kui palju aega on vaja, et teha piisavalt testimist?);

- sisend kolmandalt osapoolelt (andmete migratsioon, erinevad liidesed);

- testandmed.

Testplaani abil planeeritakse mitte ainult testijate tööd, vaid osaliselt ka arendajate, analüütikute ja projektijuhi tööd.

Arendusse võetavate tööülesannete järjekord on osaliselt määratud testplaanis tehtud hinnangutega riskide ja prioriteetide suhtes. Suuremad on need riskid, mis on seotud konkreetse tööülesandega: mida tehniliselt keerulisem on arendatav funktsionaalsus, seda varem peab alustama selle programmeerimist. Sellisel juhul jääb probleemide tekkimisel aega korralikule testimisele, parandamisele ja üle testimisele. Analüütikud saavad testplaanilt oma ülesanded, mis on seotud näiteks staatilise testimisega. Testplaan on samuti väga kasulik vahend projektijuhtimises - protsessi juhtimise, ressursside planeerimise ja tähtaegade fikseerimise suhtes erinevate otsuste vastuvõtmiseks. Lisaks on testplaan testijate jaoks vahend, mille abil saab testimiseks vajalikke ressursse (aeg, andmed, tarkvara, jne) dokumenteerida ja projektijuhilt nõuda. Testplaan on oluline testimise (ja ka arenduse) protsessi ja progressi jälgimise vahend.

Meeskond võib projekti jooksul muutuda ja selline inimfaktor ei tohi projekti edukusele mõjuda - testplaan võimaldab igal projektiga seotud inimesel olla kursis sündmustega, mis on toimunud ja mis on planeeritud projektis testimise poolel.

Üksinda töötamise korral võimaldab testplaan saada selgema ülevaate projektist ja aitab seetõttu oma tööd paremini organiseerida, kuid ei pea olema väga detailselt ja ametlikult vormistatud.

4.2.3 Master testplaan

Master testplaani koostab vastutav testija koostöös *QA* meeskonnaga. Lisas 3 on esitatud näide ühe iteratsiooni testimiseks master testplaanis.

Master testplaan kirjeldab üldiselt kogu testimise protsessi. Selle abil defineeritakse, millal ja millest alustatakse automattestimisega ja andmete migratsioonidega (juhul, kui see on

vajalik), loetletakse kõik kasutatavad testimise meetodid. Samas jagatakse testimise protsess iteratsioonideks ja otsustatakse funktsionaalsete ja mittefunktsionaalsete nõuete testimise järjekord. Edasi planeeritakse regressioontestimisele vajalik aeg: esimeses iteratsioonis puudub vana funktsionaalsus ja pole vaja teha regressioontestimist, iga järgmise iteratsiooniga ja uue arendatava funktsionaalsusega regressioontestimise aeg hakkab suurenema.

Master testplaani eesmärgiks on mõelda kogu testimise protsess põhjalikult läbi, panna kõik planeeritavad tegevused kirja, et mitte unustada midagi olulist hetkeks, mil tuleb projekti pingeline aeg.

Master testplaan peab vastama järgmistele küsimustele:

- mida testida - rakenduse ja liideste, funktsionaalsuse ja komponentide kirjeldus;
- kuidas testida - testimise strateegia (valitud meetodid);
- millal testida - mis hetkel millised projekti osad peavad olema läbi testitud;
- kogu projekti testimise alustamise kriteeriumid - millal alustada testimist;
- kogu projekti testimise lõppemise kriteeriumid - millal lõpetada testimine.

Master testplaanis on vaja kirjeldada ka vajalik varustus (riist- ja tarkvara).

4.2.4 Iteratsiooni testplaan

Iteratsiooni testplaan koostatakse iga iteratsiooni algul. Testplaani pannakse detailselt kirja kõik iteratsiooni käigus teostatavad testimise tegevused - milliseid funktsionaalseid ja mittefunktsionaalseid nõudeid testitakse ja milliseid osi testitakse staatiliselt. Iteratsiooni testplaanis koostatakse ja uuendatakse automaatteste, uuendatakse dokumentatsiooni jne. Testplaani sisendiks on iteratsiooni käigus arenduses olevad tööülesanded. Testplaanis planeeritakse regressioontestimisele vajalik aeg. Testplaani iga rea kohta hinnatakse riskid (nii projekti- kui ka tooteriskid), määratakse testimise põhjalikkus, testimise meetod, vastutav testija, hinnatakse testimisele kuluvat aega ja tehakse kindlaks tööülesannete vahel olevad seosed. Lisas 4 on esitatud iteratsiooni testplaani näide.

Testimise alustamise kriteeriumid (iteratsiooni kohta):

- eelmise iteratsiooni testimine on lõpetatud (kui selline oli);
- iteratsiooni testplaan on koostatud;

- tööülesannetele on määratud vastutavad testijad;
- mõned spetsifikatsioonid on valmis (oluline staatilise testimise jaoks);
- mõned tööülesanded on arendaja poolt teostatud (oluline dünaamilise testimise jaoks).

Testimise lõpetamise kriteeriumid (iteratsiooni kohta):

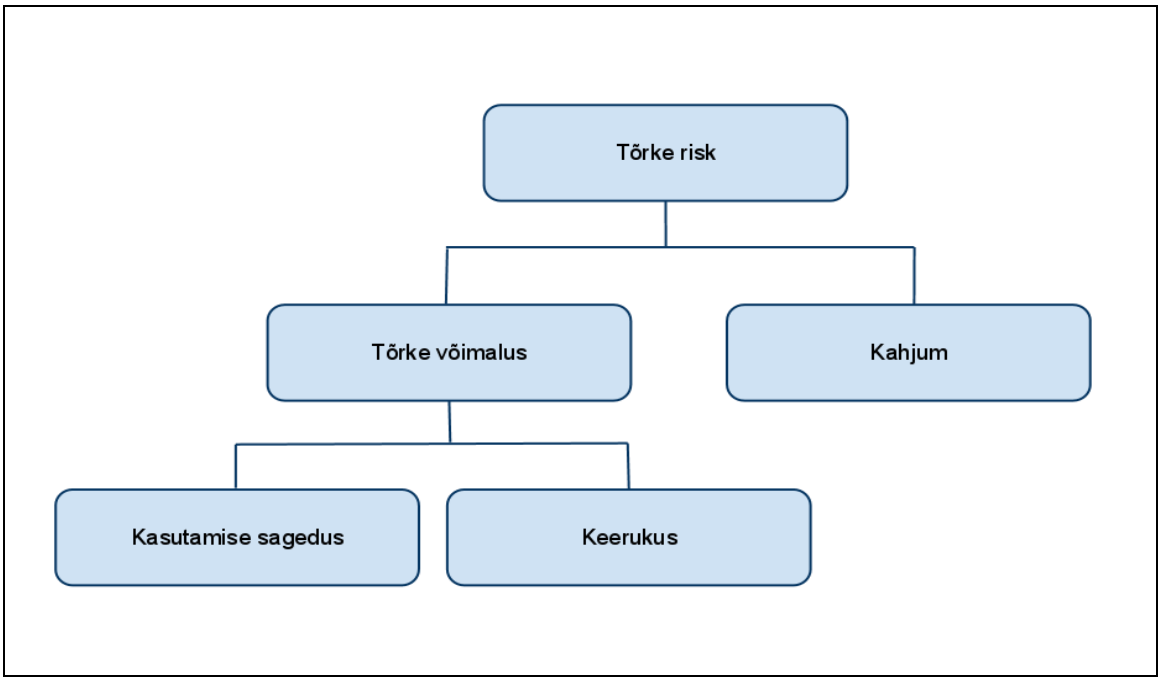
- iteratsiooni testplaanis hinnatud riskid piisavalt maandatud;
- kõik iteratsiooni tööülesanded on suletud või planeeritud järgmisse tsüklisse;
- projektijuht otsustab, et rakendus on valmis tarnimiseks.

4.2.5 Riskide analüüs

Riskide analüüsiks nimetatakse testimise protsessi, mille käigus tehakse kindlaks toote riskid ja pannakse paika testimise prioriteedid.

Valmiva funktsionaalsuse võimalike tõrgete riski hindamiseks kasutatakse toote riskianalüüsi (p*roduct risk analysis* (*PRA*)) mudelit. Toote riskianalüüs on üks analüüsi strateegiast, mida tavaliselt kasutatakse teaduses ja tarkvaraarenduse tööstuses. Toote riskianalüüs (*PRA*) on analüüsiprotsess, mille jooksul testjuhi ja teiste osapoolte vahel saavutatakse kokkulepe, millised toote kvaliteedi omadused või toote komponendid on rohkem riskantsed, millised vähem ning kui põhjalikult tuleb testida neid komponente. *PRA* põhiküsimuseks on, millised on riskid kliendi jaoks, kui tootes puudub kindel oodatud omadus. *PRA* tulemuseks on dokument, mille alusel saab hiljem otsustada, milliseid tooteobjekte (komponente) ja toote kvaliteedi omadusi võib testida pinnapealselt ja milliseid peab testima põhjalikult. *PRA* põhimõte on „pole riski, pole testi“ [27].

Toote risk on rakenduse tõrke võimaluse seos oodatud kahjumiga vea ilmnemisel. Tõrke võimalus määratakse funktsionaalsuse kasutamise sagedusega ja keerukusega [28]. Joonisel 5 esitatud riskide hindamise diagramm:



Joonis 5. Riskide hindamise diagramm.

Funktsionaalsuse keerukuse kohta päritakse infot arhitektilt, arendajatelt ja andmebaasi spetsialistidelt – tehnilise lahenduse keerukuse kohta aga projektijuhilt ning äriloogika keerukuse kohta analüütikutelt.

Kahjumiks võib olla:

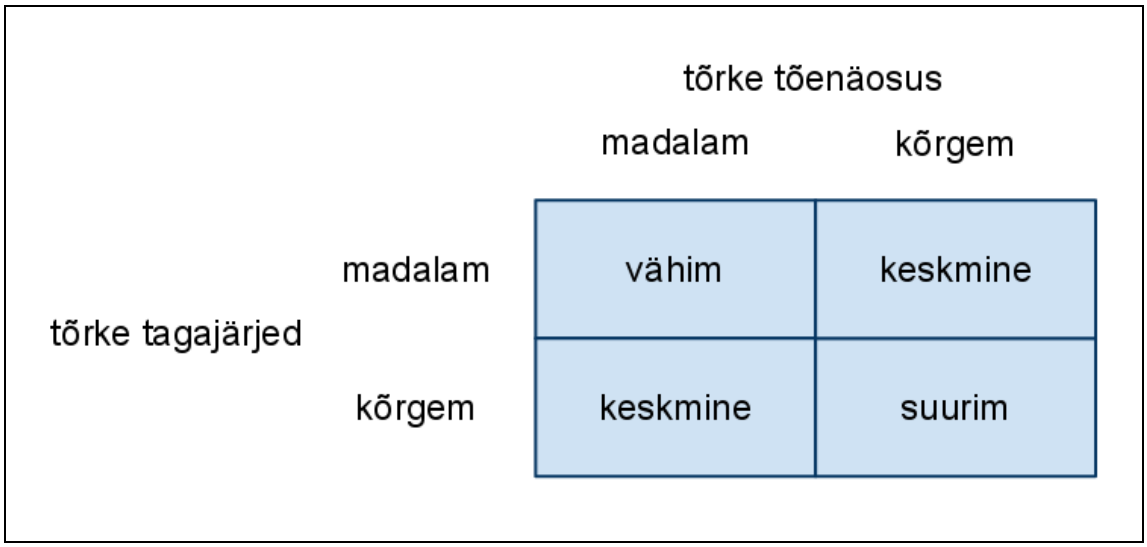
- tulu kaotamine;
- kolmandate osapoolte kahjum;
- keskkonna ja füüsiline kahju;
- paranduste maksumus;
- usalduse kadumine, maine rikkumine;
- kasutajatoe ülekoormus.

Riskifaktorid, mida peab meeles pidama, on:

- keerukas funktsionaalsus;
- täiesti uus funktsionaalsus;
- kõrge ajapinge;
- kogenematud arendajad;
- uued (tundmatud) arendusvahendid ja keskkonnad;
- suured meeskonnad;
- halb suhtlus (kas kliendiga või/ja meeskonna sees);

- funktsioonid erinevate liidestega;
- ebaadekvaatne kvaliteedijuhtimine arenduse tsüklis;
- ebaadekvaatsed automaattestid;
- süsteem on suureks kasvanud – vajalik põhjalik regressioontestimine.

Testimine annab informatsiooni, mis võib võimaldada minimeerida riske. Testimise mahud sõltuvad tõrgete tagajärgedest ja tõenäosusest, mis on näidatud Joonisel 6 esitatud tabelis [18]:



Joonis 6. Testimise mahtude sõltuvus tõrgete tagajärgedest ja tõenäosusest.

4.3 Mittefunktsionaalsete nõuete testimine

Mittefunktsionaalsete nõuete testimine on osa süsteemi testimisest. Mittefunktsionaalsed nõuded võivad olla erinevad ja mõned on raskesti testitavad. Tabelis 1 on esitatud erinevad mittefunktsionaalsete nõuete tüübid selgitustega [29].

Tüüp	**Selgitus**
Ligipääsetavus	Rakenduse võime pääseda ligi rakenduse funktsionaalsusele.
Audit ja kontroll	Rakenduse võime kergesti üle vaadata ajaloolist töövoogu ja kontrolljälge.
Käideldavus	Rakenduse käideldavus vastab teenusetaseme lepingul (SLA) sätestatule.
Ühilduvus	Rakendus sobib olemasoleva (vanema) keskkonnaga.
Dokumentatsioon	Rakenduse kasutusjuhendid annavad õigeid juhiseid.
Paigaldamine	Rakendus töötab ettekirjutatud raamistike ja versioonidega.
Koostalitlusvõime	Rakendus töötab pärast olulise komponendi muutmist.
Kõide	Rakendus suudab töödelda nõutud tehingute arvu antud ajahetkel.
Hooldatavus	Rakendust on lihtne hallata peale toodangusse paigaldamist.
Jõudlus	Rakenduse kriteeriumid nagu reaktsiooniaeg, jõudlus, kokkulangevus jne vastavad nõuetele.
Töökindlus	Rakendus võib töötada koormatud toodangu keskkonnas.
Skaleeritavus	Rakendus võib rahuldada kasvavaid ärivajadusi.
Turvalisus	Rakendus on piisavalt turvaline, et kaitsta informatsiooni vargust.
Hooldatavus	Rakenduse käitumise silumine ei avalda mõju kliendi ärile.
Kasutatavus	Rakendus on kasutatav lõpptarbija seisukohast.

Tabel 1. Näited mittefunktsionaalsetest nõuetest.

Teoreetiliselt saab testida kõiki neid nõuded. See on keeruline, kuid teostatav. Testijad peavad minimaalselt kontrollima järgmisi süsteemi mittefunktsionaalseid omadusi:

- süsteemi jõudlus;
- süsteemi kasutatavus.

Lisaks, kui tegemist on avaliku rakendusega, mida saavad interneti abil kasutada kõik soovijad, siis tuleb kindlasti testida ka brauseri ühilduvust (*browser compatibility)* - kas süsteem näeb välja ja toimib identselt erinevate veebilehitsejate puhul. Testimise aluseks võetakse värske brauserite kasutuse statistika ja süsteemi testitakse kõige levinumate brauseritega. Seda saab teha nii käsitsi kui ka automaatselt erinevaid vahendeid kasutades:

- Janitor (WM enda poolt loodud tarkvara);
- Page speed [30];
- Browser shots [31].

Testimisel lähtutakse süsteemi nõuetest – otsustakse (veel testimise planeerimise faasis), mis tüüpi testide käivitamine on vajalik; ja kui on, siis milliste parameetritega.

4.3.1 Jõudluse testimine (*performance testing*)

Jõudluse testimist on võimalik jagada koormuse ja stressi testimiseks. Need teostatakse testija või arendaja poolt testplaani järgi.

Koormustestide sisenditeks on järgmised parameetrid:

- andmete maht;
- kasutajate arv;
- päringute tegemise sagedus;
- nõutav/oodatav rakenduse kiirus;
- funktsionaalsuse (komponentide) kasutatavuse sagedus;
- kasutatava platvormi (riistvara ja tarkvara) kirjeldus.

Koormuse testimise põhiliseks eesmärgiks on verifitseerida, et rakendus vastab nõuetele. Näiteks on nõudes kirjas, et kui 100 paralleelset kasutajat teostavad otsinguid viie erineva andmebaasi tabelitest iga kolme sekundi järel, kusjuures summaarne andmemaht on 1Gb, siis süsteemi komponent (‘otsing x’) peab tagastama otsigu tulemusi maksimaalselt 2 sekundi jooksul. Sellisel juhul kontrollitakse spetsiaalse koormuse testimise vahendi abil, kas süsteem vastab esitatud nõuetele või mitte.

Stressi testimise käigus uuritakse süsteemi võimsust ja proovitakse leida süsteemi võimsuse piire (ehk otsitakse, millise koormusega süsteem ei saa hakkama ja variseb kokku). Lisaks kontrollitakse, millisel viisil süsteem kokku variseb – kas ainult testitav süsteem ise või koos sellega ka rakenduse server, füüsiline masin, andmebaas ja mis juhtub andmetega, jne. Testimiseks koormatakse süsteemi aeglaselt ja jälgitakse rakenduse käitumist. Teades rakenduse jõudluspiire, on võimalik operatiivselt toimida süsteemi ressursside lõppemisel.

4.3.2 Kasutatavuse testimine (*usability testing*)

Kasutatavus näitab, mil määral „kindlaksmääratud kasutajad saavad toodet kasutada kindlaksmääratud eesmärkide saavutamiseks efektiivselt, tulemuslikult ja rahuldustpakkuvalt kindlaksmääratud kasutuskontekstis“ [32,33]. Kasutatavuse testimise

käigus pannakse tähele just neid aspekte. Kasutatavuse testimisel on eesmärk teha kindlaks, mil määral toode on arusaadav, lihtne õppimiseks ja atraktiivne kasutajatele [34].

Kasutatavuse testimise protsess kujutab endast peamiste ülesannete, mille jaoks see toode on mõeldud, lahendamist süsteemi potentsiaalse kasutaja poolt. Kasutaja tegevusi jälgitakse kasutatavuse spetsialisti poolt ja palutakse kasutajal kommenteerida oma tegevusi nende katsete täitmise ajal. Selleks, et saada adekvaatseid testimise tulemusi, katsetatakse sama protsessi erinevate kasutajate peal - mitu esindajat iga potentsiaalse kasutajate grupi kohta. Testimise tulemusi analüüsitakse ja selle alusel teostatakse vastavad parandused kasutajaliideses või isegi süsteemi loogikas.

Mõningal määral tegeldakse kasutatavuse testimisega iga testimise käigus. Testija peab ennast alati panema süsteemi kasutaja rolli ja pöörama tähelepanu toote kasutatavusele.

5. Testimise protsessi ja progressi jälgimine

Testimine on aega ja inimressurssi vajav protsess. Testimine on samas ka väga mahukas protsess, millega tihti tegeleb testija mitte üksinda, vaid koostöös teiste testijatega.

Testimise jälgimiseks kasutatakse:

- testplaane;
- rakenduse protsesside nimekirja;
- tööülesannete/vigade haldussüsteemi;
- meetrikaid.

Testimise protsessi ja progressi jälgimise vahendeid kasutavad nii testijad kui ka projektijuht. Teoreetiliselt võiks seda kasutada ka tellija, et saada pilti toimuvast. Kõik testimise progressi jälgimise vahendid on kasulikud vaid juhul, kui neid jooksvalt uuendatakse ja vajadusel korrigeeritakse.

Testimise protsessi jälgimine võimaldab igal ajahetkel näha, kui kaugel ollakse testimisega - kui palju on tehtud ja mida on vaja veel teha. Samas jooksva situatsiooni jälgimine annab piisavalt informatsiooni, et võimalikult vara teada saada kriitilisest olukorrast, et jääks aega selle kõrvaldamiseks.

5.1 Kliendipoolne valideerimine

Kliendipoolne valideerimine on protsess, mille käigus kontrollitakse, kas süsteem, rakendus või toode, vastab ärinõuetele. Selline protsess on kohustuslik osa arenduse ja testimise protsessis. See võib toimuda kolmel tasemel:

1. spetsifikatsioonid;
2. prototüübid;
3. rakendus.

Spetsifikatsioonide ja prototüüpide tasemel on valideerimise eesmärkideks saada teostamiseks õige ülesanne, maksimaalselt maandada vigu ja vähendada probleemide tekkimise võimalusi. Rakenduse tasemel valideerimine peab garanteerima, et klient saab endale funktsionaalsuse, mis sobib olemasoleva protsessiga, lihtsustab tellija tööd ja toob lisaväärtust.

Tellijapoolse testimise käigus klient raporteerib vigadest. Need vead tulevad mitte ainult vigasest ja mittetöötavast funktsionaalsusest, vaid ka ebapiisavast analüüsist. Analüüsi tasemel probleemide tekkimise põhjusteks on rakenduse keerukus – tihti pole võimalik tootjal (arendajal) ega tellijal saada „suurt pilti“ enne kui teatud funktsionaalsus on valmis arendatud.

Kliendipoolne valideerimine tagab hea kvaliteediga toote, kuna klient teavitab kiiresti, juhul kui rakenduses on midagi puudu või valesti. Samas teeb tellija omapoolsed ettepanekud rakenduse parendamiseks, täiendamiseks ja arendamiseks. Arendaja seisukohast on tellijapoolne valideerimine riskide maandamise üks meetodeid ja sellest tulenevalt on parim praktika tellija tihe kaasamine arendusprotsessi. Webmedia üritab hoida oma arendusprotsessi tellijale läbipaistvana, et tellijal oleks hea ülevaade protsessist, progressist ja valmivast funktsionaalsusest. Seda saavutatakse tihedate tarnete abil. Tellijale tarnitakse uut ja uuendatud funktsionaalsust iga iteratsiooni lõpus – keskmiselt kord kuus.

6. Testjuhtumid

Testjuhtum on tegevuse kirjeldus, sellega kaasnevate tingimuste ja muutujate hulk, mille alusel kontrollitakse, kas rakendus või tarkvara toimib ootuspäraselt või mitte ja milles kirjeldatakse, kuidas test läbi viiakse.

Testjuhtum peab sisaldama järgmiseid kohustuslikke punkte [35]:

- testjuhtumi identifikaator;
- testjuhtumi nimetus;
- vajalikud sammud ettevalmistuseks;
- tegevuste kirjeldus;
- testi sisendid/väljundid;
- testi läbimise/mitteläbimise kriteeriumid.

6.1 Testjuhtumite koostamise vajadus

Testijad kasutavad testjuhtumeid verifitseerimiseks, kuna need võimaldavad kontrollida, kas mingi kindel funktsionaalsus töötab korrektselt.

Suurtes projektides kasutavad testijad testjuhtumeid ka vanema funktsionaalsuse testimiseks, kuna ei ole võimalik meeles pidada kogu projekti funktsionaalsust ja käitumist. Lihtsam on vajadusel ette võtta vastav testjuhtum ja värskendada seda oma mälus.

Testjuhtumeid kirjutatakse ka kliendi jaoks. Näiteks, klient võib kasutada testjuhtumeid endale kasutusjuhendite kirjutamiseks. Klient võib kasutada testjuhtumeid ka rakenduse valideerimiseks, milleks nad väga hästi sobivad. Kui testjuhtumid koostatakse ja saadetakse kliendile piisavalt vara, siis on kliendil võimalik enne funktsionaalsuse arendamist muuta oma nägemust funktsionaalsuse kohta, kohanda tellimust ja teha teisi tähtsaid ettepanekuid.

Juhul, kui klient osaleb testimise protsessis, siis testjuhtumid annavad tootjale (arendajale) võimaluse juhtida klienttestimist vajaliku suunda. Näiteks, kui tekivad mingid kahtlused kasutusmugavuses, siis testjuhtumite abil on võimalik kliendi tähelepanu pöörata kindlale süsteemi osale.

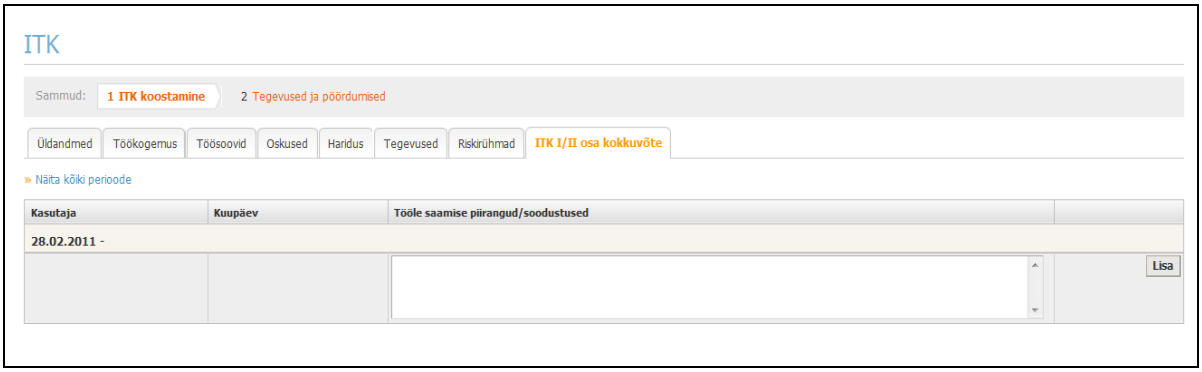
Testjuhtumitel on ka lisaväärtus. Tavaliselt on testijad ainsad inimesed, kes tunnevad hästi kogu rakendust ja seepärast väga tihti teised meeskonnaliikmed küsivad testijate käest, kuidas mingi komponent töötab. Meeskonnale kättesaadavad testlood hoiavad kokku testijate aega küsimustele vastamisel.

Arendajad kasutavad testjuhtumeid automaattestide kirjutamiseks.

6.1.2 Näide testjuhtumi kasutamisest

Allpool on toodud näide Töötukassa infosüsteemist „EMPIS“ [36]. Näidiseks on toodud kogu tsükkel testitava funktsionaalsuse valmimisest kuni automaattesti kirjutamiseni.

Joonisel 7 on esitatud valitud rakenduse kuva väljaga, mida on vaja testida.



Joonis 7. Rakenduse kuva testimiseks.

Antud funktsionaalsus sai valitud sellepärast, et see on süsteemi kasutajate jaoks oluline funktsionaalsus, seda kasutatakse piisavalt tihti ja tõenäoliselt tulevikus ei muutu, mis tähendab, et see funktsionaalsus peab olema kaetud automaattestiga. Selleks on vaja koostada testjuhtum, kus testitavad protsessid on „kokkuvõtte lisamine ja muutmine“:

Testjuhtum №1 – KokkuvotteLisamineJaMuutmineToootsijanaArvelOlevIsik

Vajalikud ettevalmistuse sammud: isik peab olema lisatud süsteemi

Tegevus	**Oodatav tulemus**
Isiku otsingus valitakse töötuna arvel oleva isik.	Avatakse valitud isiku kontekst.
Vasakpoolses menüüs valitakse menüüpunkt ’Detailandmed’.	Avatakse isiku detailandmete kuva. Vaikimisi avatud ’Üldandmete’ sakk.
Valitakse sakk’ITK I/II osa kokkuvõte’.	Avatakse sisestatud kokkuvõtete nimekiri.
Vajutatakse nuppu ’Lisa’	Kuva jääb samaks ja lisatakse järgmised veateated: Väli 'Tööle saamise piirangud/soodustused' on kohustuslik.
Vorm 'Tööle saamise piirangud/soodustused' täidetakse järgmiselt: • Sisestatakse tekst „Tööle saamise piirangute ja soodustuste informatsioon puudub“. Vajutatakse nuppu ’Lisa’.	Tekib uus link ’Muuda’.
Vajutatakse linki „Muuda“ eelmises punktis testi käigus lisatud kirje juures.	Avatakse kokkuvõtte muutmise komponent ja ilmuvad lingid Salvesta/Katkesta.
Välja muudetakse järgmiselt: • „Tööle saamise piiranguid ei paista olevat". Vajutatakse „Salvesta“ linki.	Nimekirjas kuvatakse muudetud kokkuvõtte kirje.

Testi läbimise kriteeriumid: test on läbitud, kui kõik oodatavad tulemused on saavutatud.

Testjuhtumi põhjal kirjutatakse Selenium [37] automaattest:

```
public class KokkuvoteLisamineMuutmine extends
ee.tootukassa.empis.SeleniumBaseSingleTest {

  @Override

  protected void testMain() {

    getSelenium().type("m.f0.menu.f0.c.f0.form.legalCode","37810011757");

    getSelenium().click("search");

    getSelenium().click("link=Dylan, Bob");

    getSelenium().waitForPageToLoad("30000");

    getSelenium().click("link=Detailandmed");

    getSelenium().waitForPageToLoad("30000");

    getSelenium().click("//span[@id='tabCountRgn']/ul/li[8]/a/span");

      getSelenium().type("m.f0.menu.f1.c.f0.sidemenu.f1.menuWidget.f0.lis
      t.formList.addForm.kokkuvote", "Tööle saamise piirangute ja
      soodustuste informatsioon puudub");

      getSelenium().click("//input[@value='Lisa']");

      assertTrue(getSelenium().isTextPresent("Tööle saamise piirangute ja
      soodustuste informatsioon puudub"));

      getSelenium().click("m.f0.menu.f1.c.f0.sidemenu.f1.menuWidget.f0.li
      st.formList.rowForm2.editSave");

      getSelenium().type("m.f0.menu.f1.c.f0.sidemenu.f1.menuWidget.f0.lis
      t.formList.rowForm2.kokkuvote", "Tööle saamise piirangud ei paista
      olevat");

      getSelenium().click("m.f0.menu.f1.c.f0.sidemenu.f1.menuWidget.f0.li
      st.formList.rowForm2.save");

      assertTrue(getSelenium().isTextPresent("Tööle saamise piirangud ei
      paista olevat"));

      }
```

6.2 Testjuhtumi koostamine

Testjuhtumite kirjutamiseks on vaja selgeks teha, kelle jaoks neid koostatakse. Kui testjuhtum on mõeldud teiste testijate jaoks, kes tunnevad projekti hästi, siis juhtum võiks olla lihtne ning mitte üle koormatud informatsiooniga. Näiteks, iga testija saab ise testandmeid genereerida, aga arendajate jaoks tuleb kindlasti juhtum täpsemalt kirjeldada -

tegevuste käik koos testandmetega. Teisisõnu, testjuhtumite detailsus võib varieeruda lähtuvalt juhtumite koostamise eesmärgist. Üks näide detailsest testjuhtumist on toodud Lisas 5.

Kogu rakenduse funktsionaalsuse katmine testjuhtumitega on väga töömahukas, seetõttu on vaja läbi mõelda ja valida välja need osad, mis vajavad katmist. Näiteks, see funktsionaalsus, mille peale hakatakse kirjutama automaatteste, peab kindlasti olema kaetud testjuhtumitega, kuna arendajad kirjutavad teste neid kasutades.

Testjuhtumid tuleb kindlasti koostada keerulise funktsionaalsuse korral, lihtne (triviaalne) funktsionaalsus ei vaja tavaliselt testjuhtumitega katmist. Samas pole ka mõtet koostada testjuhtumeid, mida ei hakata taaskasutama. Alati tuleb arvestada sellega, et testjuhtumite koostamine on ajamahukas töö ja tuleb kaaluda, kas konkreetse testjuhtumi kirjutamine oleks mõistlik antud projekti jaoks.

7. Testimine arendaja poolt

Enne kui testija saab uue funktsionaalsuse enda kätte, peab see olema arendaja poolt juba läbi testitud. Arendaja poolt teostatud testimine erineb testija poolt teostatud testimisest - arendaja tavaliselt ei veendu selles, kas terve rakendus töötab 100% korrektselt. Ta veendub, et tema poolt arendatud funktsionaalsusega saab töötada, sealhulgas ka seda testida.

Arendaja testimine on teostatav järgmiste meetoditega:

- komponenttestimine (*component testing*);
- integratsioontestimine (*integration testing*);
- ühiktestimine (*unit testing*);
- koodi staatiline analüüs;
- koodi ülevaatus.

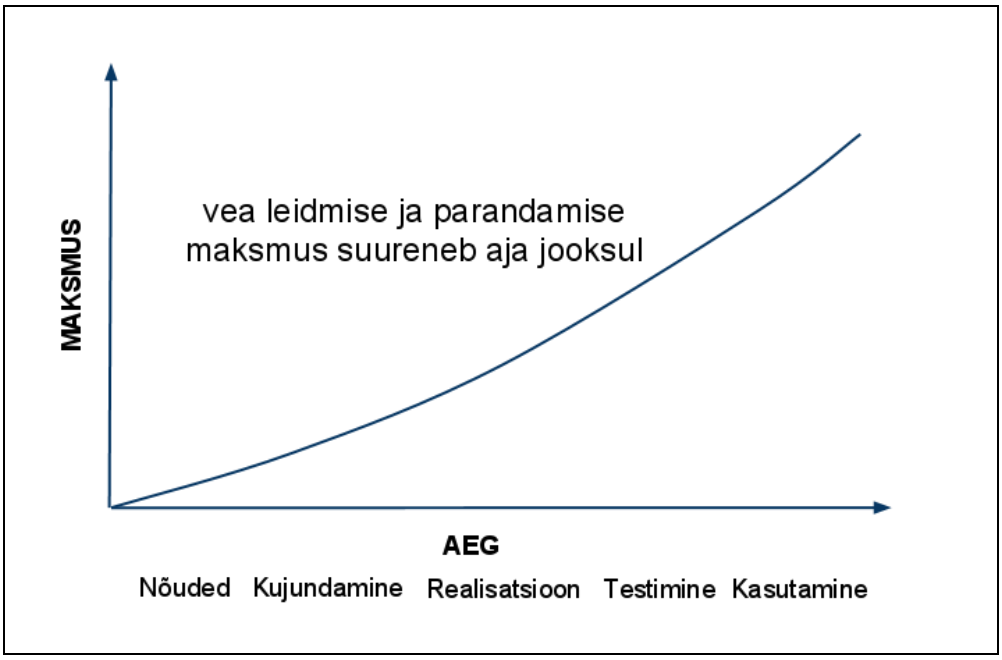
Samas tegeleb arendaja ka automaattestide kirjutamisega.

Kõige tähtsam eesmärk arendaja testimises on, et peale arendaja testimist ei tohi rakenduses esineda veaekraane.

7.1.1 Komponenttestimine

Ülesanded, mida täidab arendaja, on tavaliselt seotud kindla komponendiga rakenduses. Pole oluline, kas tegemist on uue funktsionaalsuse loomisega või vigade parandamisega, mingi osa rakendusest võib hakata käituma erinevalt varasemast.

Vea parandamise hind sõltub sellest, millises faasis (dokumentatsioon, arendus, arendaja poolne testimine, tellijapoolne testimine, toodang) seda avastatakse. Joonisel 8 on esitatud diagramm, mis näitab vea parandamise maksumuse sõltuvust ajast ja arendusprotsessi etapist [34].



Joonis 8. Vea parandamise hind.

Seetõttu oleks odavam, kui arendaja ise leiaks kõik vead. Reaalsuses pole see võimalik ega ka otstarbekas, kuid triviaalsemate vigade leidmiseks selline lähenemine töötab. Seega arendaja kontrollib peale oma lahenduse loomist, kas ülesande raames loodud komponentides on rahuldatud järgmised nõuded:

- komponent avaneb;
- lingid, nupud (kõik teised elemendid, mis kutsuvad esile mingit tegevust) töötavad ja ei anna veaekraani (see tähendab, et midagi, mis näeb välja nagu korrektne tegevus, toimub);
- rakendus töötleb korrektset sisendit ilma veata;
- rakendus töötleb mittekorrektset sisendit ilma veata.

Lisaks, kuna arendaja teab, kuidas funktsionaalsus on rakenduses realiseeritud, võib ta kontrollida juhtumit, milles käivitatakse kõige kahtlasem osa rakenduse koodist (*white-box testing*). See valik toimub arendaja sisetunde järgi ja piisava kogemusega arendaja saab sellega hakkama. Oluline on, et arendaja ei kontrolli kogu funktsionaalsust, ta lihtsalt teeb kindlaks, et tema poolt arendatud funktsionaalsus töötab ja “ei lähe katki”.

Kogu komponenttestimise eesmärk on kõrvaldada olukord, mil arendaja saadab testimisele väga toore lahenduse, mille tõttu hakkab arenduse – testimise tsükkel väga pikaks venima.

7.1.2 Integratsioontestimine

Rakendust arendatakse komponentide kaupa, mis lõpuks ühendatakse üheks tervikuks. Äriprotsessid nõuavad, et need komponendid töötaksid ja vahetaksid üksteisega andmeid. Integratsioontestimisega saab kontrollida, kuidas komponendid (või suuremad rakenduse osad, nt. paigaldatud rakendus ja andmebaas) koos töötavad.

Arendaja eesmärgiks ei ole ka siin veenduda, et funktsionaalsus töötab korrektselt, vaid integratsioonitesti kasutatakse selleks, et kontrollida, kas komponendid on kooskõlas - milliseid andmeid komponentide vahel vahetatakse. Näiteks, kui ühe komponendi initsialiseerimiseks on vaja kasutaja identifikaatorit, siis kontrollitakse, kas teised komponendid, mis selle komponendiga suhtlevad, annavad välja identifikaatori (*ID*) või mitte. Kui komponent asub mingis ärivoos ja neid komponente, mis peavad temaga suhtlema, pole veel valmis, siis asendatakse puuduvad komponendid mängukomponentidega: "*driver*". „*Driver*" asendab komponente, mis pöörduvad antud komponendi poole ja "*stub*" asendab komponente, mis asuvad ärivoos tagapool.

7.1.3 Ühiktestimine (*unit testing*)

Ühiktestimist kasutatakse selleks, et kontrollida, kas rakenduse kood käitub nii, nagu oodatakse. Kuid kontrollida, et mingi kood käitub korrektselt, ei ole lihtne - arvuti saab teha ainult seda, mida talle on öeldud.

Ühiktest koosneb kolmest osast:

- etteantud sisend;
- funktsionaalsuse väljakutse;
- väljundi kontroll.

Ühiktesti loomiseks valib arendaja mingi osa funktsionaalsusest, mida ta asub kontrollima. Seejärel luuakse etteantud sisend ja oodatud väljund. Edasi kontrollib test, kas funktsionaalsus antud sisendiga väljastab oodatava tulemuse.

Väga tähtis on, et üks ühiktest kontrolliks võimalikult väikest koodiosa. See lubab kontrollida käitumist iga sisendi juhul eraldi. Kui testitav koodiosa on suurem, siis on keerulisem luua sellist testide komplekti, mis hõlmaks igat koodikäsku ja kontrolliks, kas

see käitub nii, nagu oodatud. Ühiktestide põhimõte on selles, et arendaja kontrollib, kas kood töötab nii, nagu ta ette kujutab.

Sellised testid on väga kasulikud siis, kui keegi hakkab testidega kaetud koodi muutma. Siis on kiiresti võimalik teada saada, kui midagi läheb katki (sest testid ei tööta enam). Vastupidi, kui kood on muutunud, aga testid on korras (nn. rohelised), siis saab loota, et rakendus ei ole katki.

Ühiktestidega ei ole mõistlik katta kogu rakenduse koodi. Nagu testjuhtumite korral, on olemas kohad, mida võiks ühiktestidega katta peaaegu täielikult - need on:

- andmete ligipääsu kiht (*data access layer*) - mis suhtleb nt. andmebaasiga või veebiteenustega - siin saab kontrollida, näiteks, et *SQL* päringud on valiidsed (õigsus kontrollitud) või et andmed, mis saabusid andmebaasist, sisestatakse objektidesse korrektselt, jne;
- taaskasutatavad kasulikud meetodid (*utility methods*) - näiteks, kui rakenduses on olemas kood, mis arvutab tuletist, siis seda on lihtne välja kutsuda erinevate funktsioonidega ja kontrollida, kas ta vastab õigesti;
- ärikihi loogika meetodid - võivad olla osaliselt kaetud ühiktestidega, sõltuvalt keerukusest ja suurusest (juhul kui nad on väga suured, siis on neid ühiktestidega testida väga keeruline).

Ühiktestid on koodiga samaväärsed osad ja asuvad tavaliselt rakenduse koodi kõrval versioneerimise süsteemis. Ühikteste soovitatakse käivitada võimalikult tihti - arendaja jaoks iga 5 minuti pärast. Aga isegi siis, kui neid ei käivitata nii tihti, on need väga kasulikud koodi regressioontestimiseks.

7.1.4 Koodi staatiline analüüs

Staatilise analüüsi käigus koodi ei käivitata. Analüüsiks kasutatakse rakendusi ja raamistikke (*frameworks*), mis loevad koodi ja annavad hoiatusi, kus vead võivad esineda. Näiteks:

- raamistik näeb, et koodis toimub jagamine nulliga (operatsioon, mis ei ole tavaliselt lubatud), siis on peaaegu kindel, et tulevikus selles kohas tekib rakenduses viga;

- raamistik loeb koodist, et rakendus hakkab mingil hetkel 1000 korda baasist andmeid lugema ("*for*" tsükli sees näiteks); siis on selge, et siin tõenäoliselt võivad tekkida probleemid jõudlusega.

Üldjuhul need raamistikud annavad võimaluse koostada reeglite sõnastikku, kus on kirjas, mida nad peavad koodist otsima.

7.1.4.1 Koodi ülevaatus

Veel üks tegevus, millega tegelevad arendajad koodi kõrge kvaliteedi tagamiseks, on projekti teiste arendajate koodi ülevaatus. Koodi ülevaatus on staatilise testimise meetod.

Koodi ülevaatusel on mitu põhilist eesmärki:

- kõrvaldada ilmsed vead koodis;
- võimalikult vara leida ning ümber kirjutada ebakvaliteetne kood: enne kui arendaja hakkab teiste ülesannetega tegelema või kui halba koodi hakatakse kasutama teistes kohtades;
- kõrvaldada koodi dubleerimine;
- projekti arhitektuuri raamistikust ja arenduse reeglitest kinnipidamise jälgimine;
- tutvustada teist arendajat uue koodiga;
- ülevaatuse protsessis osalejate väljaõpe.

Koodi ülevaatust soovitatakse teha enne muudatuse integreerimist, aga see on kasulik igal etapil.

7.1.5 Automaattestide kirjutamine

Lisaks sellele, et on võimalik koodi lugeda ja veenduda, et ta töötab, on oluline osa testimisest automaattestimisel. Automaattestimine on protsess, mil testide käivitamiseks kasutatakse spetsiaalset tarkvara tööriista (nt. Selenium). Lisaks sellele tavaliselt seatakse testide eeltingimusi ja raporteerimise võimalusi, kontrollitakse testide käitumist, võrreldakse oodatud ja tegelikke tulemusi [38]. Automaattestimine lubab testimise aega kokku hoida ja testimise protsessi lihtsustada.

Automaattestid suhtlevad rakendusega kasutajaliidese kaudu ja imiteerivad kasutaja tegevusi. Tavaliselt koostatakse automaatteste mingi teegi abil, mis töötab rakenduse formaadiga: kui rakenduseks on veebileht - siis millegagi, mis oskab veebilehte avada ja

lugeda. Automaattestid kontrollivad, kuidas rakendus reageerib etteantud tegevustele: näiteks, peale rakenduse käivitamist peaks olema nähtav sisselogimise vorm. Kui selle vormi sisse kirjutada korrektne kasutajanimi ja parool ja vajutada nuppu “logi sisse” - peab tekkima kasutajale tervitus.

Automaatteste kirjutatakse testjuhtumite põhjal ja tavaliselt ei kaeta nendega kogu funktsionaalsust - see on väga ajamahukas. Nendega saab kontrollida, et olulisemad asjad on omal kohal ja reageerivad klikkidele jne. Nii nagu ühiktestidki, on ka automaattestid väga kasulikud regressioontestimise faasis.

8. Vigade raporteerimine

Vigade raportid on oluline ja kõige silmapaistvam osa testijate tööst. Võib ka öelda, et vearaport on testimise vahekokkuvõte või tulemus. Hea vearaport hoiab palju aega kokku mitte ainult arendajale, vaid ka testijale.

Vearaport on üks viisidest teavitada arendajat veast tema töös. Raport peab olema selge, lakooniline, kuid piisavalt detailne vea reprodutseerimise jaoks, kasutatav keel peab olema arusaadav, viisakas ja erapooletu [39]. Arendajad tagastavad tihti vearaporti, kui nad ei saa viga reprodutseerida või nad ei saa aru, mis on raportis kirjutatud. Korralikult kirjutatud vigade raportid vähendavad arendajate poolt tagastavate vearaportite arvu ja kiirendavad vigade parandamist. Võib juhtuda, et viga on raporteeritud ühe testija poolt, aga järelkontrolli hakkab teostama teine testija, kes peab samuti veast aru saama.

Tuleb arvestada ka sellega, et testija kirjutab raportid mingil määral enda jaoks - kui viga parandatakse peale pika aja möödumist, siis ei saa kindel olla, et sammud vea kordamiseks ning vea sisu veel meeles on.

Vearaport peab sisaldama järgmisi andmeid [40]:

- **Pealkiri**. Pealkiri peab olema lühike ja sisukas. Pealkirja eesmärk on anda ülevaade veast ja kohast, kus viga tekib. Näiteks, pealkiri “Nupp ei tööta” ei anna mingit sisulist informatsiooni, kuid ta on väga lühike. Samas pealkiri “Töösoovi lisamisel “Salvesta” nupp ei tööta” on piisavalt informatiivne ja samas ka lühike. Hea pealkiri võib aidata hoida kokku teiste meeskonnaliikmete aega. Näiteks, kui arendaja vaatab oma tööülesannete nimekirja ja näeb, et tal on mitu ülesannet seotud sama komponendiga, siis ta saab neid teha koos, mitte hüpates ühelt komponendilt teisele.
- **Kirjeldus**. Kirjeldus on kõige olulisem ja sisulisem raporti osa, mis peab sisaldama konkreetseid samme vea reprodutseerimiseks ja kui on teada, siis ka vea tekitamise põhjust.
- **Prioriteet**. Prioriteet määrab, kui tõsine viga on ja kui kiiresti seda tuleb parandada. Kui raport omab prioriteeti number üks - see tähendab, et viga on kriitiline ja seda on vaja viivitamatult parandada. Tavaliselt prioriteedi number üks saavad süsteemi

kasutajate poolt raporteeritud vead (*live*-keskkonnast) või need, mis oluliselt takistavad testimist.

Raport peab tingimata sisaldama testitava süsteemi versiooni numbrit ja testimiskeskkonna nimetust - siis on teada, millises iteratsioonis tuleb viga parandada ja hiljem parandusi testida; samas kuna viga võib sõltuda keskkonna seadistustest, siis on teada, millises keskkonnas on vaja kontrollida. Peale kohustuslike aspektide lisatakse mõnikord tööülesande juurde erinevaid lisasid. Väga kasulik on lisada rapordi juurde manusena pilte (nt. kui komponendi kujundus on katki), videot ja veaekraanide tekste (*exception stacktrace*). Mõnikord, eriti keerulise juhtumi korral, soovitatakse arendaja jaoks ette valmistada vea kordamiseks vajalike andmete seis. Olukorras, mil viga on raske või võimatu korrata, peab kindlasti märkima vea tekke põhjuse kirjeldus, vaatamata sellele, et kirjeldus ei ole eriti veenev. Lisaks saab raportisse panna väljavõtte rakenduse logidest. Logid on väga olulised vigade põhjuste tuvastamisel.

Tavaliselt vearaporti elutsükkel järgib mustreid. Lisas 6 on toodud vearaporti/tööülesande elutsükli joonis. Peale vearaporti loomist analüüsitakse seda ja seatakse vastav prioriteet. Kui tehakse otsus vea parandamiseks, siis raport suunatakse vastutavale arendajale, kes omalt poolt viga uurib ja juhul kui see on tõesti viga – parandab selle. Vea parandamisele järgneb vea suunamine tagasi testimisele ja selle verifitseerimine. Kui viga on parandatud, siis raport pannakse kinni.

Vearaportitest luuakse kas uued tööülesanded või lisatakse see juba olemasoleva tööülesande juurde ja suunatakse arendajale. Peale vea parandamist tagastatakse tööülesanne testijale. Selleks, et oleks arusaadav, millal tööülesanne on valmis kontrollimiseks ja millal kontrollimist on võimalik lõpetada, on testimise alustamise ja lõpetamise kriteeriumid.

Tööülesande testimise alustamise kriteeriumid:

- testimisega alustatakse üldjuhul siis, kui vastav tööülesanne on järelkontrolli oote staatuses;
- vastav funktsionaalsus on kaetud ühiktestidega (*unit test*) arendaja poolt;
- vastav funktsionaalsus on testitud arendaja poolt.

Tööülesande testimise lõpetamise kriteeriumid:

- funktsionaalsus on verifitseeritud spetsifikatsiooni järgi;
- avastatud vead on parandatud.

8.1 Testimise abivahendid

Testijad kasutavad erinevaid vahendeid oma töö lihtsustamiseks, kiirendamiseks ja progressi jälgimiseks.

Pildid

Väga kasulik on vearaporti juurde panna pilte, mille abil saavad arendajad kiiremini aru, milles on viga. Pildi tegemiseks on võimalik kasutada standardseid vahendeid (nt. MS Paint), aga mugavam on kasutada vahendeid, millel on laiendatud võimalused.

Näiteks, üks sellistest vahenditest on „SnagIt" [41], kus on olemas pildile kommentaaride ja erinevate noolte lisamise võimalus. Selle plussiks on võimalus pildistada erinevaid kuva osasid, mis eelnevalt vajavad mingit tegevust või *scrolling windows*'i. Näiteks valikmenüüd (*input suggest, select menu*).

Video

Eriti keeruliste vigade korral on võimalik lindistada ekraanilt video ja panna see vearaporti juurde. Üheks võimaluseks on programm „Extra Screen Capture Free" [42], mis on tasuta ja võimaldab salvestada heliga videot.

Video sobib hästi kasutatavuse testimiseks, kus kasutaja tegevused salvestatakse ekraanilt ja tema kommentaarid jäädvustatakse mikrofoniga.

Andmete genereerimine

Süsteemi sisestusväljadele tekstiliste ja numbriliste parameetrite sisestamisel on kindlasti vaja kontrollida väljade suurust ja sisendite valideerimist. Selleks eksisteerivad erinevad generaatorid, mille abil on võimalik genereerida vajaliku pikkusega ridu, mis sisaldavad numbreid ja tekste koos suurte ja väikeste tähtedega (nt. Random String Generator [43]). Selle plussiks on võimalus genereerida unikaalseid ja mitteunikaalseid ridu.

Isikukoodi genereerimine

Erinevate Eesti süsteemide testimiseks on tihti vaja kasutada palju isikukoode (nt. kasutajate loomiseks). Isikukood vastab kindlatele reeglitele [44]. Etteantud soo ja sünniajaga sobivate isikukoodide genereerimiseks on olemas Joosep-Georg Järvemaa poolt loodud isikukoodi generaator [45].

9. Testimiskeskkonnad

Mugava, objektiivse ja produktiivse testimise jaoks on vaja testimise tegevustest ja eesmärkidest lähtudes hoida mitut testkeskkonda, kuhu on võimalik paigaldada vajalikud versioonid. Näiteks:

- Ühes keskkonnas võib olla versioon, mis on praegu kliendil kasutusel. See on vajalik selleks, et juhul kui *live*-rakenduses tekib viga, oleks võimalik sama seisu korrata ja kiirendada probleemi lahendust.
- Teises keskkonnas peab olema versioon, kus toimub käesoleva iteratsiooni testimine. Seda keskkonda tuleb pidevalt uuendada - alati võiks olla viimane versioon.
- Kolmas keskkond on tavaliselt automaattestide jaoks. Kuna automaattestid nõuavad kindlat andmete seisu, siis testidele õigete vastuste saamiseks on vaja rangelt keelata andmete muutmist andmebaasi skeemis, mida automaattestid kasutavad.
- Tihti tekib vajadus erinevate liideste testimise jaoks luua lisaks uusi testkeskkondi;
- Väga kasulik on hoida ka kliendi jaoks veel üks keskkond, mille seadistused vastaksid 100% keskkonnale, kus töötab kliendirakendus. See keskkond on mõeldud kliendile selleks, et ta saaks üle vaadata uut funktsionaalsust ja õigeaegselt teha omapoolseid ettepanekuid ja märkusi. Sellesse keskkonda uue versiooni üles panemiseks peab ta olema juba eelnevalt läbi testitud: klient ei tohi näha veaekraane!

Kõik testkeskkonnad peavad olema kirjeldatud kasutades *QA* strateegiat. Seda infot peab hoidma aktuaalsena, st aja jooksul tekkinud muudatused peavad olema dokumendis kajastatud. Selle jaoks, et meeskonnaliikmete tööd lihtsustada, tuleb iga keskkonna juures märkida ka sisselogimise info (nt. kasutajatunnus ja parool).

Testkeskkonda paigaldatakse ainult repositooriumi salvestatud kood – jooksvalt repositooriumi salvestamata koodiga ei tohi testkeskkonna seisu muuta. Ainult selliselt saab testkeskkonna adekvaatset seisu tagada ja olla kindel, et kliendi testkeskkonda ja toodangkeskkonda läheb süsteem samas seisus.

Testkeskkonna uuendamine kuulub testija kohustuste hulka. Testkeskkonna uuendamise juhendid peavad olema koostatud projekti testijate poolt ja operatiivselt uuendatud

muudatuste korral. Nagu teised testimisega seotud dokumendid, peab keskkonnade uuendamise dokument olema kõigile meeskonnaliikmetele kättesaadav. Dokument peab sisaldama nimekirja kõikidest keskkondadest, kus on kohustuslik määrata rakenduse veebiaadress, edasi *XML*, andmebaasi skeemi ja logide kausta nimi ning testkasutaja kasutajatunnus ja parool:

Arenduses olev versioon on kättesaadav: http://testimine.webmedia.int:1010/first-test/
Schema: test/ first-test
Build kaust: test/test/ first-test
Build xml: first-test.xml
Logid: tomcat/logs/ first-test.log
Testkasutaja: john.doe/test0987

10. Testijate väljaõppe

Kaasaegsed süsteemid on mahukad, keerulised (tegevusloogika ja tehnoloogia poolest) ja tulevikus muutuvad veel keerulisemaks, samal ajal muutuvad süsteemide valmimise tähtajad lühemaks. Seepärast peab testija alati arendama oma oskusi (tähelepanu, mälu, kriitilist suhtumist testimises, süsteemset mõtlemist) ja teadmisi. Testijate arendamise suurim osa on iseseisev õppimine ja enesearendamine, mida võib saavutada järgmiste tegevustega:

- raamatute ja ajakirjade lugemine (näidisnimekiri esitatud Lisas 7);
- internetiallikate jälgimine (näidisnimekiri esitatud Lisas 7);
- uue meetodite katsetamine;
- kogemuse jagamine teiste testijatega;
- koolitused, seminarid;
- konverentsid;
- testimise võistlused (*Weekend testers* [46], *Topcoder* [47]);
- ühistestimine.

10.1 Ühistestimine

Testijate arendamiseks leidub palju võimalusi, aga üks neist, mis ei ole ainult kasulik vaid samas ka huvitav, on ühistestimine. Selline tegevus toob kasu reaalsetele projektidele, mille peal ühistestimist läbi viiakse. Näiteks, riigiportaali „eesti.ee“ [48] ühistestimise käigus seitse testijate paari lisasid 77 tööülesannet (vead, arenduse soovid, ettepanekud). Umbes 20 viga sai parandatud ja 13 ettepanekut edastati kliendile või suunati teostamisele.

10.1.1 Ühistestimise kirjeldus

Ühistestimine on tegevus, mis koosneb kahest erinevast metoodikast: paaristestimisest (*paired testing*) ja uurivtestimisest (*exploratory testing*). Paaristestimise käigus on ühe masina taga kaks testijat, kes testivad sama funktsionaalsust. Paaristestimise plussideks on ideede genereerimine ja kogemuste jagamine, paremad vearaportid (kõik vead on teise inimese poolt üle vaadatud) [49]. Uurivtestimine on lähenemine tarkvara testimisele, mida lakooniliselt võib kirjeldada kui samaaegset õppimist, testide kavandamist ja rakendamist [50]. Testimise juhtimiseks kasutatakse seansipõhist testimise juhtimist (*session based test*

management), kus testimine toimub seanssides. Seanss on katkematu ja dokumenteeritud testimise katse kindla eesmärgiga [51].

10.1.2 Ühistestimise läbiviimise vajadus

On mitu põhjust, miks ühistestimine on väga kasulik nii testijatele kui ka projektidele.

Uued, projekti mittekuuluvad testijad esinevad kasutajate rollis ja saavad paremini testida kasutajamugavust, mis on eriti kasulik projektidele, kus seda eraldi tavatestimise käigus ei tehta. Näiteks, kuna seansi aeg on piiratud, siis testijad saavad kohe näha, kas kasutajaliides on lihtne ja arusaadav.

Uue inimese värske pilk lubab leida vigu, mis jäid märkamata projekti testijate poolt. Juhul, kui testimiseks ei ole väga palju aega või/ja projekt on piisavalt suur, siis ühistestimine on kiire võimalus leida kavalaid kriitilisi vigu. Plussiks on ka see, et ei ole teada, kuidas klient ja arendajad näevad rakenduse tööd ja sellepärast testija peab ise otsima viisi, kuidas rakendus töötab.

Testijate jaoks on ühistestimine mugav viis kogemuse vahetamiseks, oskuste rakendamiseks ja arendamiseks. Uued projektid lubavad väljaspool harjumuspäraseid raame proovida uusi ideid ja meetodeid, mida ei ole lihtne rakendada projektides, kus testija igapäevaselt töötab.

10.1.3 Ühistestimise läbiviimine

Ühistestimine kujutab endast projekti testimist mitme projekti mittekuuluva testijatega. Testijad jagatakse paarideks ja iga paar saab ühe rakendusega seotud ülesande, mida teostatakse ühe tunni jooksul. Ülesannete näidisteks võiksid olla järgmised tegevused:

- avalehe verifitseerimine spetsifikatsiooni järgi (brauser - IE7);
- brauserite ühilduvus (*cross browser compatibility*);
- uudiste lisamine (peatoimetaja rollis, brauser - FF3.6);

Detailne ühistestimise ülesande näide on toodud Lisas 8. Testimise tundi nimetatakse seansiks [52] ja tavaliselt ühe ühistestimise käigus viiakse läbi mitu seanssi. Tingimuseks on täielik tähelepanu testimisele, mis tähendab, et kõik faktorid, mis võivad segada, kõrvaldatakse (näiteks telefonid ja suhtlemisvahendid lülitatakse välja). Kõik protsessi jooksul leitud vead ja tekkinud ettepanekud dokumenteeritakse. Oluline on mitte tulemus –

vea kirjeldus või ettepanek, vaid protsess, kuidas selleni jõuti. Lisaks peab pidama testimistegevuse logi.

Peale testimise seanssi räägib iga paar oma testimisest: kuidas ja miks just niimoodi testiti, millised vead avastati, millised ettepanekud tekkisid ning mida tahaks/võiks veel testida selle protsessiga seoses. Kõige olulisem on rääkida teistele oma protsessist – kuidas testiti ja miks just nii. Tegemist on kogemuste ja mõtete jagamise temaatilise ürituse ja harjutusega.

Projekti testijad kasutavad hiljem saadud informatsiooni: vaatavad läbi kõik leitud vead ja vajadusel suunavad arendajatele tööülesandeid; ettepanekud vaadatakse üle koos analüütikute ja projektijuhiga, võimalusel ka kliendiga.

11. Kokkuvõte

Tarkvara testimine on lai ja kiiresti arenev ala, mis sisaldab keerukaid osi ja omapärasusi. Käesolevas töös on tehtud katse kirjeldada süstematiseeritud testimise protsessi läbiviimist ja anda ülevaade protsessidest ja printsiipidest, mida tavaliselt nimetatakse “testimiseks”.

Töös on analüüsitud süstematiseeritud testimise põhimõtteid ja põhjusi, miks kaootiline lähenemine testimisele ei vii toodet soovitud kvaliteedini ning kuidas on võimalik organiseerida kaasaegset testimise protsessi. Töös on kirjeldatud tööprotsessis loodavaid artefakte, võimalikke töövahendeid ning nende mõju toote elutsüklile.

Töös on koostatud põhjalik testimise protsessi analüüs, mis annab kogu vajaliku informatsiooni projekti testimise edukaks läbiviimiseks. Antud töö käsitleb mitte ainult testija poolt teostatavaid tegevusi, vaid annab ka ülevaate tervest arendusprotsessist ning programmeerija poolt teostatavast testimisest. Protsesside paremaks mõistmiseks on koostatud diagrammid ja toodud ka vajalike dokumentide näidiseid. Lisaks on toodud näidiseid tehnilistest abivahenditest, mis aitavad testijatel oma tööd kiiremini teha.

Üks põhiline eesmärk oli luua testimise protsessi kirjeldus, mis oleks arusaadav Webmedia klientidele. Selline eesmärk on saavutatud. Antud töö peaks looma kliendile lisaväärtust ja tegema testimise protsessi selgeks ja läbipaistvaks, mis aitab luua paremaid suhteid klientidega ja tõsta toodete kvaliteedi.

Kuna testimise valdkond areneb kiiresti, siis antud tööd hakatakse jooksvalt täiendama vastavalt testimise protsessi muutumisele. Plaanis on näidisteks lisada töö juurde reaalsed testimise dokumendid (väga mahukad antud töö jaoks) ja kirjeldada teisi keerulisemaid kasutatavaid vahendeid (nt. Hudson [53], Changelogic). Testijate väljaõppe ressursse hakatakse täiendama uute viidetega, tehnoloogiliste lahendustega ja raamatutega.

Systemized testing process on the example of Webmedia LC

Master thesis (30 EAP)

Polina Morozova

Summary

Testing is now one of the fastest growing areas in the software development world. Compared to the rest of the world, testing is still young and developing area in Estonia. Because of that we can highlight several major factors that decrease result of quality assurance (testing process):

- software quality is becoming increasingly important with each passing day, because the software becomes more complex, as well as the technical aspects and the business logic;
- people who begin working as testers (many of which are students) have no knowledge of testing theory and practical experience;
- customers do not have complete understanding of testing and software development in general;
- lack of materials in Estonian;
- flow of people in large companies is usually active and companies organize their work process differently.

Taking into account aspects of the software development process mentioned above and complexity of organising high-quality testing process it was decided to write a descriptive overview paper. This thesis analyses many aspects of testing process that take place during development, however we do not go into details of different testing techniques. The practical part of the thesis is a preparation guide for newcoming quality assurance engineers in Webmedia LC.

The present thesis is an attempt to present a systematic approach to testing process and gives overview of subprocesses and general principles that are commonly referred with an umberella term "testing". It is discussed why chaotic approach does not lead to stable product quality and how one could organize modern testing processes. Additionally, the review of artefacts being produced during development, possible productivity tools used by

quality assurance engineers and their impact on product's lifecycle, is presented. Examples of different testing artefacts and tools, which will make testing faster and less complicated, are provided.

It is hoped that this thesis will provide an extra value to customers and make the testing process clear and transparent, which will help to build better relationships between the customers and developers and increase products quality.

Sõnastik

Audit - volitatud isiku sooritatav revisjon tarkvaratoodete ja protsesside sõltumatuks hindamiseks nende nõuetekohasuse otsustamise eesmärgil [5].

Automaattestimine - protsess, mil testide käivitamiseks kasutatakse spetsiaalset tarkvara tööriista. Lisaks sellele tavaliselt seatakse testide eeltingimusi ja raporteerimise võimalusi, kontrollitakse testide käitumist, võrreldakse oodatud ja tegelikke tulemusi.

Artefakt – tarkvaraarenduse käigus loodav ja/või kasutatav tulem.

Funktsionaalne nõue - defineerib tarkvarasüsteemi või selle komponendi funktsionaalsust.

Iteratsioon - lühike ja hästiplaneeritud etapp projektis.

Integratsioon - tervikliku süsteemi järkjärguline koostamine komponentidest [5].

Järelkontroll – testimise meetod, mille eesmärk ei ole vigade otsimine, vaid ainult kinnitus, et vajalik arendus/parandus on teostatud.

*Kvaliteedi tagami*ne - kavandatud süstemaatilised toimingud, mis on vajalikud, et tagada komponendi või süsteemi vastavust kehtestatud tehnilistele nõuetele [5].

Mittefunktsionaalsed nõuded - süsteemsed kvaliteedimõõtmed, mis defineerivad teenuse kvaliteedi.

Nõue – dokumenteeritud vajaduse kirjeldus selle kohta, milline süsteem või tulem peab olema, millist funktsionaalsust võimaldama ja kuidas toimima [5].

Pakkumus - lepingu sõlmimise ettepanek.

Prototüüp - HTML leht, mis näitab, kuidas rakendus ja selle komponendid hakkavad välja nägema ja kuidas protsesse läbitakse.

Regressioontestimine - kogu rakenduse või selle kõige olulisemate protsesside läbi testimist.

Risk – võimalus, et tegevus toob kaasa kahju.

Silumine – leitud vea kõrvaldamine [5].

Spetsifikatsioon - dokumendi vormis detailne formuleering, mis annab süsteemi väljatöötamiseks ja vastuvõtu katsetusteks süsteemi määratleva kirjelduse [5].

Testjuhtum - tegevuse kirjeldus, sellega kaasnevate tingimuste ja muutujate hulk, mille alusel kontrollitakse, kas rakendus või tarkvara toimib ootuspäraselt või mitte ja milles kirjeldatakse, kuidas test läbi viiakse.

Testimine - protsess, mis koosneb nii staatilistest kui ka dünaamilistest elutsükli tegevustest. Testimine puudutab korraga nii tarkvara planeerimist, ettevalmistamist, hindamist kui ka nendega seotud töid. Testimine aitab teha kindlaks, kas valmistatav tarkvara vastab kindlaksmääratud nõuetele (sealhulgas ettenähtud otstarbele) ning avastab defekte.

Testplaan - dokument, mille abil kirjeldatakse ja planeeritakse testimise protsessi.

Valideerimine – protsess, mille käigus kontrollitakse, kas süsteem, rakendus või toode vastab nõuetele.

Verifitseerimine – protsess, mille käigus kontrollitakse, kas süsteem, rakendus või toode vastab spetsifikatsioonile.

Viga - arvutatud, vaadeldud või mõõdetud väärtuse või oleku ning tegeliku, spetsifitseeritud või teoreetiliselt õige väärtuse või oleku lahknevus [5].

Kasutatud allikad

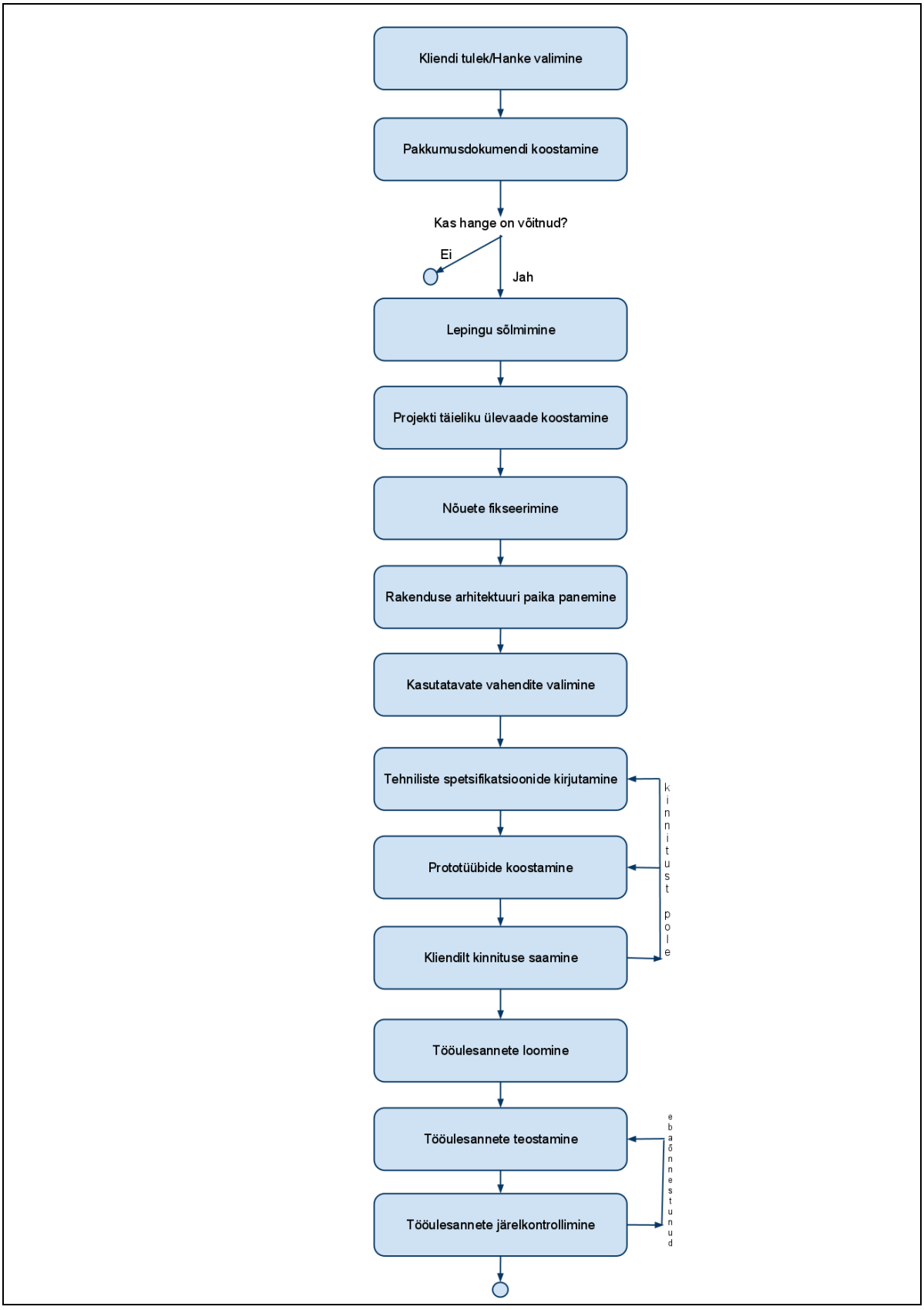
1. Õppekava "Informaatika (2476)" sisu 2011/2012 sisseastunutele. [Veebis]. https://www.is.ut.ee/pls/ois/tere.tulemast?viit=2183882
2. Õppekava "Infotehnoloogia (2477)" sisu 2011/2012 sisseastunutele. [Veebis]. https://www.is.ut.ee/pls/ois/tere.tulemast?viit=2183883
3. „Vikipeedia arutelu: Vikiprojekt Tõlkimine/Tarkvara testimine“. [Veebis, 2.05.2011]. http://et.wikipedia.org/wiki/Vikipeedia_arutelu:Vikiprojekt_T%C3%B5lkimine/Tarkvara_testimine
4. K. Kaljula, "Tarkvara testimine nõuete formuleerimise, analüüsi ja disaini etapis.", Tartu Ülikool, Matemaatika-informaatika teaduskond, magistritöö, 2004. [Veebis 2.05.2011]. http://dspace.utlib.ee/dspace/bitstream/handle/10062/650/Kaljula.pdf
5. J. Tepandi, „Tarkvara kvaliteet ja standardid (IDX5721, IDX5722).“, 2010. [Veebis 2.05.2011]. http://deepthought.ttu.ee/users/tepandi/pdf/tks-loeng.pdf
6. C. Kaner, J. Bach, B. Petticord, „Lessons learned in software testing“, Wiley Computer Publishing, 2002.
7. C. Keith, M. Cohn, „How to fail with agile“, „Better Software“, juuli/august 2008.
8. Agile Modeling, 2001-2010. [Veebis 11.05.2011]. www.agilemodeling.com
9. C. Kaner, J. Bach, „The Seven Basic Principles of the Context-Driven School“. [Veebis 12.05.2011]. http://www.context-driven-testing.com/
10. International Software Testing Qualifications Board. [Veebis 12.05.2011]. http://istqb.org/display/ISTQB/Home
11. Test Maturity Model integrated (*TMMi*). [Veebis 12.05.2011]. http://www.tmmifoundation.org/downloads/tmmi/TMMi%20Framework.pdf
12. Test Management Approach (*TMap*). [Veebis 12.05.2011]. http://eng.tmap.net/Home/
13. E. van Veenendaal, „Standard glossary of terms used in Software Testing.“, 2010. [Veebis 2.05.2011]. http://istqb.org/download/attachments/2326555/ISTQB+Glossary+of+Testing+Terms+2+1.pdf
14. C. Kaner, „Exploratory Testing.“ , 2006. [Veebis 2.05.2011]. http://www.kaner.com/pdfs/ETatQAI.pdf

15. G. Myers, „The Art of Software Testing.“, John Wiley & Sons, Inc. , 2004.
16. T. Huckle, „Collection of Software Bugs.“, 2010. [Veebis 2.05.2011], http://www5.in.tum.de/~huckle/bugse.html
17. S. Garfinkel, „History's Worst Software Bugs.“, 2005. [Veebis 2.05.2011]. http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=2
18. G. W. Weinberg, „Perfect Software and other illusions about testing.“, Dorset House Publishing, 2008.
19. J. McCaffrey, „What Makes A Good Software Tester?“, 2008. [Veebis 8.05.2011]. http://msdn.microsoft.com/en-us/magazine/dd252951.aspx
20. B. Beizer, „Qualities of a Good Tester“, 2000. [Veebis 8.05.2011]. www.sqatester.com/qateam/qualitiesofagoodtester.htm
21. A. Cockburn, „Incremental and Iterative Development. How they are different and why you should be doing both.“, „Better Software“, aprill 2008.
22. Bugzilla. [Veebis 7.05.2011]. *www.bugzilla.org*
23. Mantis Bug Tracker. [Veebis 7.05.2011]. http://www.mantisbt.org/
24. Changelogic. [Veebis 2.05.2011]. http://www.changelogic.com
25. Apache Subversion. [Veebis 9.05.2011]. www.subversion.apache.org
26. ISTQB. Advanced Level Syllabus, 2007. [Veebis 12.05.2011]. http://istqb.org/display/ISTQB/Downloads?atl_token=LRTXfS3XKA
27. TMap, „Techniques.“. [Veebis 2.05.2011]. http://eng.tmap.net/Home/TMap/The_4_essentials/Complete_tool_box/Techniques.jsp
28. T. Koomen, L. Van der Aalst, B. Broekman, M. Vroon, „Tmap Next for result-driven testing.“, UTN Publishers, 2007.
29. S. Asthana, „Best practices for SOA nonfunctional testing.“, 2008. [Veebis 19.05.2011]. http://www.ibm.com/developerworks/webservices/library/ws-soa-nonfunctional/index.html
30. Google code, Page speed. [Veebis 7.05.2011]. http://code.google.com/speed/page-speed/
31. Browser shots. [Veebis 7.05.2011]. http://browsershots.org/
32. H. Vallaste, „E-teatmik on ingliskeelsete info- ja sidetehnoloogia terminite seletav sõnaraamat“, 2000-2011. [Veebis 8.05.2011]. www.vallaste.ee
33. Standard ISO 9241-11:1998. [Veebis 7.05.2011]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883

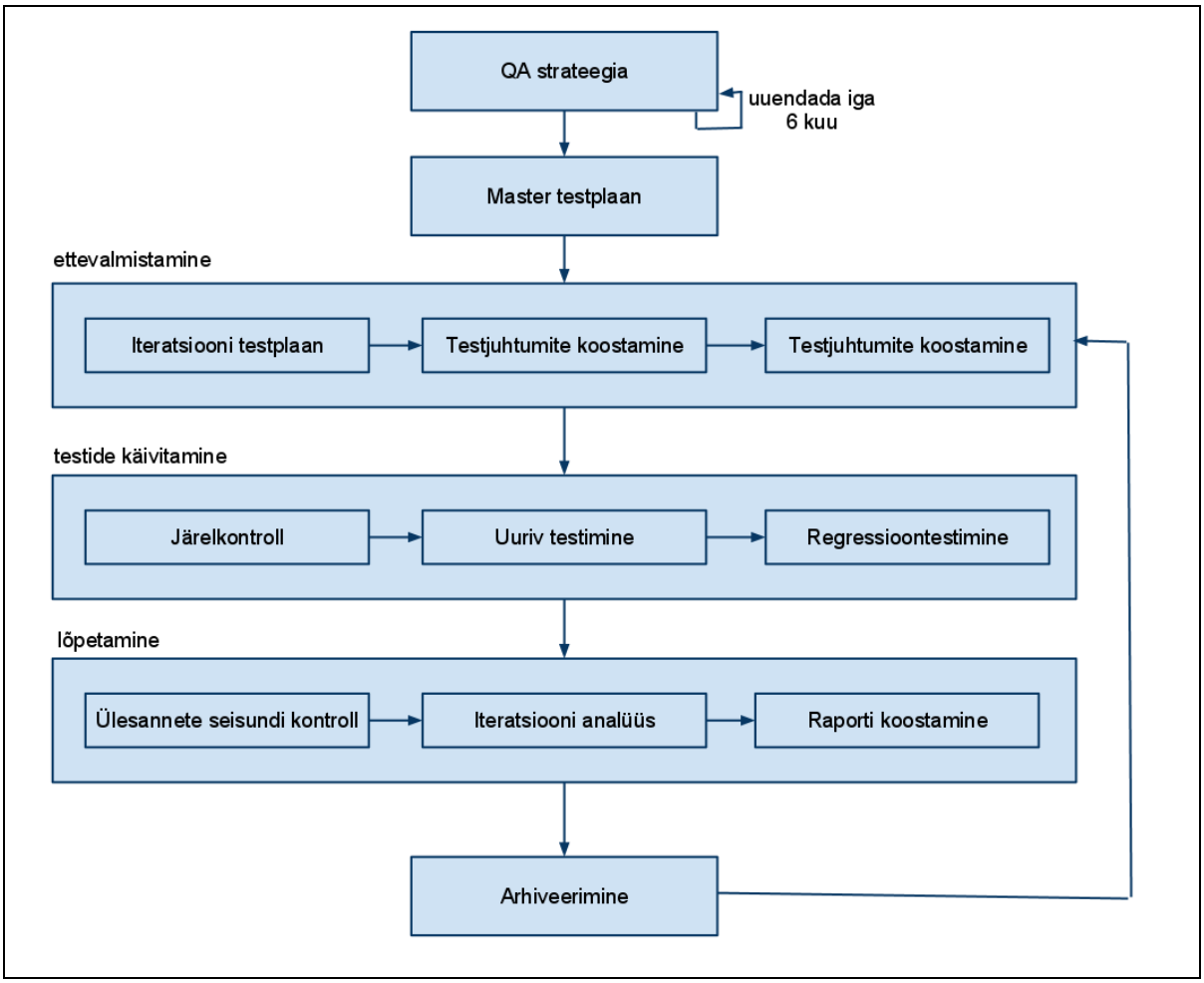
34. D. Graham, E. van Veenendaal, I. Evans, R. Black, „Foundations of Software Testing“, Cengage Learning EMEA, 2008.
35. IEEE Standard for Software Test Documentation (IEEE standard 829), 2008 [Veebis 06.05.2011]. http://www.cs.unb.ca/profs/wdu/cs3043w10/IEEE-829-2008.pdf
36. U.Visse, „IT arendustest ja nendega seotud hankemenetluste valikust töötukassas 2009-2010.“. [Veebis 2.05.2011]. www.tootukassa.ee/index.php?id=13586&highlight=EMPIS
37. Selenuim. [Veebis 07.05.2011]. http://seleniumhq.org/
38. A. Kolawa, D.Huizinga, „Automated Defect Prevention: Best Practices in Software Management.“, Wiley-IEEE Computer Society Press, 2007.
39. K. Whitmill, „So, you’we got a problem...Crafting remarks and abstracts for more effective defect reports.“, „Better Software“, september 2008.
40. T. Ryber, „Essential software test design.“, Fearless Consulting Kb, 2007.
41. SnagIt. [Veebis 2.05.2011]. www.techsmith.com/snagit/
42. Extra Screen Capture. [Veebis 2.05.2011]. http://www.dvdcopyrip.com/extra-screen-capture.html
43. Random String Generator, 2008-2011. [Veebis 2.05.2011] http://www.random.org/strings/
44. Regionaalminister, „Isikukoodide moodustamise, väljajagamise ja andmise kord.“, 2005. [Veebis 2.05.2011]. https://www.riigiteataja.ee/akt/983873?leiaKehtiv
45. J-G. Järvemaa, „Isikukood“. [Veebis 2.05.2011]. www.skip.nonsense.ee/isikukood.php
46. Riigiportaal „eesti.ee“. [Veebis 8.05.2011]. www.eesti.ee
47. Weekend Testing. [Veebis 13.05.2011]. http://weekendtesting.com/
48. Topcoder. [Veebis 13.05.2011]. apps.topcoder.com/wiki/display/tc/Testing
49. C. Kaner, J. Bach, „Exploratory testing in pairs“, 2001. [Veebis 8.05.2011]. www.kaner.com/pdfs/exptest.pdf
50. J. A. Whittaker, „Exploratory software testing“, Pearson Education, Inc, 2010.
51. J. Bach, „Session-Based Test Management“, 2000. [Veebis 8.05.2011]. www.satisfice.com/articles/sbtm.pdf
52. C. Kaner, J. Bach, B. Pettichord, „Lessons Learned in Software Testing.“, Wiley Computer Publishing, 2002.
53. Hudson. [Veebis 19.05.2011]. http://hudson-ci.org/

Lisad

Lisa 1. Arendusprotsessi diagramm



Lisa 2. Testimise protsess



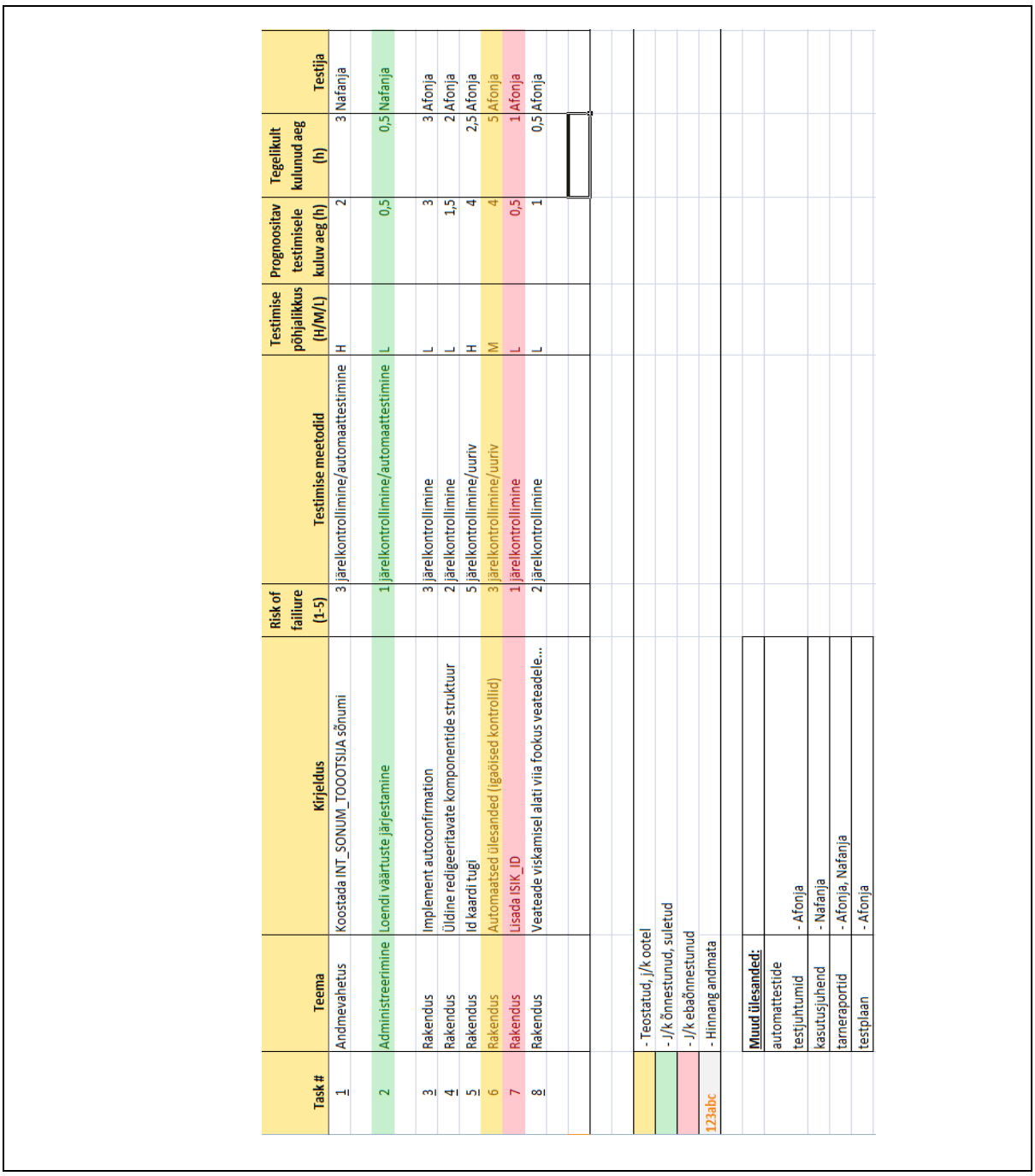
Lisa 3. I iteratsiooni näide master testplaanist

Testimise alustamise kriteeriumid: spetsifikatsioonid on valmis.

Iteratsioon I (25.02.2009 - 28.03.2009)
Staatiline testimine
Analüüs
Muudatuste ülevaatus (arendajate poolt)
Funktsionaalsuse testimine
Õiguste süsteem
Sisselogimine
Süsteemsete parameetrite haldus
Riigipühad
Pankade haldus
Asutuste funktsionaalsus
Minu ülesanded (lugemise pool ja muutmise/lisamise pool)
Isiku kontekst, inforiba, teated
Isiku otsing
Isiku detailandmed
Isiku töökogemus ja selle haldus
Seisundite nimekiri
Protsesside nimekiri
Automaattestimine
Süsteemsed (Administreerimise menüü all) komponendid kaetakse automaat web (liidese põhistega) testidega.
Kõik komponendid kaetakse Unit testidega (arendajate poolt).
Stabiliseerimine
Parandatakse kõik avastatud iteratsiooni arendustööde käigus tehtud funktsionaalsed vead, Unit testid, konfiguratsiooni vead.
Tulemus
Kõik iteratsiooni käigus arendatud komponendid avanevad ja toimivad analüüsi järgi.
Kõik avastatud arendustöö käigus tehtud vead on parandatud.
Kõik automaattestid annavad positiivset tulemust.
Stabiilne versioon on paigaldatud testimiskeskkonda.
Automaatne iga öine versiooni ehitamine, baasi genereerimine ja Unit testide käivitamine on konfigureeritud ja rakendatud.
Regressioon testimine
Vana funktsionaalsus puudub

Testimise lõppemise kriteeriumid: ülesanded on täidetud

Lisa 4. Iteratsiooni testplaan



Task #	Teema	Kirjeldus	Risk of failiure (1-5)	Testimise meetodid	Testimise põhjalikkus (H/M/L)	Prognoositav testimisele kuluv aeg (h)	Tegelikult kulumud aeg (h)	Testija
1	Andmevahetus	Koostada INT_SONUM_TOOOTSIJA sõnumi	3	järelkontrollimine/automaattestimine	H	2	3	Nafanja
2	Administreerimine	Loendi väärtuste järjestamine	1	järelkontrollimine/automaattestimine	L	0,5	0,5	Nafanja
3	Rakendus	Implement autoconfirmation	3	järelkontrollimine	L	3	3	Afonja
4	Rakendus	Üldine redigeeritavate komponentide struktuur	2	järelkontrollimine	L	1,5	2	Afonja
5	Rakendus	Id kaardi tugi	5	järelkontrollimine/uuriv	H	4	2,5	Afonja
6	Rakendus	Automaatsed ülesanded (igaöised kontrollid)	3	järelkontrollimine/uuriv	M	4	5	Afonja
7	Rakendus	Lisada ISIK_ID	1	järelkontrollimine	L	0,5	1	Afonja
8	Rakendus	Veateade viskamisel alati viia fookus veateadele...	2	järelkontrollimine	L	1	0,5	Afonja

- Teostatud, j/k ootel
- J/k õnnestunud, suletud
- J/k ebaõnnestunud
- Hinnang andmata

123abc

Muud ülesanded:	
automattestide testjuhtumid	- Afonja
kasutusjuhend	- Nafanja
tarneraportid	- Afonja, Nafanja
testplaan	- Afonja

Lisa 5. Detailne testjuhtum

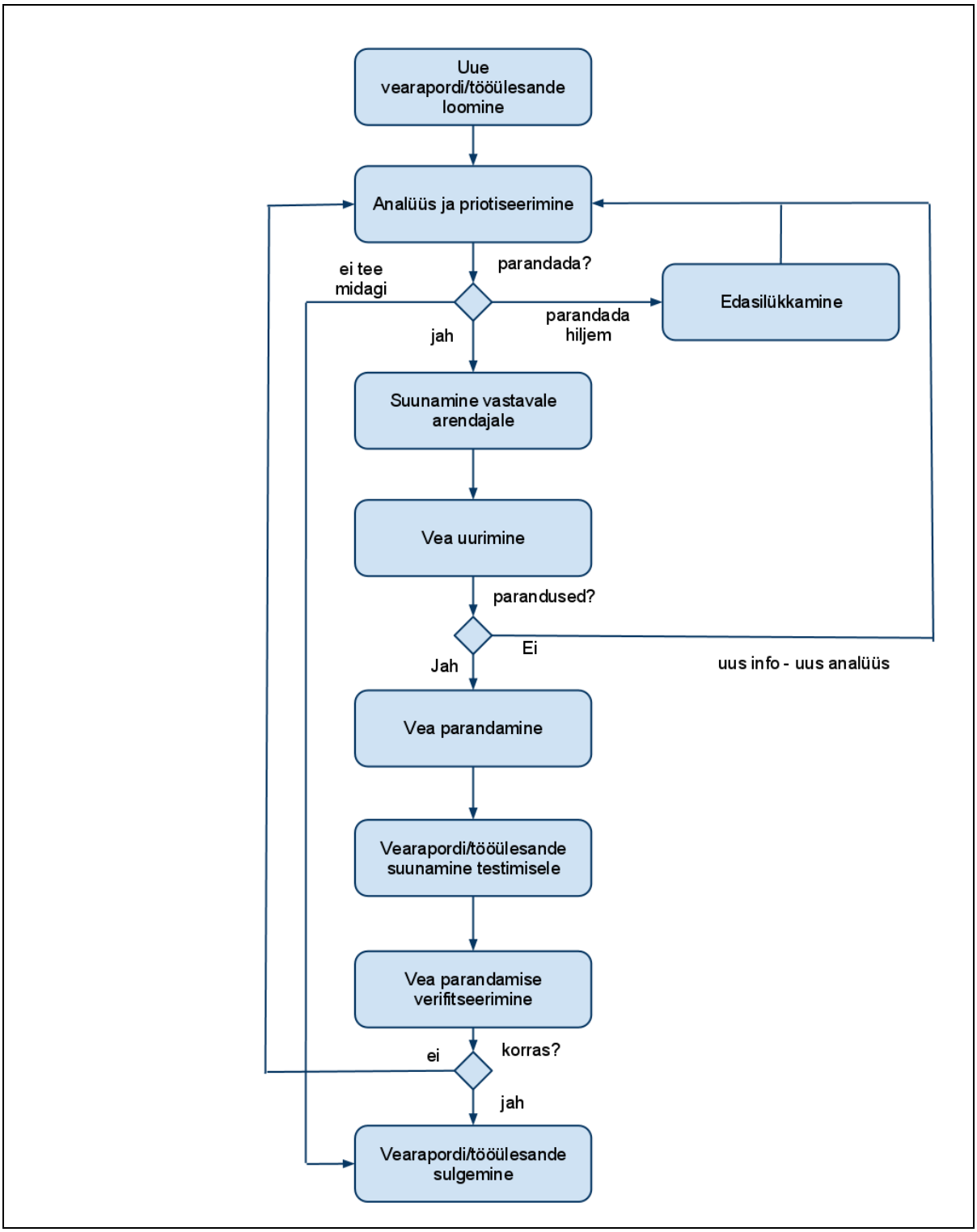
Testjuhtum №1 – KokkuvotteLisamineJaMuutmineToootsijanaArvelOlevIsik

Vajalikud ettevalmistuse sammud: isik peab olema lisatud süsteemi

Tegevus	**Oodatav tulemus**
Isiku otsingus valitakse töötuna arvel olev isik.	Avatakse valitud isiku kontekst.
Vasakpoolses menüüs valitakse menüüpunkt ’Detailandmed’.	Avatakse isiku detailandmete kuva. Vaikimisi avatud ’Üldandmete’ sakk.
Valitakse ’ITK I/II osa kokkuvõte’ sakk.	Avatakse sisestatud kokkuvõtete nimekiri.
Vajutatakse nuppu ’Lisa’	Kuva jääb samaks ja lisatakse järgmised veateated: Väli 'Tööle saamise piirangud/soodustused' on kohustuslik.
Vorm 'Tööle saamise piirangud/soodustused' täidetakse järgmiselt: • Sisestatakse tekst „Tööle saamise piirangute ja soodustuste informatsioon puudub“. Vajutatakse nuppu ’Lisa’.	Tekib uus link ’Muuda’.
Vajutatakse linki „Muuda“ eelmises punktis testi käigus lisatud kirje juures.	Avatakse kokkuvõtte muutmise komponent ja ilmuvad lingid Salvesta/Katkesta.
Välja muudetakse järgmiselt: • „Tööle saamise piiranguid ei paista olevat". Vajutatakse „Salvesta“ linki.	Nimekirjas kuvatakse muudetud kokkuvõtte kirje.

Testi läbimise kriteeriumid: test on läbitud, kui kõik oodatavad tulemused on saavutatud.

Lisa 6. Uue vearaporti/tööülesande elutsükkel



Lisa 7. Testija väljaõppe materjalid

Raamatute ja ajakirjade nimekiri:

1. Gerald M.Weinberg, „*Perfect Software and other illusions about testing*“, Dorset House Publishing, 2008.
2. Gerald M.Weinberg, „*An Introduction to General Systems Thinking*“,
3. James A.Whittaker, „*How to break software*“, Pearson Education, Inc, 2003
4. James A.Whittaker, „*Exploratory software testing*“, Pearson Education, Inc, 2010
5. Torbjörn Ryber, „*Essential software test design*“ , Fearless Consulting Kb, 2007.
6. Cem Kaner, Petticord Bach, „*Lessons learned in software testing*“, Wiley Computer Publishing, 2002.
7. David A. Levy, „*Tools of critical thinking*“, Waveland Press, Inc., 2010
8. G. Myers, „The Art of Software Testing.“, John Wiley & Sons, Inc. , 2004
9. D. Graham, E. van Veenendaal, I. Evans, R. Black, „Foundations of Software Testing“, Cengage Learning EMEA, 2008.
10. Ajakiri „*Better software*“, Software Quality Engineering

Internetiressursside nimekiri:

1. Joel on software: www.joelonsoftware.com
2. Topcoder: apps.topcoder.com/wiki/display/tc/Testing
3. StickyMinds: www.stickyminds.com
4. ISEB/ISTQB Software Testing: www.iseb-istqbsoftwaretesting.co.uk

Lisa 8. Ühistestimise ülesande näide

Avalehe verifitseerimine spetsifikatsiooni järgi

- Brauserid: IE8 + FF3.6.
- Logi pidamine.
- Tegevuste käik: alguspunkt -> testimise idee -> tulemus (pange kõik testimise ideed kirja!).
- Probleemide raporteerimine CL'i (Changelogic).
- Testimise tulemuste esitamine:
    - tegevuste käik;
    - miks ühte või teist katsetati;
    - rakenduse kahtlased kohad;
    - avastatud probleemid;
    - testimise ideed, mida ei jõutud läbi proovida.