TARTU ÜLIKOOL

Loodus- ja tehnoloogiateaduskond

Tehnoloogiainstituut

Henri Perkmann

Veebirakenduse kasutajaliidese automaatne testimine väleda arendusprotsessi kontekstis – põhimõtted ja implementatsioon

Bakalaureusetöö (12 EAP)

Juhendaja: M.Sc Vambola Leping

Luban töö kaitsmisele:

Juhendaja ………………………….

Programmijuht ………………………….

Tartu 2015

Web application user interface automated testing in an agile software development environment – principles and implementation

Henri Perkmann

ABSTRACT

As agile methodologies have crept into web application development processes, traditional testing practices and processes are being proven inadequate. Traditional manual testing is unable to cope with frequent releases promoted by agile development teams.

This research paper introduces an agile testing framework for both manual and automated testing. The testing pyramid metaphor illustrates that as much as possible of regression tests should be automated. Writing automated user interface tests should take into account certain practices, given in this paper. Developing and maintaining these tests demand certain processes, also introduced.

As most web applications lean on databases for storing data, different approaches can be taken in order to make consistent data available for user interface automated tests. Four approaches are given, each with pros and cons.

Writing and running user interface automated tests demands a suitable system. This system rests on a Ruby/Watir based programming framework. Jenkins continuous integration/delivery software is used to run test cycles and publish test reports. Tests themselves run on virtual machines which are configured for stability and recoverability.

Sõnastik ... 4
1. Sissejuhatus ... 8
1.1 Regressiooni automatiseerimine ... 9
1.2 Automaattestimise teemalised ainekursused ja väitekirjad ... 10
2. Teooria ... 12
2.1 Väleda testimise põhimõtted ... 12
2.2 Testimispüramiid ... 14
2.3 Kasutajaliidese automaattestide kirjutamise head praktikad ... 16
2.4 Kasutajaliidese automaattestide arendamise ja haldamise protsess ... 17
2.4.1 Automaatse kvaliteedi tagamise protsess: ... 17
3. Testandmed ... 19
3.1 Testandmete simuleerimine ... 19
3.2 Testandmete genereerimine ... 20
3.3 Testandmete otsimine ... 21
3.4 Testandmete koodi sisse kirjutamine (*hard-code)* ... 22
4. Praktika ... 23
4.1 Süsteemi ülevaade ... 23
4.2 Teegid ... 24
4.3 Raamistiku poolt pakutavad võimalused testide kirjutamisel ... 25
4.4 *Jenkins* ... 26
4.5 Virtuaalmasin ... 27
5. Kokkuvõte ... 28
6. Kasutatud kirjandus ... 29
7. Lisad ... 31
Testitav rakendus - www.elion.ee iseteenindus ... 31
Testide jagunemine ... 32
Jenkins'i töökäsu konfiguratsioon ... 33
Testitsüklite käivitamine ... 35
Näidistesti analüüs #1 ... 36
Näidistesti analüüs #2 ... 38
Testtsükli raporti näide ... 40
Osa autori poolt tehtud koodimuudatustest ... 41

Sõnastik

API-testid – API-testid testivad API-keskset arhitektuuri[1], kus veebirakenduse ning andmebaaside vahele on loodud äriloogikat haldav teenuskiht. API-testid võimaldavad mugavalt testida API-kihis paiknevat äriloogikat ning integratsiooni andmebaasidega.

Arenduspaar – Paarisprogrammeerimine on programmeerimise vorm, kus sama programmeerimisülesandega tegelevad korraga kaks programmeerijat. Paarisprogrammeerimise puhul on kasutusel kaks kuvarit, mis näitavad sama pilti. Tavaliselt üks arendaja programmeerib ning teine jälgib tegevust. Paarisprogrammeerimine aitab tõsta kirjutatud koodi kvaliteeti, õpetada nooremarendajaid ja muuta programmeerimist meeskonnatööks.

Automaatne testimine – Automaatne testimine võimaldab rakenduse funktsionaalsuse järjepideval automaatsel testimisel pidevalt tagada kvaliteeti. Tavapäraselt kirjutatakse uuele funktsionaalsusele erineval testitasemel automaatsed testid (ühiktestid, API-testid, kasutajaliidese testid). Võimalikult madalal testitasemel proovitakse katta võimalikult suur osa funktsionaalsusest.

Backlog – *Backlog'*is asuvad kasutajalood, mida pole veel arendama hakatud. Backlog'i haldab tavaliselt tooteomanik.

Bugi – Tarkvara arenduse või halduse käigus tekkinud probleem, mis takistab tarkvara normaalset tööd.

Kanban – *Kanban* on tarkvara väleda arendamise metoodika, mis sarnaneb *scrum* metoodikale. Erinevus on selles, et *kanban* metoodikas ei kasutata sprinte. Arendusmeeskond võtab pärast kasutajaloo valmimist *backlog*'i tipust arendamiseks järgmise kasutajaloo. Tooteomanik saab meeskonna tööd häirimata mõjutada toote arendust toote *backlog*'i tippu reprioritiseerides. Kui tooteomanik hoiab kõige olulisemad tööd *backlog*'i tipus, pakutakse kliendile võimalikult kiiresti maksimaalselt lisandväärtust.[2]

---

[1] E. Anuff, „API-Centric Architecture: All Development is API Development,“ https://blog.apigee.com/detail/api_centric_architecture_all_development_is_api_development (22.05.2015).

[2] D. Radigan, „„A brief introduction to kanban or, what software makers can learn from Japanese manufacturing,“ https://www.atlassian.com/agile/kanban (22.05.2015).

Kasutajaliidese testid (*user interface tests*) – Kasutajaliidese testimine tegeleb reaalsele kasutajale (kliendile või omatöötajale) mõeldud rakenduste kasutajaliideste testimisega. Kasutajaliidese testimisel võib välisteenuseid simuleerides keskenduda ainult testitavale rakendusele.

Kasutajalugu (*user story*) – Tooteomaniku poolt kirjeldatav rakenduse kasutajale väljapaistev funktsionaalsus, mida arendusmeeskond arendab. Hea kasutajalugu sisaldab vastuvõtukriteeriume (*acceptance criteria*) ning kogu informatsiooni võimalikult kiireks arenduseks. Välised sõltuvused on kasutajaloos kirjas ning varem valmis arendatud.

Koskmudel (*waterfall*) – Koskmudel on esimesena kirjeldatud tarkvarasüsteemi arendamise mudel. Enim kritiseeritud mudelina lähtub koskmudel tavalistest tootmisprotsessidest ehituses ja mehhaanikas. Erinevad tegevused (nõuete määratlemine, süsteemi kavandamine, arendamine, integratsioon, testimine, kasutamine, hooldus) toimuvad eraldi etappidena. Iga etapp algab pärast eelmise etapi lõppu. Projekti rangeteks faasideks jagamine ei luba protsessi käigus muudatuste tegemist ning puudulikust analüüsist või programmeerimisvigadest tulenevate probleemide parandamine on hilise testimise tõttu raskendatud.[3]

Käitumisel põhinev arendus (*behavior driven development*) - Testimisel põhinevast arendusest väljakasvanud tarkvara arenduse protsess, mille järgi tuleb automaatseid teste kirjutada äriliselt väärtuslikele protsessidele (kasutaja käitumine) ning seejärel need protsessid arendada.[4]

Lähtekoodi pidev integratsioon/tarnimine (*continuous integration/delivery*) – Selle bakalaureusetöö kontekstis *Jenkins*[5] rakendust kasutav süsteem/protsess, kus koodi repositooriumisse laaditud lähtekoodi alusel ehitatakse pidevalt uusi rakendusserverile paigaldatavaid pakke (*war*/*zip* faile) ning paigaldatakse nad rakendusserverile. Seeläbi tagatakse viimase koodiseisu olemasolu arendus- ning testprotsessides kasutatavates rakendusserverites.

Manuaalne testimine – Tüüpiliselt rakenduse kasutajaliidese funktsionaalsuse testimine. Manuaalselt saab testida nii uusi arendusi kui ka olemasolevat funktsionaalsust (regressiooni testimine).

---

[3] I. Petuhhov, „Koskmudel," http://www.e-uni.ee/e-kursused/eucip/arendus/1221_koskmudel.html (22.05.2015).
[4] BDDWiki: BehaviourDrivenDevelopment, http://behaviourdriven.org/ (22.05.2015).
[5] K. Kawaguchi, „Meet Jenkins," https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (22.05.2015).

MVP (*minimum viable product*) – Eric Ries'i poolt kasutusele võetud termin, mis tähendab kliendile/kasutajale maksimaalse funktsionaalsuse pakkumist võimalikult vähese pingutusega (investeeringuga). Sellise lähenemisega loodud rakendused hakkavad esimesel võimalusel ettevõttele tulu tooma ning nende edasisel arendusel saab keskenduda täiendava funktsionaalsuse inkrementaalsele arendamisele ning turule toomisele.[6]

Regressiooni testimine – Traditsiooniliselt rakenduse olemasoleva funktsionaalsuse manuaalne (enne rakenduse versioonivahetust) või automaatne (pidev) testimine. Üldises tähenduses terve rakenduse funktsionaalsuse testimine, seega sisalduvad regressioonis ka näiteks ühiktestid.

Scrum – 1993. aastal Jeff Sutherlandi poolt loodud üks edukaim tarkvara väleda arendamise metoodika, mis pakub arendusmeeskonnale paindliku raamistiku ning aitab keskenduda sihile. Ettevõetud ülesanded lahendatakse inkrementaalselt tsükliliste sprintide käigus kogu meeskonnaga, mille liikmeteks tooteomanik/äritellija (*product owner*), *scrum master* ja iseorganiseeruvad arendajad.[7]

Testjuhtum (*test case*) – Rakenduse ühte (äriliselt vajalikku) funktsionaalsust kirjeldav tekst või lähtekood.

Versioonivahetus (*release*) – Rakenduse uue versiooni *live'*i minek ehk hetk, kui rakenduse uus versioon avatakse kasutajatele. Tavaliselt tähendab see rakenduse uue versiooni paigaldust *live* keskkonda ja/või andmebaasi migratsiooni.

Väle tarkvaraarendus (*agile software development*) – Väle tarkvaraarendus on tarkvaraarenduse metoodikate kogum, mille kohaselt tarkvara nõuded ning lahendused arenevad iseorganiseeruvate multifunktsionaalsete meeskondade koostöös. Väle tarkvaraarendus tähendab kiiret ja paindlikku muudatustega kohanemist, pidevaid täiendusi ja varajast versioonivahetust.[8]

Ühiktestid (*unit*-testid) – Rakenduse koodi väikseimate moodulite (meetodite) testimine. Lähtekoodi muutmisel (kesksesse koodi repositooriumisse laadimisel) tuleks arendaja poolt

---

[6] A. Maurya, „Minimum viable product. Race to deliver customer value," http://leanstack.com/minimum-viable-product/ (22.05.2015).

[7] A. Kozlov, „Miks scrum?" http://www.scrum.ee/miks-scrum (22.05.2015).

[8] Agile Methodology, http://agilemethodology.org/ (22.05.2015).

käivitada ühiktestide komplekt. Eelnevalt tuleb uutele arendustele kirjutada uued ühiktestid või parandada vanu.

1. Sissejuhatus

Pidevalt areneva ning sagedate versioonivahetustega veebirakenduse kvaliteedi hoidmine nõuab olemasoleva ning arendatava funktsionaalsuse testimist erinevatel tasemetel nii käsitsi kui automaatselt. Tavalises praktikas vastutab pidevalt areneva rakenduse kvaliteedi tagamise eest kogu rakenduse elutsükli jooksul sama hulk inimesi. Uued arendused ning olemasoleva funktsionaalsuse refaktoorimine võivad rakenduse normaalses töös põhjustada probleeme ja bugisid. Siiski on mõeldamatu, et testijad suudavad rakenduse kogu funktsionaalsuse enne igat versioonivahetust manuaalselt üle kontrollida. Eriti juhul, kui rakendus peab olema valmis näiteks iganädalasteks versioonivahetusteks. Seetõttu tuleb pidevalt areneva rakenduse kvaliteedi tagamiseks kaasata lisaks manuaalsele testimisele automaatteste.

Käesoleva bakalaureusetöö eesmärk on anda praktiline ülevaade toimivast veebirakenduse kasutajaliidese automaatse testimise raamistikust. Raamistiku all on siinkohal mõeldud protsessi koos tarkvaralise ning süsteemse lahendusega.

Lõputöö autor töötab AS Eesti Telekom tarkvaratestijana, kelle peamised töökohustused on seotud www.elion.ee era- jaärikliendi iseteeninduse kvaliteedi tagamisega. Tööülesannete raames on autor tegelenud kahe veebirakenduse (www.elion.ee ja www.telekom.ee) kasutajaliidese automatiseeritud testimise raamistiku ülesseadmise ning haldamisega. Lisaks kuulub igapäevatöö hulka uute testide kirjutamine ning olemasolevate testide haldamine. Bakalaureusetöös käsitletava rakenduse kasutajaliidese testide raames on autori poolt tehtud 199 koodimuudatust ning mõjutatud 365 faili. Käesolev bakalaureusetöö koondab pooleteise aasta jooksul omandanud olulisi praktilisi kogemusi, millele tuginedes oleks töö lugejal võimalik iseseisvalt töötada välja ning implementeerida pidevalt areneva rakenduse jaoks sobilik testimise raamistik.

Bakalaureusetöö sisuline pool on jaotatud kolmeks osaks. Teises peatükis antakse ülevaade testimise rollist väledas arendusmetoodikas ja tutvustatakse erinevaid testimistasemeid kirjeldavat testimispüramiidi. Samuti vaadeldakse töö fookuses olevaid kasutajaliidese teste, täpsemalt nende kirjutamise häid praktikaid. Teise peatüki lõpetab ülevaade kasutajaliidese testide arendamise ja haldamise protsessist.

Kolmandas peatükis vaadeldakse testandmete temaatikat, andes ülevaate peamistest võimalustest testandmete kasutamisel.

Neljandas peatükis keskendutakse www.elion.ee iseteeninduste testimisele *Watir*'i ning teiste toetavate teekidega. Lisaks antakse ülevaade pideva automaatse testimise süsteemist, kus kõik testid käivitatakse pärast iga koodiuuendust omaette virtuaalmasinas, mida kontrollib *Jenkins*'i pideva integratsiooni keskserver. Sealjuures on oluline märkida, et terve süsteem peab olema täisautomaatne ning võimeline erinevate probleemide korral tööd jätkama. Iga testtsükli tulemused esitatakse hästiloetava ja informatsioonirohke raporti kujul.

Lõputöö kokkuvõttes tuuakse välja süsteemi kasutamisel saadud õppetunnid, millele toetudes on edaspidi võimalik luua paremini töötavaid süsteeme.

1.1 Regressiooni automatiseerimine

Käesolevas bakalaureusetöös kirjeldatava protsessi ja süsteemi implementeerimise vajalikkuse mõistmiseks tuleb täpsemalt kirjeldada olulisi probleeme, mis mõlema puudumisel esinesid. www.elion.ee veebirakenduse kvaliteedi tagamisega tegelevad kaks testijat, kelle töö koosneb arendusmeeskondade poolt loodava uue funktsionaalsuse testimisest ning enne versioonivahetust rakenduse manuaalsest testimisest. Olemasoleva funktsionaalsuse testimine toimus varasemalt suures osas manuaalselt. Selline protsess oli ajamahukas ning seetõttu oli versioonivahetust võimalik realiseerida maksimaalselt kord kuus.

Nii harva toimuvad versioonivahetused ei sobinud äritellijatele, kuna sellega kaasnev viivitus uue funktsionaalsuse klientidele jõudmisel suurendas arendamisega seotud kulutusi (pikemal arendusprotsessil on finantsiliselt väiksem kasutegur). Manuaalne regressiooni testimine ei sobinud testijatele, kuna selle korduv iseloom langetas oluliselt testijate motivatsiooni. Samuti tuleb manuaalse testimise juures arvestada inimliku vea võimalusega, mida on praktiliselt võimatu elimineerida. Lisaks pole pidevalt lisanduvate arenduste taustal sisuliselt võimalik manuaalse testimisega ära katta kogu rakenduse äriloogikat.

Kuna testimisressurss on konstantne, tuli testijatel pidevalt tegeleda regressiooni nimekirja uuendamisega (uute testjuhtude lisamine) ning prioritiseerimisega (vähemtähtsate testjuhtude eemaldamine). Veel olulisemaks probleemiks manuaalse testimise juures võib pidada selle toimumist vahetult enne uue versiooni väljatulekut. Vaid mõned päevad enne versioonivahetust avastatud vead on tihtipeale minimaalse ajaakna kontekstis parandamatud. Sellisel juhul tuleb rakenduse versioonivahetus kriitiliste probleemide esinemise tõttu edasi lükata.

Ülal kirjeldatud probleemide lahendamiseks otsustati rakenduste kvaliteedi paremaks tagamiseks kasutusele võtta automaatsed kasutajaliidese testid. Uue süsteemi arendamine toimus mitmes etapis ning lõplikult autonoomse lahenduse väljatöötamine eeldas lisaks innovaatilisele tehnilisele lahendusele muutust arendajate ja testijate mõtteviisis. Taustal leidis aset arendusprotsessi üleviimine *kanban* metoodikalt *scrum* arendusmetoodikale. Projektijuhtimise seisukohalt toimus mahukate pooleaastaste projektide asendamine väiksemamahuliste projektidega, mis keskendusid MVP (*minimum viable product*) lahendustele, suuremate projektide skoopideks jagamisele ja tihti toimuvatele versioonivahetustele.

1.2 Automaattestimise teemalised ainekursused ja väitekirjad

Viimastel aastatel on automaattestimise teematika muutunud teaduslikult üha populaarsemaks. 2010. aastal hakati Tallinna Tehnikaülikoolis lugema kursust Tarkvara automatiseeritud testimine, mis keskendus TTCN-3-nimelise testiarendustehnoloogia õpetamisele.[9] 2014. aasta kevadsemestril loeti Tallinna Tehnikaülikoolis Automaattestimise kursust, mis keskendus ühiktestimise kõrval ka veebirakenduse kasutajaliidese testimisele *Selenium*'i teegi baasil.[10]

2008. aastal kaitses Sergei Sergejev Eesti Infotehnoloogia Kolledžis diplomitöö, mis tutvustab testimise põhimõtteid erinevate arendusmetoodikate korral, võrdleb peamiseid automaattestimise vahendeid ning annab praktilise ülevaate *Selenium* raamistikul põhinevatest funktsionaalsetest testidest.[11] Meeli Pällin kaitses 2012. aastal Tartu Ülikoolis *Selenium'*i-teemalise bakalaureusetöö.[12] Age Kruusamägi bakalaureusetöö, mis kaitstud 2012. aastal Tallinna Ülikoolis, keskendub LHV panga maksete allsüsteemi testimise automatiseerimise väljatöötamisele *Selenium* raamistiku näitel. 2014. aastal kaitses Tallinna Ülikoolis

[9] A. Kull, „Software testing automation course in Tallin University of Technology supported by Elvior," http://www.elvior.com/news-and-blog/news/automated-test (22.05.2015).

[10] M. Kalmo, „Automaattestimine 2014," http://enos.itcollege.ee/~mkalmo/ (22.05.2015).

[11] S. Sergejev, „Funktsionaalsete automaattestide loomine uue kõrgkoolide vahelise õppeinfosüsteemi jaoks Selenium raamistiku näitel," Diplomitöö, Eesti Infotehnoloogia Kolledž, 2008, http://enos.itcollege.ee/~ssergeje/diploma/SergejevDiploma.pdf (22.05.2015).

[12] M. Pällin, „Automaattestimisvahendite kasutus ning praktiline ülevaade Seleniumi näitel," Bakalaureusetöö, Tartu ülikool, Matemaatika-informaatikateaduskond, 2012, http://comserv.cs.ut.ee/forms/ati_report/downloader.php?file=ABD4559968E0DC88727D55B95D260E9CA295F3DC (22.05.2015).

bakalaureusetöö Mait Mikkelsaar, kes toob oma töös välja *Selenium*'i raamistiku pakendi *Selenide*'i, mis lihtsustab oluliselt kasutajaliidese testide kirjutamist.[13]

Loetletud lõputööde kõrval on oluline tõsta esile käesolevas bakalaureusetöös kirjeldatud süsteemi eeloost kirjutatud Triinu Karminga tööd teemal „Elioni automaattestimise raamistiku täiendamine“.[14] Karmingu 2011. aastal Tallinna Tehnikaülikoolis kaitstud bakalaureusetöö tõi välja automaattestimisega seotud probleemid Elionis (praegune AS Eesti Telekom), lõi nõuded ja standardid automaattestidele, dokumentatsioonile ning käivitusprotsessile.

[13] M. Mikkelsaar, „Kasutajaliidese testid programmeerimisoskuse hindamiseks," Bakalaureusetöö, Tallinna Ülikool, Informaatika Instituut, 2014, http://www.cs.tlu.ee/teemad/get_file.php?id=313 (22.05.2015).

[14] T. Karming, „Elioni automaattestimise raamistiku täiendamine," Bakalaureusetöö. Tallinna Tehnikaülikool, 2011.

2. Teooria

Kuna käesolev bakalaureusetöö keskendub regressiooni automaatsele testimisele väleda tarkvaraarenduse keskkonnas, selgitatakse järgnevalt põgusalt väleda testimise omapärasid. Testimise automatiseerimise näitlikustamiseks kasutatakse testimispüramiidi analoogi. Seejärel tuuakse välja konkreetselt kasutajaliidese automaattestide programmeerimise head praktikad. Peatüki lõpetab kasutajaliidese automaattestide kirjutamise ja haldamise protsessi ülevaade.

2.1 Väleda testimise põhimõtted

Traditsioonilistes *waterfall*-projektides peab testimisosakond andma hinnangu arenduse kvaliteedile projekti arenduse lõppfaasis. Selline lahendus võib osutuda suuremate ning keerulisemate projektide puhul sobivaks, sest erinevate meeskondade poolt pika aja jooksul arendatava tarkvara omavahelist lõimumist ning integratsiooni teiste süsteemidega on võimatu testida arendustsükli jooksul.

Peamisteks probleemideks antud lahenduse juures on aga leitud bugide prioritiseerimise keerukus, vigade parandamine ilma uute bugide tekitamiseta ja versioonivahetuse võimalik edasilükkumine. Mida hiljem tarkvaras probleeme leitakse, seda keerulisem ning kulukam on neid parandada. Selliste probleemide vältimiseks võib sobiv lahendus olla väle testimine, mis testib arendusi manuaalselt kohe pärast nende valmimist ning automaatselt rakenduse elutsükli jooksul. [15]

Väledates meeskondades ei ole testimine omaette faas, vaid sellega tegeletakse pidevalt. Tihtipeale ei ole klassikalistes väledates arendusmeeskondades eraldi testijaid, vaid testimisega tegeleb jooksvalt terve meeskond. Rakenduse kvaliteediga pidev tegelemine välistab olukorra, kus bugid kuhjuvad ning tuleb tegelema hakata nende prioritiseerimisega. Samuti ei teki olukorda, kus bugide algpõhjuse leidmine võtab palju aega, sest automaatne testimine annab koheselt märku bugi põhjustanud koodimuudatusest. Seetõttu võib pidevat testimist pidada ainukeseks viisiks, millega tagada lakkamatut kvaliteeti.[16]

Nagu eelpool mainitud, eeldab pidev testimine, et sellega tegelevad kõik meeskonnaliikmed. Kindlasti vajab arendusmeeskond ühte hea testimisoskusega liiget, kuid on mõeldamatu, et

[15] E. Hendrickson,. Agile Testing. „Nine Principles and Six Concrete Practices for Testing on Agile Teams,“ lk 5, http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf (22.05.2015).

[16] *Ibid, lk 6.*

mitme arenduspaari tööd suudab manuaalselt ja automaatselt testida ainult üks inimene. Seda põhjusel, et väleda testimise juurde kuulub igale arendusele automaatsete testide kirjutamine. Need testid võivad olla koodi tasemel (ühiktestid), API-kihi tasemel (API-testid) kui ka kasutajaliidese tasemel (kasutajaliidese testid).

Terve meeskonna panus automaatsete testide kirjutamisel on vajalik, kuna kindlale arendusele suudab adekvaatseima testi luua just selle arenduse tegija. Kasvanud töömahu tõttu võib esialgu tunduda, et väleda testimise tõttu langeb üldine arenduskiirus. Kuna eesmärgiks on luua hästitoimiv ja bugide-vaba lahendus, mis rahuldaks äritellijat ning säiliks probleemivabana rakenduse elutsükli lõpuni, tuleb harjuda aeglasema tempoga. Kokkuvõttes õigustab aeglasemat arendustempot seeläbi tagatud arenduse jätkusuutlik kvaliteet.

Väleda testimise üks eesmärke on lühendada arendajale tagasiside andmisele kuluvat aega. Väledates tarkvaraprojektides programmeeritakse arendusi selliselt, et neid on koheselt võimalik testida. See toob kaasa erinevate arenduste ja rakenduste vaheliste sõltuvuste vähenemise ning arendused, mille programmeerimine võtab aega maksimaalselt paar päeva.[17]

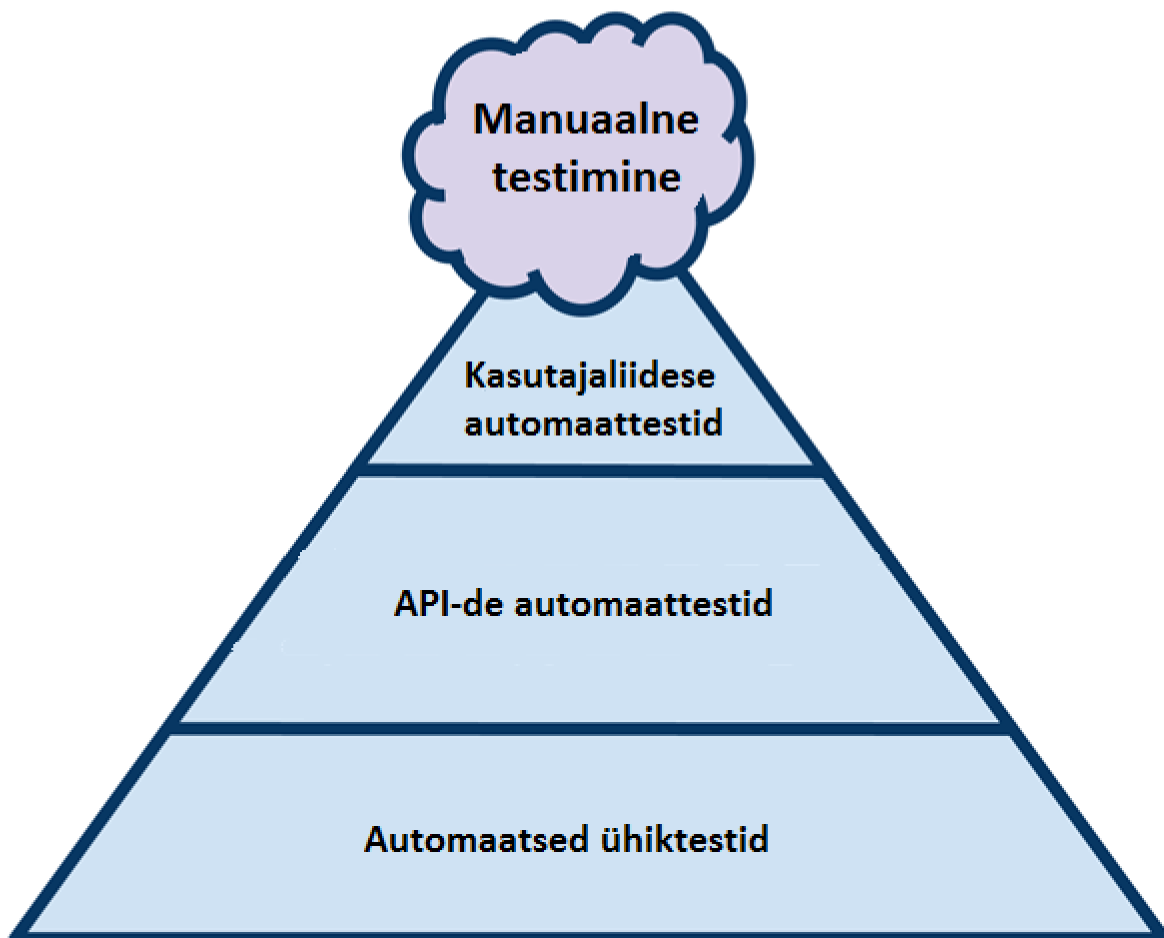
„The definition of „done““ ehk millal on kasutajalugu arendatud ehk valmis? Traditsioonilistes projektides, kus arendamine ning testimine paiknevad eraldi osakondades ja kus testimine toimub arendusprotsessi lõpus, esineb tihtipeale ettekujutus, et kasutajalugu on „valmis“, kui arendaja on oma töö lõpetanud. Tegelikult on kasutajalugu valmis alles siis, kui see toimib tõrgeteta ning ei mõjuta rakenduse olemasolevat funktsionaalsust. Väleda arendusprotsessi kasutusele võtmisega tagatakse, et kasutajalugu on „valmis“ ainult juhul, kui see vastab tegelikele valmisoleku kriteeriumitele.[18]

Väle testimine kindlustab rakenduse kvaliteedi ka olukordades, kus arendamiseelse analüüsi tulemusel pole suudetud välja tuua kõik kasutajalood või kitsaskohad. Traditsiooniliste tarkvaraprojektide puhul on selliste olukordade tekkimine paratamatu, sest eelanalüüs ei saa kunagi olla täiuslik ning arendusfaasile järgneva testimise käigus leitud probleeme arvestatakse parimal juhul järgmise versiooni skoobis. Halvimal juhul võivad eelanalüüsi puudujäägid põhjustada rakenduse versioonivahetuse edasilükkumise. Väleda testimise rakendamisel leitakse probleemid koheselt ning äritellija saab nendega arvestada juba arendustsükli käigus.

[17] *Ibid*, lk 7.
[18] *Ibid*, lk 11.

2.2 Testimispüramiid

Väledad tarkvaraprojektid kasutavad testimiseks mitmeid viise, mis moodustavad testimispüramiidi (vt joonis 1). Testimispüramiid kirjeldab ilmekalt erinevate testimistüüpide mahtu ning vormi (manuaalne/automaatne) regressioonitestimisel. Testimispüramiidi ideoloogia kohaselt proovitakse võimalikult madalal asuvas kihis testida võimalikult palju. Tipu poole liikudes muutub manuaaltestimise läbiviimine või automaattestide kirjutamine ja haldamine keerulisemaks, ajamahukamaks ning seeläbi ka kallimaks. Automaattestide üheks peamiseks eeliseks on asjaolu, et need testivad rakenduse kvaliteeti pidevalt ja seetõttu leitakse bugid koheselt. Manuaalne regressioonitestimine toimub tavaliselt arendustsükli lõpus enne versioonivahetust ning sel hetkel bugide avastamine võib osutuda kvaliteetse versioonivahetuse kontekstis liiga hiliseks.



Joonis 1. Testimispüramiid. Autori koostatud

Testimispüramiidi alumises osas on testimine lihtne ning keskendub koodi tasemel meetodite ühiktestide kirjutamisele. Ühiktestid testivad ainult rakenduse enda kvaliteeti, integratsiooni teiste süsteemidega testitakse kõrgemates kihtides. Ühikteste peaks arendaja käivitama enne uue koodi repositooriumisse laadimist. Samu teste käivitab ka lähtekoodi pideva integratsiooni süsteem, mis valmistab iga uue kooditäienduse järel rakendusserverile pealepandava *war* formaadis faili.

Püramiidi järgmisel tasemel asuvad API-testid ehk teiste süsteemidega suhtluse testimine. Veebirakenduse kontekstis, kus suurem osa ärifunktsionaalsust on viidud API-tasemele, vastutavad nende testide eest API-d ise. Seega tuleb veebirakenduse teiste süsteemidega integratsiooni testimiseks vaid kontrollida, kas veebirakendus saab API-dega suhelda. Vanema arhitektuuriga veebirakendustes puudub andmebaasiga suhtluse vahel API-kiht. Tihtipeale andmebaas ning selles sisalduv veebirakendusele väljapaistev ärifunktsionaalsus iseend automaatselt ei testi, mistõttu tuleb testimine katta näiteks kasutajaliidese (integratsiooni)testides.

Kasutajaliidese automaattestide kirjutamine ning haldamine on kõige ressursikulukam automaattestimise vorm, kuna testid kasutavad tööks reaalseid internetilehitsejaid, mis pöörduvad reaalsete veebiserverite poole. See muudab testide kirjutamise ajamahukaks, kuna nende käivitamine nõuab võrreldes teiste automaattestidega lisaaega. Mahukamate kasutajaliidese testide puhul võib testi käivitamine kesta mitu minutit. Samuti on oluline rõhutada, et kasutajaliidese automaattestide haldamine (testandmete parandamine, testide parandamine rakenduse muutumise tõttu, reaalsete bugide analüüs) võtab võrreldes püramiidi alumiste kihtide testidega rohkem aega. Ajamahukama haldamise kõrval on pikem ka testtsükli läbikäimiseks kuluv aeg. Kui tuhandete ühiktestide tsükli käivitamine võib kesta loetud sekundid, siis mahukamad kasutajaliidese testide tsüklid võivad kesta tunde, mis omakorda pikendab tagasiside andmist arendusmeeskonnale.

Testimispüramiidi tipus paikneb manuaalne testimine, mille maht peaks olema võimalikult väike. Suurem osa rakenduse testimisest peaks olema kaetud alumistes "automaatkihtides". API-kihi ja andmebaaside integratsiooni testimine (kas teenus vastab või mitte) võib jääda manuaalselt kaetuks. Samuti vajavad manuaalset testimist teatud testjuhtumid, mille automaatne testimine võib olla raskendatud või isegi võimatu (näiteks ID-kaardi või mobiil-ID sisselogimine, arvete maksmine pangalingiga).

2.3 Kasutajaliidese automaattestide kirjutamise head praktikad

Kasulike ning kergesti hallatavate kasutajaliidese testide kirjutamiseks tuleb panustada testide kvaliteedile. Kiiresti kirjutatud test võib hiljem valepositiivsete tulemuste või muu halduskulu tõttu olla ressursimahukam kui hästi kirjutatud test. Seega on mõistlik juba testi kirjutades arvesse võtta järgmiseid häid tavasid ja praktikaid:

1) Testi kirjutamisel tuleb vältida funktsionaalsuse, mida saab testida ka madalamal testimise tasemel (nt ühiktesti tasemel), testimist. Näiteks piisab kasutajaliidese testimisel tekstiväljade testimise puhul kahest testjuhtumist: korrektne sisend ning ebakorrektne sisend. Kõik teised ebakorrektsete sisendite juhtumid, mis rakenduse koodis lahendatud, tuleks testida ühiktesti tasemel.
2) Test peab olema võimalikult efektiivne – tegema võimalikult palju võimalikult vähese ajaga. Tuleb silmas pidada, et iga uus test lisab kogu testtsükli läbimise kestvusele aega juurde.
3) Iga test peab testima vaid ühte funktsionaalsust. Pikad ja kohmakad testid on raskemini hallatavad: vigade analüüs on keerukam ning testi parandamine ajamahukam. Siiski leidub erandeid. Näiteks pikkades toodete tellimise voogudes tuleb viimasesse sammu jõudmiseks läbi käia varasemad sammud ning iga sammu eraldi testimine võib tähendada liigset ajakulu. Sellistel juhtudel on erinevaid funktsionaalsusi testivad mahukad testid lubatud.
4) Testi kirjutamisel tuleb peamiselt pöörata tähelepanu selle loetavusele. Test peab olema sedavõrd lihtne ja arusaadav, et teised meeskonnaliikmed suudaksid sellest pikalt süvenemata aru saada. Test ei ole ülesanne, milles peaks demonstreerima keeruliste algoritmide kirjutamise oskust.
5) Testi kirjutamisel tuleb pöörata tähelepanu selle stabiilsusele. Veebilehte testides võivad tekkida ooteajad sisu laadimisel ning sellega tuleb arvestada. Kindlasti ei ole lahenduseks testi n sekundiks ootama panemine. Tuleks kasutada testiraamistiku funktsionaalsust, mis ootab terve veebilehe või teatud elementide laadimist enne testiga jätkamist. Enne koodi repositooriumisse laadimist peaks autor testi enda masinas mõned korrad läbi jooksutama, veendumaks selle stabiilsuses.
6) Testi stabiilsust mõjutavad ka testandmed. Eelisjuhul peaks testidel välised sõltuvused teenustest olema eemaldatud neid näiteks lokaalse HTTP serveriga simuleerides. Järgmine soovitus on enne testi testandmete loomine andmebaasides, kolmandaks

nõuandeks on sobivate testandmete otsimine andmebaasist. Viimase variandina tuleks testkasutajad ja –tooted testi sisse kirjutada.

7) Testi kirjutamisel tuleks arvestada võimalike muudatustega veebisisu andmebaasis. Kui veebirakendus on loodud selliselt, et tekstid asuvad andmebaasis, võib neid kergesti jooksvalt muuta ning tihtipeale seda ka tehakse. Sellisel juhul peaksid testid tekste pärima andmebaasist sarnaselt sellele, kuidas veebirakendus ise tekste andmebaasist pärib.

2.4 Kasutajaliidese automaattestide arendamise ja haldamise protsess

Kasutajaliidese testide kirjutamine on ressursimahukas töö, lisaks omavad testid esmapilgul varjatud halduskulu. Võrreldes ühiktestidega, mille testtsükli kestvus on vaid mõned sekundid ning mida arendajad uuendavad tavaliselt enne koodi kesksesse repositooriumisse laadimist, võib kasutajaliidese testide testtsükkel kesta tunde. Vigade märkamine, analüüs ja parandamine ei toimu järelikult koheselt ning vea põhjustanud arendaja poolt. Seetõttu on vaja inimest, kes analüüsiks näiteks iga tööpäeva hommikul testraportite tulemusi. Kuigi vigade võimalikult kiire avastamine on oluline prioriteet, ei anna tihedam analüüs praktilistele kogemustele toetudes paremaid tulemusi. Seda põhjusel, et a) vigast koodi laaditakse koodi repositooriumisse harva, b) valepositiivsed tulemused võivad kaasa tuua tarbetu analüüsi, c) testihalduri tööpäev on vähem killustatud ning ta saab tegeleda muude kohustustega.

2.4.1 Automaatse kvaliteedi tagamise protsess:

1) Arendajad ja testijad kirjutavad uuele funktsionaalsusele automaatsed testid.
2) Pidevalt täienevat testikogumit käivitatakse tihti, võimalusel kohe pärast eelmise testtsükli lõppu või pärast arendaja poolt uue koodi repositooriumisse laadimist.
3) Testihaldur analüüsib teste iga tööpäeva hommikul ning vigase testi analüüsi järgselt
    a) parandab uute arendustega katki läinud testi (test on katki),
    b) pöördub rakenduse reaalse bugi korral vea parandamiseks vigase koodi autori poole (rakendus on katki).
4) Lisaks analüüsib testihaldur jooksvalt testide katvust ning vajaduse korral kirjutab teste juurde või palub arendajatel seda teha.

Testiraportite analüüsil tuleb pöörata tähelepanu vigade järjestikusele esinemisele, et vältida valepositiivsete tulemuste analüüsi. Kui viga esineb kahes või enamas järjestikuses testtsüklis, on ilmselt tegu reaalse veaga, mis vajab tähelepanu. Testide raportite juures peaksid kindlasti

olema koodimuudatused võrreldes eelmise tsükliga, kuna seeläbi on võimalik hõlpsasti leida vigane kood ning selle autor.

3. Testandmed

Testid kasutavad tihtipeale andmeid, mis asuvad andmebaasides. Kuna tüüpilise veebirakenduse funktsionaalsus toetub andmete muutmisele ning muudatuste kuvamisele, siis tuuakse siinkohal välja erinevad testandmete kasutamise võimalused.

3.1 Testandmete simuleerimine.

Testandmete simuleerimine on rakenduse kvaliteedi testimiseks kõige parem viis. Simuleeritakse välisteenuseid, näiteks suhtlust erinevate andmebaaside ja API-dega. Tavaliselt simuleeritakse kasutajaliidese testide andmeid kas testitavas rakenduses endas, kasutatakse selleks teste ja rakendust jooksutavas masinas üles seatud HTTP serverit või mõnes muus masinas üles seatud HTTP serverit.

Testandmete simuleerimise peamised eelised on järgmised:

a) Täielik kontroll testandmete üle. Andmebaasides toimuvad muudatused ei mõjuta testide stabiilsust.
b) Täielik puutumatus välisteenuste maasolekute, võrguprobleemide või versioonivahetuste suhtes.
c) Lokaalses masinas nii rakendust kui välisteenuste simulatsioone jooksutades on rakendusel andmete kättesaamine palju kiirem, võrreldes päris teenuste, mille baasipäringud võtavad aega ning mis asuvad võrguliikluse kiiruse kontekstis kaugemal kui lokaalne masin, poole pöördumisega.

Testandmete simuleerimise puudused:

a) Testandmete simuleerimise peamine puudus on asjaolu, et testitakse ainult rakenduse enda kvaliteeti. See tähendab, et välisteenustes toimuvad muudatused või nende maasolek jääb kasutajaliidese testidel märkamata. Siiski tuleb rõhutada, et ebastabiilses arenduskeskkonnas on terve süsteemidevahelise integratsiooni testimine palju töömahukam, sest põhjustab teadmatust ning suhtlust kolmandate osapooltega (välisteenuste halduritega ja arendajatega). Veelgi enam, baasipäringute ja API-kihi kvaliteedi testimist ei tohiks implementeerida kasutajaliidese tasemel, sest veebilehel liikumine võtab kümnetes kordades rohkem aega võrreldes lahendusega, kus testitavaid baasipöördumisi ja teenuspäringuid tehakse otse vastu testitavat rakendust. Iga rakendus

peaks enda automaatse testimisega ise tegelema ning rakenduste vaheliste integratsioonide testimist tuleks implementeerida võimalikult väikeses mahus ja madalal tasemel.

b) Testandmete simuleerimine eeldab algset ressurssi simuleerimissüsteemi sisseseadmisel ning pidevat ressurssi uute simulatsioonide lisamisel. Samas tuleb eelisena märkida asjaolu, et selle tegevuse käigus õpitakse selgeks rakenduse suhtlus välisteenustega.
c) Kui välisteenused muutuvad, peab ka simulatsioone uuendama.

3.2 Testandmete genereerimine

Testandmete genereerimise korral kasutavad testid teatud baasiprotseduure, millega genereeritakse enne testi vajalikud testandmed. Kasutusel on teegid, mis võimaldavad testide otsesuhtlust andmebaasiga.

Testandmete genereerimise peamised eelised:

a) Selline lahendus on kasulik testjuhtumite puhul, kus testi käigus andmed baasis muutuvad (näiteks kontaktandmed, toodete tellimine, lepingute sõlmimine jne)
b) Lahendus aitab vältida olukorda, kus baasis olevad andmed võivad väliste osapoolte tõttu muutuda. Näiteks võib andmebaasis olla küll tuhandeid testkliente, kuid juhuslikult võivad teised baasikasutajad erinevate tegevuste tulemusel andmeid muuta. Test läheb muudetud andmete tõttu katki ning testihaldur peab analüüsima valepositiivset tulemust.
c) Testandmete genereerimiseks vajalikud protseduurid on tavaliselt andmebaasides juba olemas, sest neid kasutatakse baasi enda või selle poole pöörduvate rakenduste poolt.
d) Lahendus aitab kaasa testide stabiilsusele, sest testid saavad alati kasutada sama andmestikku.
e) Testitakse süsteemide integratsiooni (veebirakenduse ja andmebaasi vahelist suhtlust).

Testandmete genereerimise puudused:

a) Tuleb leida sobiv teek ning realiseerida selle implementatsioon.
b) Andmebaasides võivad protseduurid aja jooksul muutuda ning neid võib juurde tekkida. Järelikult nõuab lahenduse haldamine lisaressurssi. Tüüpiliselt on andmebaasid võrdlemisi stabiilsed ning praktikas on lisanduv töömaht pea olematu.
c) Testandmete genereerimine tagab küll alati testide jaoks sobiva andmestiku, kuid reaalselt võivad baasis esineda andmevead. Lahenduse puhul jäävad sellised olukorrad kontrollimata. Puudust leevendab asjaolu, et testide jooksutamiseks kasutatav

arenduskeskkond tihtipeale sisaldabki vigast andmestikku ning see võib põhjustada testidel valepositiivseid tulemusi.

d) Andmebaas võib olla maas või ei saada sellega võrguprobleemide tõttu ühendust. See põhjustab olukorra, kust lahendust kasutavad testid ei toimi. Samas reaalselt ei esine sellist olukorda tihti, sest baasid on võrdlemisi stabiilsed. Võimalikud probleemid lahendatakse kiiresti, kuna need mõjutavad erinevaid huvigruppe ning arendusmeeskondi.
e) Testandmeid tekib genereerimise käigus baasi pidevalt juurde. Seega tuleb tavaliselt baasi tasemel implementeerida funktsionaalsus, mis juba kasutatud testandmeid kustutab. Testi enda sees sellise lahenduse sisseviimine ei pruugi olla süsteemi ebastabiilsuse tõttu piisavalt töökindel.

3.3 Testandmete otsimine

Testandmete andmebaasist otsimine on lahendus, mille puhul testi läbiviimiseks vajalikud andmed otsitakse enne testi käivitamist vastavate päringutega andmebaasist. Lahenduse toimimiseks on vaja teeki, mis võimaldaks testil andmebaasiga suhelda. Testandmete otsimine sobib seetõttu kasutamiseks koos testandmete genereerimisega, sest mõlemat ülesannet saab lahendada sama teegiga.

Testandmete otsimise eelised:

a) Testandmete otsimist kasutades võib avastada andmevigu, mis võivad osutada tõsisematele andmeprobleemidele. Lahendus aitab seega kaasa andmekvaliteedi parandamisele.
b) Lahendus on võrdlemisi stabiilne, sest otsib testandmeid alati teatud parameetrite järgi. Kui andmebaasis on andmed muutunud, siis päring neid ei tagasta ning test neid ei kasuta.
c) Testitakse süsteemide integratsiooni (veebirakenduse ja andmebaasi vahelist suhtlust).

Testandmete otsimise puudused:

a) Andmebaas peab lahenduse kasutamiseks sisaldama võimalikult vähe andmevigu vältimaks valepositiivseid testitulemusi. Seega tuleks kasutada andmebaase, kuhu on kopeeritud reaalsete klientide andmed. See võib aga olla juriidilistel põhjustel (andmekaitse) raskendatud.
b) Lahendus ei võimalda alati täpselt samade andmete kasutamist, sest enamikel juhtudel ei ole võimalik või otstarbekas kasutada liiga spetsiifilisi päringuid. Seega peavad testid arvestama erinevate juhtumitega, mis lisab testide kirjutamisele keerukust ning võib

tekitada olukorra, kus reaalsed vead võivad liiga üldiste kontrollide tõttu jääda tähelepanuta.

c) Lahenduse kasutamine eeldab teadmisi kasutatavatest andmebaasidest ning nende struktuurist. Puudust leevendava momendina võib välja tuua asjaolu, et sellised teadmised aitavad kaasa tehniliste oskuste arenemisele.

3.4 Testandmete koodi sisse kirjutamine (*hard-code)*

Testandmete koodi sisse kirjutamine tähendab seda, et test kasutab alati samu andmeid. Näiteks võib test alati kasutada sama kliendi andmeid. Tegemist on esmapilgul lihtsa ja mugava lahendusega, kuid see võib pikas perspektiivis suurt halduskulu põhjustada. Seetõttu sobib antud lahendus pigem ajutiselt testi kirjutamise lihtsustamiseks. Pikas perspektiivis tuleb testandmete kasutajal valida eelnevalt kirjeldatud kolme lahenduse vahel.

Testandmete koodi sisse kirjutamise eelised:

a) Kuna andmebaasis olevad andmed on juba kasutusel, ei ole vaja rakendada lisameetmeid andmete simuleerimiseks, genereerimiseks või otsimiseks. Esmane implementatsioon on lihtne ning kiire.

b) Lahenduse kasutamiseks pole vaja teadmisi andmebaasidest ega nendega suhtlemisest.

c) Testide parandamine on lihtne. Näiteks tuleb muutunud andmete puhul lihtsalt asendada testi sissekirjutatud testklient.

d) Testitakse süsteemide integratsiooni (veebirakenduse ja andmebaasi vahelist suhtlust).

Testandmete koodi sisse kirjutamise puudused:

a) Andmebaasis võivad andmed muutuda ning testid lähevad seetõttu tihti katki, põhjustades valepositiivseid tulemusi. Seega kasvab testide halduskulu aja jooksul.

b) Vaatamata asjaolule, et testides on andmete muutmine lihtne (näiteks testkliendi isikukoodi asendamine uuega), tuleb sobivad andmed andmebaasi luua või nad otsida. See pole küll tavaliselt keeruline, kuid võtab aega.

c) Ei sobi kasutada, kui testi käigus andmed baasis muutuvad. Isegi, kui test need andmed ise suudab kasutajaliidese funktsionaalsust kasutades taastada, võib test erinevatel põhjustel ebaõnnestuda ning järgmisest tsüklist alates jääbki test vigaste andmete tõttu ebaõnnestuma.

4. Praktika

Peatükis kirjeldatakse www.elion.ee era- ja ärikliendi iseteeninduse automaattestimise süsteemi. Esmalt antakse ülevaade süsteemi üldisest struktuurist ning seejärel kirjeldatakse kasutatavaid teeke. Edasi laskutakse üksiktesti tasemele, misjärel kirjeldatakse terve testtsükli käivitamist pideva integratsiooni keskkonnas *Jenkins*. Peatüki lõpus tutvustatakse teste jooksutava virtuaalmasina konfiguratsiooni.

4.1 Süsteemi ülevaade

Uue funktsionaalsuse arendamisel www.elion.ee veebirakendusele kirjutatakse tavaliselt koos lähtekoodiga ühiktestid ning kasutajaliidese testid. Peamiselt keskendutakse rakenduse enda kvaliteedile, seega integratsioon teiste süsteemidega on pigem sekundaarne. Ühiktestid (ja *PhantomJS* testid) kasutavad täielikult simuleeritud andmeid, kasutajaliidese testid kasutavad andmebaasidega suhtlusel testandmete genereerimist ja koodi sisse kirjutamist, API-kihiga suhtlusel simuleeritud andmeid. Järgnevalt antakse ülevaade kasutajaliidese testimise süsteemist.

Kasutajaliidese testid asuvad koos rakenduse koodiga ühes repositooriumis. See lihtsustab testide sünkroonis olekut arendusega, sest teste saab mugavamalt parandada. Testid jagunevad analoogselt rakendusega kaheks – erakliendi ning ärikliendi testideks ehk testtsükliteks. Testid grupeeritakse kahe üldjaotuse all sarnaste lehtede või funktsionaalsuste alusel. Üldiselt on igas testfailis keskmiselt 3-4 testi.

Pideva integratsiooni keskkonnas *Jenkins* asuvad testtsükleid käivitavad töökäsud, mis ühenduvad teste jooksutavate virtuaalmasinatega, laadivad sinna repositooriumist viimase koodiseisu, käivitavad selle baasil rakenduse ning seejärel käivitavad vastava testtsükli. Virtuaalmasinas käivitatakse veebilehitseja, mis asub testima samas masinas käivitatud veebirakendust. Testtsükli lõpus saadetakse rakenduse logid ja raport õnnestunud ning ebaõnnestunud testidest *Jenkins*'isse. Mainitud töökäsud käivitatakse pärast iga koodiuuendust. Testtsüklid kestavad 1-2 tundi.

4.2 Teegid

Järgnevalt on kirjeldatud peamised *Ruby* teegid (*gem*-id), mis kasutusel kasutajaliidese automaattestides.

Bundler

Bundler[19] pakub mugava viisi hallata kõiki projektis kasutatavaid *Ruby* teeke. *Bundler* lubab pääsu „sõltuvuste põrgust" ning hoolitseb selle eest, et defineeritud raamistikud oleksid igas keskkonnas olemas täpselt sellise versiooni või versioonivahemikuga nagu vaja. *Bundler* aitab lahendada probleeme, mis võivad testide arendajatel tekkida puuduolevate või vale versiooniga teekide tõttu.

RSpec

RSpec[20] on käitumisel põhineva arenduse *Ruby* teek, mis muudab testide kirjutamise loomulikuks käitumisel põhinevaks protsessiks. Teek võimaldab kirjeldada testjuhtumeid loogiliselt ning struktureeritult, lisab konfigureerimise võimalused erinevate testtsüklite käivitamiseks, võimaldab enne ja pärast teste käivitada teatud käsklusi (testimise eelsete ja järgsete olukordade lahendamiseks) ning mis peamine, võimaldab oodatavate tulemuste kontrollimist.

Watir, ***Watir-rspec***, ***Watir-webdriver***

Watir[21][22] on avatud lähtekoodiga *Ruby* teek, mille eesmärgiks on internetilehitseja automatiseerimine. *Watir* kasutab veebilehitsejaid nagu inimesed: klikkides linkidele, täites tekstivälju, vajutades nuppe. *Watir*'iga saab kontrollida tulemusi, näiteks kas oodatav tekst ilmub veebilehele või mitte. *Watir* toetab ainult *Internet Explorer*'it, seega muude veebilehitsejate automatiseerimiseks on kasutusele võetud *watir-webdriver*[23]. Viimane toetab *Chrome*'i, *Firefox*'i, *Internet Explorer'it*, *Opera*'t ning *HTMLUnit*'it (kasutajaliideseta testide jooksutamiseks). Watir-webdriver on mugavaim viis kasutada *Selenium WebDriver*'it *Ruby*'s. *Watir-rspec*[24] on Jarmo

---

[19] Bundler, „What is Bundler," http://bundler.io/ (22.05.2015).

[20] D. Chelimsky et al, *The R-Spec Book: Behaviour-Driven Development with R-Spec, Cucumber and Friends* (The Pragmatic Programmers, 2010)

[21] Watir, „Automated testing that doesn't hurt," http://watir.com/ (22.05.2015).

[22] Vt lisaks näited *Watir* teegi kasutamisest http://watir.com/examples/ (22.05.2015).

[23] Watir WebDriver, http://watirwebdriver.com/ (22.05.2015).

[24] J. Pertman, „Watir/watir-rspec," https://github.com/watir/watir-rspec (22.05.2015).

Pertmani kirjutatud teek *Watir*'i ning *RSpec*'i paremaks integreerimiseks, muutes testikoodi kirjutamise mugavamaks ning lihtsamaks. Märkimisväärsena lisab *watir-rspec* loetavad raportid, kus lisandväärtust pakuvad vigaste testide ekraanitõmmised ning HTML lähtekood.

Ruby-oci8 **ja** ***ruby-plsql-spec***

Ruby-oci8[25] teegi abil on *Ruby* programmides võimalik suhelda *Oracle* andmebaasidega. Teek lubab testides kasutada andmebaasis olevate testandmete loomise ning haldamise funktsionaalsust. *Ruby-oci8* abil on loodud klientide genereerimine/kustutamine, toodete ja teenuste lisamine/kustutamine ja palju muud. *Ruby-plsql-spec*[26] täiendab *Oracle* andmebaasiga suhtlust, võimaldades PL/SQL protseduuride väljakutsumist

WEBrick

WEBrick on *Ruby* teek, mille abil on võimalik jooksutada HTTP veebiserverit. *WEBrick* toetab HTTPS-i ning testimisel saab seda kasutada lokaalselt välisteenuste simuleerimiseks. Tegemist on asendamatu tööriistaga, sest aitab suurendada testide stabiilsust ning vähendada haldusmahtu.

4.3 Raamistiku poolt pakutavad võimalused testide kirjutamisel

Kasutatav testimise raamistik pakub rakenduse kasutajaliidese automaatseks testimiseks mitmeid võimalusi. Peamised teststsenaariumid näevad ette veebilehtede vahelist liikumist ning lehtedel asuvate elementide olemasolu kontrollimist. Raamistik võimaldab lehel asuvate elementidega manipuleerimist, näiteks tekstiväljadele kirjutamist, raadionuppude seadmist, „linnukeste“ (*checkbox*) seadmist, nuppude vajutamist, linkidele vajutamist, nimekirjadest (*select list*) valikute tegemist jne.

Teste alustatakse tavaliselt eelmise testi järelt „koristamisega“ (näiteks, kui kasutaja ei ole rakendusest välja loginud, peaks seda tegema). Seejärel liigutakse testitavale veebilehele ning kontrollitakse lehe teatud sisuelementide olemasolu. Kui element on olemas, siis lehe funktsionaalsus võib eeldada ka sellega manipuleerimist. Näiteks tekstivälju peab saama täita ning nuppudele või linkidele vajutamine pea viima järgmisele (alam)lehele. Testid on

[25] Vt lisaks Ruby oci8 tutvustus http://ruby-oci8.rubyforge.org/en/ (22.05.2015).
[26] Vt lisaks Ruby-plsql-spec https://github.com/rsim/ruby-plsql-spec (22.05.2015).

tüüpiliselt ehitatud üles kasutajalugude järgi: näiteks peab klient saama sõlmida kliendilepingut, tellida teenust, maksta arveid jne.

Üksiktestide koodinäited asuvad bakalaureusetöö lisas.

4.4 *Jenkins*

Jenkins on pideva integratsiooni keskkond, mis võimaldab lisaks rakenduste pakkide ehitamisele ning serveritele paigaldamisele lihtsalt ja mugavalt konfigureerida ja käivitada erinevaid automaatseid tegevusi. Antud bakalaureusetöös kirjeldatava süsteemi kontekstis on *Jenkins* kasutusele võetud kasutajaliidese testide jooksutamiseks. Testtsükli töökäsu konfiguratsioon asub bakalaureusetöö lisas.

Jenkins'i testide käivitamise töökäsk kasutab *Slave Agent* funktsionaalsust, millega ühendub virtuaalmasinaga. Funktsionaalsus on mugav peamiselt seetõttu, et võimaldab testtsükli käivitamist virtuaalmasinas ning tulemuste ja logide saatmist *Jenkins*'i kesksüsteemi.

J*enkins*'i SCM *plugin* võimaldab konfigureerida koodi laadimist virtuaalmasinasse. Iga 15 minuti tagant päritakse koodi repositooriumist, kas lähtekoodi on uuendatud. Kui koodi on uuendatud, siis laaditakse virtuaalmasinasse koodi viimane seis ning käivitatakse testtsükkel.

Enne koodi virtuaalmasinasse laadimist hoolitseb töökäsk selle eest, et võimaliku teadmata põhjusel pooleli jäänud testtsükli järelt on „koristatud". Kui millegipärast on virtuaalmasinas käima jäänud testides kasutatav veebilehitseja või leidub *Ruby* või *Java* protsesse, siis need lõpetatakse. Vastasel juhul võib tekkida olukord, kus rakenduse koodi ei ole võimalik virtuaalmasinal uuendada mõne soovimatult jooksma jäänud protsessi tõttu.

Seejärel käivitab *Jenkins*'i töökäsk virtuaalmasinal start_testing.cmd faili, mille argumendiks testtsükli nimi. Tegemist on lihtsa skriptiga, mis esmalt käivitab *ant*[27] käsklusega veebiserveri. Serverit käivitades võetakse (alternatiivset konfiguratsiooni kasutades) arvesse asjaolu, et kohalikus masinas simuleeritavate välisteenuste URL-id tuleb vahetada *localhost*'i vastu. Kui server on käivitatud, käivitatakse veebiserveri lõplikku tööleminekut kontrolliv *Ruby* skript wait_for_server.rb. Skripti töö lõppedes on veebiserver üles seatud ning käivitatakse testtsükkel.

[27] Vt lisaks Apache ant http://ant.apache.org/ (22.05.2015).

Start_testing.cmd skriptis või *Jenkins*'is on võimalik konfigureerida serveri logifailide kopeerimine kausta, mis saadetakse virtuaalmasinast *Jenkins*'i kesksüsteemi. Antud süsteemi puhul on kasutatud selleks *xcopy* käsklust. Skript asub bakalaureusetöö lisas.

Testtsükli tulemusel loodud raportid, ekraanitõmmised ning logid saadetakse *Slave Agent*'i poolt tagasi Jenkins kesksüsteemi. *Jenkins* lisab tulemused testtsükli töökäsu juurde, et neid oleks hiljem mugav analüüsida.

4.5 Virtuaalmasin

Testide jooksutamiseks kasutatakse *Windows Server 2008* virtuaalmasinaid. Virtualiseerimise tarkvarana on kasutusel *VMWare*. Virtuaalmasinatega saab ühenduda *Remote Desktop Connection*'it kasutades. Alternatiivina on töös ka *TightVNC* tunneldamistarkvara. Süsteem on sedavõrd automatiseeritud, et üldiselt ei ole vaja virtuaalmasinaga ühenduda. Võimalust kasutati süsteemi algsel konfigureerimisel.

Teste jooksutavad virtuaalmasinad peavad olema võimalikult stabiilsed. See tähendab, et nad peavad olema alati töökorras ja võimalike vigade korral suutma taastada testide jaoks sobiva algseisu.

Virtuaalmasinasse sisselogimine on automatiseeritud. Kui masin peaks mingil põhjusel taaskäivituma, siis logitakse automaatselt korrektse kasutajatunnuse ja parooliga sisse.

Edukal sisselogimisel käivitatakse *Jenkins Slave Agent*, mis loob ühenduse *Jenkins*'i keskserveriga. Kui ühendus keskserveriga on loodud, siis omab *Jenkins* täielikku kontrolli virtuaalmasina üle, võimaldades käivitada testtsükleid sobival hetkel.

Virtuaalmasina stabiilse töö tagamiseks on loobutud *Windows*'i automaatsetest uuendustest. Vajadusel saab uuendust siiski manuaalselt käivitada.

5. Kokkuvõte

Kasutajaliidese automaatne testimine on rakenduse pideva arengu ning aastate pikkuse eluea kontekstis püsiva kvaliteedi tagamiseks hädavajalik lahendus. Automaatsete testidega on pea täielikult võimalik välja vahetada manuaalne regressiooni testimine. Bakalaureusetöös kirjeldatud süsteem võimaldab tagada järjepidevat rakenduse kvaliteeti. Programmeerimisel tehtud vead tulevad välja kiiresti ning reaalseid arenduskeskkonna andmebaase kasutavad testid märkavad vigu ka nendes.

Automaatne kasutajaliidese testimine on pidev protsess. Testraporteid tuleb pidevalt jälgida ning analüüsida tekkinud vigu. Eesmärgiks on vigade võimalikult kiire likvideerimine ehk tagasiside programmeerijale.

Peamise õppetunnina võib käesolevast kirjatööst kaasa võtta asjaolu, et kasutajaliidese testimine peaks keskenduma testitavale rakendusele. Sõltuvused teistest rakendustest ja andmebaasidest tuleks elimineerida ehk kasutada võimalikult palju testandmete simuleerimist. Veebirakenduse poolt kasutatavad teenused peaksid oma kvaliteeti ise tagama ning nende testimiseks on olemas efektiivsemad võimalused, selle asemel, et seda teha neid kasutava rakenduse kasutajaliideses. Kindlasti tuleb tähelepanu pöörata rakenduste vahelisele integratsioonile, kuid seda tuleks teha võimalikult minimaalses mahus.

Teise õppetunnina tuleb märkida, et rakendus ning selle testid peaksid olema kirjutatud samas programmeerimiskeeles. Seda põhjusel, et arendajal oleks võimalikult lihtne oma arendusele teste kirjutada. Käesolevas töös on rakendus kirjutatud *Java*'s ning testid *Ruby*'s, mistõttu mõned programmeerijad on pidanud õppima selgeks uue programmeerimiskeele. Lisaks võimaldavad samas keeles kirjutatud rakendus ja testid jagada erinevatel testitasemetel kasutatavaid teeke. Näiteks ühik- ja kasutajaliidese testide ühisosa võiks olla teste grupeeriv ning jooksutav teek *JUnit* või *TestNG*. Autor soovitab kasutusele võtta *Selenium*/*Selenide* baasil Java's kirjutatava kasutajaliidese testide raamistiku, kus teste jooksutab sama teek, mis ühikteste ning välisteenuste simuleerimisel kasutusel näiteks *MockServer*'i nimeline teek.

6. Kasutatud kirjandus

A. Kozlov, „Miks scrum?“ http://www.scrum.ee/miks-scrum (22.05.2015).

A. Kull, „Software testing automation course in Tallin University of Technology supported by Elvior,“ http://www.elvior.com/news-and-blog/news/automated-test (22.05.2015).

A. Maurya, „Minimum viable product. Race to deliver customer value,“ http://leanstack.com/minimum-viable-product/ (22.05.2015).

Agile Methodology, http://agilemethodology.org/ (22.05.2015).

BDDWiki: BehaviourDrivenDevelopment, http://behaviourdriven.org/ (22.05.2015).

Bundler, „What is Bundler,“ http://bundler.io/ (22.05.2015).

D. Chelimsky et al, The R-Spec Book: Behaviour-Driven Development with R-Spec, Cucumber and Friends (The Pragmatic Programmers, 2010)

D. Radigan, „„A brief introduction to kanban or, what software makers can learn from Japanese manufacturing,“ https://www.atlassian.com/agile/kanban (22.05.2015).

E. Anuff, „API-Centric Architecture: All Development is API Development,“ https://blog.apigee.com/detail/api_centric_architecture_all_development_is_api_development (22.05.2015).

E. Hendrickson,. Agile Testing. „Nine Principles and Six Concrete Practices for Testing on Agile Teams,“ http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf (22.05.2015).

https://blog.apigee.com/detail/api_centric_architecture_all_development_is_api_development (22.05.2015).

I. Petuhhov, „Koskmudel,“ http://www.e-uni.ee/e-kursused/eucip/arendus/1221_koskmudel.html (22.05.2015).

J. Pertman, „Watir/watir-rspec,“ https://github.com/watir/watir-rspec (22.05.2015).

K. Kawaguchi, „Meet Jenkins,“ https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (22.05.2015).

M. Kalmo, „Automaattestimine 2014," http://enos.itcollege.ee/~mkalmo/ (22.05.2015).

M. Mikkelsaar, „Kasutajaliidese testid programmeerimisoskuse hindamiseks,"
Bakalaureusetöö, Tallinna Ülikool, Informaatika Instituut, 2014,
http://www.cs.tlu.ee/teemad/get_file.php?id=313 (22.05.2015).

M. Pällin, „Automaattestimisvahendite kasutus ning praktiline ülevaade Seleniumi näitel,"
Bakalaureusetöö, Tartu ülikool, Matemaatika-informaatikateaduskond, 2012,
http://comserv.cs.ut.ee/forms/ati_report/downloader.php?file=ABD4559968E0DC88727D55
B95D260E9CA295F3DC (22.05.2015).

S. Sergejev, „Funktsionaalsete automaattestide loomine uue kõrgkoolide vahelise
õppeinfosüsteemi jaoks Selenium raamistiku näitel," Diplomitöö, Eesti Infotehnoloogia
Kolledž, 2008, http://enos.itcollege.ee/~ssergeje/diploma/SergejevDiploma.pdf (22.05.2015).

T. Karming, „Elioni automaattestimise raamistiku täiendamine," Bakalaureusetöö. Tallinna
Tehnikaülikool, 2011.

Watir WebDriver, http://watirwebdriver.com/ (22.05.2015).

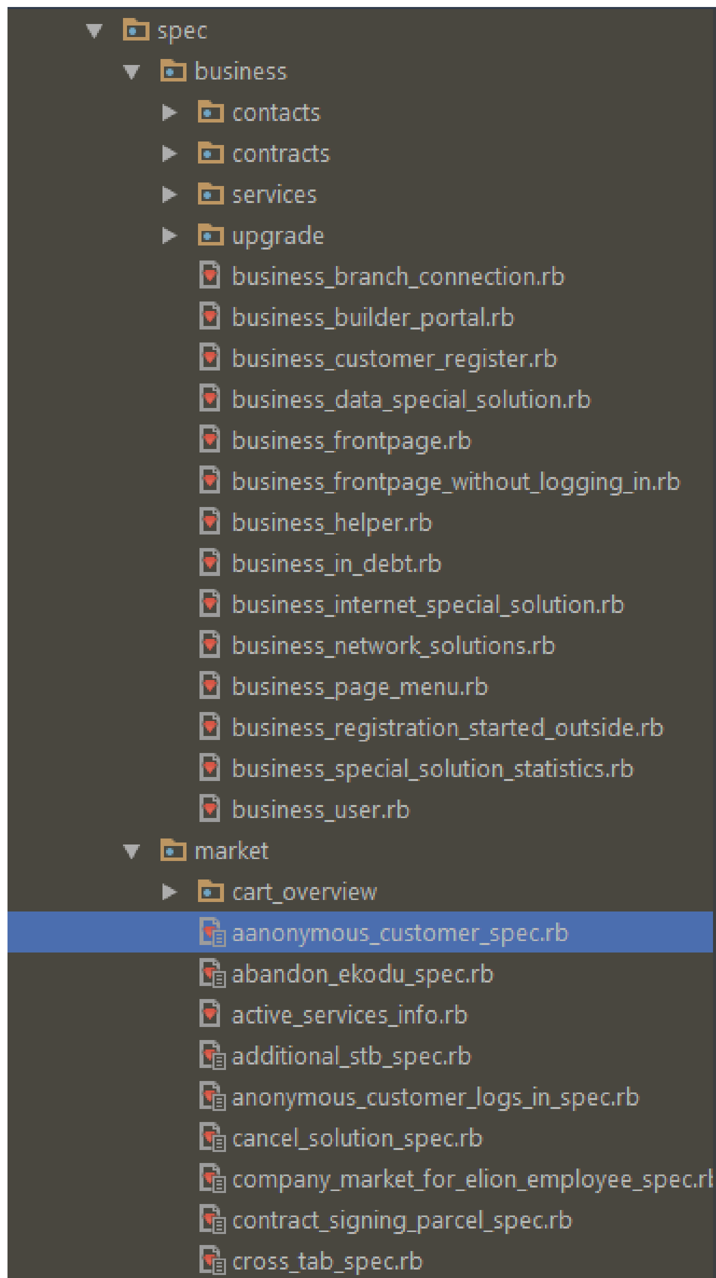
Watir, „Automated testing that doesn't hurt," http://watir.com/ (22.05.2015).

7. Lisad

Testitav rakendus - www.elion.ee iseteenindus

Joonis 2. www.elion.ee iseteeninduse ekraanitõmmis

Testide jagunemine



Joonis 3. Testide jagunemine. Allikas: www.elion.ee iseteeninduse testide lähtekoodi failide ekraanitõmmis.

Jenkins'i töökäsu konfiguratsioon

Source Code Management

- None
- CVS
- CVS Projectset
- Clone Workspace
- Git

Repositories — Repository URL: ; Credentials: ; Add

Branches to build — Branch Specifier (blank for 'any'): master

Repository browser: (Auto)

Additional Behaviours: Wipe out repository & force clone

Add

- Mercurial
- Subversion

Build Triggers

- Trigger builds remotely (e.g., from scripts)
- Build after other projects are built
- Build periodically
- Poll SCM

Schedule: H/15 * * * *

Would last have run at Sunday, May 17, 2015 3:00:27 PM EEST; would next run at Sunday, May 17, 2015 3:00:27 PM EEST.

Ignore post-commit hooks

Build Environment

- Abort the build if it's stuck

Time-out strategy: Absolute

Timeout minutes: 300

Time-out variable:

Set a build timeout environment variable

Time-out actions: Abort the build

Add action

- Assign unique TCP ports to avoid collisions
- Generate Release Notes
- Provide Node & npm bin/ folder to PATH
- Run buildstep before SCM runs

Execute Windows batch command

Command

```
tskill java
tskill ruby
tskill rubyw
tskill iexplore
```

See the list of available environment variables

Build

Execute Windows batch command

Command

```
ui-test-dev\scripts\start_testing.cmd ui-test-market
```

See the list of available environment variables

Execute Windows batch command

Command

```
xcopy C:\jenkins\workspace\web3-ui-test-market\logs\general.log C:\jenkins\workspace\web3-ui-test-market\ui-test-dev\results
```

See the list of available environment variables

Add build step

Post-build Actions

Archive the artifacts

Files to archive: ui-test-dev/results/**/*

Publish JUnit test result report

Test report XMLs: **/results.xml

Did not manage to validate **/results.xml (may be too slow)

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.

Retain long standard output/error

Health report amplification factor: 1.0

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Add post-build action

Save Apply

Joonis 4. *Jenkins*'i töökäsu konfiguratsioon. Allikas: www.elion.ee Jenkins keskserveri töökäsu konfiguratsiooni ekraanitõmmis.

Testitsüklite käivitamine

```
set hudson=1
set IGNORE_HOLD=1

tskill rubyw
tskill ruby
tskill java
RMDIR C:\Users\hudson\AppData\Local\Temp /S /Q

cd C:/jenkins/workspace/web3-%1
cmd /C ant run-ui-test -Dweb3.env=ui-test-zorro2 -Djetty.port=9091 -Djetty.ssl.port=9092

ruby c:/jenkins/workspace/web3-%1/ui-test-dev/scripts/wait_for_server.rb
cd c:/jenkins/workspace/web3-%1/ui-test-dev
cmd /C bundle install

IF %1 == ui-test-market (cmd /C bundle exec rspec -I . -O scripts/market.rspec spec/market/**/*spec.rb
& cmd /C bundle exec rspec -I . -O scripts/market_aux.rspec spec/market_aux/*.rb)
IF %1 == ui-test-business (cmd /C bundle exec rspec -I . -O scripts/business.rspec spec/business/**/*.rb)
IF %1 == megatests (cmd /C bundle exec rspec -I . -O scripts/megatests.rspec spec/megatests/*.rb)

IF %1 == ui-test-business (xcopy C:\jenkins\workspace\web3-%1\logs\general.log C:\jenkins\workspace\web3-%1\ui-test-dev\results)

tskill java
```

Joonis 5. Testitsüklite käivitamise skript start_testing.cmd. Allikas: www.elion.ee iseteeninduse testide lähtekoodi ekraanitõmmis.

Näidistesti analüüs #1

Watir teegi abil Ruby programmeerimiskeeles kirjutatud kasutajaliidese test keskendub Elioni teenuse minuTV tellimisprotsessile, mis on lahendatud lehel AngularJS rakendusena Elioni veebilehel.[28] Tellimisprotsess on äriliselt oluline komponent ning kasutajaliidese automaattest aitab tagada lahenduse toimimise rakenduse eluea jooksul.

Näidistestide analüüsile on lisatud mõningad kommentaarid. Testide kirjutamisel on autor lähtunud loetavusest ning arusaadavusest. Testid on ühtlasi testitava funktsionaalsuse dokumentatsioon.

Testifail mytv_order_spec.rb testib kaheksa testiga teenuse tellimise protsessi läbitavust ning kontrollib teatud elementide olemasolu erinevates sammudes.

```ruby
describe "Client on my tv ordering page" do # Terve testfaili
  include MarketHelper # Helper faili hõlmamine. Siin asuvad või on viidatud kõik(idele) selles testfailis kasutatavad meetodid.

  before :all do # Selles funktsioonis defineeritakse enne kõiki teste tehtav tegevus (test setup).

    # Näitetesti lihtsustamise huvides on siin olevad api-teenuse mockid eemaldatud. Mockimine on kaetud eraldi peatükis.

    logout # Kuna test on suurema testtsükli osa, siis tuleb eelneva testi järgselt kasutaja välja logida.
    goto_page '/eraklient/nutitv/minutv' # Internetilehitseja suunamine soovitud lehele.
    login '34711010288' # Sisselogimine teatud kasutajaga. Ainult sisselogitud kasutaja võib minuTV teenust tellida.
  end

  it "should see correct texts" do # Esimene testijuhtum.
    # Test kontrollib erinevate lehe sektsioonide olemasolu ning neis sisalduvaid tekste. Leht peab olema laetud 15 sekundi jooksul (time_out_short).
    # Tekste kontrollitakse andmebaasis olevate tekstidega võrreldes.
    hero_section.should be_present.within(time_out_short)
    h1(:id => 'mytvBannerTextDesktop').should be_present.within(time_out_short)
    hero_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'heroBannerWithRecordingDiscount')
    first_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'orderingFirstBannerNew')
    second_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'orderingSecondBannerNew')
    third_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'myTvChannelsDescription')
    fourth_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'orderingFourthBannerNew')
    fifth_section.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'orderingFifthBannerNew')
    order_section.text.should include fetch_market_label('portlets.tv.mtv-order', 'myTvAdditionalInformation')
    order_section.text.should include fetch_market_label('portlets.tv.mtv-order', 'viewingLocationsAdditionalInformation')
  end

  it "should see buttons to apps" do # Teine testijuhtum.
    # Test kontrollib linkide olemasolu lehe neljandas sektsioonis
    fourth_section.link(:href => /www.minutv.ee/).should be_present.within(time_out_short)
    fourth_section.link(:href => "https://play.google.com/store").should be_present.within(time_out_short)
    fourth_section.link(:href => "https://www.apple.com/itunes/").should be_present.within(time_out_short)
  end

  it "can see and click buttons to scroll to ordering section" do # Kolmas testijuhtum.
    # Test kontrollib nuppude olemasolu, proovides neid vajutada.
    hero_section.link(:class => "btn btn-primary banner-btn").click
    first_section.link(:class => "btn btn-primary").click
  end

  it "should see five offering icons" do # Neljas testijuhtum.
    # Test kontrollib viie erineva pakkumise olemasolu. Kontrollitakse, et poleks kuuendat pakkumist.
    order_section.li(:class => /selectBlock/, :index => 0).should be_present
    order_section.li(:class => /selectBlock/, :index => 1).should be_present
    order_section.li(:class => /selectBlock/, :index => 2).should be_present
    order_section.li(:class => /selectBlock/, :index => 3).should be_present
    order_section.li(:class => /selectBlock/, :index => 4).should be_present
    order_section.li(:class => /selectBlock/, :index => 5).should_not be_present
  end
```

[28] https://www.elion.ee/eraklient/nutitv/minutv#/

```
it "should be able to order my tv with 2 viewing places" do # Viies testijuhtum.
  # Test vajutab teenuse tellimise nupule ning selle tulemusel peaks 15 sekundi jooksul (time_out_short) ilmuma teenuse tellimise kinnitamise samm.
  order_section.li(:class => /selectBlock/, :index => 1).link(:class => "btn btn-primary proceedButton myTvOrderRow ng-scope").click
  div(:id => "contractConclusionContainer").should be_present.within(time_out_short)
end

it "should see contact data page" do # Kuues testijuhtum.
  # Test
  div(:class => 'registrationPage').should be_present.within(time_out_short)
  button(:id => 'register_continue').should be_present.within(time_out_short)
  sleep 3
  button(:id => 'register_continue').click
end

it "should see correct information in order confirmation step" do # Seitsmes testijuhtum.
  sleep 3
  div(:id => 'productChoices').should be_present.within(time_out_short)
  div(:id => 'productChoices').text.should include "Valitud teenused\r\n\r\nKuutasu\r\n\r\nLisainfo\r\n\r\n minuTVKasutaja5,00 \u20AC Salvestamise kliendina
  on sinu jaoks kuutasu 0,00 \u20AC/kuu \r\n minuTVLisaekraan3,00 \u20AC  3,00 \u20AC/kuu \r\n minuTVLisaekraan3,00 \u20AC
  3,00 \u20AC/kuu \r\n \r\n\r\nKokku\r\n\r\n11,00 \u20AC/kuu\r\n\r\nSoodushind sinule: 6,00 \u20AC/kuu"
end

it "should accept terms and conditions before being able to order" do # Kaheksas testijuhtum.
  terms_and_conditions.should be_present
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'mustAgreeWithMyTvConditions')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'elionTermsAndConditions')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'conditionsTextGeneralTerms')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'accountingFeesInfoText')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'mustAgreeWithMyTvConditions')
  terms_and_conditions.checkbox(:id => 'acceptOrder').should_not be_set
  terms_and_conditions.checkbox(:id => 'acceptTermsAndConditions').should_not be_set
  button(:class => /confirmBtn/).should be_disabled
  terms_and_conditions.checkbox(:id => 'acceptOrder').set
  terms_and_conditions.checkbox(:id => 'acceptTermsAndConditions').set
  button(:class => /confirmBtn/).should_not be_disabled
end

# Muutujate defineerimine. Koodi kirjustamise lihtustamiseks on korduvkasutatavad elemendid muutujatena defineeritud.

def hero_section
  div(:id => 'heroBanner')
end

def first_section
  div(:id => 'orderingFirstBannerNew')
end

def second_section
  div(:id => 'orderingSecondBannerNew')
end
```

```
  def third_section
    div(:id => 'myTvChannelsDescription')
  end

  def fourth_section
    div(:id => 'orderingFourthBannerNew')
  end

  def fifth_section
    div(:id => 'orderingFifthBannerNew')
  end

  def order_section
    div(:id => 'mtvOrderBeginning')
  end

  def terms_and_conditions
    div(:class => 'border ui-corner-all padding-20 lightGreyBg')
  end
end
```

Joonis 6. Näidistest 1. Allikas: www.elion.ee iseteeninduse testide lähtekoodi ekraanitõmmis.

Näidistesti analüüs #2

Testifail customer_contract_signing_spec.rb testib erakliendi kliendilepingu sõlmimise protsessi – kontaktandmete salvestamist ning lepingu sõlmimist.

```ruby
describe 'Client on customer contract page' do
  include MarketHelper

  before :all do # Enne kõiki teste genereeritakse andmebaasi
                 # uus klient ning logitakse temaga sisse veebirakendusse.
    logout
    customer_name = rand.to_s[2..11]
    @client = ArgosHelper::Client.generate_customer_without_clients_contract customer_name
    ArgosHelper::Client.remove_all_from_customer @client.personal_code

    execute_script "$.cookie('JSESSIONID', null, {path:'/'})"
    execute_script "$.cookie('WEBSESSIONID', null, {path:'/'})"''

    login @client.personal_code
    goto_page 'eraklient/iseteenindus/minu-andmed'
  end

  # Test verifitseerib andmebaasis olevate andmete korrektset kuvamist veebirakenduses. Samasugused
  # andmed genereeritakse iga kord enne testi selle faili alguses.
  it 'can see contact data information fetched from database' do
    div(:class => 'registrationPage').should be_present.within(time_out_short)
    top_div = div(:class => 'topBlock ui-corner-top border-left border-top border-right')
    top_div.should be_present
    top_div.text.gsub(/\r\n|\r|\n/,'').should match /Nimi:TEST \d+ Isikukood:\d+/
    div(:class => 'ui-corner-bottom border-left border-right border-bottom').should be_present
    f.div(:index => 1).text.gsub(/\r\n|\r|\n/,'').should match /Aadress/
    f.div(:index => 2).text_field(:name => 'contactAddressQuery', :value => /AASA TN 7, TALLINN/).should be_present
    f.div(:index => 4).text.gsub(/\r\n|\r|\n/,'').should match /E-post:/
    f.div(:index => 5).text_field(:id => 'customerEmail', :value => 'test2@elion.ee').should be_present
    f.div(:index => 7).text.gsub(/\r\n|\r|\n/,'').should match /Mobiil:/
    f.div(:index => 8).text_field(:id => 'customerMobile', :value => '53737933').should be_present
    f.div(:index => 10).text.gsub(/\r\n|\r|\n/,'').should match /Telefon:/
    f.div(:index => 11).text_field(:id => 'customerPhone', :value => '6402888').should be_present
    f.div(:index => 13).text.gsub(/\r\n|\r|\n/,'').should match /Keel:/
    execute_javascript("document.getElementById('customerLanguage').style.display='block';")
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('eesti').should eq true
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('vene').should eq true
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('soome').should eq true
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('inglise').should eq true
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('saksa').should eq true
    f.div(:index => 14).select_list(:id => 'customerLanguage').include?('klingon').should eq false
    f.div(:index => 14).select_list(:id => 'customerLanguage').select_value '1'
  end

  # Test jätkab eelmise testi lõpust ning verifitseerib turunduslike andmete raadionuppude
  # olemasolu. Test proovib nuppe seada ning kontrollida nuppude seadmist.
  it 'can see marketing information radio buttons' do
    div(:class => 'border ui-corner-all block').should be_present.within(time_out_short)
    radio(:id => 'allowMarketingRadioButton').should be_present.within(time_out_short)
    not radio(:id => 'allowMarketingRadioButton').set?
    radio(:id => 'doNotAllowMarketingRadioButton').should be_present.within(time_out_short)
    radio(:id => 'doNotAllowMarketingRadioButton').set?
    radio(:id => 'allowMarketingRadioButton').set
    radio(:id => 'allowMarketingRadioButton').set?
  end
```

```ruby
it "should be able to order my tv with 2 viewing places" do # Viies testijuhtum.
  # Test vajutab teenuse tellimise nupule ning selle tulemusel peaks 15 sekundi jooksul (time_out_short) ilmuma teenuse tellimise kinnitamise samm.
  order_section.li(:class => /selectBlock/, :index => 1).link(:class => "btn btn-primary proceedButton myTvOrderRow ng-scope").click
  div(:id => "contractConclusionContainer").should be_present.within(time_out_short)
end

it "should see contact data page" do # Kuues testijuhtum.
  # Test
  div(:class => 'registrationPage').should be_present.within(time_out_short)
  button(:id => 'register_continue').should be_present.within(time_out_short)
  sleep 3
  button(:id => 'register_continue').click
end

it "should see correct information in order confirmation step" do # Seitsmes testijuhtum.
  sleep 3
  div(:id => 'productChoices').should be_present.within(time_out_short)
  div(:id => 'productChoices').text.should include "Valitud teenused\r\n\r\nKuutasu\r\n\r\nLisainfo\r\n\r\n minuTVKasutaja5,00 \u20AC Salvestamise kliendina
  on sinu jaoks kuutasu 0,00 \u20AC/kuu \r\n minuTVLisaekraan3,00 \u20AC  3,00 \u20AC/kuu \r\n minuTVLisaekraan3,00 \u20AC
  3,00 \u20AC/kuu \r\n \r\n\r\nKokku\r\n\r\n11,00 \u20AC/kuu\r\n\r\nSoodushind sinule: 6,00 \u20AC/kuu"
end

it "should accept terms and conditions before being able to order" do # Kaheksas testijuhtum.
  terms_and_conditions.should be_present
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'mustAgreeWithMyTvConditions')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'elionTermsAndConditions')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'conditionsTextGeneralTerms')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'accountingFeesInfoText')
  terms_and_conditions.text.gsub(/\r\n|\r|\n/,'').should include fetch_market_label('portlets.tv.mtv-order', 'mustAgreeWithMyTvConditions')
  terms_and_conditions.checkbox(:id => 'acceptOrder').should_not be_set
  terms_and_conditions.checkbox(:id => 'acceptTermsAndConditions').should_not be_set
  button(:class => /confirmBtn/).should be_disabled
  terms_and_conditions.checkbox(:id => 'acceptOrder').set
  terms_and_conditions.checkbox(:id => 'acceptTermsAndConditions').set
  button(:class => /confirmBtn/).should_not be_disabled
end

# Muutujate defineerimine. Koodi kirjustamise lihtustamiseks on korduvkasutatavad elemendid muutujatena defineeritud.

def hero_section
  div(:id => 'heroBanner')
end

def first_section
  div(:id => 'orderingFirstBannerNew')
end

def second_section
  div(:id => 'orderingSecondBannerNew')
end
```

Joonis 7. Näidistest 2. Allikas: www.elion.ee iseteeninduse testide lähtekoodi ekraanitõmmis.

Testtsükli raporti näide

RSpec Code Examples

☑ Passed ☑ Failed ☑ Pending

59 examples, 12 failures
Finished in **7170.69982 seconds**

Market for anonymous customer

should verify that personal data block doesn't exist. TMT testcase id: 01034 — 55.23152s

should verify that anonymous customer can only add a new address from the address selection list. TMT testcase id: 01035 — 8.99790s

eKodu abandon

should abandon 'eKodu' offer, Internet at first. (The main product will stay). — 135.81378s

```
timed out after 15 seconds

(eval):3:in `wait_until'
./spec/market/market_helper.rb:852:in `open_selected_product_dialog_and_click_button'
./spec/market/market_helper.rb:868:in `unselect_product_with_one_click'
./spec/market/abandon_ekodu_spec.rb:43:in `block (2 levels) in <top (required)>'

1 # Couldn't get snippet for (eval)
2 # gem install syntax to get syntax highlighting
```

HTML Screenshot

Additional STB-s

should add 3 STB-s (aasa 1) — 223.14831s

Market for anonymous customer after login

displays existing eKodu on selected address. TMT testcase id: 01304 — 92.76247s

keeps new selections if no existing eKodu on address. TMT testcase id: 01305 — 84.78548s

Solution cancel button

should remove all selected products from cart (Regress). TMT testcase id: 01639 — 198.20602s

Product conditions for anonymous user

should appear in conclusion step. TMT testcase id: 01563 — 334.64005s

```
timed out after 120 seconds

(eval):3:in `wait_until'
./spec/market/cart_overview/anonymous_user_product_conditions_spec.rb:82:in `fill_device_selection'
./spec/market/cart_overview/anonymous_user_product_conditions_spec.rb:55:in `block (2 levels) in <top (requir

1 # Couldn't get snippet for (eval)
2 # gem install syntax to get syntax highlighting
```

HTML Screenshot

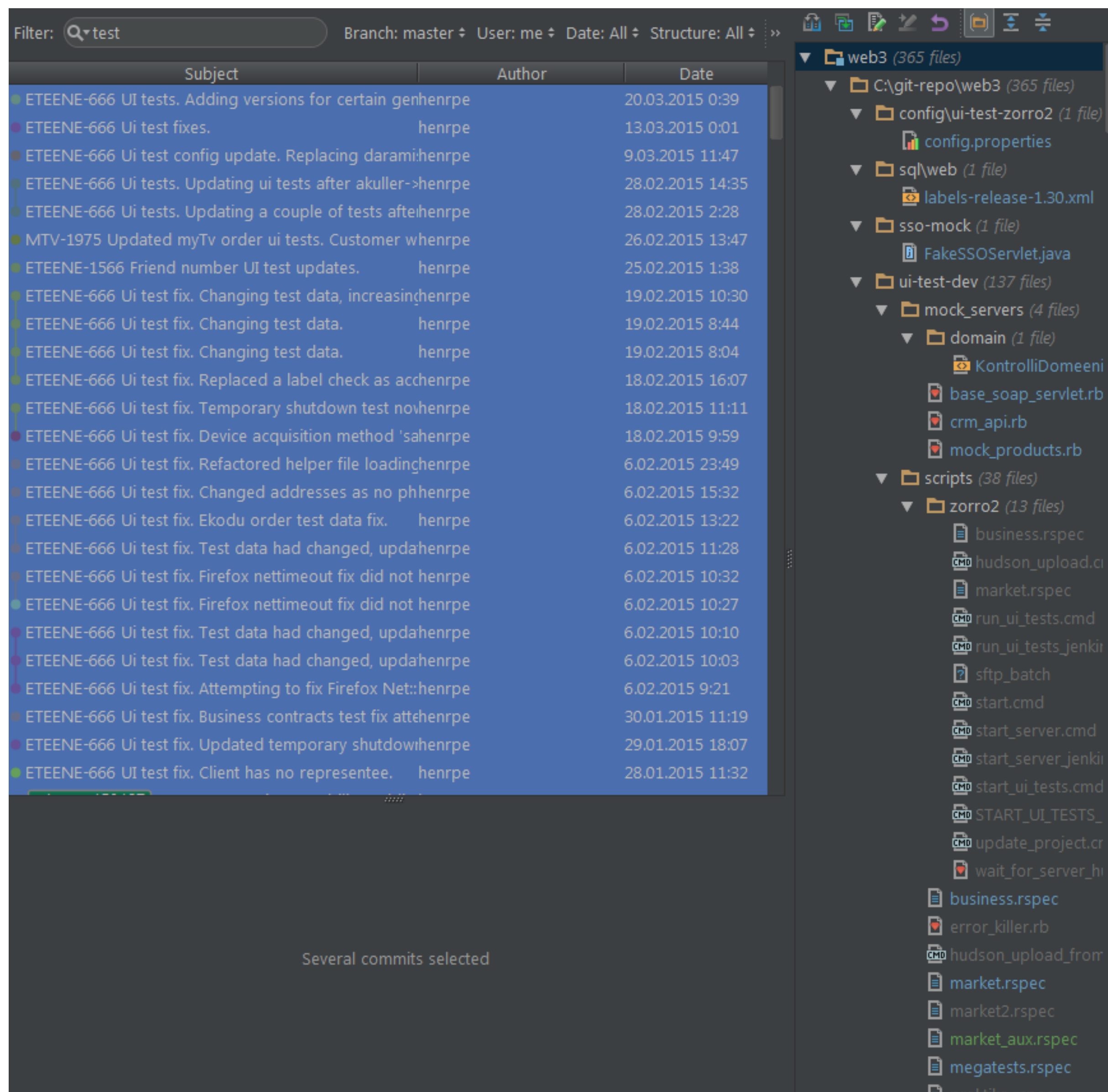
Cart overview flow contract contact data

should check invoice channel 'email' fields and checkboxes — 8.68726s

should check invoice channel 'bank' fields and checkboxes — 16.35273s

Joonis 8. Testtsükli raport. Allikas: www.elion.ee Jenkins keskserveri töökäsu testraporti ekraanitõmmis.

Osa autori poolt tehtud koodimuudatustest



Joonis 9. Koodimuudatused. Allikas: www.elion.ee iseteeninduse testide lähtekoodi muudatuste ekraanitõmmis.

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Henri Perkmann,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

„Veebirakenduse kasutajaliidese automaatne testimine väleda arendusprotsessi kontekstis – põhimõtted ja implementatsioon“,

mille juhendaja on Vambola Leping,

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **22.05.2015**