T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Infotehnoloogia eriala

Arli Türk

Kasutajaliidesepõhiste automatiseeritud regressioontestide komplekti loomine

Bakalaureusetöö (6EAP)

Juhendajad: dotsent Helle Hein

Artur Assor, AS Nortal

Autor: ……..... "….." mai 2013

Juhendaja: ……..... "….." mai 2013

Juhendaja: ……..... "….." mai 2013

Lubada kaitsmisele

Professor : ..... "….." mai 2013

TARTU 2013

Sisukord

Sissejuhatus

Loodava tarkvara keerukus kasvab pidevalt ning samuti kasvab ka selle osatähtsus meie elus, kuna üha enam valdkondi kasutab mingil moel arvutiprogramme, et tööd efektiivsemalt teha. Tähtis osa tarkvaraarenduses on testimisel, kuna loodavalt tarkvaralt oodatakse, et see töötaks selliselt, nagu algselt planeeritud oli, ning arvestatud oleks ka juhtudega, mille peale tellija ise ei tulnud. Eriti tähtis on testimine projektides, kus kaalul on inimelud. Valdkondades nagu näiteks lennundus ja meditsiin ei saa kvaliteedi osas järelandmisi teha. Seoses sellega kasvavad ka testimise mahud ning otsitakse lahendusi, kuidas selle väljakutsega paremini hakkama saada.

Käesoleva bakalaureusetöö eesmärgiks on luua kasutajaliidesepõhiste regressioontestide algne komplekt, mida tulevikus edasi arendada. Loodavad testid peaksid katma tähtsamad funktsionaalsused, et vähendada testimise mahtu ning kiirendada arendusprotsessi.

Antud töö esimeses peatükis antakse ülevaade regressioontestimisest ja testjuhtumite loomisest ning seletatakse, miks nad on testimise vajalikud osad. Samuti tutvustatakse lähenemist, mille abil saab määrata, millised testjuhtumid tuleks esimestena läbida, vältimaks tähtsate funktsionaalsuste testimata jätmist ajapuuduse tõttu. Teises peatükis tutvustatakse automaattestimist ning tuuakse välja juhiseid, mida võiks meeles pidada, et automatiseerimine edukalt kasutusele võtta. Kolmandas peatükis tutvustatakse Selenium tööriista Webdriver, mida kasutati automaattestide loomiseks antud töö raames. Testide loomine toimub Nortal AS ettevõtte projekti EAK (eArvekeskus) näitel. EAK on veebipõhine e-arvete operaator, mis tegeleb e-arvete saatmise, vastuvõtmise ning säilitamisega [32]. Rakenduse põhilisteks funktsionaalsusteks on müügiarvete loomine ja nende saatmine, ostuarvete sisestamine ning nende vastuvõtmine ja ostuarvete kinnitamine erinevate isikute poolt.

Lisadena on kaasas lõputöö raames EAK projekti jaoks loodud testjuhtumid, funktsionaalsuste nimekiri ning automaattestid koos lähtekoodiga.

1. Regressioontestimine

1.1 Vajadus

Regressioontestimise eesmärk on peale lähtekoodi täiendamist üle kontrollida, kas uute arenduste või parandustega ei tekitatud tahtmatuid muudatusi rakenduse vanas funktsionaalsuses. Tahtmatu muudatus võib olla uute vigade tekitamine või ka kunagiste vigade paranduste muutmise tõttu nende uuesti esile kutsumine. Põhilised tegevused, mille tulemusena olemasolevat koodi muudetakse ja peale mida oleks regressioontestimine vaja läbi viia, on olemasoleva funktsionaalsuse täiendamine, uute funktsionaalsuste lisamine, vigade parandamine ja rakenduse optimeerimine. Korralik regressioontestimine tagab nii tellija kui ka arendaja poolele kindluse, et uute arenduste tellimisel jääb vana funktsionaalsus korralikult tööle.

Reaalseks näiteks võib tuua olukorra, kus mingi süsteemi administraatori komponendile, mis võimaldab kasutajatele ettevõtetesse esindusõigusi anda, tehakse täiendus, et administraator saaks kasutajale eriõiguseid anda. Tahtmatult lisatakse see funktsionaalsus aga ka tavakasutaja komponendile, mille kaudu on võimalik enda esindusõiguseid hallata. Tulemuseks on viga, mis võimaldab kasutajal endale ise eriõiguseid anda. Testides ainult seda komponenti, kuhu arendus algselt oli planeeritud, ei oleks viga üles leitud. Antud vea leidmiseks on vaja teostada regressioontestimist. Regressioontestimise osa tarnitavast versioonist illustreerib Joonis 1.

Joonis 1. Regressioontestimine.

1.2 Testjuhtumite loomine

Testjuhtum on dokument, mis kirjeldab sisendit, oodatavaid tulemusi ja tingimusi, mille raames mingit testi läbi viiakse [9]. Korralike testjuhtumite loomine aitab tulevikus sama funktsionaalsust efektiivsemalt testida, kuna planeerimise osa on ära tehtud ja kirja pandud ning järjekordsel testimisel saab tugineda loodud dokumendile. See on ka üks suuremaid eeliseid, mida testjuhtumite loomine annab: kirjutamise ajal tuleb põhjalikult läbi mõelda, kuidas oleks kõige parem antud funktsionaalsust testida, milliste andmetega testida, kus antud komponent kogu protsessis asub ja kuidas ta teisi komponente mõjutab ning mis tegevused kindlasti läbi peaks tegema [10]. Lisaks on loodud testjuhtumid väga kasulikud, kui projektiga liituvad uued testijad, kes saavad neid kasutada kui juhendeid komponentide testimiseks ning nendega tutvumiseks. Kuna antud testjuhtumeid hakkavad ilmselt kasutama ka teised inimesed peale nende looja, siis tasub kirjutamisel olla piisavalt detailne, kirjeldades samme, mida on vaja teha. Samas peaks testjuhtum olema nii lühike kui võimalik, et seda poleks väga raske järgida ning hooldamine ei nõuaks ülemäära palju tööd. Lõplik testjuhtumi ülesehitus ja detailsus sõltub sellest, kes seda kasutama hakkavad ning mis eesmärgil. Luues testjuhtumit ainult enda jaoks ning olles testitava rakendusega pikemat aega töötanud, ei ole vaja alati palju detaile sinna sisse panna – piisab näiteks protsesside diagrammist. Kui loodavate testjuhtumite sihtgrupp on tellija poolsed inimesed, siis peaks kirjeldus olema väga detailne, kuna enamasti näevad inimesed esimest korda uut funktsionaalsust ja detailne juhend kiirendab nendepoolset testimist.

Testjuhtumit kirjutades peaks mõtlema, millist informatsiooni on võimalik kasutada, saamaks teada, mis nõuetele testitav funktsionaalsus peab vastama. Lisaks tellija poolt kinnitatud nõuetele on võimalik kasutada ka kaudseid spetsifikatsioone, mis pole algselt kliendi poolt kinnitatud, aga omavad siiski autoriteeti. Sellised dokumendid omavad tavaliselt autoriteeti, kuna nende sisu on veenev ning usaldusväärne. Loomulikult tuleks selliste dokumentide põhjal tehtavad muudatused kliendiga kooskõlastada. Järgnevalt on toodud mõned näited kaudsetest spetsifikatsioonidest [4, lk 22-23]:

1) konkureerivad tooted;
2) seotud tooted;

3) sama toote vanemad versioonid;
4) arutelud, mida on peetud projekti siseselt e-maili või mõne muu suhtluskanali kaudu;
5) kasutajate kommentaarid;
6) raamatud ja artiklid, mis on teemaga seotud (näiteks raamatupidamise raamatut saab kasutada rakenduse puhul, mis tegeleb raamatupidamisega);
7) kasutajaliidese standardid;
8) operatsioonisüsteemi ühilduvuse nõuded;
9) testija enda kogemus.

Kui rakenduses leidub viga, mis rikub kliendi poolt kinnitatud dokumentatsiooni, siis on suhteliselt lihtne leitud viga raporteerida, kuna saab viidata ametlikule dokumendile, mida projektis kasutatakse. Leides aga probleemi, mis rikub kaudset spetsifikatsiooni, on raporteerimine raskem, sest tuleb rohkem põhjendada, miks antud parandus vajalik on. Sellisteks probleemideks on näiteks kasutajamugavusega seotud teemad, kus loodud rakenduses saab mingit tegevust teha teistmoodi kui üldlevinud viisil, aga kliendi poolt pole otsest nõuet ette antud [15].

Luues testjuhtumeid, peab arvestama sellega, kas antud testjuhtumit hakatakse läbima manuaalselt või see automatiseeritakse. Manuaalsel läbimisel ei tohiks ette anda liiga detailset informatsiooni selle kohta, mida tegema ja kontrollima peab. Testijale peaks jääma vabadus kasutada enda loomingulisust. Selline lähenemine aitab kaasa ka uute vigade leidmisele, kuna iga kord, kui testjuhtumit läbitakse, tehakse seda natukene teistmoodi kui eelmisel korral [4, lk 29]. Kui testjuhtum on mõeldud automatiseerimiseks, siis peaks see sisaldama võimalikult detailset informatsiooni sammude kohta, mida läbitakse ja mis on oodatav tulemus. Selline lähenemine teeb testi automatiseerija töö lihtsamaks ja valmib rohkem läbimõeldud test, sest tihti on automatiseerijaks programmeerija, mitte testija taustaga inimene.

Antud töö raames loodud testjuhtumid on mõeldud automatiseerimiseks, seega on üritatud võimalikult detailselt lahti kirjutada sammud, mida peab tegema ning mis on neile järgnevad oodatavad tulemused. Valminud testjuhtumid koosnevad järgmisest infost:

1) antud komponendi spetsifikatsiooni kood ning link antud spetsifikatsiooni asukohale;

2) eesmärgid, mis püstitatakse enne testjuhtumi sammude kirjeldamist, et oleks selge, mida antud testjuhtumiga kontrollida tahetakse;
3) nimekiri eeldustest, mis on vajalikud antud testjuhtumi läbimiseks;
4) nimekiri sammudest, mis on vaja läbida testjuhtumi raames. Iga sammu juures on kirjeldatud tema järjekorranumber, tegevus, testandmed, mida kasutatakse, oodatav tulemus ning lisaks veel kommentaari lahter juhuks, kui mingi sammuga tahetakse kaasa panna lisainformatsiooni (näiteks kui mingi samm vajab parandamist, aga paranduse tegemiseks pole koheselt aega).

Testjuhtumid on loodud kasutades Atlassian Confluence keskkonda. Tegemist on wiki tüüpi rakendusega, mis võimaldab veebilehitsejat kasutades luua keskkonda uusi lehekülgi või muuta olemasolevaid, lihtsustades seeläbi meeskonnasisese informatsiooni jagamist [11]. Testjuhtumid on Confluence keskkonnast eksporditud .docx formaati failidesse ning lisadena antud tööle kaasa pandud. Joonis 2 [2] illustreerib loodud testjuhtumite ehitust Confluence keskkonnas.

Dokumentatsioon :

KEK08

Eesmärgid :

1. Ilma õigete andmeteta ei saa keskkonda siseneda
2. Õigete andmetega saab edukalt rakendusse siseneda
3. Kasutajale avaneb oodatud funktsionaalsus ning puudub ligipääs üleliigsele funktsionaalsusele

	Eeldused sammude läbimiseks
1.	4. sammu eelduseks on, et infoteade peab 'Back-offic -> Teavitused' alt seadistatud olema.

	Tegevus	Testandmed	Oodatav tulemus	Kommentaar
1.	• Avada rakenduse esileht			
2.	• Vajutada nuppu "Sisene" (ilma kasutajanime ja parooli lahtrit väärtustamata)		• Kuvatakse kahte veateadet : 1) Väli 'Isikukood' on kohustuslik. 2) Väli 'Salasõna' on kohustuslik. • Ei tohi eksisteerida esindatava valikut.	
3.	• Väärtustada kasutajanime lahter	• Kasutajanimi :	• Kuvatakse veateade :"Sisselogimine	

Joonis 2. Testjuhtumi ülesehitus [2].

1.3 Regressioontestimise väljakutsed

Regressioontestimise põhiline eesmärk on võimalikult lühikese aja jooksul võimalikult palju kriitilisi vigu tarkvarast üles leida ja seda regressioontestimise tsüklis nii vara kui võimalik. Antud eesmärk toob endaga kaasa ka mõned väljakutsed [1].

1. Regressioontestimise maht kasvab uute funktsionaalsuste arendamisega. Kuna arenduse käigus lisatakse rakendustele aina rohkem funktsionaalsust, siis tähendab see ka regressioontestimise mahu kasvamist järgmises tsüklis. Selle tõttu võib aastaid aktiivselt arendatud projekti puhul regressioontestimise maht kasvada raskesti hallatavaks [1].
2. Regressioontestimine võtab palju ressursse. Tulenevalt sellest, et testimise maht kasvab iga tsükliga, tähendab see järjest suuremat ajalist ja seega ka rahalist kulu [1].
3. Regressioontestimine on korduv ja rutiinne töö. Kuna regressioontestimine tähendab varasemate arenduste pidevat ülevaatamist, siis tähendab see samade testide läbiviimist igas tsüklis. Siin on oht, et peale mitmeid tsükleid muutub antud töö testija jaoks igavaks ja vähendab seega ka motivatsiooni ning tähelepanu, mis on mõlemad vajalikud efektiivseks testimiseks [1].
4. Tähendusrikaste tulemuste saamine nõuab palju aega. Testide suur maht tähendab, et kõigi nende läbimine võtab palju aega ja seega võib kuluda päevi või nädalaid teada saamaks, kas antud versioonis eksisteerib suuri probleeme, mis vajavad lahendamist. Tihti tahetakse aga informatsiooni versiooni kvaliteedi kohta kiiremini, et vastu võtta otsust, kas versiooni kliendile tarnida või mitte.
5. Iga muudatus koodis võib tähendada uusi vigu. Vea leidmisel võib tunduda, et tegemist on isoleeritud kohaga ega mõjuta ülejäänud rakendust. Reaalsuses võib antud viga aga mõjutada paljusid rakenduse komponente, mille peale alguses ei tulda. See tähendab, et regressioontestimise käigus leitud vigade parandused võisid ära lõhkuda funktsionaalsuse komponentides, kus regressioontestimine juba läbi viidi. Seetõttu on vaja uuesti läbi viia ka varasemalt teostatud testid [1].
6. Tähtsamad testid teostatakse alles lõpus. Kuna regressioontestide maht võib olla väga suur, siis on võimalik, et osa tähtsamaid teste jäävad testimistsükli lõppu. Selle tõttu võivad aga kriitilised probleemid välja tulla liiga hilja ning jätta parandamiseks liiga

vähe aega [1]. Vähene aeg aga suurendab võimalust, et arendaja viib parandusega sisse uued vead, kuna tal pole aega lahendus korralikult läbi mõelda. Seetõttu tuleks läbi mõelda, millised testid on kõige kriitilisemad ja keskenduda esmalt neile.

EAK projektis on suurimaks probleemiks suur regressioontestimise maht, kuna projekti on arendatud alates 2009. aastast. Samuti on kogu selle aja testimine olnud ühe inimese vastutada, seega on regresioontestimine kujunenud üpris rutiinseks tööks. Esineb ka täiendusi arendustsüklite lõpu poole, mis tähendab, et rakendus on vaja kiirelt enne tarnet uuesti üle vaadata.

1.4 Tehnikad

Regressioontestimise läbiviimiseks on mitmeid tehnikaid, igaühel neist on omad eelised ja puudused ning seetõttu tasub lähenemine valida arvestades projekti ressursse ja riske. Näiteks tuleb arvestada, kas projekti jaoks on tavalised ootamatud täiendused, mida tellija soovib saada kiirelt toodangukeskkonda, või on projektis rangemad reeglid, mis nõuavad täienduste sisseviimisel pikemat etteteatamist tellija poolt. Samuti tuleb arvestada sellega, millistel tingimustel on tarnijal õigus tarnekuupäeva edasi lükata, kui peaks aega regressioontestimiseks liiga vähe jääma. Järgnevalt on toodud mõned regressioontestimise tehnikad:

1. Täielik regressioontestimine (*The complete regression test*) - antud lähenemise raames läbitakse kõik olemasolevad regressioontestid uuesti ja seda iga uue versiooni puhul. Kui regressioontestimise käigus leitakse viga ning parandusega luuakse uus versioon, siis kõigepealt lõpetatakse regressioontestimine vanale versioonile ning peale seda alustatakse algusest peale uue versiooni testimist [1]. Vana versiooni lõpuni testimise eelis on see, et jõutakse käivitada kõik testid ja seetõttu leitakse viimaste testidega avastatavad probleemid võimalikult vara. Selline lähenemine on väga suure ajakuluga, kuid põhjalik ja sobib seetõttu projektidele, kus on kõrged riskid (näiteks oht inimelule).
2. Uuesti algav regressioontestimine (*The restartable regression test*) – erinevalt täielikust regressioontestimisest, alustatakse regressioontestimisega algusest peale, kui

paigaldatakse uus versioon. Enamasti alustatakse testimist algusest peale siis, kui arvatakse, et valmimas on viimane versioon. Juhul kui avastatakse suuremat sorti probleeme, peab testimist jälle algusest peale alustama. Probleemiks on asjaolu, et mingi arv teste võidaks jõuda läbida alles viimaste versioonide juures [1].

3. Jätkuv regressioontestimine (*The continuous regression test*) - antud juhul ei katkestata testimist, kui uus versioon välja tuleb. Peale versiooni valmimist pannakse see testkeskkonda ning jätkatakse testide läbimist sealt, kus eelmise versiooni juures pooleli jäädi. Selline käitumine hoiab aega kokku, aga lõpptulemusena pole ükski versioon täielikult läbi testitud, sest regressioontestid on hajutatud mitme versioonide peale [1].
4. Regressioon suitsutestimine (*The regression smoke test*) – ühe lähenemisena tehakse uutele versioonidele ainult pinnapealsed regressioontestid, mis tavaliselt võtavad paar tundi aega (täpsem ajakulu oleneb projektist). Viimasele versioonile tehakse aga põhjalikum testimine, mille käigus läbitakse kõik testid. Probleemiks on asjaolu, et kunagi ei saa täiesti kindlalt ette teada, milline versioon on viimane, sest testimise käigus võib välja tulla vigu, mis nõuavad uuesti täieliku regressioontestimist. Samuti võib juhtuda, et kriitilised vead tulevad välja alles lõpus, kui aega nende parandamiseks on vähe. Eeliseks on väike ajakulu testtsükli alguses [1].

EAK projektis on kasutusel agiilne iteratiivne ja inkrementaalne arendus. See tähendab, et kliendile ei tarnita kogu funktsionaalsust ühekorraga, vaid inkrementaalselt ehk osade kaupa. Iga tarne jooksul rakendatakse iteratiivset metoodikat, mille käigus loodud funktsionaalsuse arendamisest ja kasutamisest saadakse infot selle kohta, milliseid täiendusi ja parandusi on vaja sisse viia [28]. Arenduse käigus järgitakse agiilse arenduse põhimõtteid, mis on ära toodud agiilse tarkvaraarenduse manifestis [29].

1) Inimesi ja nendevahelist suhtlust tuleb hinnata rohkem kui protsesse ja arendusvahendeid.
2) Töötav tarkvara on tähtsam kui kõikehõlmav dokumentatsioon.
3) Koostööd kliendiga tuleb rohkem hinnata kui läbirääkimisi lepingute üle.
4) Tähtsam on reageerida muutunud oludele, kui järgida algset plaani.

Antud lähenemist kasutades võib päeva jooksul tekkida üle kümne uue versiooni. Versioon EAK projekti kontekstis tähendab uut seisu rakendusest, mille arendaja on oma muudatuste lisamisega loonud. Projektis on põhiliselt kasutusel jätkuva regressioontestimise meetod. Antud lähenemine on suhteliselt väikese ajakuluga võrreldes täieliku ja uuesti algava regressioontestimisega, ning põhjalikum kui regressioon suitsutestimine. Kui regressioontestimise käigus luuakse uus versioon mingi vea parandamiseks, siis mõeldakse läbi, milliseid teisi komponente antud parandus mõjutada võib. Siin on kasulik saada sisendit paranduse teinud arendajalt, kes oskab lähtekoodi vaadates öelda, kus parandatud koodijuppi veel rakenduses kasutatakse. Sellega üritatakse kahandada riski, et muudatused uutes versioonides lõhuksid ära juba läbi testitud komponendid.

1.5 Testidele prioriteetide määramise kriteeriumid

Antud tehnikad on ajamahukad ning nõuavad seetõttu palju ressursse, välja arvatud regressioon suitsutestimine, mis pole aga nii põhjalik kui teised tehnikad ja seetõttu võib seda tehnikat kasutades rohkem kriitilisi vigu testimises leidmata jääda. Kõigi nende tehnikatega saab kasutada lähenemist, kus testidele antakse prioriteedid. Prioriteetide määramine aitab kaasa eesmärgile võimalikult lühikese aja jooksul võimalikult palju kriitilisi vigu tarkvarast üles leida ja seda nii kiirelt kui võimalik. See vähendab riski, et ajapuuduse tõttu jääb mingi kriitiline funktsionaalsus läbi testimata. Määramaks, millised funktsionaalsused on kõrgema prioriteediga, tuleb vaadata nende nähtavust, kasutatavust ning võimalikku kulu tõrke korral ehk kui kriitiline on antud funktsionaalsus [2].

Kriitilised funktsionaalsused – on osad rakendusest, mille tõrke puhul oleksid tõsised rahalised või muud sorti kahjud. Nende funktsionaalsuste leidmiseks tuleb mõelda, milliseid tõrkeid võib rakenduses esineda ning mis oleksid nende tõrgete tulemused. Sõltuvalt võimalike tõrgete tõsidusest, võib funktsionaalsused jaotada gruppidesse [2].

1. Tõrke tulemus oleks katastroofiline. Siia kuuluvad funktsionaalsused, mille tõrkumisel oleksid tõsised tagajärjed, nagu näiteks terve süsteemi seiskumine, juriidilised probleemid ja kahju inimelule [2]. Näitena võib tuua tõrke lennujuhtimissüsteemis,

mille tulemusena ei kuvata õiget lennukite asukohta teiste lennukite suhtes. Tulemuseks võib olla inimeste hukkumine.

2. Tõrke tulemus oleks kahjustav. Siia kuuluvad funktsionaalsused, mille tõrked ei peataks kogu rakenduse tööd, aga põhjustaks näiteks andmekadu või osalist funktsionaalsuse kadu [2]. Selline tõrge oleks näiteks arvetega tegelevas süsteemis olemasolevate andmete tahtmatu ülekirjutamine, mis põhjustaks andmekao, mida varukoopia puudumisel või mitte töötamisel poleks võimalik taastada.
3. Tõrke tulemus oleks takistav. Siia kuuluvad funktsionaalsused, mille puudumisel oleks kasutajatel tööd võimalik edasi teha, kasutades alternatiivseid funktsionaalsuseid [2]. Näiteks võib tuua funktsionaalsuse, mis sisestusväljal pakub kasutajale registris olevaid võimalike väärtuseid, mille seast saab valida sobiva. Antud funktsionaalsuse mittetöötamisel peab kasutaja registrist ise järele vaatama, mis väärtuste seast on võimalik valida – töö oleks aeglustatud, aga seda oleks siiski võimalik teha.
4. Tõrke tulemus oleks häiriv. Funktsionaalsus poleks mõjutatud, aga rakendus poleks kasutaja jaoks nii atraktiivne [2]. Siia kuuluvad probleemid rakenduse visuaalse osaga, näiteks vead menüü paigutuses.

Funktsionaalsuse nähtavus – näitab, kui paljud kasutajad oleksid tõrkest mõjutatud, kui see esineks. Siin tuleb arvestada ka sellega, kui andestav on kasutaja tõrgete korral. Näiteks avalikult kasutatavatel süsteemidel on kasutajaliidese osatähtsus suurem kui majasiseselt kasutataval süsteemil [2]. Arvetega tegeleva süsteemi puhul oleks suure nähtavusega tõrge arve sisestamisel. Väikese nähtavusega oleks tõrge arvete kustutamisel, kuna antud funktsionaalsust kasutatakse tunduvalt vähem.

Funktsionaalsuse kasutatavus – näitab, kui tihti mingit funktsionaalsust kasutatakse reaalsete kasutajate poolt. Kõrgema prioriteediga on komponendid, mida kasutatakse igapäevaselt ja paljude kasutajate poolt. Funktsionaalsused võib kasutatavuse poolest jaotada nelja gruppi: vältimatud, tihti kasutatavad, vahel kasutatavad ning harva kasutatavad [2]. Enne rakenduse kasutusele võtmist on raske aimata, milliseid funktsionaalsuseid hakatakse tihedamini kasutama ja milliseid vähem. Sellisel juhul on suureks abiks informatsioon rakenduse tellijalt, kellel on rohkem informatsiooni inimeste kohta, kes rakendust kasutama hakkavad. Juba kasutusel oleva rakenduse puhul on võimalik kasutatavust täpsemini määrata,

seda näiteks rakenduse logide põhjal, millest saab väljavõtteid teha ning selle põhjal statistikat koostada. Samuti on võimalik vaadelda andmeid andmebaasis – kas mingi funktsionaalsuse poolt loodavate andmete maht on suur või väike?

Suurt osa testide järjekorra määramisel mängib ka vigade esinemise võimalus. Siin tuleb arvestada erinevate faktoritega, mis suurendavad võimalust, et antud komponent võib sisaldada vigu. Järgnevalt on toodud mõned faktorid, mida analüüsides on meil võimalik teha põhjendatud eelduseid, mis aitavad suurema riskiga komponente välja valida ja neile testimisel rohkem tähelepanu pöörata.

1. Suure keerukusega komponendid – suure keerukusega komponendid võivad sisaldada vigu [4, lk 61-62]. Eksisteerib palju erinevaid meetodeid keerukuse määramiseks, näiteks muutujate arv lähtekoodis ja suurte andmemahtudega töötamine [2]. Üks viis lähtekoodi keerukust mõõta on kasutades tingimusliku keerukust (*cyclomatic complexity*), mis mõõdab lineaarselt sõltumatute radade arvu lähtekoodis [3]. Kõige lihtsam ja kiirem viis saada aimu komponentide keerukuse kohta on küsida informatsiooni seda arendanud arendajalt või kasutada mõnda staatilist koodi analüsaatorit nagu Atlassian Clover, mis lisaks muudele funktsionaalsustele võimaldab näha, millised osad lähtekoodis on suure keerukusega ja vajava rohkem testimist [30].
2. Muudetud komponendid – muudetud kohad programmi koodis võivad olemasoleva funktsionaalsuse katki teha [4]. Tüüpilisemad põhjused, miks muudatused olemasoleva funktsionaalsuse ära lõhuvad, on kiirustamine, puudulik analüüs selle kohta, mis kohti muudatus rakenduses mõjutab, või üldiselt puudulik analüüs enne arendust. Muudatuste mõju analüüsi tehnikat kasutades on võimalik vähendada riski, et peale muudatusi lähtekoodis jääb testimata mõni komponent, mida antud muudatus mõjutas. Selle raames analüüsib arendaja tehtud muudatusi ning annab testijale sisendit selle kohta, mida on vaja rakenduses üle testida.
3. Uued tehnoloogiad [4] – uued tehnoloogiad võivad põhjustada vigu, kuna arendajad pole antud tööriistadega varem kokku puutunud ning sellest tulenevalt on risk vigade tekkimiseks.
4. Ajaline surve – kiirustamine võib väga halvasti kvaliteedile mõjuda, kuna inimesed üritavad töö võimalikult kiirelt valmis saada ning tulenevalt sellest üritavad aega hetkel

mitte tähtsatena tunduvate tegevuste pealt kokku hoida. Tegemata jäetud tegevused võivad aga olla vajalikud vigade ennetamiseks või leidmiseks. Ajaline surve võib panna inimesi ka ületunde tegema, mis omakorda väsitab inimest ning pärsib keskendumisvõimet [4].

5. Vigaderohked komponendid – komponendid, kust on leitud palju vigu, võivad omada ka veel avastamata vigu [4]. Kui on teada komponente, millel on minevikus palju vigu olnud, siis tasub nendele kohtadele rohkem tähelepanu pöörata peale uusi muudatusi [3].
6. Uued inimesed projektis – kui projektiga on liitunud näiteks uus arendaja või analüütik, siis tuleb nende inimeste töö põhjalikumalt üle vaadata. Kuna uued inimesed pole veel projekti ärilise poole ja tehniliste lahendustega nii tuttavad, siis võivad nad kergemini vigu teha.

1.6 Prioriteetide määramine testidele

Eelnevaid punkte silmas pidades võib funktsionaalsustele prioriteedid määrata intuitiivselt, kasutades näiteks prioriteete vahemikus 1-3, kus 1 tähistab kõrgema-, 2 tähistab keskmise- ning 3 madalama prioriteediga funktsionaalsuseid. Olenevalt projektist võib kasutada ka suuremat prioriteetide vahemikku, et funktsionaalsused rohkematesse gruppidesse jagada. Selline lähenemine on soovitatav ainult siis, kui kas testija ise on rakenduse arendamisel pikemat aega kaasa löönud ning oskab seetõttu funktsionaalsuse erinevaid kriteeriume hinnata või on võimalik sisendit saada kelleltki teiselt, kes seda kompetentselt teha oskaks (näiteks tellija või projekti analüütik).

Kui vajalik on detailsem funktsionaalsuse analüüs enne prioriteetide määramist, siis on võimalik koostada tabel riskide arvutamiseks, kasutades valemit: risk = kahju * vea esinemise võimalus [2], [6] - slaid nr 5, [5] - peatükk 9.4.1. Tabeli veergudeks on kriteeriumid, mis on välja valitud olenevalt projektist, ning nendele kriteeriumitele antakse kaal olenevalt sellest, kui tähtis ta on riski arvutamisel. Tabeli read esindavad rakenduse erinevaid funktsionaalsuseid. Igale funktsionaalsusele antakse väärtused kõigi kriteeriumite raames. Näitena Tabelis 1 on toodud kolm funktsionaalsust: rakendusse sisse logimine, arve

sisestamine ning arve kustutamine. Kriteeriumiteks on valitud äriline tähtsus ja nähtavus, mis esindavad võimaliku kahju väärtust, ning keerukus ja muutmise tihedus, mis esindavad vea esinemise võimalust. Kõige suurem kaal on antud ärilisele tähtsusele ning kõige väiksem nähtavusele. Edasi tuleb iga funktsionaalsuse punkt korrutada kriteeriumi kaaluga. Seejärel tuleb summeerida ärilise tähtsuse ning nähtavuse punktid, et saada võimaliku kahju väärtus, ning keerukuse ja muutmise tiheduse väärtused, et saada vea esinemise võimaluse väärtus. Lõpuks tuleb korrutada saadud võimaliku kahju väärtus vea esinemise võimaluse väärtusega, et saada lõplik risk [2].

Tabel 1. Prioriteetide määramine.

Funktsionaalsus	**Äriline tähtsus**	**Nähtavus**	**Keerukus**	**Muutmise tihedus**	**RISK**
Kaal	10	1	3	3	
Rakendusse sisse logimine	5	5	2	1	55*9=495
Arve sisestamine	5	5	5	2	55*21=1155
Arve kustutamine	1	1	2	1	11*9=99

Antud näite puhul on näha, et kõige suurema riskiga on arvete sisestamise funktsionaalsus, mis tuleks esimesena ja kõige põhjalikumalt läbi testida, ning kõige väiksema riskiga arvete kustutamise funktsionaalsus, mida tuleks viimasena testida ja ei vaja nii põhjaliku testimist kui ülejäänud funktsionaalsused.

EAK projektis on loodud Confluence keskkonda nimekiri rakenduse funktsionaalsustest. Antud nimekirja illustreerib Joonis 3.

Funktsionaalsused

Sisselogimine

Funktsionaalsus	Prioriteet	Automaattest	Testjuhtum	Tulemus	Kommentaar
Kasutajanimi ja parool (KEK08)	1	✅	TC01 Sisselogimine kasutajanime ja parooliga	Tulemus	
ID kaart (KEK08)	1			Tulemus	Viga kirjasEAK-1212 tööülesandes
MOBID (KEK08)	1			Tulemus	
Panga kaudu (KEK08)	1			Tulemus	

Eraisik

Funktsionaalsus	Prioriteet	Automaattest	Testjuhtum	Tulemus	Kommentaar
Kasutaja seaded (KEK18)	2			Tulemus	
B-kaardi päring (KEK10)	1			Tulemus	

Joonis 3. Funktsionaalsuste nimekiri.

Nimekirjas olevad funktsionaalsused on jaotatud gruppidesse nagu näiteks pildil olevad „Sisselogimine“ ja „Eraisik“ funktsionaalsuste grupid. Iga funktsionaalsuse kohta on eraldi rida, mis koosneb järgmistest veergudest:

1) funktsionaalsus – väärtuseks on funktsionaalsuse nimi ning vastava funktsionaalsuse dokumentatsiooni kood, mis on link spetsifikatsiooni faili asukohale.
2) prioriteet – omab väärtuseid vahemikus 1-3, kus 1 on kõige kõrgem prioriteet ning 3 kõige madalam. Prioriteedid on määratud intuitiivselt koostöös kliendiga, kuna meeskonnas olevad inimesed on projektiga pikalt tegelenud ja omavad head ülevaadet sellest, mis on kriitilisemad ja mis vähem kriitilisemad funktsionaalsused.
3) automaattest – kui vastav funktsionaalsus omab automaattesti, siis on linnuke antud veerus.
4) testjuhtum – kui antud funktsionaalsus omab testjuhtumit, siis siin veerus on link sellele dokumendile Confluence keskkonnas.
5) tulemus – märgib, millises seisus antud funktsionaalsuse testimine hetkel on. Hall värv tähendab, et funktsionaalsus on testimata. Punane värv, et leidus probleeme. Roheline värv, et test läbiti edukalt ning kollane värv märgistab hetkel testimises olevaid funktsionaalsuseid.

6) kommentaar – antud veergu saab ükskõik millist lisainformatsiooni märkida. Põhiliselt kasutatakse seda märkimaks, millistes tööülesannetes antud funktsionaalsuse testimise käigus leitud vead asuvad.

Funktsionaalsuste nimekirja saab kasutada regressioontestimise plaanina ning testimise hetkeseisu jälgimiseks. Iga uue tsükliga luuakse uus nimekiri olemasoleva malli alusel. Samuti annab nimekiri hea ülevaate uutele inimestele, kes projektiga liituvad, kuna välja on toodud suuremad funktsionaalsused ning lingid nende spetsifikatsioonidele. Antud funktsionaalsuste nimekiri on Confluence keskkonnas eksporditud .docx formaati ning lisana kaasa pandud.

2. Automatiseerimine

2.1 Automatiseerimise vajadus

Manuaalne regressioontestimine võib olla väga rutiine ja suure ajakuluga töö, mille maht kasvab iga uue tsükliga, kuna lisanduvad uued arendused, mida testida lisaks olemasolevatele. Üks viis nende probleemidega tegelemiseks on regressioontestimise automatiseerimine, millel on järgmised eelised :

1. Automatiseerimine võimaldab kasutusjuhte tunduvalt kiiremini läbida, kui seda teeks testija, kes neid manuaalselt läbi käiks. Selline ajaline kokkuhoid võimaldab uute versioonide väljatulemisel tunduvalt kiiremini kriitilised kohad uuesti üle käia ja saada kindlust, et muudatused mingit funktsionaalsust katki ei teinud.
2. Kuna teste saab käivitada kohe peale uue versiooni valmimist, siis saab arendaja väga kiiresti tagasisidet, kas tema muudatused olemasolevat funktsionaalsust negatiivselt mõjutasid. See annab võimaluse arendajal kohe probleemiga tegeleda ja kuna muudatus alles valmis, siis ka kiiremini lahendus leida, sest arendajal on värskelt meeles, mis muudatusi ta tegi. Selline kiire reageerimine vigadele säästab terve meeskonna aega, kuna pikema aja peale satuvad ka teised inimesed sama vea otsa ning peavad aega raiskama selle raporteerimiseks või parandamiseks [4, lk 94]. Halvimal juhul võidakse viga leida alles siis, kui muudatuse sisse viinud arendaja pole enam kättesaadav ning probleemiga peab tegelema keegi teine, kellel selle lahendamiseks enamasti rohkem aega läheb.
3. Esineb kohti, kus automatiseerimine annab paremaid tulemusi kui manuaalne testimine. Näiteks suurte andmemahtude kontrollimine rakenduses, kus inimene võib rutiinse töö peale kaotada tähelepanu ja mitte märgata valesid väärtuseid vormil - automaattest sellist inimlikku viga aga ei tee.
4. Valminud automaatteste saab hiljem kerge vaevaga kasutada ka rakenduse monitooringu jaoks. Näiteks testi, mis saadab arve ja kontrollib, kas see ka kohale jõuab, saab kasutada toodangukeskkonnas jälgimaks, kas arved liiguvad korralikult või on tõrkeid rakenduse töös ning arved ei jõua enam kohale.

2.2 Automatiseerimise riskid

Automatiseerimine võib säästa aega, kiirendada arendusprotsessi, võimaldada testida funktsionaalsuseid, mida manuaalne testimine ei võimalda, ning teha testimist efektiivsemaks. Siiski ei tohi ära unustada, et kaasnevad ka teatud riskid, millele mitte tähelepanu pöörates võib automatiseerimine üleliia tähelepanu röövima hakata ning seeläbi meeskonna ressurssi raisata [4], lk 93. Alustades automatiseerimist, tuleks kindlasti tähelepanu pöörata järgmistele punktidele.

1) Automatiseerimisega tuleb alustada võimalikult vara. Automatiseerimine nõuab planeerimist, uurimist ja testide kavandamist. Samuti tuleb võimalikult vara tähelepanu pöörata rakenduse testitavusele, kuna projekti alguses on arendajad vastuvõtlikumad muudatustele rakenduse disainis. Varajane alustamine võimaldab projektijuhil eelarves automatiseerimisega arvestada ning seda planeerida. Mida hiljem automatiseerimisega alustada, seda raskem on ressurssi ümber suunata automatiseerimisele ning inimesed pole muudatustele enam nii vastuvõtlikud, sest selleks ajaks on juba välja kujunenud töövoog, millega meeskonnaliikmed on harjunud. Varajane algus ei tähenda, et loodavate funktsionaalsuste testimine tuleks kohe ka automatiseerida. Funktsionaalsuseid, mille testimist automatiseerida, tuleks väga hoolikalt valida, aga automaattestide infrastruktuuri loomisega saab alustada juba varakult [4], lk 124-125.
2) Ära automatiseeri halbu teste. Kui projektis eksisteerivad kehvad testid, siis nende automatiseerimine ei too kaasa muud, kui kehvade testide kiirema läbimise. Enne automatiseerimist tuleb üle vaadata projekti testimise protsess, vastasel juhul automatiseerimine juhib tähelepanu kõrvale probleemidelt, mis projektis tegelikult eksisteerivad ja mis vajavad lahendamist enne automatiseerimise alustamist. Näiteks kui testijad ei tea või ei ütle mida nad testivad, siis loovad nad automaattestid, mida keegi teine ei mõista ega oska kasutada [4], lk 98-99.
3) Testjuhud tuleb luua enne automatiseerimist. Selline lähenemine aitab vältida olukorda, kus automatiseeritakse testjuhud, mida on kerge automatiseerida, aga annavad vähe informatsiooni rakenduse kohta [4], lk 93]. Testjuhtude loomisel tuleb keskenduda sellele, et tegevused, mida läbitakse, ning andmed, mida kasutatakse, oleksid põhjalikult läbi mõeldud. Alustades automatiseerimist enne või paralleelselt

testjuhtude loomisega, on oht, et kaob fookus ning hakatakse liiga palju keskenduma tehnilisele lahendusele ning automaattesti tööle saamisele.

4) Vajaliku ressursi olemasolu. Tuleb uurida, kas projektis eksisteerib vajalik ressurss automaattestide loomiseks. Vajalik on ressurss nii testjuhtude loomiseks kui ka tehniline teadmine testide programmeerimiseks. Samuti tuleb arvestada, et peale testide loomist on vaja ressurssi ka testide hooldamiseks, kuna tulevikus tehtavad muudatused rakenduses võivad vajada ka muudatusi automaattestides. Kui vajalikud oskused projektis puuduvad, siis tuleb arvestada ajaga, mis kulub inimestel automatiseerimise tööriistadega tutvumiseks.

2.3 Milliseid teste automatiseerida?

Planeerides automaatteste, tuleb olemasolevate kasutusjuhtude seast hoolikalt välja valida need, mis sobivad automatiseerimiseks ja mille automatiseerimisest kõige enam kasu saab. Võib juhtuda, et valitakse välja kasutusjuhud, mida on kõige lihtsam programmeerida. Selline lähenemine pole tingimata vale, aga ei tohi unustada, et tulemus peab ka pikas perspektiivis kasulik olema. Testjuhu automatiseerimine võtab kordades rohkem aega, kui selle manuaalselt läbimine, seega peab tulemuseks saadud automaattest tulevikus võimalikult kaua kasutusel olema ja seejuures ka kasuliku informatsiooni rakenduse kohta andma, et sinna investeeritud raha tagasi teenida.

Eesmärk ei tohiks olla kogu rakenduse testimise automatiseerimine, kuna automaattestid ei suuda täielikult asendada inimese poolt läbi viidud manuaalset testimist. Seda seetõttu, et automaattest teeb ainult seda, mis käsud talle ette on antud; eksisteerib aga situatsioone ja on vaja läbi viia kontrolle, mida pole võimalik ette programmeerida. Seega tuleb mõelda, milliseid testjuhte on kasulik läbida manuaalselt ja milliseid automaatselt. Järgnevalt on välja toodud punktid, millega tuleks arvestada, kui valitakse testjuhte, mida automatiseerida.

1. Kui tihti antud testjuhtumit läbitakse ja kas oleks kasulik, kui seda tihedamini läbitakse? Kas testi läbitakse peale igat uut rakenduse versiooni või jooksutatakse seda ainult üks kord enne tarnet? Juhul kui testi jooksutatakse üks kord enne tarnet, siis

suure tõenäosusega ei ole mõtet seda automatiseerida. Rakendusse sisse logimist tehakse näiteks väga tihedalt ning antud funktsionaalsust tasuks automatiseerida [7].

2. Kui palju andmeid antud testjuhtumi raames sisestama peab? Testid, mis nõuavad suurte andmemahtude sisestamist, on kindlasti head kandidaadid automatiseerimiseks, kuna inimese jaoks on see tüütu tegevus ning ajaline kokkuhoid sellistes kohtades on väga suur [7].
3. Kui lihtne on testjuhtumi tulemust rakenduses lugeda? Teste, mille väljundit tuleb kontrollida mitmetest kohtadest üle rakenduse, on keerukam automatiseerida kui teste, mille väljund kuvatakse ühel kuval [7]. Manuaalsel testimisel võib inimene vea teha, kui kuvatavaid väärtuseid on väga palju. Sellisel juhul tuleks kindlasti eelistada automaattesti, mis kontrollib tulemuse kiiremini ja ei tee inimlikke vigu.
4. Kas testjuhtumi tulemust saab objektiivselt hinnata? Kui testi tulemuseks on täpne väärtust, siis enamasti saab seda automaattestiga kontrollida. Juhul kui testi tulemuseks genereeritakse pilt, siis on väga raske automaattestiga tulemust kontrollida. Sellisel juhul peab testija manuaalselt testides kinnitama, et tulemuseks saadud pilt vastab nõuetele [7].
5. Kas funktsionaalsus, mida soovitakse automatiseerida, on piisavalt stabiilne? Valides funktsionaalsuseid, mida planeeritakse automatiseerida, tuleks kindlasti uurida, kas funktsionaalsus on hetkeseisuga valmis ning ei vaja parandusi. Samuti tuleks uurida, kas lähiajal on plaanis mingeid suuremaid täiendusi antud komponendile. Kui funktsionaalsus vajab parandusi või planeeritakse lähiajal täiendusi, siis suure tõenäosusega ei tasu automatiseerimine hetkel ära. Muudatused rakenduse koodis võivad tähendada, et peab ka automaatteste ümber tegema ning tulenevalt sellest kulutab testide hooldamine ressurssi, mida saaks mujal kasutada.
6. Kui palju seadistatavaid kontrolle test omab? Testjuhtum on automatiseerimiseks parem kandidaat siis, kui see omab kindlaid kontrolle ning selle läbimisel kasutatakse tavalisi nuppe, linke, tekstikaste jne. Juhul kui test sisaldab seadistatavaid kontrolle ja funktsionaalsuseid, siis ei ole selle automatiseerimine võimatu, aga ilmselt keerukam kui tavaliste funktsionaalsuste puhul, sest seadistused võivad mingil põhjusel muutuda ja selle tulemusena test ebaõnnestub [7].
7. Kas test vajab improviseerimist või uurivat testimist? Automaattest teeb ainult seda, mis talle ette programmeeritud on ja seetõttu ei suuda olla loominguline nagu inimene

[7]. Kui on teada, et mingi funktsionaalsus on paremini testitav kasutades uurivat testimist, siis tuleks see kas tervenisti üle vaadata manuaalselt või automatiseerida ainult osaliselt ning läbi viia lisaks veel manuaalne testimine.

8. Kui kiiresti on antud funktsionaalsuse mittetöötamisel vaja see korda teha? Kui funktsionaalsus on rakenduses väga kriitiline ja ilma selleta ei saa kasutada suurt osa ülejäänud rakendusest, siis on vajalik kiire reageerimine selle mittetöötamise korral. Näiteks võivad vead sellistes kohtades takistada ülejäänud rakenduse edasi testimist ja seega ohustada tähtajast kinni pidamist. Sellised kohad on head kandidaadid automatiseerimiseks, sest rakenduse uue versiooni valmimisel saab automaatselt käivitada ka automaattestid, mis annavad kiiret tagasisidest selle kohta, kas antud kriitilised kohad uues versioonis töötavad või mitte. Tulemusena saab vea korral kiiresti seda parandama asuda.
9. Testjuhtumid, mida tuleb läbida erinevate andmetega [8]. Kui eksisteerivad testid, mis sisaldavad samu tegevusi, aga neid tuleb läbida mitu korda erinevate andmetega, siis tasub kaaluda nende automatiseerimist. Sellisel juhul tuleb valmis kirjutada üks automaattest ning lihtsalt käivitada seda erinevate andmetega. Manuaalne testimise sellisel juhul oleks väga tüütu ning tunduvalt suurema ajakuluga.

2.4 Automaattestide loomise ja täiendamise protsess

Automaattestide loomise ja täiendamise protsess võib projektides erineda sõltuvalt kasutada olevatest ressurssidest. Testjuhtumid, mida testide loomisel kasutatakse, tuleks luua testija poolt, kes oskaks mõelda, kuidas funktsionaalsust kõige parem testida oleks. Testide automatiseerimise osas on tihti vaja programmeerimise oskust ja seega on selleks vaja programmeerijat või testijat, kes valdab vajalikku programmeerimiskeelt. EAK projektis on testjuhtumid loodud testija poolt ning programmeerimine on tehtud testija ning osaliselt ka arendajate poolt.

Arenduse käigus võib juhtuda, et olemasolevad testid enam ei tööta, kuna rakendus on muutunud. Sellisel juhul tuleb paika panna, kes sellises olukorras testide täiendamise eest vastutab. Kui rakenduse täiendamise käigus muutus funktsionaalsuse kasutamine kasutaja

jaoks, siis on vaja täiendada testjuhtumit ning selle eest peaks vastutama testija. Lisaks tuleb täiendada automaattestide lähtekoodi vastavalt testjuhtumi muudatustele. Ühe võimalusena võib seda teha arendaja, kes vastavad täiendused ka rakenduse poole pealt sisse viis, või keegi kindel inimene, kes selle jaoks välja valitud on. EAK projektis vastutab testide täiendamise eest testija, aga ajapuuduse korral võib suunata selle tegevuse ka arendajale, kellel vähem tööd on. Tähtis on meeles pidada, et kui mingi test enam ei tööta muudatuste tõttu, siis tuleks see võimalikult kiiresti ära parandada. Katkine test võib takistada kõigi ülejäänud automaattestide jooksutamist ja see takistab testidesse investeeritud aja tagasi teenimist. Lisaks eksisteerib oht, et pikemalt hooldamata testid võivadki jääda seisma, kuna muudatusi on aja jooksul tekkinud nii palju, et kellelgi pole aega teste enam järele arendada. EAK projektis tuleb automaattestid ära parandada sama arendustsükli jooksul, kus nad katki läksid ehk enne kliendile uue versiooni tarnimist.

3. Selenium Webdriver

3.1 Selenium Webdriveri omadused

Valides automatiseerimise tööriista, tuleb arvestada, et see ei õpeta inimesi testima; seega tuleb enne automatiseerimist kindlasti üldine testimise protsess korda teha [4], lk 98. Mõeldes, millist automatiseerimise tööriista muretseda, tuleb enne hoolikalt mõelda, mis on need nõuded, mida tööriist täitma peab, ja sõltuvalt sellest oma valiku tegema. EAK projektis olid nõuded järgmised.

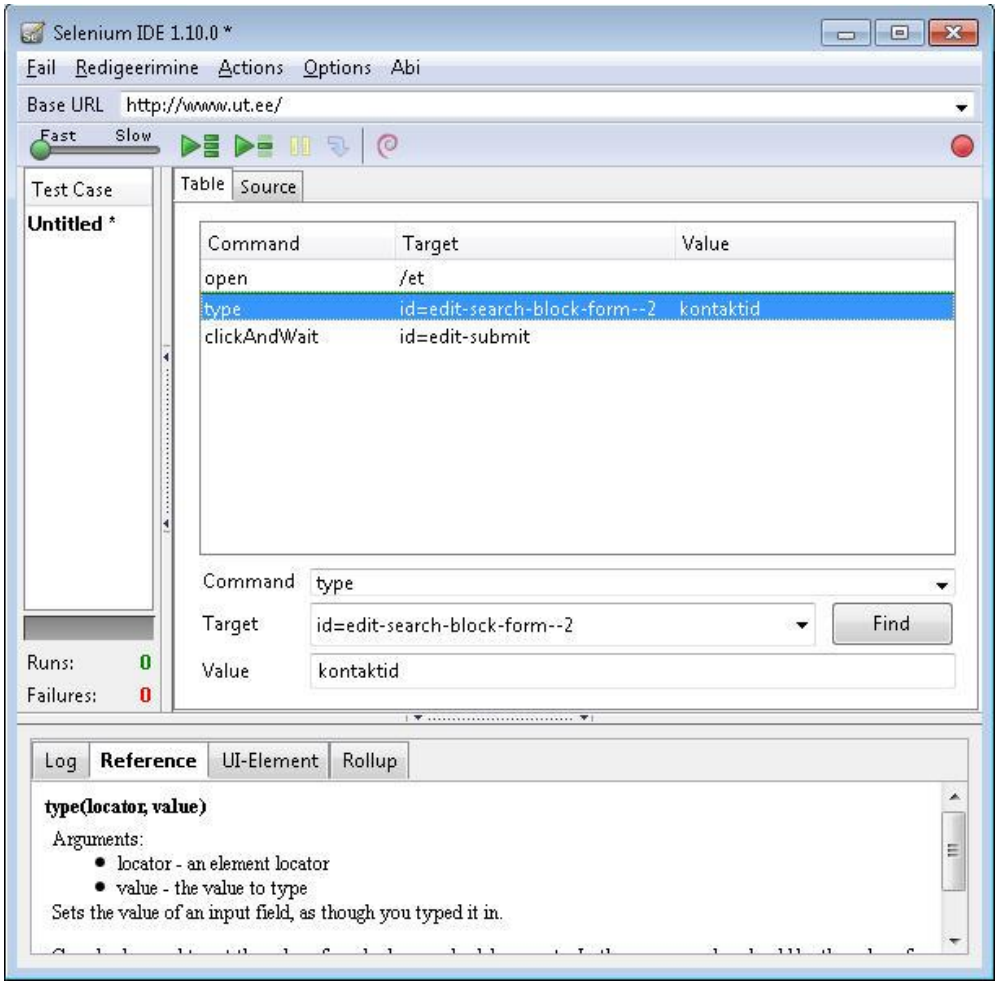
1) Tööriist peab võimaldama koostada kasutajaliidesepõhiseid funktsionaalseid teste.
2) Loodavaid teste peab olema võimalik lülitada pideva integratsiooni keskkonda.
3) Teste peab olema võimalik luua lindistades tegevusi rakenduses.
4) Keerulisemaid teste peab olema võimalik luua programmeerimiskeeles Java, kuna meeskond valdab seda kõige paremini.
5) Tööriist peab oskama koostada raportit testide jooksutamise kohta.
6) Tööriist peab omama korralikku API dokumentatsiooni.
7) Tööriist võiks olla vabavaraline.

Nõudeid silmas pidades valiti välja Selenium Webdriver. Antud tarkvara on põhiliselt mõeldud loomaks kasutajaliidese põhiseid automatiseeritud teste, mida saab veebilehitsejas käitada [12]. Seega saab antud tööriista kasutada funktsionaalsete testide loomiseks. Keerulisemaid teste on võimalik koostada, kasutades Selenium Webdriverit, mis suhtleb veebilehitsejaga otse, kasutades veebilehitseja enda poolt pakutavaid funktsionaalsuseid automatiseerimiseks. See, kuidas WebDriver veebilehitsejaga täpsemalt suhtleb, sõltub seega veebilehitseja enda võimalustest. Toetatud on Firefox, Internet Explorer, Chrome ja Opera veebilehitsejad. Populaarsematest operatsioonisüsteemidest on toetatud Windows, OS X, Linux ja Solaris. Programmeerimiskeeltest on toetatud Java, C#, Python ja Ruby [15]. Webdriverit kasutades on võimalik raporteid koostada näiteks kasutades Apache Ant koos TestNG pluginaga ReportNG [18]. WebDriver on vabatarkvaraline ning omab ka korraliku API lehekülge [16]. Webdriveri teste on võimalik kasutada koos Atlassian Bambooga , mida EAK projektis kasutatakse pideva integratsiooni keskkonnana (*continuous integration environment*) [13]. Antud tarkvaral on olemas ka integreeritud arenduskeskkond Selenium

IDE, mida saab kasutada lisana Firefox veebilehitsejaga. Selenium IDE võimaldab salvestada veebilehitsejas tehtavaid tegevusi, lihtsustades sellega automaattestide loomist [14].

3.2 Selenium IDE

Lihtsamaid automaatteste on võimalik luua ning jooksutada ainult *Selenium IDE* tarkvara kasutades, mida saab alla laadida http://docs.seleniumhq.org/download/ leheküljelt. Keerulisemate testide koostamiseks jääb antud tööriistast väheks, samuti on seda võimalik kasutada ainult Firefox veebilehitsejaga. Siiski on tegemist kasuliku ning mugava tööriistaga, mida saab näiteks kasutada toetavas rollis, kui luua keerulisema loogikaga teste Webdriveriga. Selenium IDE on võimalik käivitada Firefox veebilehitsejas „Tools -> Selenium IDE“ valiku alt. Illustreeriv pilt Selenium IDE kasutajaliidesest on esitatud Joonisel 4.



Joonis 4. Selenium IDE kasutajaliides.

Antud pildil on salvestatud testjuhtum, kus kasutaja läks www.ut.ee veebilehele, kirjutas otsingu lahtrisse sõna „kontaktid“ ning vajutas otsingu nuppu. Peale testjuhtumi salvestamist on ühe nupuvajutusega võimalik kas kogu salvestatud test või ainult teatud sammud sellest uuesti automaatselt läbi teha. Samuti on võimalik valida, millise kiirusega antud tegevused läbitakse. Teste saab muuta otse Selenium IDE kasutajaliideses, muutes siis olemasolevaid käske või lisades täiesti uusi. Uusi käske saab valida „Command“ lahtri rippmenüüst, mis tähendab, et kasutaja ei pea käske tingimata peast teadma ega neid mujalt otsima. Lisaks kuvatakse mingi käsu valimisel „Reference“ sakis selle käsu kirjeldust. Kasutades „Find“ nuppu on võimalik täpselt näha, kus valitud element veebilehel visuaalselt asub.

Keerulisemate testide valmistamisel võib Selenium IDE kasutada algse testi loomiseks, testi eksportimiseks teise formaati ning seejärel sellele keerulisemat loogikat juurde lisades. Teste saab eksportida valides „Options -> Clipboard format“ valikust soovitava formaadi ning lihtsalt kopeerida testis olevatest käskudest kas kõik või mingi alamhulk. Vaikimisi on olemas Java, HTML, Ruby, C# ja Python formaatide valikud, aga kasutajal on võimalik neid ise juurde defineerida seadete alt.

Lisaks on võimalik jooksvalt näha, kuidas mingis formaadis hetkel loodud testjuhtum välja näeb. Soovitavat formaati saab valida „Options -> Format“ valikust ning väljundit saab näha „Source“ sakilt. Vaikimisi on „Options -> Format“ valik tühi, kuna antud funktsionaalsus on algselt välja lülitatud. Põhjuseks on asjaolu, et tegemist on veel pooleli oleva funktsionaalsusega ning formaadi vahetamisel võib kasutaja töö kaotsi minna. Siiski on võimalik antud funktsionaalsus omal valikul sisse lülitada „Options -> Options...“ kuvalt valides „Enable experimental features“ märkeruudu [17].

3.3 Nõuanded automatiseerimiseks

Antud peatükis antakse nõuandeid seoses automatiseerimisega ning tutvustatakse tähtsamaid Webdriveri võimalusi. Üldise lähenemisena tuleks üritada automaattestid võimalikult lihtsatena hoida, kuna keeruline loogika teeb testidest arusaamise ja nende ülevaatamise raskemaks ning samuti on suurem võimalus, et tekivad vead automaattestides [4], lk 113.

Andmetepõhine testimine. Kui soovitakse, et loodud teste oleks võimalikult lihtne käitada erinevate andmetega, siis on hea mõte kasutada andmetepõhist testide automatiseerimist. Põhimõte seisneb selles, et andmed hoitakse eraldi failis ning automaattestide lähtekoodis küsitakse väärtuseid mingist kindlast kohast andmete failist. Selline lahendus võimaldab samu teste käitada erinevate andmetega, lihtsalt failis olevaid väärtuseid muutes või kasutades erinevaid faile, mis omavad sama formaati andmete paigutuse osas [4], lk 114. Samuti on seda tüüpi teste mugav anda tellijale kaasa, kui selline nõue või soov eksisteerib. Tellija saab siis testide andmeid vahetada ilma testide lähtekoodi muutmata. EAK projektis hoitakse andmed .xls formaadiga failis.

Järgnev koodijupp näitab, kuidas .xls formaadis fail automaattestile ette anda programmeerimiskeeles Java [19, 20]:

```
Workbook workbook = Workbook.getWorkbook(new File("myfile.xls"));
Sheet sheet = workbook.getSheet("Autentimine");

Cell kasutajanimi = sheet.getCell("B2");
String kasutajanimiString = kasutajanimi.getContents();
```

Siit on näha, et antakse ette fail nimega *myfile.xls* ning kasutatakse failis olevat „Autentimine“ nimelist lehte. Võimalik on kasutada ka numbrilisi väärtusi lehe määramiseks, sellisel juhul algab esimene leht numbrist 0. Järgmisena võetakse lehelt mingi kindla lahtri väärtus, antud näites on selleks veerus B, real 2 olev lahter. Ka siin on võimalik kasutada numbrilisi väärtuseid rea ja veeru määramiseks. Lõpuks määratakse leitud väärtus muutujale „kasutajanimiString“. Edasi saab antud muutujat kasutada koodis nagu teisigi java muutujaid:

```
typeToElement("m.f0.root.menu.f0.loginForm.code", kasutajanimiString);
```

Kaudsed ja otsesed ootamise võimalused. Ootamine automaattestide juures tähendab kindla aja möödumist peale mingi tegevuse läbimist, et jätkata järgmise sammuga. Ootamised on automaattestide juures tähtis osa, kuna halvasti loodud ootamise loogika võib automaattestide jooksutamise aega suurendada tunduvalt ja mõjutada ka testide töökindlust. WebDriver pakub erinevaid kasulikke võimalusi, kuidas automaattestides olevaid ootamisi üles ehitada.

Otsesed ootamised (*explicit waits*) on koodijupid, mida saab defineerida ootama mingit kindlat tingimust, enne kui liigutakse edasi järgmise koodirea juurde. Kõige halvem näide sellisest lähenemisest oleks kasutada `Thread.sleep()`, mis ootab täpselt nii kaua, kui palju millisekundeid talle ette antakse enne, kui edasi liigutakse. Sellise meetodi kasutamine paneb testid ilma põhjuseta ootele, pikendades testide jooksutamise aega, seega ei ole antud meetodi kasutamine soovitatav. Mõistlikum oleks kirjutada automaattestid põhimõttel, et oodatakse ainult nii kaua, kui on vaja enne, kui liigutakse edasi. Ühe võimalusena pakub WebDriver selliseid klasse nagu `WebDriverWait` ja `ExpectedCondition`, mille kombineerimisel saab luua paremaid lahendusi ootamiseks [22]. Näide otsese ootamise lahendusest:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement = (new WebDriverWait(driver, 10))
.until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicElement"
)));
```

Antud lahenduses antakse käsk avada http://somedomain/url_that_delays_loading veebileht. Järgmisena antakse käsk oodata seni, kuni leitakse veebielement, mille ID on *myDynamicElement*. Oodatakse 0 – 10 sekundit, enne kui visatakse `TimeoutException` või tagastatakse otsitud veebielement [22, 23].

Kaudsed ootamised (*implicit waits*) on lahendus, kus ise mingi loogika väljamõtlemise asemel öeldakse WebDriverile, et kui otsitavat elementi esimesel korral üles ei leitud, siis tuleb etteantud aja jooksul seda jätkuvalt pärida DOM puust. Näide kaudse ootamise lahendusest:

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement =
driver.findElement(By.id("myDynamicElement"));
```

Siin antakse käsk elementi, mille ID on *myDynamicElement* otsida 10 sekundit, kui esimesel korral elementi ei leitud. Kui element leitakse varem kui 10 sekundi jooksul, siis liigutakse edasi mitte ei jääda lõpuni ootama. Kaudne ootamine seadistatakse näites teisel real oleva käsuga ning tuleb arvestada, et antud seadistus jääb kehtima ka edasi ning mõjub kõigile `findElement` ja `findElements` meetoditele [22, 23].

Ebaõnnestunud testi uurimisel tuleb arvestada ka käitamise keskkonnaga. Võib juhtuda, et automaattestid ei ebaõnnestu mitte loodud testis oleva vea tõttu, vaid Webdriveri enda probleemide tõttu, mis tulevad esile, kui teste käitada kindlates tingimustes. EAK projekti testide kirjutamisel esines probleem, kus näiliselt juhuslikel hetkedel ebaõnnestus automaattestidel liikuda rakenduse menüüs. Peale mitmeid testide jooksutamisi tuli välja, et testidel ei õnnestunud menüüs liikuda juhul, kui hiire kursor polnud teste jooksutava veebilehitseja akna piirides. Lähemal uurimisel selgus, et antud probleem esineb kindlate veebilehitsejatega, sealhulgas ka Firefoxiga, mida kasutatakse EAK projektis testide jooksutamiseks [21].

Erinevate lokaatorite kasutamine. Et automaattestid töötada saaks, on vaja ette anda vajalikud veebielemendid nagu nupud, lingid, tekstiväljad jne. WebDrvier sisaldab klassi `By`, mis pakub palju erinevaid lokaatoreid elementide leidmiseks [26, 27]:

1. `driver.findElement(By.id(„ID“))` – otsib elementi tema ID väärtuse järgi. W3C standardite järgi peavad leheküljel olevad ID olema unikaalsed, seega on see hea võimalus elemente leida. Siiski ei lisa arendajad tavaliselt kõigile elementidele ID külge, seega tuleks arendajatel lasta need lisada elementidele, mida kasutatakse automaattestides, et teste oleks kergem koostada.
2. `driver.findElement(By.name("Name"))` – otsib elementi „Name“ atribuudi järgi. Standardi järgi ei pea antud atribuut olema unikaalne. Seega võivad testid katki minna, kui lisandub teine element sama atribuudi väärtusega. Sellisel juhul valitakse esimene element.
3. `driver.findElement(By.linkText("Link name"))` – otsib linke, mis on etteantud nimega. Mitme sama nimega lingi olemasolul valitakse esimene. Samuti on linke võimalik otsida osalise teksti järgi käsuga `By.partialLinkText(„Partial name“)`.
4. `driver.findElement(By.xpath("//input[contains(@class='required')]"))` – otsib elemente kasutades XPath keelt. Antud võimalust kasutatakse EAK projektis kõige enam, kuna arendusel kasutatud tehnoloogia tõttu on paljud ID väärtused dünaamilised. XPath lokaatori abil saab otsida elemente ID stabiilsena püsiva osa järgi.

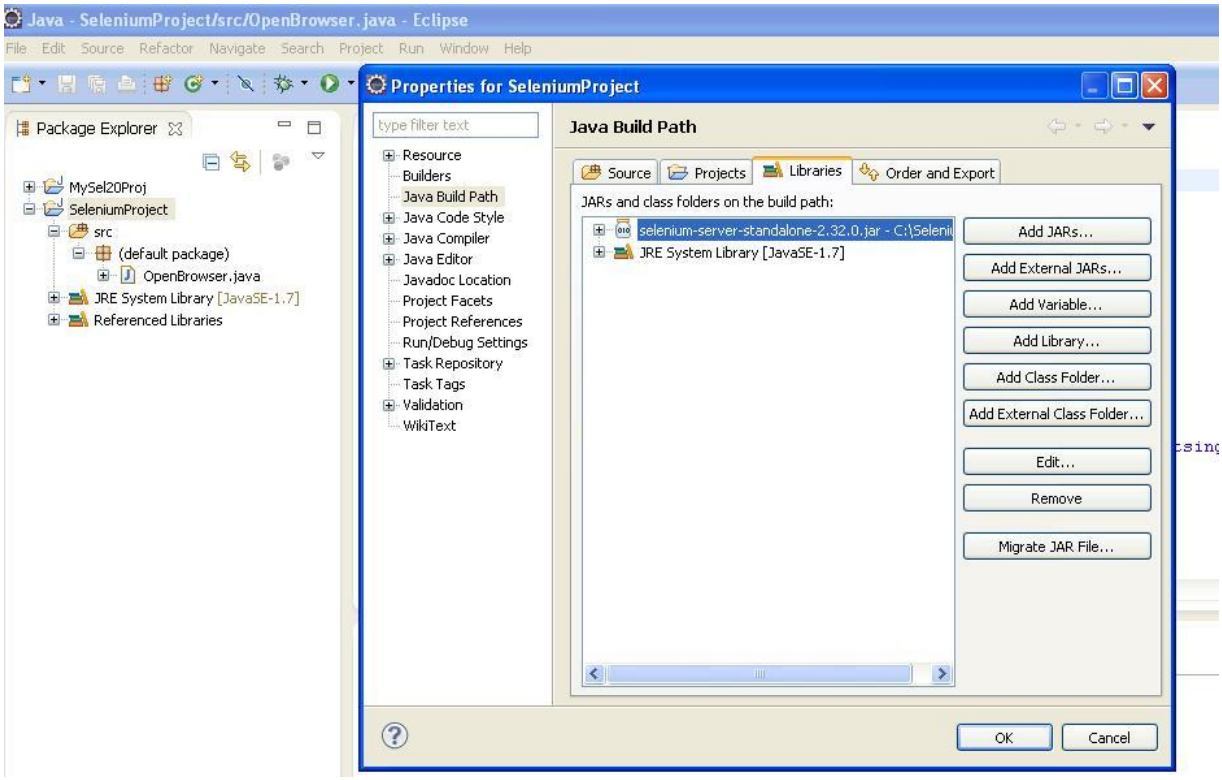
5. `driver.findElement(By.cssSelector("input[name=\"q\"]"))` – otsib elemente kasutades CSS lokaatoreid.
6. `driver.findElements(By.class("required"))` – otsib elemente klasside järgi.

Tulemuste kontrollimine. WebDriver pakub oodatava tulemuse ja reaalselt saadud tulemuse võrdlemiseks kahte tüüpi võimalusi. *Assert* tüüpi käsud peatavad testide käitamise, kui kontrolli käigus leitakse, et leitud tulemus pole sama mis oodatud tulemus. *Verify* tüüpi käskude puhul kuvatakse ebaõnnestunud kontrollide vead peale testide jooksutamist, st testide käitamist ei peatata kui mõni kontroll ebaõnnestub. *Assert* tüüpi käskude eelis on kohene tagasiside selle kohta, et mingi test ebaõnnestus ning kasutades TestNG raamistikku on võimalik lihtsalt näha milline test ebaõnnestus. Puuduseks on aga asjaolu, et kui leitakse viga, siis mingi osa teste võib jääda käitamata,. *Verify* tüüpi käskude puhul jooksutatakse leitud vea korral testid lõpuni (eeldusel, et ei esine vigu, mis takistavad järgmiste sammude läbimist). Puuduseks on asjaolu, et peab rohkem vaeva nägema vea põhjuse tuvastamiseks. Kuna TestNG ei suuda antud juhul välja tuua, milline test konkreetselt ebaõnnestus, siis on vaja iga vea puhul uurida väljundiks antud infot ning ise kindlaks teha, milline test ebaõnnestus. See võib aga suurte testide arvu korral väga koormav töö olla [31].

3.4 Testide käitamine

Kuna WebDriver on oma olemuselt lihtsalt klassiteek, siis on suhteliselt lihtne seda üles seada ning kasutada. Kõige kiirem lahendus on arenduskeskkonnas nagu näiteks Eclipse luua uus projekt, alla laadida WebDriver ning see teekide alla lisada. Järgnevalt on toodud põhilised sammud kõige lihtsama keskkonna üles seadmiseks.

1. Tuleb alla laadida Eclipse http://www.eclipse.org/downloads/ leheküljelt (Eclipse IDE for Java EE Developers).
2. Tuleb alla laadida WebDriver http://docs.seleniumhq.org/download/ leheküljelt (Selenium Server).
3. Eclipse keskkonnas tuleb luua uus projekt ning projekti seadete alt lisada alla laetud WebDriveri teek (.jar fail) nagu kujutatud Joonisel 5.



Joonis 5. Eclipse seadistus.

4. Järgnevalt tuleb luua uus java klass, kuhu saab kirjeldada enda testi. Järgnevalt on toodud näitekood, mis käitamisel läheb saidile www.ut.ee, kasutades veebilehitsejat Firefox ning kirjutab seal olevasse otsingu lahtrisse sõna „Otsing“:

```
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class OpenBrowser {

	public static void main(String[] args) throws InterruptedException {

		WebDriver driver = new FirefoxDriver();
		driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
		driver.get("http://www.ut.ee");
		driver.findElement(By.id("edit-search-block-form--
2")).sendKeys("Otsing");

	}
}
```

5. Peale testi valmimist saab seda käivitada lihtsalt Eclipse keskkonnast „Run“ nupu vajutamisega, mille peale käivitatakse Firefox veebilehitseja ning tehakse testis kirjeldatud tegevusi.

EAK projekti raames on kasutatud veel lisaks muud tarkvara, et testide jooksutamine mugavam oleks. Antud tarkvarade seadistamine ei ole osa käesolevast lõputööst ega ole tehtud töö autori poolt. Siiski kirjeldatakse lühidalt, mis tarkvara on kasutatud ning mis kasu neist on.

Apache Ant on tarkvara, mis automatiseerib tarkvara kokku ehitamist. WebDriveri puhul tähendab see, et teste saab käitada käsurealt ja seetõttu pole testide jooksutamiseks vaja arenduskeskkonda [24]. Antud töö raames loodud ja Lisades esitatud teste täies mahus jooksutada ei saa, kuna neid käitatakse kliendi testkeskkonnas ning ligipääsu projektivälistele inimestele jagada ei ole võimalik (testid käivituvad, aga sisenemine rakendusse ebaõnnestub puuduvate korrektsete kasutajaandmete tõttu). EAK teste on võimalik jooksutada enda arvutis, kui on täidetud järgmised tingimused (juhised on tehtud operatsioonisüsteemi Windows 7 näitel):

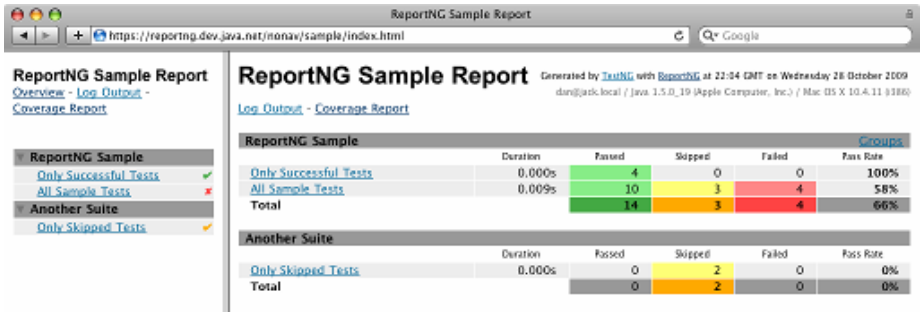
1. Installeerida Apache Ant : http://ant.apache.org/bindownload.cgi;
2. Installeerida Firefox veebilehitseja : http://www.mozilla.org/en-US/firefox/new/;
3. Installeerida Java JDK http://www.oracle.com/technetwork/java/javase/downloads/index.html;
4. Lisada vajalikud süsteemsed muutujad. Start menüü → „Computer“ valiku peal pearem hiireklõps → „Properties“ → vasakult menüüst avada „Advanced system settings“ → avanevas aknas avada „Environment Variables“. Lisada/Muuta järgmised muutujad :
    - „System vairables“ all muuta/lisada „Path“ muutujat ning lisada sinna ANT „bin“ kataloogi asukoht. Näiteks „C:\apache-ant-1.8.1\bin“ (muutujate väärtused tuleb lisada ilma jutumärkideta). Kui muutujal on mitu väärtust, siis eraldada need semikooloniga (’;’).

- „System variables“ alla lisada „ANT_HOME’“muutuja, mille väärtuseks panna Ant kodukataloogi asukoht. Näiteks „C:\apache-ant-1.8.1\bin“.
- „System variables“ alla lisada „JAVA_HOME“ muutuja, mille väärtuseks panna java JDK kataloogi asukoht. Näiteks „C:\Program Files (x86)\Java\jdk1.6.0_20“.
- „System variables“ all muuta „Path“ muutujat ning lisada sinna java JDK bin kataloogi asukoht. Näiteks „C:\Program Files (x86)\Java\jdk1.6.0_20\bin“;

5. Automaattestide kodukataloogis olevas „conf“ kataloogis olevas „arli.test.properties“ failis tuleb ära muuta Firefox browseri asukoht arvutis. Asukoht on kirjas „firefox.home“ muutujas, mis asub faili viimasel real, vaikimisi on selleks „C:\\Program Files\\Mozilla Firefox\\firefox.exe“;
6. Avada Windows Command Prompt ning liikuda seal automaattestide kodukataloogi (näiteks ’C:\Automaattestid\EAK’). Kataloogis käivitada automaattestid käsuga ’ant clean tests’. Selle peale käivitatakse automaatselt Firefox veebilehitseja ning jooksutatakse seal automaatteste.

TestNG on raamistik, mis aitab seadistada WebDriveri testide jooksutamist. Näiteks saab määrata, millises järjekorras testid töötama peavad, mis on kasulik juhul, kui eksisteerivad sõltuvused testide vahel. Samuti on võimalik TestNG abil genereerida raporteid testide jooksutamise tulemuste kohta [25]. Sõltuvuse seadistamiseks piisab ühest reast loodava meetodi juures, näiteks : `dependsOnMethods = "addSaleInvoiceInfo"`.

ReportNG on TestNG lisa, mille eesmärk on testide jooksutamise järel genereeritud raportid rohkem loetavamaks teha. Genereeritavad raportid on HTML kujul. Samuti on võimalik raporteid genereerida XML kujul, mida on võimalik ette anda pideva integratsiooni keskkondadele, et testide tulemusi kuvada. Joonisel 6 on toodud üks näide raportist:



Joonis 6. ReportNG.

Kokkuvõte

Antud töö esimeses osas tutvustati regressioontestimist üldisemalt, selgitades selle vajadust tulenevalt pidevalt muutuvast lähtekoodist ja sellega kaasnevast vigade ohust. Analüüsiti, kuidas testjuhtumite loomine aitab kaasa regressioontestimisele, sest nende loomisel teostatakse planeerimise osa funktsionaalsuste testimiseks ning tulevikus saab loodud teste kergesti taaskasutada. Toodi välja kaudseid spetsifikatsioone nagu konkureerivad tooted, kasutajate kommentaarid, kasutajaliidese standardid ja testijate enda kogemus, mida võiks testjuhtumite kirjutamisel kasutada, ning selgitati, kuidas kirjutatava dokumendi detailsus peaks sõltuma sellest, kas loodav testjuhtum on manuaalseks testimiseks mõeldud või kasutatav sisendina automatiseerimisele. Eraldi analüüsiti väljakutseid, mida regressioontestimine endaga kaasa toob ning erinevaid tehnikaid regressioontestimise läbiviimiseks. Üks suurimaid probleeme oli risk, et tähtis osa funktsionaalsusest jääb testimata tulenevalt ajapuudusest. Selle riski vähendamiseks tutvustati lähenemist, kus testjuhtumitele määratakse prioriteedid. Tulemusena on testijal alati teada, millised funktsionaalsused vajavad enim tähelepanu, ning nende testimisele pannakse suuremat rõhku kui nende funktsionaalsuste testimisele, mis omavad madalamat prioriteeti.

Teises osas tutvustati testide automatiseerimist, mis aitab tegeleda kasvava funktsionaalsuse, rutiinseks kujuneva töö ja võimalike inimvigadega testimisel. Selgitati, et kuigi automatiseerimine on võimas tööriist, siis valesti kasutamisel võib see projektile rohkem kahju kui kasu tuua, raisates ressursse, mida saaks mujal paremini kasutada. Vältida tuleks selliseid eemärke nagu kogu rakenduse testimise automatiseerimine, kuna enamikel juhtudel eksisteerib funktsionaalsuseid, kus manuaalne testimine annab paremaid tulemusi kui automatiseeritud testimine. Tähtis on meeles pidada, et automatiseerimine ei õpeta testijaid paremini testima, seega tuleks enne automatiseerimist üle vaadata projekti käesolev testimisprotsess ning olemasolevate probleemide korral need enne ära lahendada. Valides testjuhtumeid, mida automatiseerida, tuleks mõelda, kas antud funktsionaalsuse testimine vajab inimlikku lähenemist ja võimet erinevaid otsuseid teha tulenevalt saadavatest tulemustest. Kui jah, siis tuleks eelistada manuaalset testimist, sest automaattestid teevad ainult seda, mis neile ette on öeldud. Samuti tuleks mõelda, kas saadava tulemuse õigsust on arvutiprogrammil lihtne hinnata. Näiteks piltide puhul on tulemust arvutil väga keeruline hinnata ning selliseid teste

peaks läbi viima inimene. Testjuhtumid, mis omavad suuri andmemahte ja mille läbiviimine on väga rutiine töö, sobivad hästi automatiseerimiseks, eeldusel, et funktsionaalsus on stabiilne ning seda ei muudeta lähitulevikus. Testjuhtumite koostamine ning hooldamine peaks jääma testija vastutuseks ning automaattestide loomine ja täiendamine võiks olla arendaja töö, olenevalt sellest, milliste oskustega inimesed projektis on.

Kolmandas osas tutvustati Selenium Webdriver tööriista, mida kasutati antud töö raames loodud testide automatiseerimiseks. Antud tööriist valiti, kuna ta võimaldab automatiseerida funktsionaalseid teste, kirjutada teste programmeerimiskeeles Java, genereerida tulemustest raporteid ning on vabavaraline tarkvara. Lühidalt tutvustati andmetepõhist testimist ning tähtsamatest Webdriveri funktsionaalsustest uuriti automaattestide ootamise loogika loomise, veebielementide asukoha määramise ja tulemuste kontrollimise võimalusi. Lisaks kirjeldati eraldi Selenium IDE tööriista, mida saab kasutada algsete testide automatiseerimiseks või abivahendina keerulisemate testide loomisel. Loodi reaalne juhis Webdriver testide käitamiseks Eclipse arenduskeskkonnas ning kirjeldati, millised sammud tuleb läbida, kui eksisteerivad testid, mille käitamiseks kasutatakse Apache Ant tarkvara (töö käigus ei kirjeldatud, kuidas antud tarkvara seadistada testidega suhtlemiseks). Lühidalt tutvustati ka TestNG ning ReportNG tarkvara, mida saab kasutada testide omavaheliste sõltuvuste määramiseks ning loetavamate raportite koostamiseks.

Töö tulemusena valmis algne kasutajaliidesepõhiste automaattestide komplekt, mida saab EAK projektis kasutada, abistamaks regressioontestimist ning kiirendamaks arendusprotsessi, andes tekkivate vigade korral sellest võimalikult vara teada. Automaattestide komplekt sisaldab teste sisselogimise, müügiarvete sisestamise ja ostuarvete sisestamise funktsionaalsuste kohta. Samuti valmisid testjuhtumid ja nimekiri rakenduse funktsionaalsustest, mida saab kasutada regressioontestimise plaanina.

Creating user interface based automated regression test suite

Bachelor's Thesis (6 ECTS)

Arli Türk

Summary

The main goal of this Thesis was to create initial user interface based automated regression test suite for EAK project to cope with testing of growing functionality and speed up the developement process. First, the regression testing was examined to understand that it is needed because possibility to introduce new bugs to old functionality while developing new components for application. While regression testing is vital part of software testing there are also some problems that it will create. One major problem was risk that critical bugs would not be found because there would not be enough time to test everything. To deal with this risk, the idea of priority assigning to tests cases was introduced. Having priorities, the tester will test most important functionality first and if time runs out then functionalities with lower priorities would not be tested as thoroughly as were test cases with higher priorities.

At second part of this Thesis automation was introduced to deal with growing functionality, routine work and human errors at testing. While automation is powerful tool to use if it is used in the correct way it also may cause loss of valuble resources if it is done in the wrong way. Most important thing is to remember that automation will not teach testers how to test, so testing process should be reviewed and if there are any existing problems then they should be dealt before starting to automate tests. In most cases it is not realistic to automate all the testing in project, so tests for automation should be chosen carefully. Testers should think if testing of some functionality requires exploratory testing, if yes then it is not a good candidate for automation. Also if output of test is hard to confirm by automated test (for example picture is generated) then it should be tested manually. But test cases that have large amount of data or that require a lot of routine are excelent candidates for automation (assuming that components to be tested are very stable and will not be changed in near future).

At third part of this Thesis Selenium Webdriver was introduced as the automation tool which was used to create automated tests for EAK project. It was explained that this tool was selected because of its ability to automate functional tests, posibility to create tests in Java programming language, report generating ability and because it is freeware application. Overview of Selenium IDE was given, that can be used to record trivial automated tests or to use as supportive tool for creating more complex tests with Webdriver. Data driven testing was shortly intorduced and also most important parts of Webdriver functionality were investigated like what possibilities are to create waiting logic, how to locate webelements and how to confirm results. It was explained how to set up environment for developing Webdriver tests in Eclipse developing environment. TestNG and ReportNG frameworks were intorduced to create dependencies between automated tests and to generate more readable reports.

As result of this thesis initial set of automated regression tests were created for EAK project along with test cases and list of application functionalities that can be used as regression test plan.

Kasutatud kirjandus

[1] One hour regression testing (19.01.2013) [online]
http://qatarun.wordpress.com/category/regression-testing/
[2] Hans Schaefer. Risk Based Testing, Strategies for Prioritizing Tests against Deadlines (27.01.2013) [online]
http://www.methodsandtools.com/archive/archive.php?id=31
[3] Cyclomatic complexity (03.02.2013) [online]
http://en.wikipedia.org/wiki/Cyclomatic_complexity
[4] Cem Kaner, James Bach, Bret Pettichord. Lessons learned in software testing, Wiley Computer Publishing. 2002.
[5] Andreas Spillner, Tilo Linz, Thomas Rossner, Mario Winter. Software testing practice : test management : a study guide for the certified tester exam : advanced level, ISTQB compliant. 2007. (03.02.2013) [online]
http://books.google.ee/books?id=tuzDZiEGRLgC&printsec=frontcover#v=onepage&q&f=false
[6] Hans Schaefer. Risk based testing (01.04.2013) [online]
http://www.cs.tut.fi/tapahtumat/testaus04/schaefer.pdf
[7] Pete Vasiliauskas. Which Tests Do I Automate? (02.04.2013) [online]
http://blogs.seapine.com/2009/07/which-tests-do-i-automate/
[8] 6 Tips to Get Started with Automated Testing (02.04.2013) [online]
http://smartbear.com/SmartBear/media/pdfs/6_Tips_for_Automated_Test.pdf
[9] Glossary of Vulnerability Testing Terminology (03.04.2013) [online]
https://www.ee.oulu.fi/research/ouspg/Glossary
[10] Pawel Brodzinski. What Is the Main Benefit of Writing Test Cases? (03.04.2013) [online]
http://brodzinski.com/2009/05/what-is-main-benefit-of-writing-test.html
[11] About Confluence (03.04.2013) [online]
https://confluence.atlassian.com/display/DOC/About+Confluence
[12] What is Selenium ? (04.04.2013) [online]
http://docs.seleniumhq.org/
[13] Continuous Integration and Release Management (04.04.2013) [online]
http://www.atlassian.com/software/bamboo/overview/beyond-build-automation

[14]Selenium IDE (04.04.2013) [online]
http://docs.seleniumhq.org/projects/ide/
[15] Platforms supported by Selenium (04.04.2013) [online]
http://www.seleniumhq.org/about/platforms.jsp
[16] Selenium documentation (04.04.2013) [online]
http://code.google.com/p/selenium/
[17] Samit Badle. Does Selenium IDE v1.0.11 Support Changing Formats? (04.04.2013) [online]
http://blog.reallysimplethoughts.com/2011/06/10/does-selenium-ide-v1-0-11-support-changing-formats/
[18] Daniel W. Dyer. ReportNg (05.04.2013) [online]
http://reportng.uncommons.org/
[19] Java Excel API Tutorial (05.04.2013) [online]
http://www.andykhan.com/jexcelapi/tutorial.html
[20] JExcel API (05.04.2013) [online]
http://jexcelapi.sourceforge.net/resources/javadocs/2_6_10/docs/index.html
[21] Issue 2067 - Hover with native events on Windows is not persisted (10.04.2013) [online]
http://code.google.com/p/selenium/issues/detail?id=2067
[22] WebDriver: Advanced usage (11.04.2013) [online]
http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp
[23] Rahul Poonekar. Waiting after web page interactions (11.04.2013) [online]
http://www.qaautomation.net/?p=490
[24] Apache Ant (11.04.2013) [online]
http://ant.apache.org/
[25] Selenium and TestNG (11.04.2013) [online]
http://testng.org/doc/selenium.html
[26] WebDriver API (12.04.2013) [online]
http://selenium.googlecode.com/svn/trunk/docs/api/java/index.html
[27] Rahul Poonekar. Locating page elements using WebDriver (12.04.2013) [online]
http://www.qaautomation.net/?p=388
[28] Iterative and incremental development (24.04.2013) [online]
http://en.wikipedia.org/wiki/Iterative_and_incremental_development

[29] Agiilse tarkvaraarenduse manifest (24.04.2013) [online]
http://agilemanifesto.org/iso/et/
[30] Atlassian Clover (24.04.2013) [online]
http://www.atlassian.com/software/clover/overview/find-risky-code
[31] Test design considerations (24.04.2013) [online]
http://docs.seleniumhq.org/docs/06_test_design_considerations.jsp#validating-results
[32] Mis on eArvekeskus? (02.05.2013) [online]
http://www.arvekeskus.ee/earvekeskus/

Lisad

Lisa 1. DVD bakalaureusetöö juurde kuuluvate failidega

Käesoleva bakalaureusetöö juurde kuuluv DVD sisaldab järgmisi faile:

1. EAK projekti raames arendatava rakenduse funktsionaalsuste nimekiri Funktsionaalsuste nimekiri.pdf;
2. Testjuhtum rakendusse sisenemise kohta Rakendusse sisenemine.pdf;
3. Testjuhtum müügiarve sisestamise kohta Müügiarve sisestamine.pdf;
4. Testjuhtum ostuarve sisestamise kohta Ostuarve sisestamine.pdf;
5. Automaattestide kaust „EAK automaattestid“ mis sisaldab testide lähtekoodi.

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina _____Arli Türk_____
(*autori nimi*)
(sünnikuupäev: _____05.02.1987_____)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
_____Kasutajaliidesepõhiste automatiseeritud regressioonitestide komplekti loomine _____,
(*lõputöö pealkiri*)

mille juhendaja on _____Helle Hein ja Artur Assor _____,
(*juhendaja nimi*)

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, **04.05.2013**