TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Johann Juhanson 142856IAPB

AUTOMAATTESTIDE JUURUTAMINE *ORACLE CONTENT MANAGEMENT* SÜSTEEMI NÄITEL

Bakalaureusetöö

Juhendaja: Maili Markvardt
MSc

Kaasjuhendaja: Irina Astrova
MSc

Tallinn 2017

Autorideklaratsioon

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

Autor: Johann Juhanson

21.05.2017

Annotatsioon

Lõputöö peamised eesmärgid olid: muuta *Content Management* teenuse testimist kiiremaks ning luua süsteemile automaattestid. Töö lahendatavateks probleemideks olid: kuidas automaatselt testida teenust, mis on ehitatud *Oracle Content management*`i peale ja mis on tähtsamad funktsionaalsused, mida peab testima ning milliseid töövahendeid automatiseerimiseks kasutada. Olulisim tulemus oli, et süsteemile loodi automaattestid ning testimise aeg vähenes kolmekordselt.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 29 leheküljel, 8 peatükki, 13 joonist, 0 tabelit.

Abstract

Automating tests on example of the Oracle Content Management system

This thesis is about automating a service what is built on Enterprise Content Management system called Oracle Webcenter Content. Oracle Webcenter Content is document management system, where users can check-in, delete, update and manage their documents. Until now testing was manual. Regression tests were performed maximum two times a month and it took 32 hours to complete it, because it took so long, i decided to automate it using Selenium and Selenide frameworks.

The main purpose of this thesis was to automate regression tests and make the testing process faster. The issues that needed to solve were: how to test a system what was built on Oracle Content Management system and what was the most important functionalities that needed to automate. Also important question was which automating frameworks to use for automating this service. The result of this thesis is automated tests. The time that was needed to complete regression tests was reduced three times. Also the tests became more trustworthy because the human error was removed. The automated tests benefit was valuated mostly by how much time did it consume compared to manual testing. The time cost to maintain tests were not included in the comparison because it is hard to evaluate.

The first chapter will introduce the concept of Enterprise Content Management and will show some solutions to it, like Oracle Webcenter Content. Then i will explain how the testing is currently performed and how the regression tests are done. After this i will look into automating tests in general. Then i will point out the most important functionalities that needs to be automated and look what kind of frameworks i used for it. The last part of this thesis are the realistion of tests, explaining it's architecture.

The thesis is in Estonian and contains 29 pages of text, 8 chapters, 13 figures, 0 tables.

Lühendite ja mõistete sõnastik

ECM	*Enterprise Content Management*
Kontent	Süsteemi kirje koos dokumendi ja tema metaandmetega

Sisukord

1 Sissejuhatus 8

2 Enterprise Content Management 9

2.1 Oracle WebCenter Content 9

3 Teenuse testimine praegu 11

4 Automaattestimine 12

4.1 Must ja valge kasti testimine 12

4.2 Testimise tüübid 12

4.3 Automaattestimise reeglid 14

4.4 Teenuse testide automatiseerimine 15

4.5 Kasulikkuse hindamine 15

5 Automatiseeritavad funktsionaalsused 17

6 Kasutatud tehnoloogiad 21

6.1 Selenium Webdriver 21

6.2 Selenide 21

6.3 J-Unit 21

6.4 JXL 22

7 Realisatsioon 23

7.1 *Check-in* testid 25

7.2 *Check-out* testid 26

7.3 Dokumendi operatsioonide testid 27

7.4 Versiooni kustutamise testid 28

7.5 Metaandmete asetuse testid 30

7.6 Kontode ja registreerimise numbrite testid 31

7.7 Otsimise testid 33

7.8 Dokumentide muutmise testid 34

7.9 Testide automatiseerimise järgmine etapp 35

8 Kokkuvõte 36

Kasutatud kirjandus 37

Lisa 1 – Kontode ja registreerimise numbrite testide kood 39

Jooniste loetelu

Joonis 1. *Check-in* leht. 20
Joonis 2. *Content info* leht. 20
Joonis 3. Automaattestide arhitektuur. 24
Joonis 4. *Check-in* testide tegevusskeem. 25
Joonis 5. *Check-out* testide tegevusskeem. 26
Joonis 6. *Check-out undo* testide tegevusskeem. 27
Joonis 7. Dokumendi operatsioonide testide tegevusskeem. 28
Joonis 8. Versioonide kustutamise testide tegevusskeem. 29
Joonis 9. Metaandmete kasutuse testide tegevusskeem. 30
Joonis 10. Kontode ja registreerimise numbrite testide tegevusskeem. 32
Joonis 11. Otsimise testide tegevusskeem. 33
Joonis 12. Dokumendi muutmise testide tegevusskeem. 34

1 Sissejuhatus

Oracle Content management on dokumentide halduse süsteem, kuhu kasutajad salvestavad ja kus haldavad oma dokumente. Praeguseni toimus teenuse testimine manuaalselt. Teenust testitakse musta kasti põhimõttel ehk kasutajaliidese põhiselt. Regressioonitestide tegemiseks kulus aega 32 tundi ja teste tehti maksimaalselt kaks korda kuus, mis võttis liiga palju aega. Seetõttu otsustati teenus automatiseerida, kasutades selleks Seleniumi ja Selenide teeke.

Bakalaureusetöö eesmärgiks on luua *Oracle Content management* süsteemi peale ehitatud teenusele automaattestid, mis muudaksid teenuse testimise kiiremaks, võrreldes testide manuaalselt tegemisega. Samuti on eesmärgiks muuta testid rohkem usaldusväärsemaks.

Peamine probleem, mille bakalaureusetöös lahendasin, oli - kuidas automaatselt testida teenust, mis on ehitatud *Oracle Content management* peale. Samuti oli ülesandeks leida, mis on süsteemi tähtsamad funktsionaalsused, mida peab automatiseerima. Nimelt automatiseerimise hea praktika näeb ette, et esimeste asjadena tuleb valida välja need funktsionaalsused, mis on teenuse toimimiseks kriitilised. Lõpuks pakkusin välja, kuidas luua antud automaatteste ja mis töövahendeid selleks kasutada.

Töö koosneb kuuest peatükist, milles esimeses vaatlen lähemalt, mis asi täpsemalt on *Enterprise Content management* ja selle lahendus *Oracle Content management*. Siis käsitlen töös teenuse testimist enne automaattestide loomist ning uurin põgusalt automaattestimise mudeleid, tüüpe, reegleid ja häid tavasid, kuidas teenust peaks automatiseerima. Neljandas peatükis käsitlen funktsionaalsusi, mida otsustasin automatiseerida. Viiendas peatükis näitan peamisi teeke ja tööriistu, mida kasutasin töö eesmärgi saavutamiseks. Viimases töö peatükis vaatlen loodud testide koodi struktuure iga testi lõikes ning kirjeldan testimise protsesse.

2 Enterprise Content Management

Enterprise content management-il (ECM) on palju definitsioone. Selle otsetõlge on - äri sisu haldamine. Sisu võib mõista mitmeti, kuid peamiselt tähendab see erinevaid ärile vajaminevaid dokumente. Definitsiooni erinevused lähtuvad peamiselt iga kasutaja kogemustest ja taustast. Definitsioon erineb ka sõltuvalt sellest, millise firma lahendust kasutatakse või mida mõeldakse sisu all. Näiteks arendaja iseloomustab seda teistmoodi, kui firmajuhid või isegi kasutajad. Üheks definitsiooniks võib võtta, et see on süsteemi lahendus, mis on disainitud haldama erinevaid organisatsiooni dokumente. Olgu need Wordi dokumendid, Exceli tabelid, PDF-dokumendid või skaneeritud pildid. ECM-i peamine ülesanne on muuta organisatsioonide töö paberdokumentidest vabaks, mis võimaldab neil rohkem keskenduda tulu teenimisele, kui paberite haldamisele. ECM-i peamisteks operatsioonideks on dokumentide salvestus, nende haldus ning hoidmine ja samas ka dokumentide hävitamine, kui dokumendid on aegunud või nende säilitustähtaeg möödub. ECM-i lahendusi on erinevaid. Näiteks pakub antud lahendusi Microsoft Sharepoint või Adobe Experience Manager, kus peamine *content* on veebidokumendid või veebilehel asuvad erinevad artiklid või plokid, mis omakorda koosnevad pealkirjadest, tekstist ja valikuliselt ka piltidest või videotest [1], [2]. Antud töös käsitlen peamiselt Oracle organisatsiooni poolt pakutud ECM lahendust [3], [4].

2.1 Oracle WebCenter Content

Oracle WebCenter Content on Oracle poolt pakutud ECM-i lahendus. WebCenter Contenti jaoks tähendab mõiste *content* virtuaalsel kujul olevaid dokumente. Need dokumendid võivad olla erinevat formaati tekstidokumendid. Üldlevinud formaatideks on pdf, odt ja doc. Et salvestada dokumente WebCenter contenti, on vaja teha uus *check-in*, mille käigus salvestatakse dokumentide metaandmed ja ka dokument ise. Metaandmed on andmed, mis iseloomustavad antud dokumenti. Näiteks võivad kuuluda pildi metaandmetesse pildi loomise aeg, koht, isik, pealkiri ja kirjeldus. Samamoodi on ka dokumentidega. Oracle WebCenter Content võimaldab lisaks dokumentide salvestamisele ka dokumentide otsimist, versioneerimist, kustutamist ja ka töö voogude

tegemist. Uue versiooni loomiseks on vaja teha eelmisest versioonist *check out* ja siis saab muuta dokumenti. Peale seda peab uuesti tegema *check in*. Kõik eelmised dokumentide versioonid hoitakse alles ning näidatakse kasutajale, kui kasutaja seda soovib. Versioone saab ka kustutada ja ka nendesse naasta, kui dokumendi muutus ei osutunud vajalikuks [4].

Dokumendi kustutamist võib teha manuaalselt, kuid seda saab ka teha automaatselt – sel juhul pannakse kirja, kui kaua dokumenti säilitatakse. Teatud kuupäeva möödudes saadetakse automaatselt kasutajale või süsteemi administreerijale kiri, et vastavad dokumendid kustutataks, mille peale võib antud kirja saaja otsustada, kas see dokument kustutada või mitte. Kui peale kirja saatmist ei tehta midagi, siis järgmisel päeval süsteemi poolt kindlaks määratud ajal dokument kustutatakse [4].

Töövoog (inglise keeles *workflow)* on tegevuste jada, mida teevad erinevad osapooled. Voogu võib alustada dokumendi *check in* või kasutaja enda poolt tehtud tegevus. Tavalises töövoos on mitu sammu, kus kasutajad peavad antud dokumendi kas kinnitama või tagasi lükkama. Kindlates sammudes peavad kasutajad muutma dokumenti või antud ülesande delegeerima kellelegi teisele, kui nad ise seda täita ei saa. Mingis sammus võib süsteem aga teavitada kõiki inimesi, kes olid dokumendiga seotud. Tavaliselt kasutatakse seda töövoo lõpus, et teavitada kasutajaid, et töövoog on lõppenud. Lisaks nendele töövoogude funktsionaalsustele võib süsteem teha ka teisi operatsioone antud dokumendiga. Näiteks dokumendi registreerimine, mille jooksul lisatakse unikaalne väärtus dokumendi metaandmetesse [4].

3 Teenuse testimine praegu

Peamiselt jaguneb teenuse testimine kaheks: *taski*-põhised ja regressioonitestid. Testimine toimub manuaalselt - see tähendab, et testija teeb kõik võimalikud operatsioonid käsitsi, kasutades antud teenust. Kuna toimub ainult kasutajaliideste põhine testimine, siis koodi kuju ei peeta tähtsaks, vaid kasutajatel peab olema töötav funktsionaalsus.

Taski-põhine testimine tähendab, et kui luuakse uut arendust, siis ülesande nõuete põhjal mõeldakse välja võimalikult palju erinevaid situatsioone, mismoodi antud arendus võis teenuse erinevaid funktsionaalsusi lõhkuda. Mispeale proovitakse need läbi mängida. Samuti testitakse, et kogu funktsionaalsus, mis on nõuetes kirjas, oleks ka reaalsuses implementeeritud. Kui arenduses luuakse mingit funktsionaalsust, mis on piisavalt kriitiline teenuse töötamiseks, lisatakse antud testi juhud ka regressioonitestide hulka.

Regressiooniteste tehakse maksimaalselt kaks korda kuus, et oleks kindel, et süsteem töötab korralikult. Regressioonitesti juhud on kirjeldatud Exceli testitabelis, kuhu on kirja pandud testi kirjeldus ja oodatav tulemus ning samuti, kas test on läbitud või mitte. Kokku on testi juhtusid tabelis 200. Nende alla kuuluvad *check-in-i*, otsingu, kasutaja õiguste, kasutaja kontode genereerimine, lehe asetuse, sisu operatsioonide, keelte, töövoo ja ühendatud dokumentide testid.

Automatiseerimise käigus soovisime automatiseerida regressiooniteste. Põhjuseks oli, et testide tegemine kaks korda kuus võttis liiga palju aega ja osad testid polnud nii usaldusväärsed, kuna testija ei suuda kõiki vigasid avastada. Siiani võtsid testid aega 32 tundi. Kõige rohkem võtsid aega lehe asetuse, kasutaja õiguste ja kontode genereerimise testid, kus keskmiselt õiguste peale kulus 4 tundi, kontode peale 14 tundi ja asetuse peale 6 tundi. Kõige rohkem testija vigu tekkis kontode genereerimise testimise käigus.

4 Automaattestimine

Automaattestimine on tarkvara testimine, mida teeb kindel valmiskirjutatud tarkvara, mitte inimene. On olemas erinevad tarkvara lahendusi, mis pakuvad erinevaid testimise liike. Oma testimise automatiseerimiseks on just vajalik leida endale õige soovitud testimise liik ja tarkvara lahendus.

4.1 Must ja valge kasti testimine

Esimeseks mooduseks, kuidas jagada testimist, on must ja valge kast, mis tulevad ingliskeelsetest sõnadest *black and white box*. Nii must kui ka valge kast on testimise meetodid. Must kast tähendab seda, et testides ei teata koodi struktuuri või disaini, vaid tehakse kogu testimine lõpp-produkti põhjal. Tavaliselt tehakse musta kasti testimist tarkvara vastuvõtu testides. Samas aga valge kast tähendab, et testimise ajal näeme me koodi struktuuri ja testime ka koodi implementatsiooni. Tavaliselt kasutatakse valge kasti testimist ühik-testimise tasemel, vastuvõtu või integratsioonide testimisel. Kahe testi vahe selgitamiseks toon veebilehe näite, mida testitakse. Valge kasti testimisel teame, kuidas on veebilehe html-struktuur üles ehitatud ning vaatame ka html-koodi struktuuri või erinevate vajalike koodi klassidesse. Musta kasti puhul meid kood ei huvita, vaid vaatame, et terve rakenduse funktsionaalsus toimuks õigesti. Tavaliselt on ka kolmas kast, mida nimetatakse halliks kastiks, mis on musta ja valge kasti segu [5].

4.2 Testimise tüübid

Peamiselt võib jagada testimise tüübid järgnevatesse kategooriatesse: funktsionaalne, ühik-, võtmesõna juhitud testimine, andmetest juhitud testimine, regressiooni-, visuaalne, koormustestimine ja jõudluse testimine [6].

Funktsionaalne testimine testib antud süsteemi funktsioneerimist ning kuidas ta suhtleb kasutajaga. Teisisõnu - funktsionaalne testimine testib seda, kuidas süsteem töötab ning kas temal on kõik vajalikud operatsioonid olemas [7].

Ühiktestimine on süsteemi kõige väiksemate osade testimine - see tähendab, et testitakse süsteemi koodi ja meetodite korrektset käitumist ning on vajalik selleks, et

teada, kus täpselt viga võib tekkida. Tavaliselt kirjutab ühik-teste mitte testija, vaid koodi arendaja, kes siis testide abil saab kontrollida, kas tema kood käitub õigesti [8].

Võtmesõna juhitud testimine (inglise keeles *keyword-driven tests*) on testimise viis, kus kõik tegevused on võtmesõnad. Võtmesõnadeks on erinevad tegevused testitavas süsteemis, näiteks nupule vajutamine või hiire rulliku liigutamine. Põhimõtteliselt määrame ära nimekirja vajalikest operatsioonidest, mida peab süsteem tegema, et käituda nagu kasutaja. Samuti läheb võtmesõna juhitud testimise alla olukord, kus lastakse test-tarkvaral salvestada kasutaja tegevusi, et siis hiljem ise need läbi mängida [9].

Andmetest juhitud testimine tähendab, et testimisel kasutatakse testidevälist informatsiooni. Näiteks võib tekkida olukord, kui funktsionaalse testimise jooksul mingisuguse vormi täitmisel paneme vormi väljadesse ainult mingid kindlad väärtused, aga nende test-väärtustega ei tule viga välja. Sel juhul kasutatakse andmetest juhitud testimist, mis koosneb mingist suuremast hulgast andmetest, mida sisestatakse antud süsteemi. Teisisõnu lisatakse süsteemi palju erinevaid väärtusi, testimaks, et süsteem neid aktsepteerib [10].

Regressioonitestimine tähendab testimist, mida tehakse regulaarselt, et kontrollida praeguse tehtud süsteemi ja arenduse korrektset käitumist. Erinevalt ühik-testidest teeb regressiooniteste süsteemi testija. Antud teste peaks tegema iga natukese aja tagant, kuna funktsionaalse testimise käigus ei pruugi testija avastada vea tekkimist antud arendusest sõltuvas teises arenduses [11].

Visuaalne testimine tähendab olukorda, kui testitakse antud teenuse või rakenduse visuaalset kujundust, peamiselt erinevaid kasutajaliideseid. Testitakse nii taustavärve, kui ka fonte ja teisi kujundust mõjutavaid elemente. Üks võimalus, kuidas testida, on teha eelmisest kujundusest pilti ning siis võrrelda seda praeguse kujundusega. Erinevused nende kahe pildi vahel tuuakse välja [12].

Koormustestimine tähendab, et testitavale süsteemile genereeritakse kindla suurusega koormus ja vaadatakse, kuidas süsteem sellega hakkama saab. Koormuseks võib olla kas virtuaalsed kasutajad või andmete voog. Peamine on, et süsteem täidaks oma ülesandeid ja koormuse testimise ajal ei lakkaks töötamast [13].

Jõudluse testimine - erinevalt koormuse testimisest - tähendab, et vaadeldakse, kui kiiresti süsteem suudab mingeid kindlaid operatsioone teha. Vajalik selleks, et tuvastada, kui kaua kasutaja peab ootama vastust antud süsteemi kasutades. Samuti selleks, et avastada koodi jõudluse nõrkkohti, et koodi kiiremaks refaktoreerida [13].

4.3 Automaattestimise reeglid

Testide põhimõtteline struktuur peab olema jaotatud kolmeks. Esimeseks sammuks testide käivitamisel on erinevate operatsioonide tegemine, milles me valmistame testitava keskkonna ette testimiseks. Näiteks võime selles sammus logida süsteemi sisse. Teiseks sammuks on antud keskkonnaga operatsiooni tegemine. Näiteks nupule vajutus. Kolmandaks ja viimaseks sammuks on tehtud operatsiooni tulemusel tekkiva staatuse kontroll, mis antud näites võiks olla veebilehe muutus. Tavaliselt on soovitatav kontrolli sammus teha ainult üks võrdlus ehk *assert*. Antud reeglit ei pea alati jälgima, aga kindlasti peaks jälgima, et testi funktsioonis testitaks ainult ühe peakoodi kontseptsiooni [14].

Samuti on tähtis, et testide loomisel kasutataks samasugust lähenemist, nagu testitava teenuse koodi kirjutamisel. Eriti tähtis on, et testide kirjutamisel kasutataks *clean code´i* põhimõtteid. Seda seetõttu, et kui kood pole puhas, siis temast on raske aru saada ning seetõttu võtab testide koodi haldamine kauem aega ning võib tekkida olukord, et peab automaattestide koodi uuesti kirjutama [14].

Testid peaksid jälgima *FIRST* põhimõtet. *FIRST* tähendab, et testid peaksid olema *fast* (kiired), *independent* (sõltumatud), *repeatable* (korratavad), *self-validating* (iseennast kontrollivad) ja *timely* (ajakohased). Testid peavad olema kiired seetõttu, et testijatel saaks testitava süsteemi kohta võimalikult kiiresti tagasisidet. Samuti - kui testid on kiired, siis on suurem tõenäosus, et neid käivitatakse tihedamini. Testid peavad olema teineteisest sõltumatud, see tähendab seda, et üks test ei loo teisele testile vajalikke algstaadiume. Mis on omakorda tähtis sellepärast, et kui üks test peaks läbi kukkuma, siis see ei mõjuta teist testi. Kõik testid peaksid ka olema *repeatable* ehk korratavad. Põhimõtteliselt tähendab see seda, et iga kord, kui testid käivitada, siis peaks süsteem käituma alati samamoodi ning võimaldama testidel testida alati samu funktsionaalsusi. *Self-validating* tähendab olukorda, et testide väljund peaks olema alati *boolean* tüüpi, muidu on raske aru saada, kas testid läbiti. Viimaseks kriteeriumiks on, et testid peaksid

olema *timely* ehk õigesti ajastatud. See on vajalik, sest kui testid kirjutatakse peale koodi tegemist, võib välja tulla, et kirjutatud koodi ei ole võimalik testida. Samuti peaks olema testi kood kirjutatud ja hooldatud samaaegselt, kui funktsionaalsus, sest muidu võivad testid vananeda [14].

4.4 Teenuse testide automatiseerimine

Teenuse automatiseerimiseks on vaja teha peamiselt järgnevad sammud. Esiteks peab otsustama, mis test juhud üldse automatiseerida. Kõige tähtsam on automatiseerida sellised test juhud, kus võib tekkida inimvigu või testid, mis vajavad mitmeid erinevaid andmete sisendeid erinevatest kohtadest. Samuti on tähtis automatiseerida test juhud, mida on raske manuaalselt testida või test juhud, mis on seotud tähtsaima funktsionaalsusega, kuhu kindlasti ei tohi vigu tekkida. Teiseks sammuks on valida õige automatiseerimise vahend ning seda sõltuvalt firma vajadusest ja testimise tüübist. Järgmiseks peab looma head ja kvaliteetsed test-andmed. Test-andmeid hoitakse tavaliselt testide välises failis, kust andmeid loevad automaattestid. Test-andmete loomisel peab silmas pidama erinevaid testimise piirjuhte ehk kõik erinevad kombinatsioonid, mis võivad rakenduse lõhkuda, peavad esindatud olema. Viimaseks sammuks on automaattestide kirjutamine ning siin peab samuti arvestama asjaoluga, et kood töötaks iga brauseri või kasutajaliidese muutusest sõltumata [6].

4.5 Kasulikkuse hindamine

Peamine viis, kuidas hinnata automaattestide kasulikkust, on nende kuluaeg, võrreldes manuaalselt tegemise peale kulunud ajaga. Automaattestide aja sisse arvestatakse tavaliselt ka automaattestide muutmine ja uute lisamine. See tähendab, et peab arvestama lisaks kulunud testide käimise ajale testide haldamise aja, et saada teada, palju aega automaattestidele kulus. Kui testidele kulus aega rohkem, kui manuaalselt tehes, siis peab automaattestide vajalikkuse üle vaatama või neid refaktoreerima, sest muidu tekib testijatele lisatööd juurde. Loomulikult peab arvestama asjaoluga, et mõningates olukordades on automaattestid kasulikumad, sest nad avastavad vigu, mis võib inimvea tõttu märkamata jääda. Kuid automaattestidel on ka miinuseid. Nimelt nad ei tuvasta nuppude asukohti või ärivigu rakenduses, kui just ei ole tegu visuaalse testimisega. Niisiis automaattestide kasulikkust saab hinnata peamiselt aja põhjal ja

nende abil saab teha teste, mis käsitsi ei oleks võimalikud või oleks väga keerulised. Näiteks võib tuua süsteemi koormustestid.

5 Automatiseeritavad funktsionaalsused

Valisin automatiseerimiseks järgnevad funktsionaalsused: *check-in*, *check-out*, otsing, kasutaja kontode genereerimine, dokumendi registreerimise numbrid, dokumendi muutmine, versioonide kustutamine, lehe metaandmete asetus ja operatsioonid. Automaattestidega kaeti 72% funktsionaalsusest, mis olid kirja pandud eelnevalt regressioonitestide Excel tabelisse. Antud protsent arvutati selle Excel tabeli põhjal. Osa testidest välja jäänud funktsionaalsust otsustasin praegu mitte automatiseerida. Samuti jätsin välja osa funktsionaalsust, mida pole mõistlik automatiseerida, kuna nende testide loomisele ja haldamisele kuluv aeg on liiga suur, võrreldes testide manuaalselt tegemisega. Lisaks oleks antud funktsionaalsuse testimise implementeerimine struktuurselt liiga keeruline ja automatiseeritud testide jooksmisele kuluv aeg ei ole palju lühem kui manuaalselt tehes.

Check-in funktsionaalsus on antud teenuse üks tähtsamaid funktsionaalsusi, kuna selle käigus salvestatakse süsteemi uusi dokumente. Seetõttu otsustasin antud testid ka automatiseerida. Et salvestada dokumente teenuses, on vaja kasutajal esiteks valida profiil, mille dokumenti ta soovib sisestada. Mõiste „profiil" tähendab valmis määratud metaandmete hulka, mille andmeid me tahame antud dokumendi jaoks salvestada. Näiteks võime võtta kaks erineva formaadiga faili. Oletame, et üheks on pilt ja teine on tekstidokument. Mõlemal on metaandmeteks loomise kuupäev, aga erinevalt tekstidokumendist on pildil lisaks metaväli „isikud pildil". Tähendab, et pildi profiili sisse kuuluvad samad väljad kui tekstidokumenti, aga on lisaks metaväli „isikud pildil". Peale profiili valimist peab kasutaja täitma kõik kohustuslikud väljad ja soovi korral ka mittekohustuslikud ning valima oma arvutist dokumendi, mida ta soovib salvestada. Antud lehte on ka võimalik näha joonisel 1. „Esita" nuppu vajutades näidatakse kasutajale kõik salvestatud andmed. Antud menüüd nimetatakse *content menu*-ks, mida on ka näha joonisel 2.

Check-out funktsionaalsus on vajalik just versioneerimiseks. Antud operatsiooni tegemiseks on vaja valida kontent, millele tahetakse uus versioon luua. Peale mida kasutaja valib operatsiooni *check-out*. Siis saab lisada vana dokumendi asemele uue ja

vajadusel täita osa metaandmeid ning peale seda peab vajutama „esita“ nuppu, mille järel salvestatakse dokument uue versioonina. Vana versioon aga jääb alles. Otsustasin antud funktsionaalsuse automatiseerida, sest seda oli võimalik piisavalt kiiresti ja lihtsalt teha.

Oracle Webcenter Contenti kasutaja õigused on implementeeritud kasutaja kontode ja kasutaja gruppide ühisosana. See tähendab, et kasutajal peab olema kindlale kontole ligipääs ja lisaks peab ta kuuluma kindlasse kasutajagruppi. See on lahendatud niimoodi, sest kui meil on osakond, kelle töötajad kuuluvad samasse gruppi, siis nad saavad oma dokumente lisada teenusesse, aga näiteks ainult ülemusel on õigus neid dokumente kustutada. Selle jaoks antakse tervele grupile õigus dokumente lisada, aga ainult kindla kasutajakontoga saab dokumente kustutada. Seetõttu on tähtis, et kontod genereeritaks õigesti, sest neid võrreldakse kasutaja omaga ja kui need on samad, siis antud kasutajal on õigus antud dokumendile. Testitavas teenuses aga genereeritakse kontod paari metavälja järgi. Need metaväljad on *select*-tüüpi väljad. Kokku on erinevaid kombinatsioone ligi 300. Otsustasin antud funktsionaalsust automatiseerida seetõttu, et manuaalselt testides kulus selle testimiseks 14 tundi ning kuna kombinatsioone on palju, siis tekkis ka palju vigu manuaalselt testides.

Kasutaja õiguste funktsionaalsus tähendab seda, et võrreldakse kasutajate kontosid ja gruppe ja selle põhjal lastakse kasutajal teenusega mingeid funktsionaalsuseid teha. Võrreldes antud teste kasutaja kontode genereerimise testidega, siis siin testitakse, kas kasutaja saab oma kontoga talle võimalikke operatsioone teha ja mitte midagi muud. Antud funktsionaalsust otsustasin mitte automatiseerida, kuna see ei oleks muutnud testijate tööd kiiremaks, kuna testide haldamisele kulunud aeg ei oleks tasunud ära, võrreldes manuaalselt tegemisega. Lisaks oleks struktuur olnud mitu korda keerukam.

Registreerimise numbrid on vajalikud kasutajale dokumendi otsimiseks. Registreerimise numbrid genereeritakse süsteemi poolt automaatselt ja salvestatakse metavälja. Genereerimist võib alustada kasutaja ise, kui ka töövoog ja dokumendi *check in*. Samamoodi kontode genereerimisega on registreerimisnumbritel palju erinevaid kombinatsioone, mis teeb selle testimise samuti pikaks ja keeruliseks. Seetõttu otsustasin ka registreerimise numbri genereerimise automatiseerida.

Dokumendi muutmise operatsioon on üks lihtsamalt automatiseeritavaid funktsionaalsusi, kuna selle jaoks tehtavad operatsioonid on suhteliselt sarnased *check in* ja *check out* funktsionaalsusele. Otsustasin antud funktsionaalsuse automatiseerida, kuna muutmine on kasutajatele tähtis funktsionaalsus ja automatiseerides sai antud operatsiooni testimist kiirendada. Selle jaoks, et dokumenti muuta, peab kasutaja valima kontenti ja sealt valima nupu „muuda“, mispeale saab metaandmeid muuta. Võrreldes muutmisoperatsiooni *check-out* omaga, siis siin ei saa muuta dokumenti, vaid ainult dokumendi metaandmeid.

Lisaks dokumendi versioonide loomisele on tähtsal kohal ka vanade või valesti sisestatud versioonide kustutamine. Kustutada saab Webcenter Contentis nii vahepealseid kui viimaseid versioone. Kui kasutaja on *content info* menüüs, siis näeb kasutaja kõiki versioone, mille kõrval on ka kustutamise operatsioon. Kui kasutaja otsustab kustutada, küsib süsteem kasutajalt enne kustutamist kinnitust. Kui otsustatakse antud dokument ikkagi kustutada, siis kaob see versioonide loendist. Otsustasin antud operatsiooni lisaks *check out* operatsioonile automatiseerida, kuna nad on kasutajale sama tähtsad.

Dokumentide asetus lehtedel on tähtis kasutaja mugavuseks. Asetuse all mõistame nii metaandmete järjestatust, kui ka, et ei oleks lisa-metaandmete välju, ega neid ei oleks puudu. Antud funktsionaalsust oli ajamahukas testida, kuna igal lehel on oma metaandmete järjestatus ning me pidime võrdlema kogu aeg lehe kujundust teiste keskkondadega, millele ei olnud viimast arendust veel peale pandud, või oli välja trükitud kujundusega paberid. Samuti kuna osaliselt on lehe kujundused sarnased, siis ei tuvastatud alati viga. Kujunduse alla lähevad ka *check in* lehe kui ka *content info* lehe metaandmete asetus. Antud funktsionaalsus automatiseeriti.

Dokumendi operatsioonide testimise alla lähevad erinevad operatsioonid, mis lisaks Webcenter Contenti poolt pakutud lahenduse alla ei lähe. Kuna operatsioonid on väga profiilipõhised ja erinevad, siis praegu otsustasin ainult automatiseerida publitseerimise funktsionaalsuse. Vajalik on see sellepärast, et antud teenuses hoitakse ka dokumente, mida kasutavad teised teenused ning kui dokument on publitseeritud, siis näidatakse seda teises teenuses.

Üks funktsionaalsus, mida otsustasin mitte automatiseerida, on töövoog. Põhjuseks on, et praegu pole töövood veel arenduse lõppstaadiumis ning seetõttu nende lahendus võib veel palju muutuda ja testijate testide arenduse aja kokkuhoiuks otsustasin oodata, kuni need valmis saavad.

Joonis 1. *Check-in* leht.

Joonis 2. *Content info* leht.

6 Kasutatud tehnoloogiad

6.1 Selenium Webdriver

Selenium on tarkvara teek, mille abil saab automatiseerida süsteemi testimist brauseris. Selle abil saab luua kasutajaliidese teste või erinevate brauseri operatsioonide skripte. Selenium Webdriveri on Seleniumi objekt-orienteeritud API, mille abil saab programmeerimiskeelest käivitada erinevaid brausereid ja teste jooksutada. Tema plussideks, võrreldes eelnevate Seleniumi versioonidega on, et temaga saab käivitada kõiki suuremaid veebibrausereid, mitte ainult Mozilla Firefox-i. Samuti on ta arhitektuurselt lihtsam ja kiirem [15]. Valisime antud teegi sellepärast, et otsustasime teha kasutajaliidese põhised testid ja polnud võimalik luua antud süsteemile ühikteste, sest suurem osa arendusest on tehtud süsteemi konfigureerides või suurem osa meetodeid ei tagastanud midagi. Lisaks sellele mõjutas antud testimise teegi valikut see, et antud teek on populaarne ja tasuta [16].

6.2 Selenide

Selenide on tarkvara teek, mis kasutab Selenium Webdriver-it testide automatiseerimiseks. Selenide plussiks on see, et ei pea ise kirjutama koodi brauseri sulgemiseks ja ei pea haldama brauseri aegumist. Samuti võimaldab see hallata kiiresti testide vigu, selleks salvestades ekraanitõmmiseid brauseri lehest, kus viga tekkis. Lisaks on selle abil kerge teste kirjutada, sest koodi ridade arv väheneb tunduvalt võrreldes Selenium Webdriveriga ja kood on loetav. Antud asjaolud ongi põhjuseks, miks ma valisin Seleniumile juurde Selenide [17].

6.3 J-Unit

J-Unit on lihtne teek, mille abil saab kirjutada javas ühik-teste. Ta pärineb xUnit teegi perekonnast. Testimise automatiseerimise juures kasutasin J-Unit teeki mitte ühik-testide loomiseks, vaid süsteemitestide jaoks. Teek määrab ära testi klasside struktuuri ja kuidas teste käivitatakse [18].

6.4 JXL

JXL ehk Java Excel API on avatud lähtekoodiga API, mille abil saab luua, lugeda, kirjutada Excel faili tabelitesse. See on väga populaarne Selenium testides, kus sellega saab lugeda infot Excel tabelitest. Samuti on see lihtsasti arusaadav ja tasuta. Antud projektis kasutatakse seda kontode osade sisselugemiseks Excel tabelist [19].

7 Realisatsioon

Testide struktuur on peamiselt jaotatud neljaks suureks paketiks. Nende pakettide nimedeks on `tests`, `metadata`, `functions`, `constants`. Testi paketi alla kuuluvad kõik testi klassid ja veel kaasaarvatud `setup test` klass, mida vaatleme lähemalt hiljem. `Metadata` paketi alla kuuluvad `forms` klass ja teda pärivad erinevate profiilide `metadata` klassid. Seal säilitatakse infot erinevate metaandmete kohta. Joonisel 3 on näha näite profiilideks `internal` ja `standard` profiile. Kolmandasse paketti kuulub `functions` klass, kus peamiselt asuvad erinevad staatilised meetodid, mille abil tehakse teste. Näiteks meetod, millega saab täita kõik metaandmete väljad. Neljandasse paketti kuuluvad erinevad konstantsed väärtused. Nende alla lähevad klassid: `accounts`, `universal` ja `operations`. Lisaks neile neljale paketile on veel `accounts` Exceli tabel, kus hoitakse kõiki kasutajakonto kombinatsioonitükke, mille abil `functions` klass kombineerib kasutajakontod.

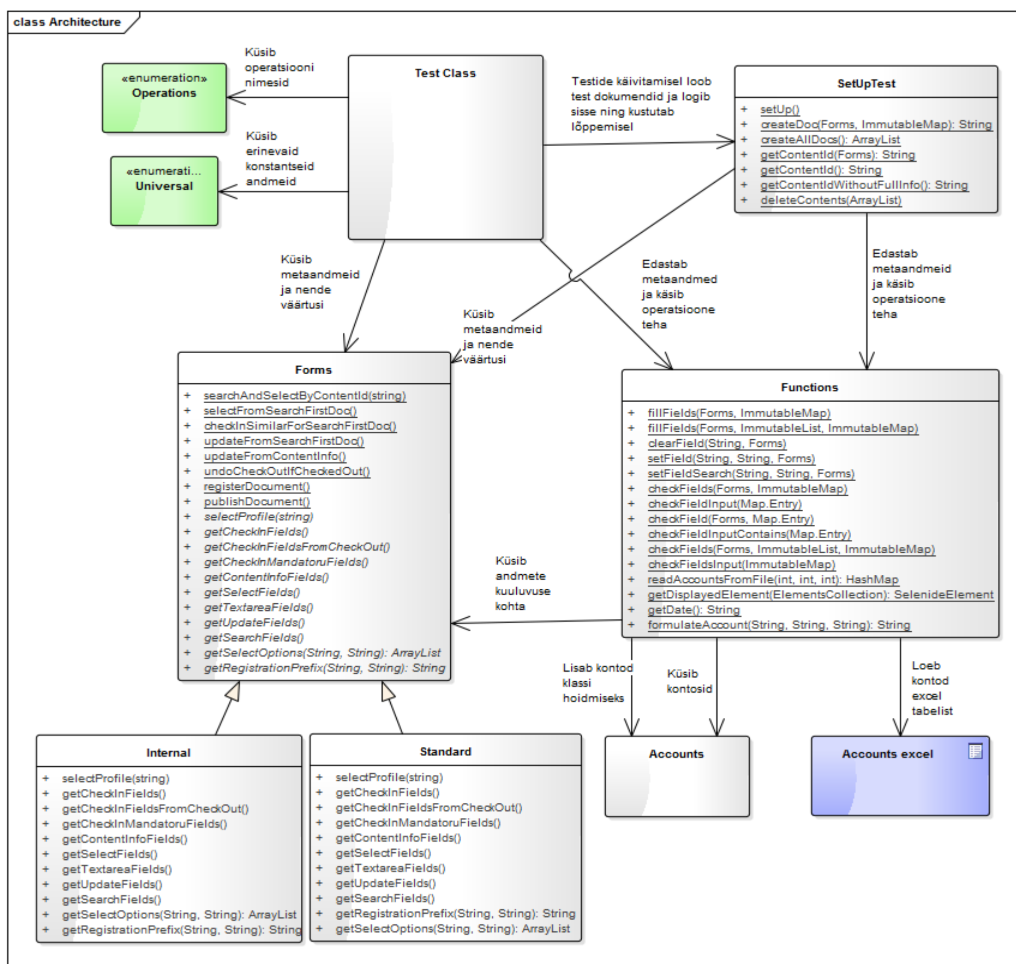
Testide üldine struktuur näeb ette, et testide tegemise ajal juhivad kõiki tegevusi testide klassis asuvad testid või abimeetodid. Kui testide klassi käivitatakse, siis alati esimese asjana käivitatakse meetodid `setup test` klassis. See on vajalik selle jaoks, et logida sisse antud vajalikku keskkonda, mis on ära määratud `universal` klassis. Antud lahendus võimaldab käivitada teste ka teistel keskkondadel. Lisaks sisselogimisele valmistab `setup` klass ette erinevad dokumendid, mida saavad siis kasutada testid, kes ise dokumente ei loo. Iga testi klass ise otsustab, kas ta laseb luua `setup testil` antud dokumendid või mitte. Alati testi lõpus kustutab `setup test` klass kõik protseduuri jooksul või alguses loodud dokumendid, seda seetõttu, et süsteemi ei jääks üleliigseid ja kasutuid dokumente ning antud keskkonna kasutaja ei näeks loodud dokumente.

Lisaks eelnimetatud klassile suhtleb testide klass funktsioonide klassiga ja metaandmete paketiga. Testide klassis on ette ära määratud, missuguse metaandmete klassiga hakkab suhtlus käima. Näiteks kui on vaja testida standard-profiili, siis test-klass suhtleb metaandmete paketis `standard` klassiga. Metaandmete pakett koosneb põhiliselt juur

klassist `forms` ja teda pärivatest profiili klassidest. `Forms` klassi on ka lisatud funktsionaalsus lehtede vahel liikumiseks. Peamiselt küsib test klass metaandmetelt erinevaid väärtusi ja metaandmeid, mille ta esitab funktsioonide klassile, kes nende andmete põhjal täidab välju või kontrollib neid. Samuti võivad funktsioonid teha erinevaid operatsioone. Funktsioonid võivad otse pärida ka metaandmete paketist infot, seda seetõttu, et vähendada testide klassi tulevat andmete voogu.

Operatsioonide klass on vajalik selle jaoks, et anda funktsioonidele ette, missugust operatsiooni peab tegema, kuna osad funktsioonid on kirjutatud nii, et teda saaks kasutada erinevate operatsioonide jaoks, kuid lisaks on vaja mõningaid teistsuguseid tegevusi teha. `Universal` klass on peamiselt konstantidest koosnev.

Funktsioonide klass suhtleb ka `accounts` Exceli tabeliga, kust ta saab kõik vajalikud kontode tükid, mille ta salvestab hoidmiseks kontode klassi. Vajadusel võtab ta sealt kontode tükid ning koostab nende põhjal vajaliku konto.

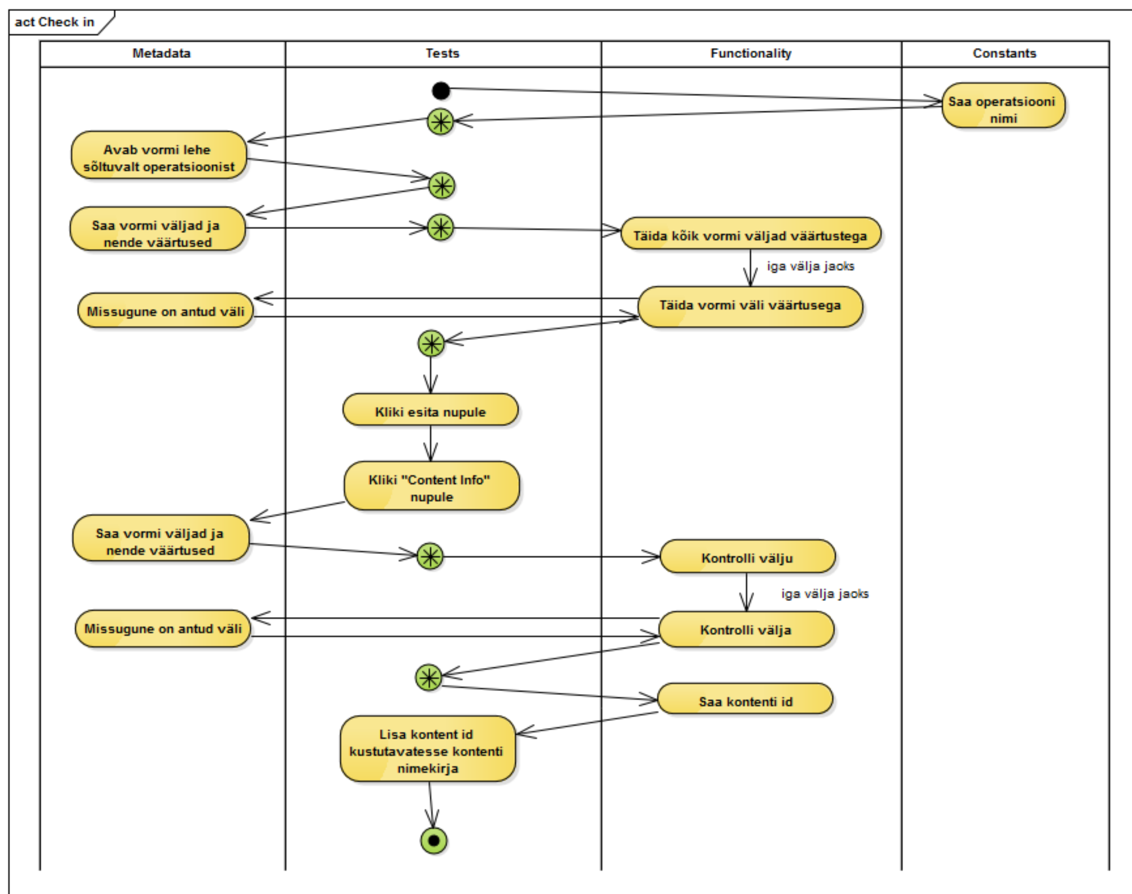


Joonis 3. Automaattestide arhitektuur.

7.1 *Check-in* testid

Antud testides testitakse *check-in* funktsionaalsust. Selle jaoks tehakse uus *check-in* ja täidetakse kõik väljad ja siis esitatakse antud profiil, peale mida kontrollitakse, kas antud metaandmete väljad salvestusid ja on lisatud dokument kontentile. Antud testid võtsid aega 14 minutit ja 54 sekundit. Manuaalselt tehes võtsid testid aga ligikaudu tund aega. Lisaks on antud testid rohkem usaldusväärsed, kui manuaalselt tehes, sest inimesel võivad minna kahe välja väärtused sassi ja kui süsteem vahetab kahe välja väärtused omavahel ära, siis seda ei tuvastata.

Joonisel 4 on näidatud rohelise märgiga olukord, kui tegevus naaseb tagasi test klassi, mis omakorda otsustab järgmise tegevuse käivitada. Samuti on järgneval joonisel näha detailsemalt, kuidas väljade täitmine ja kontrollimine käib.

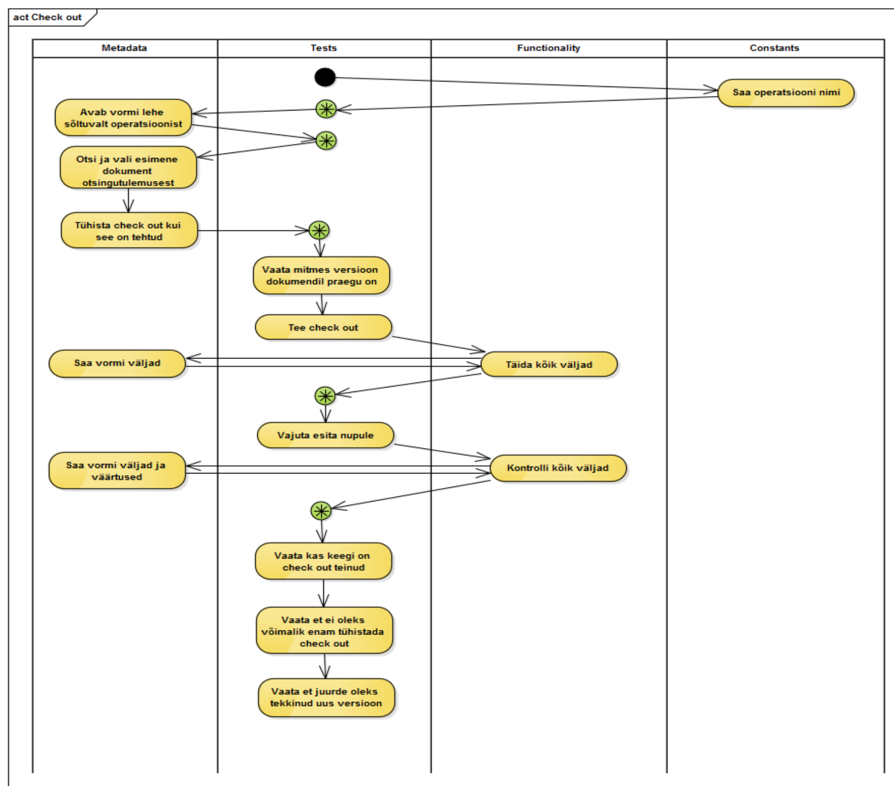


Joonis 4. *Check-in* testide tegevusskeem.

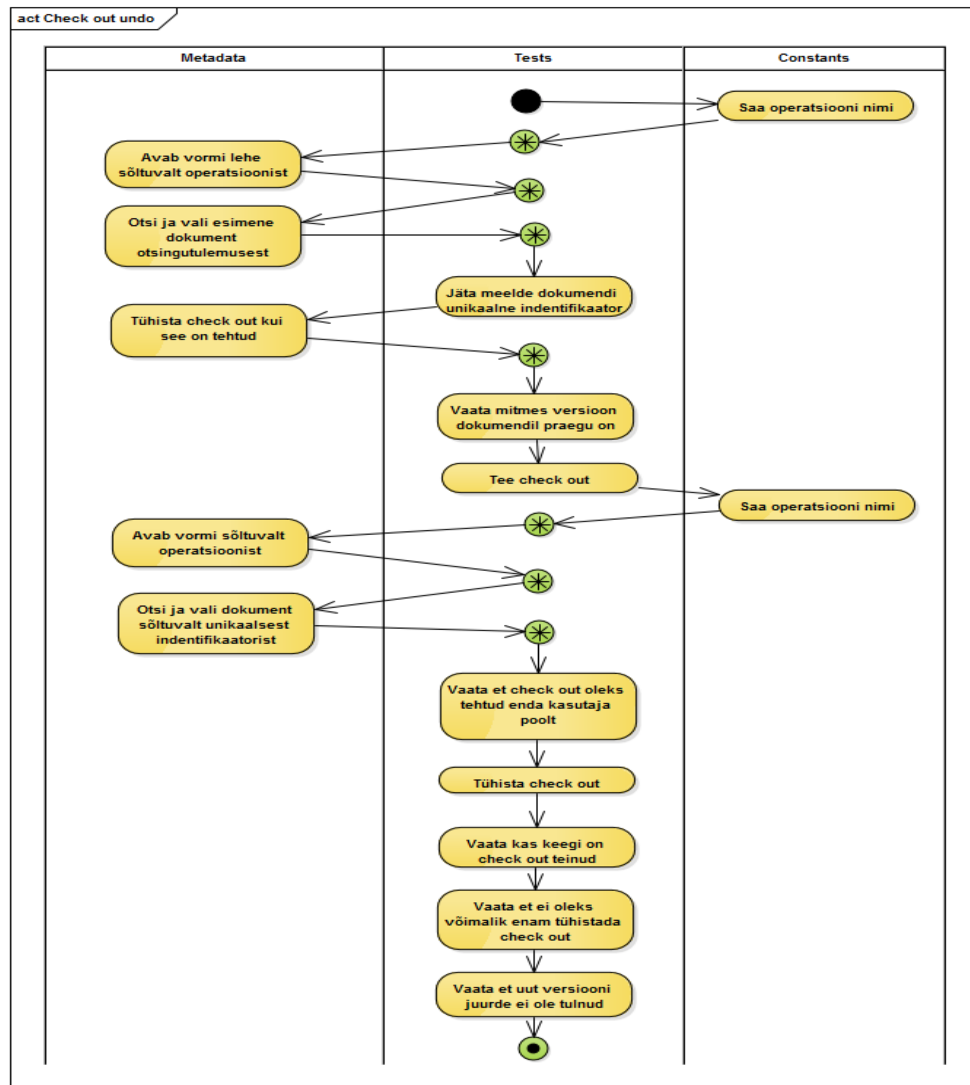
7.2 *Check-out* testid

Check-out testid jagunevad kahte kategooriasse. Esimesse kategooriasse lähevad testid, mis testivad tavalist *check-out* tegemist. See tähendab, et testid alguses jätavad meelde, mitmes versioon dokumendist praegu on ja peale seda teevad *check-out* dokumendile. Siis täidetakse kõik vajalikud väljad ja tehakse uus *check-in*, peale mida kontrollitakse, et tehtud operatsiooni ei saa tühistada ning uus versioon dokumendist on olemas. *Check-out* tegevuskeemi on näidatud joonisel 5. Teise kategooriasse lähevad testid, millel *check-out* jooksul tühistatakse *check-out*. Sellisel juhul kontrollitakse, et dokumendil enam ei oleks *check-out* tehtud ning uut versiooni ei oleks ilmunud. Antud testide tegevusskeemi joonisel 6.

Peale automatiseerimist võtsid testid aega 10 min ja 6 sekundit. *Check-out* teste algselt manuaalselt ei tehtud kõikide profiilide jaoks, vaid valiti välja kindlad profiilid, mille jaoks neid tehti. *Check-out* testid kuulusid sisu operatsioonide testide alla, mis kokku aega võtsid 2 tundi, millest ligikaudu 20 minutit võtsid *check-out* testid. Niisiis automatiseerimise käigus lisaks testide tegemise aja vähendamisele, suurendati ka süsteemi testide hulka.



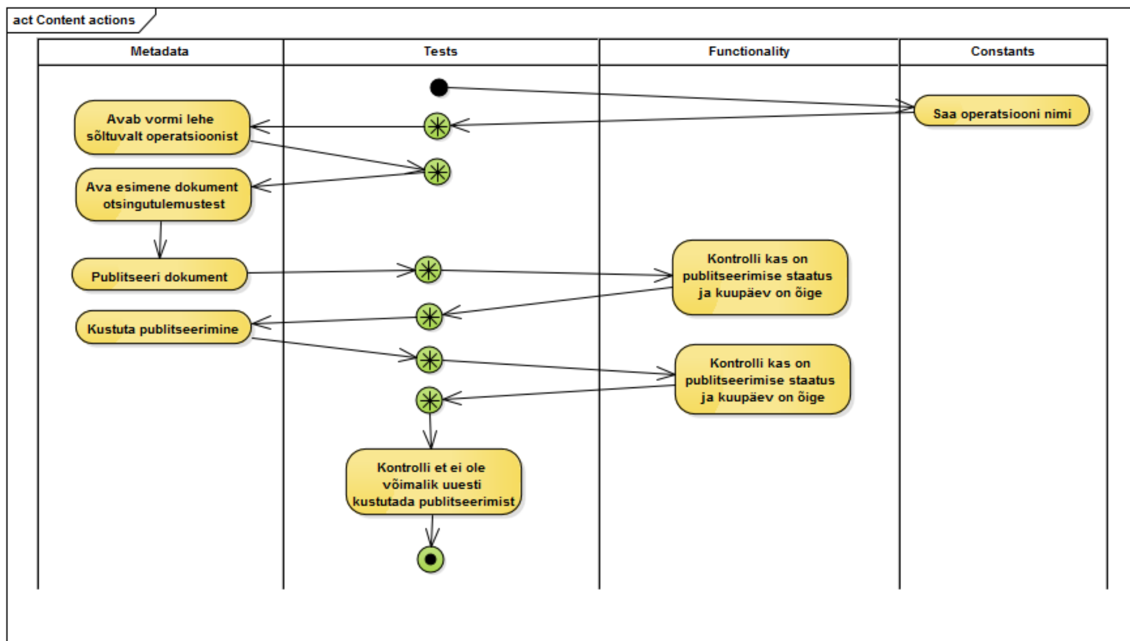
Joonis 5. *Check-out* testide tegevusskeem.



Joonis 6. *Check-out undo* testide tegevusskeem.

7.3 Dokumendi operatsioonide testid

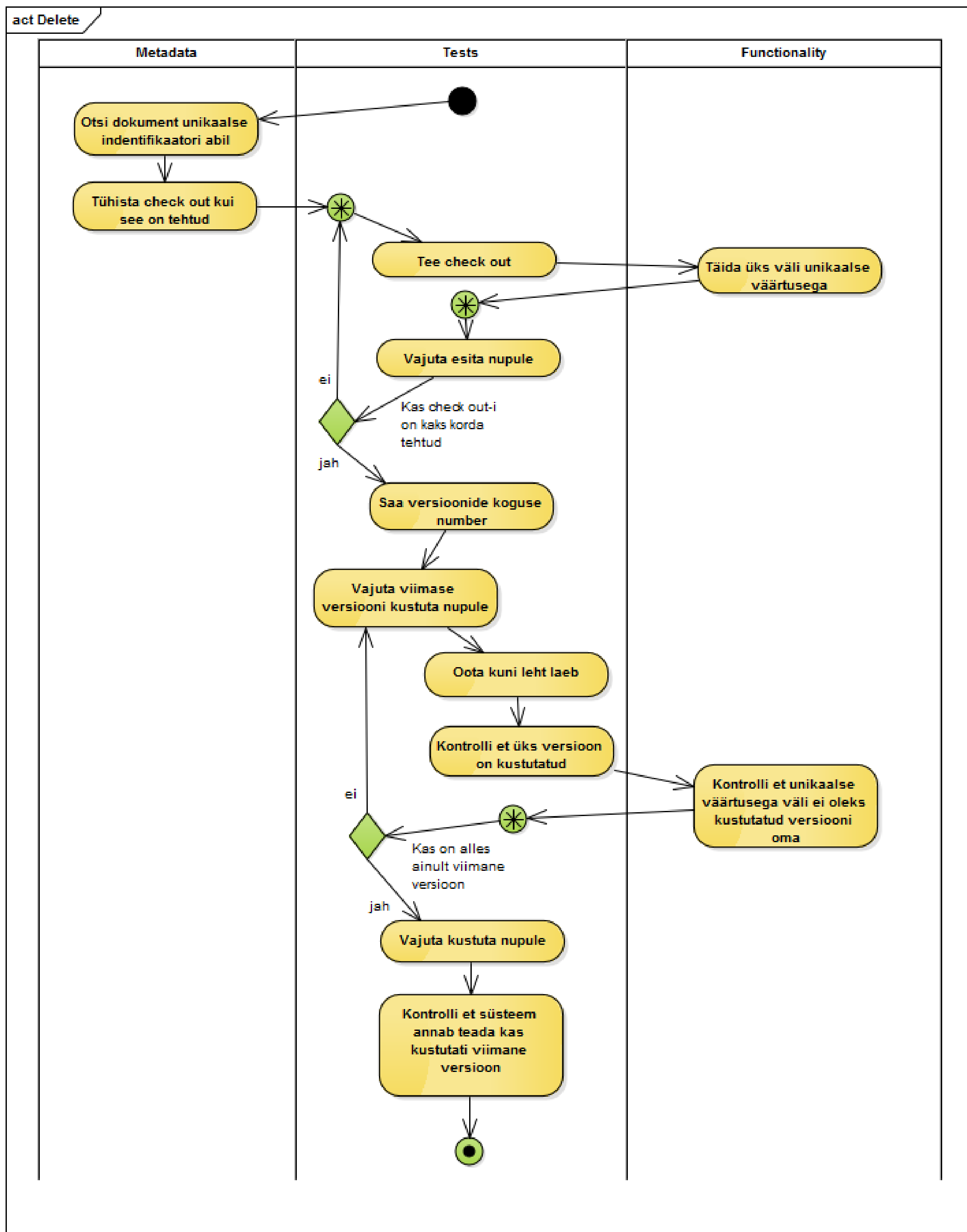
Lisaks automatiseeritud funktsionaalsusele on dokumendi operatsioonide testide sees veel teisi funktsionaalsusi, mida praegu otsustasin mitte automatiseerida. Automatiseeritud funktsionaalsuseks oli publitseerimine. Kuna publitseerimist kasutatakse ainult ühel profiilil, kuigi teistel on ka võimalus seda teha, siis automatiseerisin ainult ühe profiili publitseerimise, mille tegevusskeemi on näha joonisel 7. Testimiseks oli vaja üks dokument publitseerida ning vaadata, kas kuupäev ilmus publitseerimise kuupäeva metavälja. Peale mida kustutati publitseerimine ja vaadati, et publitseerimise kustutamise kuupäeva metavälja oleks ilmunud kuupäev. Testi tegemise aeg automatiseeritult oli 25 sekundit. Manuaalselt oleks testi testimiseks aega läinud ligi 3 minutit.



Joonis 7. Dokumendi operatsioonide testide tegevusskeem.

7.4 Versiooni kustutamise testid

Versiooni kustutamise testid kuulusid samamoodi nagu *check-out* testid ka regressioonitestide Excel tabelis dokumendi operatsioonide hulka. Versioonide kustutamise testimiseks peab alguses looma paar dokumendi versiooni, kasutades *check-out* funktsionaalsust, peale mida võib hakata dokumenti kustutama. Testi käigus kustutati sama palju versioone, kui oli neid kokku. Testimise käigus tuvastati, et kustutati õige versiooni, kasutades selleks unikaalset metaandmete väärtust. Samuti viimase asjana testiti, kas süsteem annab vastuseks, et rohkem versioone dokumendist ei ole. Kirjeldatud protsesse on näha ka joonisel 8. Automatiseeritult võtsid testid aega 6 minutit ja 5 sekundit, kuid manuaalselt läks aega 30 minutit.

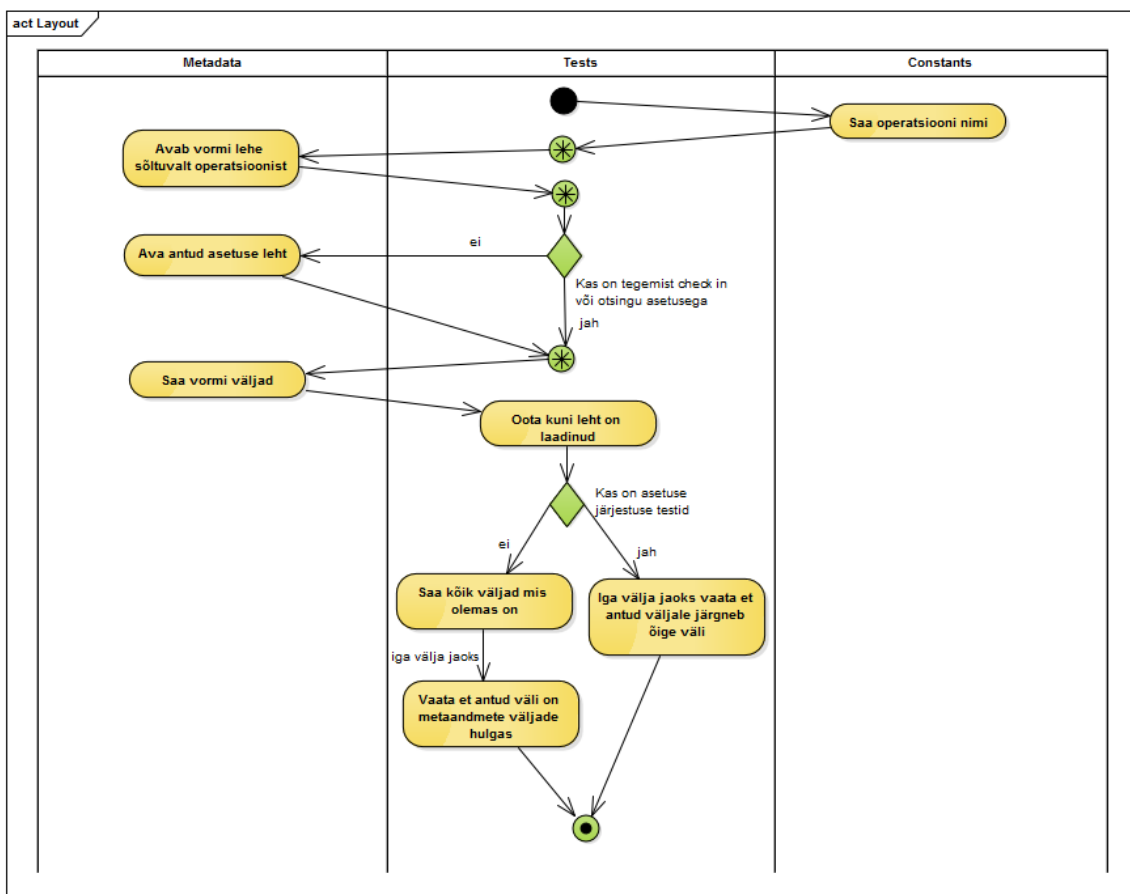


Joonis 8. Versioonide kustutamise testide tegevusskeem.

7.5 Metaandmete asetuse testid

Antud testid koosnevad kahest eraldi testide tüübist. Esimene neist on asetuse järjestuse testid. Järjestuse testid on vajalikud selleks, et teada saada, kas kõik metaandmed on olemas ja nad on kindla järjestusega. Kui peaks üks väli vales kohas olema, siis harjunud kasutajal on raske uuesti ümber harjuda. Asetuse testide alla lähevad ka testid, mis testivad, kas antud lehel on mingeid teisi metavälju peale lubatud metaväljade. Antud testide tegevusskeemi on näha joonisel 9.

Mõlemad testid võtsid kokku aega 13 minutit ja 44 sekundit. Manuaalselt tehes läks aega kuus tundi. See tähendab, et automatiseerimine muutis testimise ligi 26 korda kiiremaks ja lisaks muutusid antud testid rohkem usaldusväärsemaks. Manuaalselt tehes tekkis palju testimise vigu, kuna profiilide lehti on palju ja metaandmeid on palju, seetõttu oli lihtne teha vigu metaandmete segamini ajamisel.



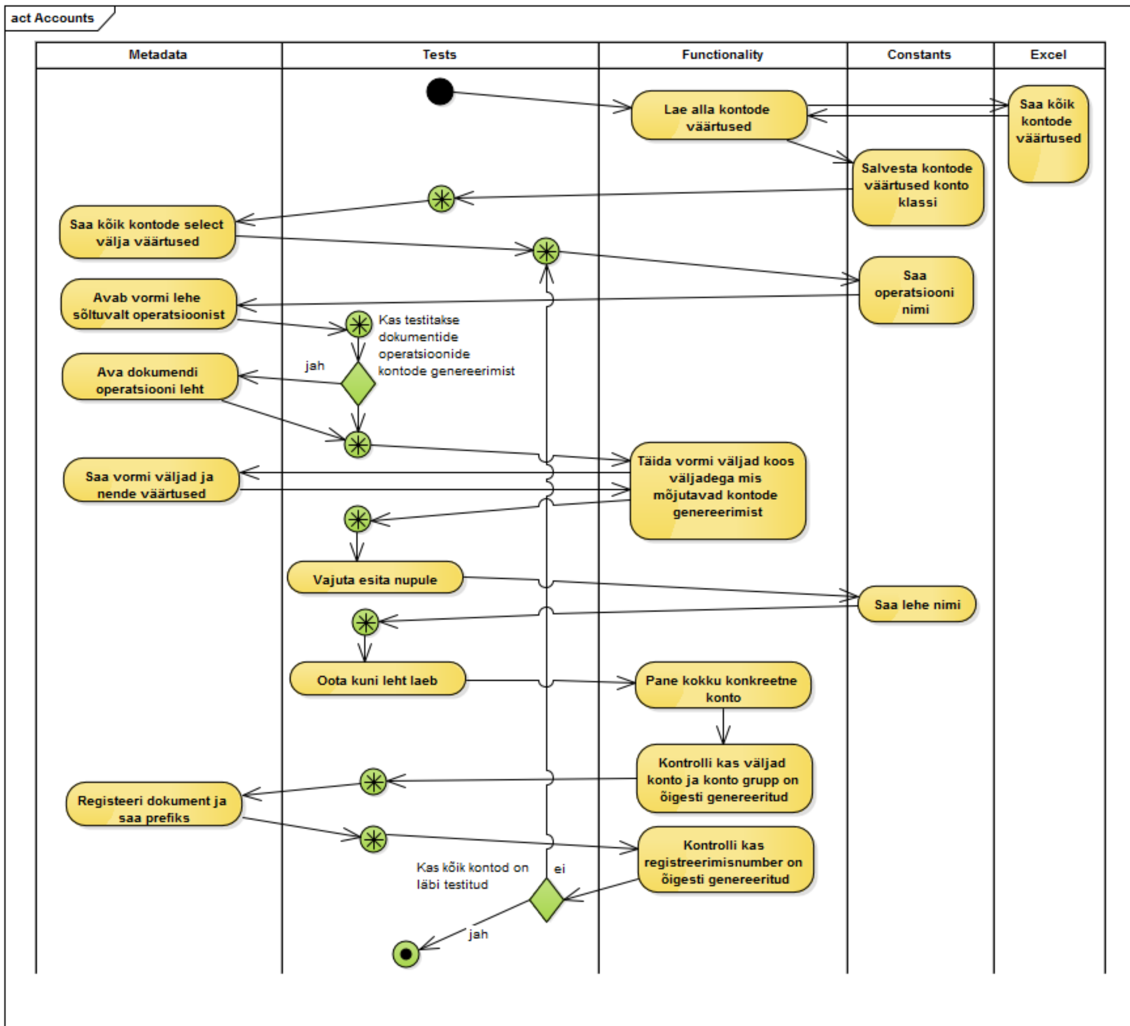
Joonis 9. Metaandmete kasutuse testide tegevusskeem.

7.6 Kontode ja registreerimise numbrite testid

Kontode ja registreerimise numbrite testid on kõige keerulisema testide struktuuriga. Testid testivad kahte eri tüüpi metaandmete välja, mida genereeritakse süsteemi poolt kindlate reeglite alusel. Test-aja kokkuhoidmise pärast on pandud kontode ja registreerimise numbrite testid kokku, sest muidu võtaks need peaaegu kaks korda rohkem aega, kui need koos võtavad. Lisaks peab testimiseks mõlema testi jaoks genereerima samad dokumendid. Seetõttu otsustasin nad panna ühe testi alla.

Esimeseks asjaks, mida on vaja teha, et läbida neid teste, on vaja laadida alla kõik kontotükkide väärtused ning salvestama konto klassi. Peale seda peab profiili metaandmed täitma väärtustega ja tegema *check-in,* peale mida kombineerime kokku kindla konto ning võrdleme seda dokumendi kontoga. Kuna *check-in-i* saab teha mitmel erineval moel, siis me peame need samuti läbi proovima ning joonisel 10 on näidatud *check-in* operatsiooni valik esimese otsuse tingmärgiga. Kui konto on testitud, siis testitakse registreerimise numbri prefiks sarnaselt, aga prefiks saadakse `metadata` klassilt. Kui konto ja registreerimisnumber on ühele dokumendiga testitud, siis tehakse teistega sama moodi, et lõpuks kõik kombinatsioonid saaksid testitud.

Peale automatiseerimist võtavad testid aega 1 tund ja 37 minuti. Manuaalselt aga tehes võtsid nad aega 14 tundi. Lisaks aja kulu vähendamisele muutusid testid kvaliteetsemaks, kuna nii paljude kombinatsioonide tõttu võivad nad segi minna. Antud testide test klassi on näha lisas 1.

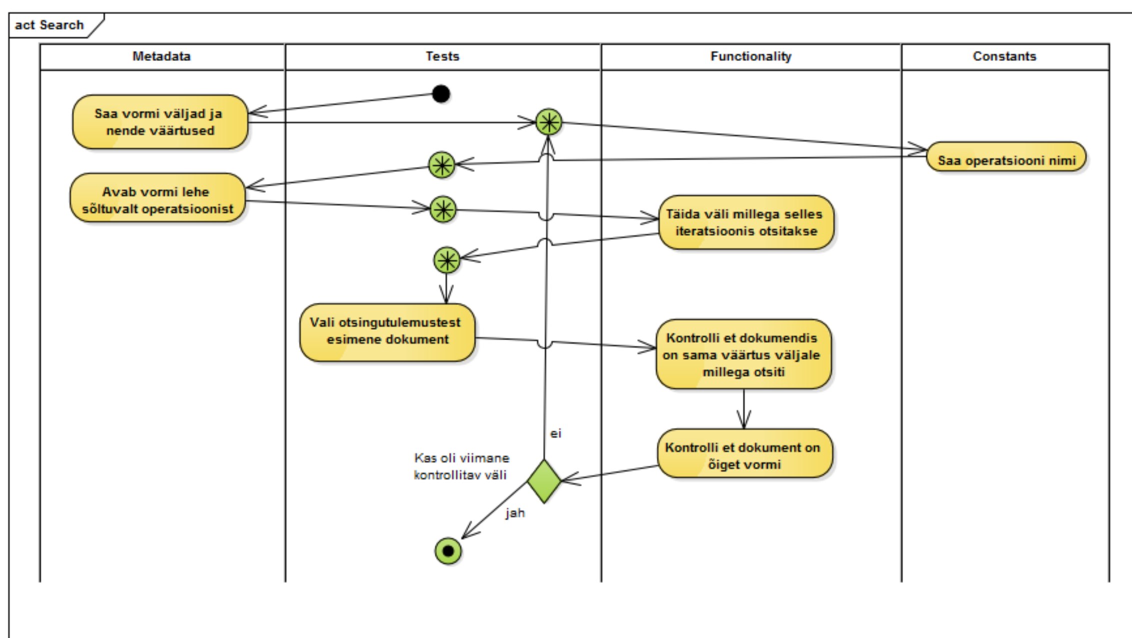


Joonis 10. Kontode ja registreerimise numbrite testide tegevusskeem.

7.7 Otsimise testid

Dokumentide otsimise testimise esimeseks asjaks võetakse lahti kindla profiili dokumendi otsimise leht, peale mida sisestatakse ühekaupa infot metaväljadesse ja otsitakse antud ühe metavälja abil dokument, peale mida vaadatakse dokumendi metaandmetesse, kas need klapivad otsituga. Siis võetakse järgmine profiili metaväli testimisse. Antud testide tegevusskeemi on näha joonisel 11.

Käsitsi tehes võttis testimine aega tund ja 30 minutit, kuid peale automatiseerimist tehti testid 20 minuti ning 56 sekundiga.

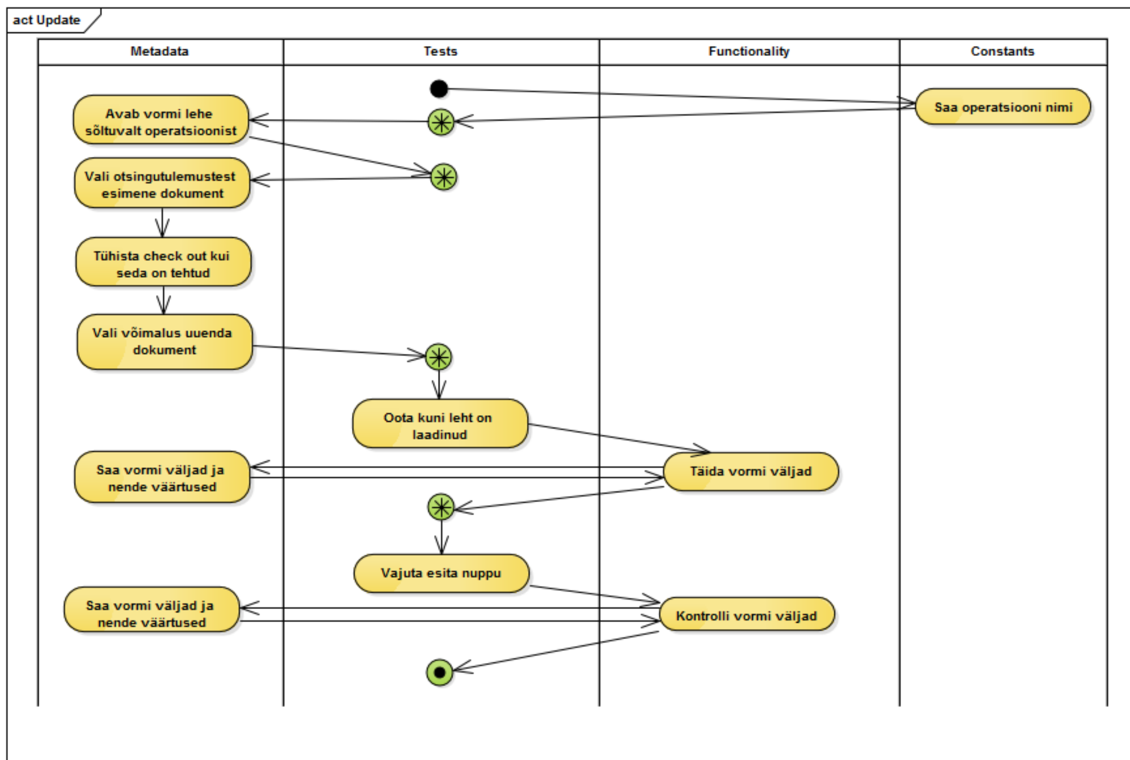


Joonis 11. Otsimise testide tegevusskeem.

7.8 Dokumentide muutmise testid

Dokumendi muutmise testimine toimub sarnaselt *check-in* ja *check-out* testimisele. Esimese asjana tühistatakse *check-out*. See on vajalik, sest kui dokumendiga on tehtud *check-out,* siis ei saa dokumendi metaandmeid muidu muuta. Peale mida muudetakse metaandmeid ja vaadatakse, kas metaandmed muutusid korrektselt. Dokumentide muutmise testide protsesse on täpsemalt näha joonisel 12.

Peale automatiseerimist võtavad testid aega 11 minutit ja 26 sekundit. Manuaalselt tehes kulus ligi pool tundi testide tegemiseks. Samuti on automatiseeritult testid usaldusväärsemad, kuna inimene võis ajada erinevad metaandme-väljad omavahel segi ning võrrelda neid valesti.



Joonis 12. Dokumendi muutmise testide tegevusskeem.

7.9 Testide automatiseerimise järgmine etapp

Automaattestide edasiseks arenguks oleks soovitatav muuta `metadata` klassid rohkem dünaamilisemaks. See tähendab, et klassis kogu informatsiooni hoidmise asemel, peaks looma klassi, mis loeb metaandmeid välisest Excel tabelist. Antud lahendus on hea seetõttu, et niimoodi saame vähendada koodi klasse ning on lihtsam teha muudatusi metaandmetes.

Teiseks arengu suunaks on lisada juurde teste, mida praegu ei eksisteeri. Sellisteks testideks oleks töövoo testid. Antud suund on kõige töömahukam, sest praegusele testide funktsionaalsusele tuleb juurde lisada funktsionaalsus, mis võimaldab testida kahe kasutajaga. Kaks kasutajat on vajalik selleks, et testida töövoo jooksul tehtud delegeerimisi. Samuti peaks automatiseerima testid, mis jäid praegu automatiseerimata.

Kolmandaks võiks lisada testidele juurde funktsionaalsuse test tulemuste kirjutamise regressiooni testide faili, kus oleks näha testi kirjeldus, käivitamise kuupäev ja tema staatus. Praegu saab test tulemusi ainult näha käivitatavas J-Unit koodi redaktori aknas.

8 Kokkuvõte

Töö peamine ülesanne oli teha automaattestid, mis säästaks testijate aega regressioonitestide tegemisel ja muudaks osad testid ka rohkem usaldusväärsemateks.

Eesmärgi saavutamiseks alustasin alguses tähtsamate funktsionaalsuste kaardistamisega ning otsustasin, kas antud funktsionaalsust on võimalik üldse mõistliku ajaga ja liigselt keerulise struktuuri tekitamiseta automatiseerida. Peale mida valisin välja töövahendid, mille abil hakkasin ülesannet realiseerima. Töö peamisteks vahenditeks olid Selenium ja Selenide, mille valisin põhjusel, et testid oleks kasutajaliidese põhised. Selenium pakub just kasutajaliidese põhiseid testimise viise ja Selenide teek muudab testide koodi lihtsamaks. Siis hakkasin teste realiseerima. Automatiseeritud testide kasulikkust hindasin peamiselt nende jooksmisele kulunud aja järgi.

Peale testide automatiseerimise kulus regressioonitestide peale aega manuaalselt testimiseks 7,5 tundi ning automaattestid jooksid 2 tundi 55 minutit. Enne automaattestide tegemist läks aega 32 tundi, see tähendab, et peale automatiseerimist testide tegemisele kuluv aeg vähenes ligi kolm korda. Lisaks testide skoop suurenes, kuna automaattestidega oli lihtne testida ka funktsionaalsust, mida peeti enne vähemtähtsaks tema peale kuluva aja tõttu. Samuti muutusid testid rohkem usaldusväärsemaks, sest vähenes inimvigade arv.

Töö eesmärk luua automatiseeritud testid, mis säästaks testijate aega, sai täidetud, kuna võrreldes varasemaga hoiti aega kokku ligi kolm korda. Samuti muutusid testid usaldusväärsemaks.

Kasutatud kirjandus

[1] Enterprise Content Management in SharePoint.- Microsoft support. https://support.office.com/en-us/article/Enterprise-Content-Management-in-SharePoint-930dd985-5bb9-447b-affd-86fcf690e994 (20. 04. 2017)

[2] What is Adobe Experience Manager.- Adobe. http://www.adobe.com/ee/marketing-cloud/enterprise-content-management.html (20. 04. 2017)

[3] What is Enterprise Content Management.- Laserfiche. https://www.laserfiche.com/ecmblog/what-is-enterprise-content-management-ecm (20 04 2017)

[4] Oracle WebCenter Content 11g: Content Server Administration.- Oracle university. http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=3 (22. 03. 2017)

[5] Differences Between Black Box Testing and White Box Testing.- Software testing fundamentals. http://softwaretestingfundamentals.com/differences-between-black-box-testing-and-white-box-testing/ (10. 05. 2017)

[6] Automated Testing Best Practices and Tips.- Smartbear. https://smartbear.com/learn/automated-testing/best-practices-for-automation/ (10. 05. 2017)

[7] Introduction to Functional Testing.- Smartbear. https://smartbear.com/learn/automated-testing/introduction-to-functional-testing/ (10. 05. 2017)

[8] What Is Unit Testing?- Smartbear. https://smartbear.com/learn/automated-testing/what-is-unit-testing/ (10. 05. 2017)

[9] Benefits of Keyword-Driven Testing.- Smartbear. https://smartbear.com/learn/automated-testing/benefits-of-keyword-testing/ (10. 05. 2017)

[10] Introduction to Data-Driven Testing.- Smartbear. https://smartbear.com/learn/automated-testing/introduction-to-data-driven-testing (10. 05. 2017)

[11] What is Regression Testing?- Smartbear. https://smartbear.com/learn/automated-testing/what-is-regression-testing/ (10. 05. 2017)

[12] Automated visual software testing.- Applitools. https://applitools.com/ (10. 05. 2017)

[13] Software performance testing.- Wikipedia. https://en.wikipedia.org/wiki/Software_performance_testing (10. 05. 2017)

[14] Martin, C, R., Clean Code. Prentice Hall, 2008.

[15] Introduction to WebDriver & Comparison with Selenium RC.- Guru99. http://www.guru99.com/introduction-webdriver-comparison-selenium-rc.html (26.

04. 2017)

[16] Selenium Webdriver.- Seleniumhq. http://www.seleniumhq.org/projects/webdriver/ (26. 04. 2017)

[17] Selenide.- Selenide. http://selenide.org/ (26. 04. 2017)

[18] JUnit.- Wikipedia. https://en.wikipedia.org/wiki/JUnit (26. 04. 2017)

[19] Java Excel API - A Java API to read, write, and modify Excel spreadsheets.- Sourceforge. http://jexcelapi.sourceforge.net/ (26. 04. 2017)

Lisa 1 – Kontode ja registreerimise numbrite testide kood

```
public class RegiNrAccountsTest {
    static ArrayList<String> contents= new ArrayList<String>();
    @BeforeClass
    public static void logIn(){
        SetUpTest.setUp();
        contents=SetUpTest.createAllDocs();
    }

    @Test
    public void internalCheckIn(){
        testAccounts(Forms.internal,"int", Operations.checkIn);
    }
    @Test
    public void internalCheckInSimilar(){
        prepareAndTestAccounts("2","2","",Forms.internal,"int",Operations.checkInSimilar);
        contents.add(SetUpTest.getContentIdWithoutFullInfo());
    }

    @AfterClass
    public static void tearDown(){
        SetUpTest.deleteContents(contents);
    }


    private void testAccounts(Forms form, String formAccount, String operation){
        String third="";
        for(String first:form.getSelectOptions("xxICCountry","")){
            ArrayList<String> department=form.getSelectOptions("xxICDepartment", first);
            for(String second:department){
                ArrayList<String> functions=form.getSelectOptions("xxICFunctions", second);
                for(int j=0;j==0||j<functions.size();j++){
                    if(formAccount.equals(""))third=functions.get(j);
                    prepareAndTestAccounts(first, second, third, form, formAccount,operation);
                    testRegistrationNumber(form, form.getRegistrationPrefix(second,third));
                    Forms.delete();
                }
            }
        }
    }
    private void prepareAndTestAccounts(String xxICCountry, String xxICDepartment, String xxICFunction, Forms form, String
formAccount, String operation){
        if(operation.equals(Operations.search)){
            form.selectProfile(operation);
            Forms.updateFromSearchFirstDoc();
        }
        else if(operation.equals(Operations.checkInSimilar)){
            form.selectProfile(Operations.search);
            Forms.checkInSimilarFromSearchFirstDoc();
        }else form.selectProfile(operation);
        String account;
        if(xxICFunction.equals(""))
            account= Functions.formulateAccount(xxICCountry,xxICDepartment,formAccount);
        else
            account= Functions.formulateAccount(xxICCountry,xxICDepartment,Accounts.accountPartsFunction.get(xxICFunction));
        Functions.fillFields(form, form.getCheckInMandatoryFields());
        Functions.setField("xxICCountry", xxICCountry, form);
        Functions.setField("xxICDepartment", xxICDepartment, form);
        if(!xxICFunction.equals(""))Functions.setField("xxICFunction", xxICFunction, form);
        $(By.name("javaSubmit")).click();
        $(By.name(Universal.contentInfoPageName)).shouldBe(Condition.visible);
        testAccount(form, account,"Room");


    }
    private void testAccount(Forms form, String expectedAccount, String expectedGroup){
        System.out.println("Testing account: "+expectedAccount);
        $(By.linkText("Full Information")).click();
        ImmutableMap<String, String> security=ImmutableMap.<String,String>builder()
                .put("dDocAccount",expectedAccount)
                .put("dSecurityGroup", expectedGroup).build();
        Functions.checkFieldsInput(security);
    }
    private void testRegistrationNumber(Forms form, String expectedRegistration){
        System.out.println("Testing registration nr: "+expectedRegistration);
        if(!form.getProfile().equals(Forms.internal.getProfile()))
            Forms.registerDocument();

        ImmutableMap<String, String> number=ImmutableMap.<String,String>builder()
                .put("xxDocumentNumber",expectedRegistration).build();
        for(Map.Entry<String,String> entry:number.entrySet())
            Functions.checkFieldInputContains(entry);
    }

}
```