TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDK40LT

Ilja Umov 104470

AUTOMATISEERITUD TESTIDE ARENDUS TARBIMISLAENU INFOSÜSTEEMI NÄITEL

bakalaureusetöö

Juhendaja: Jekaterina Tšukrejeva
Magistrikraad
Assistent

Tallinn 2016

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: Ilja Umov

15.05.2016

Annotatsioon

Töö eesmärk on automatiseerida ärireeglite testimist mitmeid riike teenindava tarbimislaenu infosüsteemi jaoks. Vajalik on defineerida kasutuslood ning nende läbimine testide kujul automatiseerida, lähtudes sealjuures lõppkasutaja seisukohast. Testide koostamise eelduseks on liidestus süsteemiga, kasutades muuhulgas brauseri juhtimist, andmebaasi, SOAP-teenuseid. Selle tarvis on vaja arendada testraamistik, mis võimaldaks süsteemi vajalikul määral juhtida.

Töös on käsitletud loodud testraamistiku komponentide valikut ning tulemusena saadud raamistiku struktuuri. Kirjeldatud on ka seadistust testide perioodiliseks käivitamiseks ning võrreldud erinevaid testiraportite genereerimise vahendeid. Samuti on kirjeldatud täiendavaid loodud komponente ja prototüüpe, mida raamistikku ei integreeritud, sealhulgas tootevaliku testide genereerimise skript ja dokumentide korrektsuse kontrollimiseks loodud prototüüp.

Töö käigus valiti välja sobilikud teegid süsteemi osade juhtimiseks ning koostati nende põhjal testraamistik. Seejärel koostati testlood vajalikud funktsionaalsuse testimiseks ning teostati need raamistiku põhjale automatiseeritud testidena. Lisaks loodi täiendavad abivahendid testimisprotsessi hõlbustamiseks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 63 leheküljel, 6 peatükki, 3 joonist.

Abstract

Development of automated tests for consumer loan system

Aim of this thesis was to automate testing of business rules for consumer loan system, which is used in different countries. It is necessary to define use cases and automate resulting tests, focusing on end user perspective. Prerequisite for implementing tests is interface for interacting with system, including controlling browser, database queries and SOAP services. This interface has to be implemented as a test framework, which would allow to interact with system and using which automated tests would be developed.

In order to develop test automation solution, necessary libraries were chosen that allowed to interface with required parts or the system. This lower-level functionality was implemented and integrated making it possible to use it from test code. Upon that additional layer of functions was built, which corresponded directly to some parts of business process. Combining previously described parts resulted in test framework, which allowed to focus on automating testing of business rules.

During development of test framework configuration for periodical running of tests in continuous integration server was done, which included evaluation of different reporting tools.

As testing some parts of system required different approach than browser based automation, some prototypes and scripts were implemented in order to accomplish that, including prototype for testing correctness of generated documents and script for generating product selection tests.

In the course of development of automated tests various issues with implementation of business rules were found in the system and reported. Running developed tests nightly allowed to detect introduced regressions promptly. Developed test framework and tools made testing of system changes faster and less reliant on manual testing.

The thesis is in Estonian and contains 63 pages of text, 6 chapters, 3 figures.

Lühendite ja mõistete sõnastik

API	***Application Programming Interface*** Programmiliides, reeglistik olemasoleva valmisprogrammiga suhtlemiseks.
Backend	***Backend*** Süsteemi osa, mis tegeleb andmete talletamise ja töötlemisega.
Disainimuster	***Design pattern*** Üldine taaskasutatav lahendus tüüpilisele probleemile tarkvara disaini juures.
Frontend	***Frontend*** Süsteemi osa mis tegeleb kasutajale informatsiooni kuvamisega ning interaktsiooniga, üldistatult kasutajaliides.
Jenkins	***Jenkins*** Avatud lähtekoodiga pideva integratsiooni server
JUnit	***JUnit*** Testimise raamistik Java programmeerimiskeele jaoks.
Mock	***Mock*** Testimiseks kasutatav objekt programmikoodis, mis realiseerib osa keerukama liidese funktsionaalsust piiratud mahus.
ORM	***Object-relational mapping*** Objektorienteeritud keeli relatsioonandmebaasidega siduv mehhanism.
Selenide	***Selenide*** Raamistik testide automatiseerimiseks, põhineb Selenium WebDriver-il.

Selenium WebDriver *Selenium WebDriver*
Veebipõhise tarkvara testimisraamistik, mis võimaldab brausereid programmikoodist juhtida

SOAP *Simple Object Access Protocol*
Andmevahetust kirjeldav protokoll, mis võimaldab veebiteenustel omavahel XML formaadis struktuurseid andmeid vahetada

SoapUI *SoapUI*
Veebiteenuste testimiseks mõeldud programm.

Jooniste nimekiri

Joonis 1. Põhiprotsessi lihtsustatud tegevusskeem. 15
Joonis 2. Ülevaade testraamistiku ülesehitusest. 21
Joonis 3. Testraamistiku osaline klassidiagramm. 25

Sisukord

1. Sissejuhatus 10
1.1 Taust ja probleem 10
1.2 Ülesande püstitus 11
1.3 Metoodika 12
1.4 Ülevaade tööst 12
2. Testitav funktsionaalsus 14
2.1 Testitava süsteemi üldine ülevaade 14
2.2 Testitava süsteemi tehniline ülevaade 16
2.3 Testide automatiseerimise alustamine 16
2.4 Testide automatiseerimise vajalikkus 17
2.5 Krediidireitingu arvutuse testimine 17
2.5.1 Uued kliendid 18
2.5.2 Teenust varem kasutanud kliendid 19
2.6 Kliendile saadetavad dokumendid 19
3. Testraamistik 21
3.1 Brauseri juhtimine 22
3.2 Andmebaas 22
3.3 SOAP-liides 23
3.4 Testraamistiku ülesehitus 24
3.5 Täiendavad testimiseks loodud abivahendid 27
3.5.1 Tootevaliku testide automaatne genereerimine 27
3.5.2 Kasutajaliides levinuimate kliendikonto tegevuste jaoks 30
3.5.3 Skriptid registreerimisprotsessi läbimiseks 31
3.5.4 Dokumentide korrektsuse automatiseeritud testimise prototüüp 32
3.6 Testid CI keskkonnas 33
3.7 Genereeritud testiraportite võrdlus 34
3.7.1 Jenkins-i testiraport 34
3.7.2 Maven Surefire Report Plugin 35
3.7.3 Allure raamistik 35
3.8 Avastatud tehnilised puudujäägid 36
4. Testlood 39
4.1 Automatiseeritavate testilugude valiku põhimõtted 39

4.2 Testlugude kirjeldus ... 39
4.2.1 Uue kliendina erinevate laenude taotlemine ... 40
4.2.2 Uue kliendina lepingu tingimuste muutmine ... 43
4.2.3 Uue kliendi perioodi lõppemine ... 44
4.2.4 Kliendile pakutavad tooted ... 45
4.2.5 Täiendav toodete filtreerimine ... 47
4.2.6 Aktiivse lepinguga kliendi lisateenused ... 48
4.2.7 Krediidikonto lisateenused erinevate kliendigruppide jaoks ... 48
4.2.8 Aktiivse lepinguga kliendi krediidireitingu arvutamine ... 49
4.2.9 Kliendile saadetavate meilide kontroll ... 49
4.2.10 Infolehtedel näidatavad laenukalkulaatorid ... 50
4.2.11 Kliendi käest vajalike dokumentide nõudmine ... 50
4.2.12 Tootegruppide valiku korrektsus ... 50
4.2.13 Maksed läbi veebiliidese ... 50
4.3 Välise krediidireitingu mooduli vastuvõtutestimine ... 51
5. Testide automatiseerimine projekti vaatepunktist ... 53
5.1 Muutuvad nõuded ... 53
5.2 Protsess ... 53
5.3 Automatiseeritud testidest saadud kasu ... 54
6. Kokkuvõte ... 55
Kasutatud kirjandus ... 56
Lisa 1 – Funktsioonide vaikimisi parameetrid ... 58
Lisa 2 – Andmebaasiga suhtleva funktsiooni näide ... 59
Lisa 3 – SOAP-liidesega suhtleva funktsiooni näide ... 60
Lisa 4 – Testi kommenteeritud koodinäide ... 61
Lisa 5 – Tootevaliku testi näide ... 62
Lisa 6 – Page Objects muster ... 63

1. Sissejuhatus

Kasutusel olevate infosüsteemide puhul, mida samas ka pidevalt edasi arendatakse, on oluline tagada süsteemi ärireeglitele vastav toimimine ning vältida arenduse käigus olemasoleva funktsionaalsuse katki minekut ehk regressioone. Süsteemi funktsionaalsuste toimimist kontrollitakse testimise protsessi abil, mida võib jaotada mitmeks kihiks. Kirjutatud koodile kõige lähedasemad on arendajate poolt koostatud *unit*-testid, millega kontrollitakse konkreetse funktsiooni või klassi toimimist. Sellele järgnevad integratsioonitestid, millega kontrollitakse mitme eri komponendi koos toimimist. Lõpuks on vaja ka kogu rakendust tervikuna kasutaja seisukohalt vaadates testida, sest eelnevad testimise liigid ei pruugi katta kõiki reaalselt tekkivaid olukordi, eriti kui rakenduse loogika on keeruline ning erinevaid komponente on palju.

Veebipõhise rakenduse manuaalsel testimisel tuleb läbi teha erinevad testilood ning jälgida, et süsteem töötaks korrektselt. Seeläbi avastatakse üldjuhul vigu ja muid puudujääke, mis rakenduse lõppkasutajani jõudma ei peaks. Selline tegevus on vajalik, kuid üsna ajamahukas, sest rakenduses navigeerimine, vormide täitmine ja samal ajal protsessi korrektsuse kontroll võtavad aega. Mida keerukam on rakendus, seda rohkem kõikide stsenaariumite läbitestimine aega võtab ning teatud keerukusastmest alates võtab kogu funktsionaalsuse käsitsi läbitestimine liiga palju ressursse. Samas on aga endiselt vajalik kindlus, et süsteem toimib korrektselt.

Eelnev probleem, eriti aktiivselt arenduses oleva süsteemi puhul, tingib vajaduse testide automatiseerimise järele, mis võimaldab testimist vajavat, kuid vähe muutuvat funktsionaalust testida vähema ressursikuluga, kui manuaalne testimine.

1.1 Taust ja probleem

Eelnevalt kirjeldatud olukord ehk siis arenduses olev infosüsteem, mille korrektset toimimist oli vaja pidevalt tagada, tekkis ühes tarbimislaenude pakkumisega tegelevas ettevõttes. Süsteemi arenduse käigus tehti muudatusi ja täiendusi, mille pidev käsitsi läbitestimine oleks olnud liialt ajamahukas. Seetõttu hakati mõtlema automatiseerimise peale, et ei oleks vaja tüüpilisemaid kasutuslugusid pidevalt käsitsi läbi teha.

Testimine on vajalik nii äripoole, kui ka arendustiimi jaoks. Äripoole jaoks on kõige tähtsam kindlustada erinevate ärireeglite korrektne teostus süsteemis arendajate poolt ning vältida sellega vigaseid väljalaskeid. Arendajate jaoks annab testimine kindluse, et nende poolt loodud kood käitub õigesti, sealhulgas erinevates piirolukordades.

Testide automatiseerimine võimaldab hilisemalt ilma oluliste täiendavate ressursikuludeta kontrollida kogu süsteemi toimimist. Näiteks lihtsalt avastada regressioone, mis tekivad kui ühe komponendi muudatused mõjutavad mõnda teist, ainult kaudselt seotud komponenti. See võimaldab testijatel keskenduda uue funktsionaalsuse testimisele ning vältida rutiinsete tegevuste kordamist.

Tööd tehti kliendi juures, arendustiimi koosseisus, algusega 2015. aasta suvel kuni 2016 aasta kevadeni. Kliendiks oli tarbimislaene pakkuv firma, mis tegutseb mitmes eri riigis, pakkudes teenuseid erinevate brändide all. Testitava süsteemi arendusest võtsid osa *frontend* tiim, milles oli projekti vältel 4-6 liiget, samuti erinevate vastutusaladega *backend* arendajad, keda projekti lõpuks oli kahe tiimi jagu. Autor sai tööülesanded testijuhilt, kes omas terviklikku ülevaadet süsteemis tehtud ja planeeritud muudatuste üle. Lisaks oli tiimis veel üks testija, kelle põhifookuseks olid mobiilirakendused ja kasutajaliidese funktsionaalsus.

Algse plaani kohaselt oli vaja automatiseerida ja sel viisil põhjalikult läbi testida suurem osa süsteemis olevatest ärireeglitest, millega tegeletakse kogu projekti vältel. Kui reeglites esineb muudatusi, keskendutakse muutunud või uue funktsionaalsuse jaoks automatiseeritud testide koostamisele, et anda võimalikult kiiret tagasisidet arendajatele.

1.2 Ülesande püstitus

Töös pannakse kokku raamistik, mis võimaldab testitava süsteemiga suhelda, seda nii läbi andmebaasi, SOAP-liidese ja ka brauseri. Selleks valitakse sobilikud teegid, kirjutatakse valmis vajalik põhifunktsionaalsus, mis moodustab raamistiku põhja.

Raamistikku kasutades koostatakse järk-järgult vajaliku funktsionaalsuse jaoks teste ning testide käivitamisega kontrollitakse süsteemi toimimise korrektsust. Tuleb arvestada ka seda, et samalaadseid teste tuleb koostada mitme erineva riigi süsteemide jaoks ehk et osa koodi saab ja tuleb taaskasutada, samas on arvestada süsteemide iseärasustega.

Eelneva tegemisel tuleb arvestada ka sellega, et testid peavad toimima ka CI-serveris, sealhulgas peab testidel olema arusaadav väljund testiraporti näol ning peab olema võimalik koostatud teste aja kokkuhoiu jaoks paralleelselt käivitada.

Töö käigus selguvad autori meelest sobilikuimad vahendid seda laadi töö tegemiseks, soovituslik metoodika samalaadse töö tegemiseks ning nõuanded testraamistiku loomisel.

1.3 Metoodika

Tööd alustatakse lihtsamate testide automatiseerimisest ning teostatakse selle jaoks vajalik funktsionaalsus süsteemi juhtimiseks, pannes sellega aluse testraamistikule. Protsessi vältel laiendatakse testraamistiku võimalusi, lisades funktsionaalsuse erinevate äriprotsessi osade läbimiseks.

Kuivõrd teatud süsteemi komponentide (näiteks dokumentide genereerimine) testimine nõuab veidi erinevat lähenemist, siis luuakse vastavad prototüübid täiendava funktsionaalsuse testimiseks.

Kuna süsteem on pidevas arenduses ning esineb tihti muutusi ärireeglites ja pakutavates toodetes, siis tekib tihti vajadus teste kohaldada uue nõuetega. Selle jaoks luuakse skriptid, mis genereerivad sisendandmete põhjal vajalikud testid, vähendades tootevalikut puudutavate testide ajakohastamisele kulunud aega.

Protsessi käigus raporteeritakse leitud vead ning testitakse parandusi üle, samuti kohaldatakse koostatud teste vastavalt süsteemis esinenud muutustele ning jälgitakse seda, et testidega kaetud funktsionaalsuses ei tekiks regressioone. Selle jaoks vajalikud testide perioodiline käivitamine ning testraportite genereerimine on samuti töös käsitletud.

1.4 Ülevaade tööst

Töös tuuakse alguses välja vaadeldava süsteemi testimist vajavad funktsionaalsused koos nende toimimise põhimõtte lühikese kirjeldusega, seda ühe riigi näitel. Samuti on üldiselt kirjeldatud meetodid, kuidas testimisele läheneti.

Järgmisena kirjeldatakse töö tehnilist teostust: koostatud testiraamistiku ülesehitust ning erinevate süsteemi osade kasutamiseks valitud teeke. Lisaks kirjeldatakse töö käigus loodud testimiseks mõeldud ja kasutatud abivahendeid, mida raamistikku ei integreeritud. Samuti on

kirjeldatud seadistused testide käivitamiseks *continuous integration* serveris ning võrreldud selle protsessi käigus genereeritud erinevaid testiraporteid.

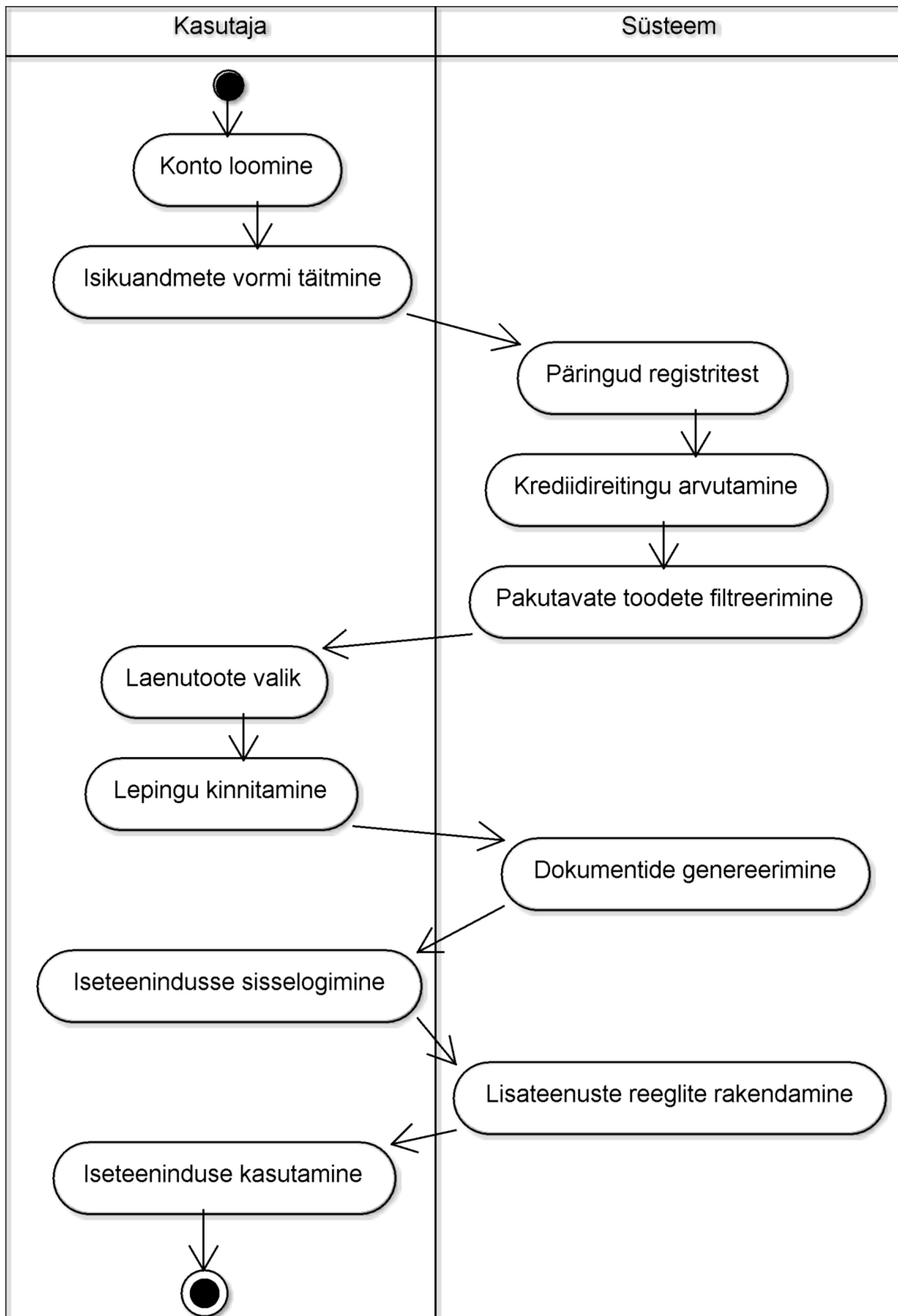
Sellele järgneb testitava funktsionaalsuse jaotus testilugudeks, milles kirjeldatakse ühe riigi testikomplekti põhjal mida täpselt testides tehti. Lisatud on ka kommenteeritud koodinäited, mis aitavad mõista kuidas juhiti testide käiku ning kuidas toimus tulemuste kontroll.

Lõpuks on välja toodud mõningad töö käigus tekkinud tehnilisemat laadi probleemid ning välja toodud mõned üldised järeldused lähtudes projektijuhtimise seisukohalt.

2. Testitav funktsionaalsus

2.1 Testitava süsteemi üldine ülevaade

Töö raames testitav süsteem on kasutaja seisukohalt vaadates üldiselt lihtne: on võimalik teha kasutajakonto, täita ära vormil enda kohta käivad andmed, seejärel valida sobilik laenutoode ning taotlus ära kinnitada. Pärast taotluse aktsepteerimist on võimalik iseteenindusse sisse logida ning kasutada pakutavaid lisateenuseid, näiteks suurendada limiiti või võtta maksepuhkus. Iga kirjeldatud sammuga käib kaasas aga hulk äriloogikat: milliseid tooteid peaks kliendile pakkuma, milliseid täiendavad lisateenused ja millal peaks klient kasutada saama ja palju muud; lisaks erinevad ärireeglid riigiti. Töös kirjeldatud testide automatiseerimine keskendubki ärireeglite testimisele süsteemis, mis teenindab mitme eri riigi kliente. Järgnevalt on toodud süsteemi põhiprotsessi lihtsustatud tegevusskeem.



Joonis 1. Põhiprotsessi lihtsustatud tegevusskeem.

2.2 Testitava süsteemi tehniline ülevaade

Terve süsteem koosnes erinevatest komponentidest, avalik veebiliides koos sisuhaldusega oli ehitatud Wordpress-i põhjale. Eraldi moodulina oli tehtud ja integreeritud funktsionaalsus laenu taotlemise ja iseteeninduse jaoks. *Backend*-i jaoks oli kasutusel Java Spring raamistik, koos teiste vajaminevate moodulitega, andmebaasina kasutati MySQL-i. Lisaks oli kasutusel Salesforce põhjale tehtud portaal, mis oli samuti läbi veebiteenuste *backend*-iga ühendatud.

Eelnevalt kirjeldatud komponendid moodustasid süsteemi, lisaks mille põhikeskkonnale hoiti käigus ka testkeskkondi. Testkeskkondades olid mõnede väliste registrite ja teenuste kasutus väljalülitatud või asendatud *mock*-iga. Omavahel erinesid testkeskkonnad põhiliselt uuenduste sageduse järgi. Ühte keskkonda sattusid kõik *master* haru muudatused, teises oli sama kood, mis praeguses põhikeskkonnas ning samuti oli ka keskkond, kuhu kogunesid järgmise väljalaske täiendused.

2.3 Testide automatiseerimise alustamine

Iga süsteemi puhul saab leida kõige tähtsama, mida süsteem teha võimaldab ning selle põhiprotsessi toimimist kontrollida. Kui põhiprotsessis esineb viga, siis avaldub see viga ka teistes testides. Põhiliste protsesside kontrolli nimetatakse ka suitsutestimiseks ning antud süsteemi puhul kuulusid sinna alla eri liiki laenude taotlemine ning seejärel lihtsamate lepingumuudatuste tegemine.

Niisamuti kui manuaalsel testimisel, on ka automaatsel testimisel suitsutestid tavalised esimesed, mida tehakse. Projekti puhul oli vaja automatiseeritud teste teha eri riike teenindavate süsteemide jaoks, millel olid omad iseärasused. Tähtsamad neist olid erinev protsess testkasutaja loomise ning sisse logimise puhul, samuti esines olulisi erinevusi taotlusvormi sisu osas. Põhiprotsessi automatiseerimise läbi loodi kõikide vajalike tegevuste läbiviimise jaoks vajalikud funktsioonid, mida oli võimalik teistes, keerukamates testides kasutada.

Testlugude ja nende põhjal testide koostamisel on vajalik, et süsteemi eeldatav käitumine oleks defineeritud ehk peavad olema konkreetsed ärireeglid, mille toimimist saab kontrollida. Tehnilise poole pealt, on testlugude automatiseerimiseks vajalik, et testitaval süsteemil oleksid olemas vajalikud liidesed ning et need oleksid piisavalt stabiilsed. Vaadeldud süsteemi puhul olid eelnevad tingimused täidetud: ärireeglid olid selgelt defineeritud ja süsteemil olid olemas testimiseks vajalikud liidesed, muutusi milles esines harva.

2.4 Testide automatiseerimise vajalikkus

Testide automatiseerimise alustamisel oli olemas teatav manuaalse testimise protsess. Kuid iga riigi põhiprotsesside pidev manuaalne testimine oleks liialt ajamahukas ning rutiinne tegevus, samuti on vea leidmise tõenäosus igal testimisel väike. Siinkohal võimaldaks tüüpilisemate testlugude automatiseerimine aega kokku hoida ning võimaldada testijatel keskenduda uuele funktsionaalsusele ning testlugudele, mida ei ole mõistlik automatiseerida, näiteks erinevad muudatused kasutajaliideses.

Laiemalt vaadatuna oli täiendav automatiseeritud testimine vajalik süsteemi ülesehituse tõttu, et kontrollida rakenduse toimimise funktsionaalsust *backend*-i ja seda kasutava front-endi piiri peal, süsteemi lõppkasutaja seisukohast. Kuna *frontend*-i ja *backend*-i arendasid erinevad tiimid, siis esines palju olukordi, kus nende ühendamisel funktsionaalsus ei toiminud või toimis valesti. Sellest lähtuvalt valiti ka vahendid testimise automatiseerimiseks: eesmärgiks oli tegevuse jada brauseris läbi teha ning ainult siis, kui see ei osutu võimalikuks, kasutada muid viise süsteemis tegevuste juhtimiseks. Projekti eesmärk ei olnud uute tehnoloogiatega eksperimenteerida, seetõttu oldi konservatiivsed ja eelistati levinud ja järeleproovitud tehnoloogiad. Täpsemalt on valitud vahendid kirjeldatud peatükis „Testraamistik“.

Lühidalt kokku võttes oli vaja testidega katta funktsionaalsus, milles kippus vigu esinema ning mida seni testiti vaid valikuliselt ja manuaalselt, lähtudes sealjuures lõppkasutaja vaatepunktist.

2.5 Krediidireitingu arvutuse testimine

Lisaks kasutajaliideses tehtavatele tegevustele oli vajalik ka süsteemi selle funktsionaalsuse testimine, mis mõjutab kasutajaliideses näidatavat. Olulist mõju kliendile pakutavate toodetele ja lisateenustele omab krediidireiting ja sellega seotud muutujad.

Kliendi oletatava maksevõime ja riski hindamiseks kasutatakse krediidireitingut. See tähendab, et kliendi kohta teada olevale infole (nt. haridus või töösuhe) omistatakse mingi arvväärtus (nt. kõrgharidus annab 10 punkti, keskharidus 6, jne..) ning liidetakse kõik saadud tulemused kokku. Seda, kas süsteem käitub nii nagu eeldatud, tuli testimise käigus kontrollida.

2.5.1 Uued kliendid

Uute klientide puhul pole süsteemis nende maksekäitumise kohta andmeid, seetõttu kombineeritakse kliendi poolt taotlusvormil esitatud ja välistest registritest pärinevaid andmeid. Kombineeritud andmete põhjal saadakse krediidireiting, mille põhjal otsustab süsteem, mis tooteid kliendile pakutakse. Täiendavalt filtreeritakse välja laenutooted, mille kuumakse on suurem kui kliendi sissetulekud ja olemasolevad finantskohustused maksta võimaldaksid.

Eelnevalt üldiselt kirjeldatud tingimused on täpselt defineeritud ärireeglites, mille põhjal koostatakse konkreetsed testlood. Üldjuhul on tegemist tabeli formaadis dokumendiga, kus on näidatud ära sisendväärtused, iga sisendväärtusele vastav väljund (arv, mida kasutatakse krediidiskoori arvutamiseks), terve krediidiskoor ja oodatavad laenutooted, mida süsteem pakkuma peaks. Pakutavatel toodetel võib olla ka täiendavaid atribuute, näiteks peab pakutav toode kuuluma kindlasse gruppi.

Testlood on üldjuhul koostatud testijuhi poolt ning koostamine on valdavalt manuaalne protsess. Ühte riiki teenindava süsteemi jaoks on oluline katta kõik muutujate väärtused. See tähendab, et kui muutuja X võib omada väärtusi 3, 5 ja 7, siis on kõikidele nendele väärtustele vastavaid sisendeid testilugude kogumis kasutatud vähemalt üks kord.

Töö käigus tuli koostatud testlood automatiseerida võimalikult väikese ajakuluga. Selleks kasutati projekti käigus kirjutatud Selenide põhjal loodud abifunktsioone konto loomise, vormi täitmise ja kasutajaliideses pakutavate toodete tuvastamiseks. Nende abil läbiti taotlusprotsess sisestades testlugudes kirjeldatud andmeid. Seejärel kontrolliti pakutavate toodete vastavust eeldatud toodetele.

Sarnane protsess tuli läbi teha kõikide testitavate riigi-brändi kombinatsioonide lõikes, mida oli kokku 4. Iga konkreetse kombinatsiooni kohta käivaid teste oli 20-30. Tavaliselt olid sellised testilood ka esimesed, mille automatiseerimist alustati. Sellise testi näide on toodud Lisas 5.

Alguses oli vaja iga süsteemi jaoks teeki kohaldada, põhjuseks teistsugune sisse logimise meetod, erinevad vormiväljad ja vahel ka erinev tulemuste kuvamise formaat. Peale seda oli võimalik testilugusid automatiseerida. Algselt olid testlood koostatud Testlink rakenduses ning nendes esitatud sisendite-väljundite ülekandmine Java koodi oli üsna rutiinne protsess. Kuid kuna testlood olid algselt loodud Excel-is, nende formaat oli vähemalt riigiti ühesugune, siis sai

kirjutatud ka skript, mis Excel-i failist olevatest testlugude definitsioonidest genereerib Java testklassi funktsioonid.

Seda liiki testid olid ühed lihtsamad ja üksiku testi käivitamine võttis ligikaudu minut aega, mis on palju vähem kui testi käsitsi läbitegemine. Nende käivitamise käigus leiti üksikuid vigu, kuid need andsid kindluse süsteemi lihtsama funktsionaalsuse toimimisest.

2.5.2 Teenust varem kasutanud kliendid

Kui kliendil on kunagi varem olnud leping või on praegune leping kestnud rohkem kui ärireeglites nõutud, siis krediidiskoori arvutamisel kasutatakse rohkem süsteemis olevaid kliendi (makse)käitumist kirjeldavaid andmeid. Samas jäävad kasutusse välistest registritest saadud andmed.

Samuti nagu uute klientide puhul, pidi siinkohal ärireeglitest välja lugema krediidiskoori arvutamisel aluseks olevad muutujad ning võimalikult palju nende sisendi-väljundi kombinatsioone läbi testima. Mõned näited seda laadi muutujatest: eelnevate laenulepingute arv, makse laekumise ja arve tähtaja vahe, kasutatud krediidilimiidi protsent.

Selle jaoks, et neid teste kirjutada saaks oli vaja teha eeltööd: kirjutada suuremat protsessi läbi tegevad abifunktsioonid. Näiteks oli vajalik meetod, mis käiks läbi kogu taotlusprotsessi saades mingi kindla krediidireitingu, kontrolliks pakutavaid tooteid ning kinnitaks laenutaotluse.

Selleks tuli teha täiendusi brauseri juhtimise, andmebaasist päringuid tegevatele ja SOAP-liidesega suhtlevatele funktsioonidele. Näiteks osutusid siinkohal vajalikuks arvete genereerimise ja maksmise funktsionaalsus. Samuti tuli klienti andmebaasis ajas tagasi “nihutada”, et arved reaalsete ajavahemike tagant genereeritaks.

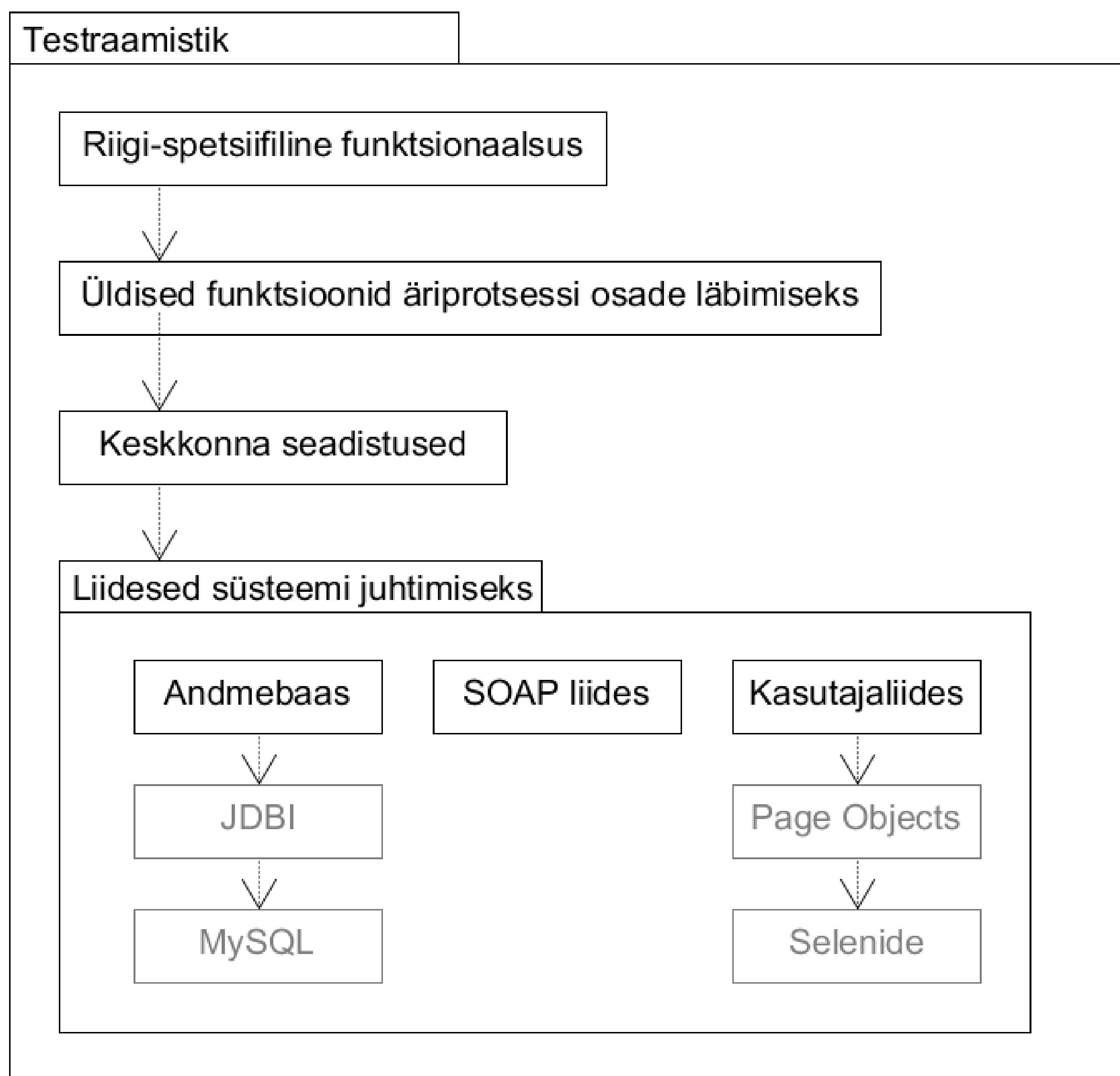
2.6 Kliendile saadetavad dokumendid

Kliendile saadetakse taotlemise käigus ning teatud tingimuste täitumise korral lepingu vältel erinevaid emaile, koos nendele lisatud PDF formaadis dokumentidega. Näideteks võib tuua individuaalsed lepingutingimused ja arved. Emaili sisu ja dokumendid genereeritakse dünaamiliselt vastavate mallide põhjal. Vahel tekib vajadus kõik ühe riigi süsteemi poolt

saadetavad dokumendid üle kontrollida, näiteks kui malle muudetakse. Samuti on võimalik, et muudetakse mingite kirjade ajastust, näiteks meeldetuletusarve saadetakse 5 päeva pärast maksetähtaja möödumisest, varasema 10 asemel. Ka seda, et vajalikud dokumendid saadetaks vajalikul ajal, on tarvis kontrollida. Kuigi tihtipeale saab testandmebaasist leida või siis luua vajalikele tingimustele vastav kasutaja ning käivitada dokumentide saatmine, siis kõigi dokumentide kontrollimine sellisel viisil osutub väga aeganõudvaks. Seetõttu on vajalik luua skript, mis teeb uue kasutajakonto, läbib sellega kõik dokumentide saatmise seisukohalt olulised olekud, suunates süsteemi poolt saadetavad meilid etteantud meiliaadressile.

3. Testraamistik

Et oleks võimalik süsteemi ärinõudeid testida, on vajalik süsteem viia seisundisse, kus need nõutavad protsessid käivituvad. Selleks on vaja juhtida brauserit, lugeda ning kirjutada andmebaasi andmeid, ning SOAP-liidese abil käske käivitada. Lisaks tuleb kirjutada abifunktsioone, mis kasutavad kõiki kolme eelnevalt kirjeldatud süsteemiga läbimise viisi, et automatiseerida pikemaid äriprotsesse. Järgnevalt on kirjeldatud põhilisi testraamistiku komponente.



Joonis 2. Ülevaade testraamistiku ülesehitusest.

3.1 Brauseri juhtimine

Brauseri juhtimiseks on levinud Selenium [1] raamistiku kasutamine, millel on liidesed paljude programmeerimiskeelte jaoks, sealhulgas Java. Kuigi Java API võimaldab teha kõike testimiseks vajalikku, nõuavad mõningad tavalised tegevused rohkem koodiridu, kui sooviks. Näiteks tuleb anda ette õiged parameetrid avatavale brauserile, teha abifunktsioonid veebilehe elementide järel ootamiseks ning kuvatõmmiste tegemiseks. Sellise funktsionaalsuse kirjutamine oleks vajalik, kuid ei annaks testimise seisukohalt midagi juurde.

Seetõttu otsiti alternatiivset Java-toega raamistikku, mida oleks mugavam kasutada ning valik langes Selenide raamistiku [2] peale. Tegemist on Java raamistikuga, mis lihtsustab tüüpilisemad Selenium-iga tehtavad tegevused, samas võimaldab vajaduse korral ka otse Selenium API meetodeid kasutada.

Kasutades Selenide raamistiku võimalusi on koostatud Page Objects mustrit [3] [4] kasutavad klassid, mille abil on loodud abstraktsioon veebilehe elementide jaoks. Näiteks on loodud meetod `enterName(String name)`, mis kontrollib, et vormi element id-ga firstName on olemas, teeb selle tühjaks ja seejärel sisestab sinna soovitud väärtuse. See võimaldab testkoodi funktsionaalsusest paremini aru saada, kuna vabastab selle kasutajaliidese ülesehituse detailidest. Samuti on kasutajaliidese muutusi lihtsam hallata, kuna on ainult üks koht, kus identifikaatoreid muuta.

Vaata ka Lisa 6: kasutajaliidese juhtimiseks mõeldud klassi koodinäide.

3.2 Andmebaas

Testitav süsteem kasutas MySQL andmebaasi, millest oli mõnedes testides kontrollimiseks andmeid lugeda, samuti oli mõningatel juhtudel vajalik andmeid andmebaasis muuta.

Et eeldatav andmebaasi kasutus oli oma iseloomult lihtne ja hõlmas vaid väikest osa andmebaasis salvestatud andmetest, lähtuti sellest ka sobiva andmebaasi teegi valikul.

Valituks osutus JDBI teek [5], mis võimaldab minimaalse seadistamise järel SQL päringuid kirjutada ning nõuab vähem koodikirjutamist kui Java JDBC API. Samuti ei eelda JDBI mingi suurema raamistiku kasutamist, vaid on eraldiseisev teek.

ORM (Object-relational mapping) lahendused antud projekti ei sobinud, kuna nõuavad liialt palju häälestamist (näiteks tuleks luua tabelitele vastavad Java klassid ja õigesti annoteerida). Samuti lisaks antud lahenduste kasutamine täiendava abstraktsioonitaseme, ilma olulist täiendavat kasu toomata. Testides vajalikud SQL päringud on üldjuhul piisavalt lihtsad, et neid käsitsi koostada; lisaks saab koostatud SQL päringuid ka mingi protsessi osa käsitsi läbi tehes lihtsalt kasutada või juba olemasolevaid päringuid projekti lisada.

Vaata ka Lisa 2: Andmebaasiga suhtleva funktsiooni koodinäide.

3.3 SOAP-liides

Selleks, et kogu laenu taotlemise ja väljastamise protsess läbi teha tuli teha osa tegevusi kliendina, aga samuti oli tarvis teha läbi tegevusi, mida käivitatakse süsteemi poolt perioodiliselt või mis on osa protsessist, mis laenuhalduritel laenu väljastamise protsessis läbi teha tuleb. Laenutaotluste läbivaatamisel ja haldamiseks kasutatakse Salesforce põhjale ehitatud lahendust, mille suhtlus testitava süsteemiga käib valdavalt läbi SOAP-liidese.

Osad süsteemi poolt pakutavad SOAP teenused osutusid testimisel vajalikuks, siinkohal mõned näited: kasutaja identifitseerimise kinnitamine, laenutaotluse aktsepteerimine, arvete genereerimine, kliendi maksete import. Testide automatiseerimise alguses olid olemas ka mõnede süsteemi SOAP-teenuste kasutamiseks näidispäringud, mis olid salvestatud SoapUI programmi projektina.

Eelnevast lähtudes oli vaja olemasolevaid SOAP päringuid testides kasutada ning aja jooksul täiendavaid lisada.

Levinuim meetod, kuidas Java koodis SOAP-liidese poole pöörduda, on käsurea vahendiga WSDL või mõne muu liidest defineeriva faili põhjal genereerida kood ning genereeritud koodi abil kasutada serveri poolt pakutavaid teenuseid. Selline lähenemine on tüüpiline ning kindlustab selle, et serveri poole pöördutakse korrektses formaadis päringud (andmetüüpe kontrollitakse) ning vastus konverteeritakse automaatselt Java objektiks. Näited vahenditest, mis on seda võimaldavad on wsimport [6], wsdl2java [7] ja maven-jaxb2-plugin [8].

Kuna testides oli vaja vaid väikest osa teenustest, päringute koodi tasemel korrektsusest oli tähtsam nendest lihtsam aru saamine ning vastustest kasutati vaid mõnda elementi, mitte keerulisi andmestruktuure, otsustati teistsuguse lahenduse kasuks. Põhiline kriteerium oli, et

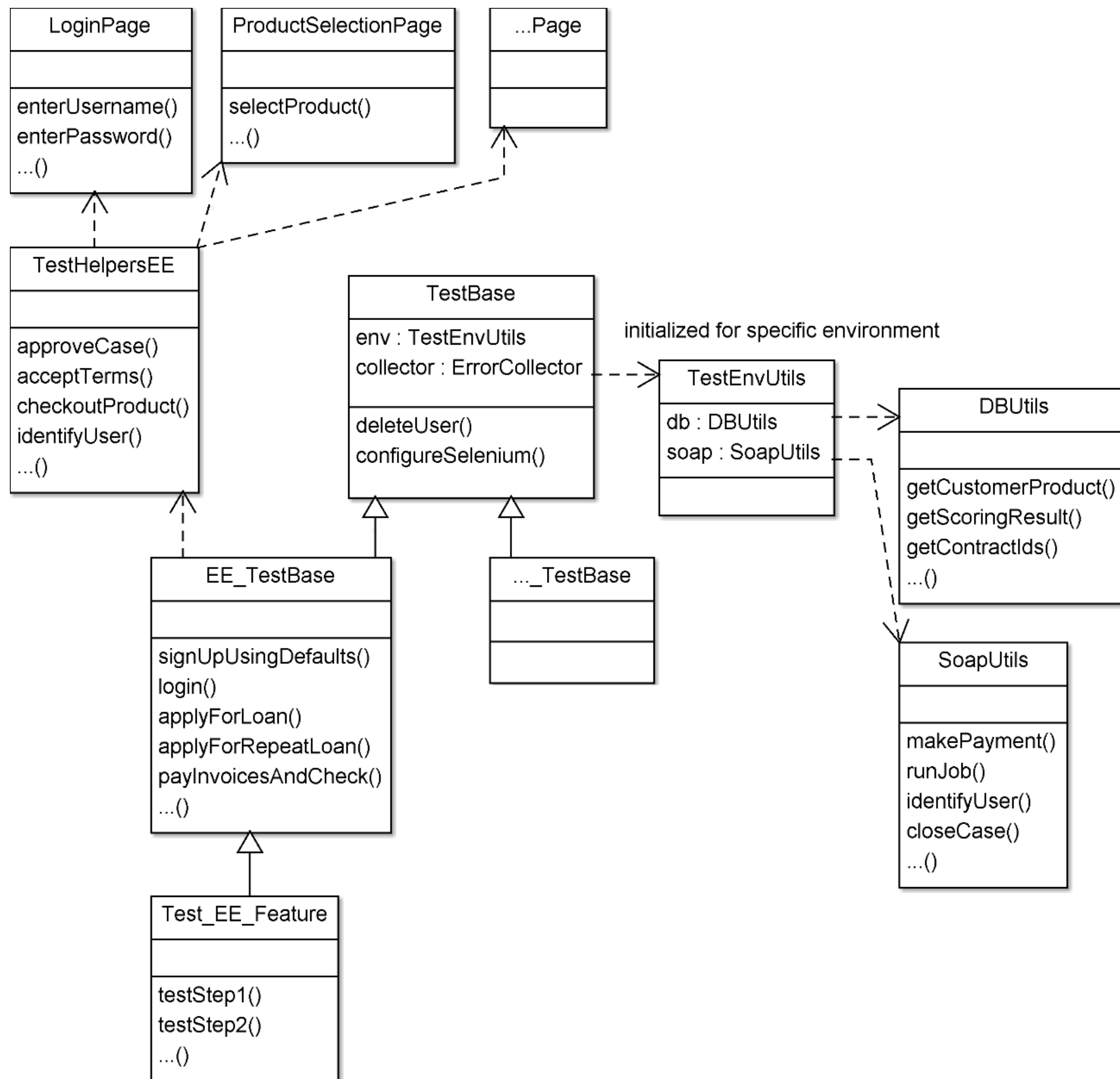
eelnevalt SoapUI-s koostatud päringuid võimalikult väheste muudatustega projekti koodis kasutada saaks.

SOAP-liidese poole pöördumine realiseeriti kasutades soap-ws teeki [9] WSDL-i põhjal XML päringu põhja genereerimiseks, jOOX teeki [10] XML päringu põhja täitmiseks vajalike andmetega ning vastusest andmete kätte saamiseks ning Apache HttpClient teeki [11] päringute tegemiseks. Kasutades eeltoodud komponente oli võimalik lihtsa vaevaga koostada ja kasutada SOAP päringud Java koodis, lähtudes sealjuures SoapUI projekti näidetest.

Vaata ka Lisa 3: SOAP-liidesega suhtleva funktsiooni näide.

3.4 Testraamistiku ülesehitus

Testide automatiseerimise käigus kirjutati palju klasse ja funktsioone, mis mingi osa testimise jaoks vajalikuks osutusid. Osa funktsionaalsust oli vaja erinevate riikide süsteemide jaoks, osa oli üldjoontest kõigi riikide jaoks sama, kuid teatava riigi-spetsiifikaga, millega oli oluline arvestada. Et kirjutatud koodi oleks võimalik lihtsalt kasutada ning täiendada, tuli kindlaks määrata kuidas seda struktureerida, mida on järgnevalt illustreeritud ja kirjeldatud.



Joonis 3. Testraamistiku osaline klassidiagramm.

Kõigi testimiseks mõeldud klasside üheks baasklassiks on klass nimega `TestBase`. See sisaldab üldist funktsionaalsust, mida kõik testid vajavad. Näiteks loob vajalikud andebaasi ja SOAP-liidese klasside instantsid vastavalt brändile ja riigile; konfigureerib käivitavad brauserid vaikimisi väärtustest veidi erinevalt. Samuti initsialiseeritakse kasutatavad JUnit-i täiendused. Nendeks on testide nimede logimine enne ja pärast testi lõppu ning ErrorCollector, mis võimaldab testi käigus tekkinud vead kokku koguda ja siis testi lõpu tehes nende põhjal teha järelduse kas test õnnestus või mitte. `TestBase` klassi on jäänud ka üksikud projekti alguses kirjutatud funktsioonid kasutaja kustutamiseks, mis sobivad sinna kontseptuaalselt.

Mõnede testide koostamiseks eelnevast baasklassist piisab, kuid paljudel juhtudel laiendatakse seda klassi täiendava funktsionaalsusega, mis on riigi või brändi spetsiifiline. Ühe näitena registreerumise ja sisse logimise protsess on riigiti erinev; samuti on teatud süsteemi osade

testimiseks vajalik luua kindlate profiilidega kasutajad ning selle jaoks vajaminevad funktsioonid kuuluvad laiendatud klassi funktsioonide hulka.

Kõigi erinevates testides vajaminevate funktsioonide koondamisel ühte klassi kaoks selge ülevaade koodist, sest klassi funktsioonide arv paisuks liialt suureks, samas funktsioonide mõistlik grupeerimine poleks võimalik. Seetõttu jaotati arendamise käigus vajaminevad funktsioonid erinevatesse abiklassidesse: eraldi klassides olid arvete genereerimise ja maksmisega seotud funktsionaalsus, mõnede lisateenuste (näiteks läbi kasutajaliidese limiidi suurendamine) kasutamiseks mõeldud kood, abiklassid kasutajaliideses pakutavate toodete ja arvutatud krediidiskoori kontrolliks. Samuti olid eraldatud riigi-spetsiifilised funktsioonid ning üldised funktsioonid, mis kuhugile paremini ei sobinud.

Eelnevas lõigus välja toodud funktsioonid olid loodud staatilistena, kasutades Java `static` võtmesõna. Ehk siis nende kasutamiseks ei olnud vaja testkoodis klassi initsialiseerida, mis tegi nende kasutamise lihtsamaks. Teisest küljest kaasnes aga iga testiga teatud hulk olekumuutujaid, mida tuli iga funktsiooni väljakutsumisel parameetritega edasi anda. Näidetena võib välja tuua viite testkeskkonna klassi instantsile (teeb päringuid õigest andmebaasist ning SOAP-liidesest), viide `ErrorCollector` klassi instantsile ja testkasutaja identifitseerimist võimaldav kood. Selle lisandusid veel funktsiooni sisulise tähendusega muutujad, tootevaliku näitel: millist toodet peaks valima, milliseid tooteid peaks üldse pakutama ning millist krediidiskoori eeldama peaks. Aja jooksul lisandusid osadele funktsioonidele täiendavad parameetrid, et neid saaks algest erinevates testides kasutada ning tulemusena tekkisid mõned üksikud funktsioonid, mis vajasid 8-10 parameetrit. Kuigi nii võimaldati koodi taaskasutamist, raskendab selline puudus funktsioonide kasutamist. Abiks oleks olnud, kui testide kirjutamiseks kasutatav programmeerimiskeel toetaks funktsioonide väljakutsumisel parameetrite vaikimisi väärtuste seadmist (default parameters) ja võtmesõnaga parameetrite väärtustamist (keyword parameters). Kuna Java seda ei võimalda, siis kasutati funktsioonide ülelaadimist, et vähendada sagedamini kasutavate funktsioonide parameetrite arvu. Näide on toodud Lisas 1.

Iga testi puhul on olemas teatud eeltingimused, mis peavad olema täidetud. Antud projekti raames on iga testi alustamisel vajalik, et testis kasutavate andmetega kasutajat ei oleks süsteemis olemas. Selle saavutamiseks on kaks võimalust: genereeritakse juhuslikke identifikaatoreid, kuni leitakse selline, mida süsteemis pole või kustutakse testis kasutavate

identifikaatoritega kasutaja enne testi algust. Lisaks kustutakse testis loodud kasutaja testi lõpus. Siinkohal prooviti algelt kasutada JUnit raamistiku võimalusi `@BeforeClass` ja `@AfterClass` annotatsioonide näol. Nende annotatsioonidega märgitud meetodid käivitatakse nagu nimest aru saada, pärast ja enne teiste klassides kasutatud funktsioonide käivitamist. Sellisel lähenemisel oli aga töös kirjeldatud projekti puhul puudusi, mistõttu ei osutunud nende kasutamine võimalikuks. Esimene probleem seisnes selles, et meetodid, millel neid annotatsioone kasutada saab, peavad olema staatilised, mida aga ei õnnestunud saavutada. Iga testklassi instantsi puhul initsialiseeritakse ka „keskkonna“ klass, mis määrab mis andmebaasi ja mis URL-e test kasutama hakkab. Keskkond võib aga olla igal instantsil erinev, mistõttu staatiliste muutujate kasutamise korral kirjutavad paralleelselt käima lastud instantsid teineteise muutujad üle. Teine probleem seisnes testide üldises ülesehituses: tervikliku testi läbi tegemiseks kulunud aeg oli minutites. Seetõttu arendamise ja paranduste tegemise hõlbustamiseks jaotati test osadeks – mitmeks funktsiooniks, et ei peaks mingi alamosa muutmiseks tervet testi läbi teha. Alamosa käivitamiseks aga ei ole vaja eelnevaid ja järgnevaid tegevusi, nagu kasutaja kustutamine. Seetõttu tuli arendamise käigus annotatsioonid välja kommenteerida ja pärast uuesti kood aktiivseks teha, mis lisas veel ühe aspekti, mida jälgida tuli. Lisaks eelnevale ei võimaldanud kasutatud töövahendid ainult *setup* meetodit (`@BeforeClass`) või ainult *teardown* (`@AfterClass`) meetodit eraldi käivitada.

3.5 Täiendavad testimiseks loodud abivahendid

Testide automatiseerimise käigus selgusid mõned aspektid, mis testimisel probleeme valmistasid ning neid püüti aja olemasolul lahendada. Järgnevaid osi ei saa otseselt testraamistikku alla liigitada, pigem on nende näol tegemist eraldiseisvate täienduste ja prototüüpidega, mille raamistikku integreerimisega tuleks edaspidi tegeleda.

3.5.1 Tootevaliku testide automaatne genereerimine

Tootevaliku testide puhul kontrollitakse, et krediidiskoori moodustavad muutujatele omistatakse õiged väärtused ja nende põhjal pakutatakse kliendile korrektseid laenutooteid. Lisaks tuleb arvestada täiendavate piirangutega, näiteks pakutavate laenude kuumakse ei tohi ületada kindlat protsenti netosissetulekust. Riigiti on pakutavad tooted ja täiendavad piirangud

erinevad. Pakutavad tooted muutusid projekti vältel korduvalt ning testide koostamine käsitsi muutus liiga suureks ajakuluks, siis otsustati testide genereerimine automatiseerida.

Vajalike skriptide koostamiseks kasutati Python programmeerimiskeelt, lahendus nägi välja järgnev: Iga riigi jaoks on krediidiskoori kujundavad muutujad erinevad, muutujate erinevad võimalikud väärtused ja väärtustele vastav skoor kirjutati programmikoodi sisse. Järgmisena lisati pakutavaid laenutooteid kirjeldavad andmed: milliseid tooteid peaks kindla krediidiskooriga kasutajatele pakkuma, millistesse gruppidesse nad kuuluvad. Eelneva puhul tuli arvestada eri toodete tüüpidega: krediidikonto ja väikelaenude puhul kehtivad erinevad reeglid. Lähteandmed saadi süsteemi kirjeldavatest dokumentidest, sh. Excel-i tabelitest ning osadel juhtudes ka andmebaasist.

Järgmisena genereeriti krediidiskooride vahemikud nii, et iga vahemiku jaoks oleks pakutavad tooted erinevad. Nüüd oli tarvis teada saada mõni krediidiskoori muutujate väärtuste kombinatsioon, mis langeks etteantud vahemikku. Selleks kasutati kitsendusreegleid ja teeki, mis nende põhjal vajalikud väärtused leiaks, konkreetsemalt python-constraint [12]. Määratud kitsenduse sõnaline kirjeldus on järgnev: leia muutujate väärtused, mille summa jääb kindlasse vahemikku, nii et kõiki muutujaid kasutataks üks kord ning võimaluse korral väldi eelnevalt valitud väärtuste kasutamist. Osade muutujate väärtused sõltusid teistele muutujatele juba omistatud väärtustest, ka sellega tuli arvestada.

Nüüd olid olemas testide jaoks vajalikud sisendandmed (milliseid väärtusi peaks vormides valima) ja eeldatud väljundid (millised tooted peaks olema pakutud). Selleks, et leitud väärtuste põhjal töötavaid teste koostada, oli aga vajalik edasine töö. Üks variant mida kaaluti, oli väärtuste salvestamine vahepealsesse andmeformaati, mida siis testidest loetaks. Selleks võis olla mingi XML-fail või Java klass vajalike andmetega. See oleks aga tähendanud veel ühe vahekihi lisamist testidesse, mis oleks testidele täiendavat keerukust lisanud.

Seetõttu otsustati koodi genereerimise kasuks: võeti aluseks eelnevalt koostatud tootevaliku testide kood ning loodi selle põhjal koodi mall *(template)*, kasutades selleks sobilikku Python standardteegis olevat funktsionaalsust [13]. Tootevaliku testide ülesehitus oli eelnevalt paika pandud ning võimaldas kontrollida kõike vajalikku: seetõttu ei pidanud siinkohal vaeva nägema näiteks andmete formaadi määramisega ja täiendavate abifunktsioonide kirjutamisega. Teste genereeriti sel viisil mitme riigi süsteemide jaoks ning nende omavahelised erinevused, eriti sisendandmete osas, olid suured. Seetõttu kasutati iga riigi süsteemi jaoks erinevat koodi malli.

Eelnevalt kirjeldatud testid kontrollisid ainult põhilist toodete pakkumise funktsionaalsust, jättes vaatluse alt välja täiendava toodete filtreerimise. Filtreerimise testimine eraldi oli taotluslik, sest võimaldas kõrvalekallete korral lihtsamini leida vea põhjuse. Kuid nende testide genereerimiseks oli vaja teha olulisi täiendusi skripti koodi.

Toodete täiendava filtreerimise põhimõte on üldjoontes järgmine: Esiteks arvutatakse välja pakutavate toodete kuumakse, see sõltub laenu suurusest, intressidest ja täiendavatest tasudest. Teiseks arvutatakse välja näiteks kliendi sissetuleku ja olemasolevate kohustuste põhjal muutuja, mille põhjal otsustatakse kas mingi laen on kliendile kättesaadav või mitte. Täpsed reeglid võivad sõltuda kliendi sissetuleku grupist, laenu liigist ning laenuperioodist. Lisaks on võimalik, et osadel piirjuhtudel suunatakse kliendi laenutaotlus täiendavaks läbivaatuseks klienditeeninduse poolt.

Skriptis tuli eelneva loogikaga arvestamiseks teha järgnevat: Esiteks genereerida loend kõikidest kliendile pakutavatest laenudest, arvutada piisava täpsusega iga laenu kuumakse, seejärel iga laenu jaoks leida maksimaalne sissetuleku/kohustuste suhtarv, mille puhul seda laenu veel pakutakse. Suhtarvu leidmiseks kirjutati funktsioon, mis tagastaks negatiivse väärtuse siis, kui sisendparameetrite põhjal peaks laenu välja filtreerima ning positiivse väärtuse siis, kui sisendparameetrite põhjal peaks laenu pakkuma. Seejärel kasutades SciPy teegi `bisect` funktsiooni [14], leiti sisendväärtus, mille puhul funktsiooni väljund oleks 0 ehk lõikepunkt, millest alates ei tohiks laenu enam pakkuda. Seda tehti kogu laenude loendi jaoks.

Nüüd oli olemas loend kõikidest pakutavatest laenudest koos „lõikepunktidega“. Nüüd tuli välja valida mingi näiteks sissetuleku väärtus ning kontrollida, et filtreerimise reeglid toimivad. Kuna arvutatud väärtused olid ligikaudsed, siis tuli välja valida loendist mingi selline element, mis põhjustaks teatud arvu toodete väljafiltreerimist ning teised tooted oleksid endiselt pakutud. Seetõttu valiti punkt nii, et kahe järjestikuse näiteks sissetuleku väärtuste vahe oleks teatud arvust suurem ning valiti nende väärtuste keskpunkt. Tuleb mainida, et ülesanne oleks olnud oluliselt lihtsam, kui kõik arvud oleksid arvutatud täpselt samal viisil, mis süsteemis.

Nüüd jäi ainult üle arvutatud väärtused koodi malli lisada, selle põhjal testid genereerida ning need käivitada. Samuti lisati genereeritud koodi kommentaarina info krediidiskoori

moodustumise ning välja filtreeritud toodete kohta, et lihtsustada eeldatust erinevate tulemuste korral olukorrast arusaamist.

Skript võimaldab muutuste korral toodetes genereerida vajalikud testid vähema vaevaga ja kiiremini, kui kogu arvutuskäiku käsitsi läbi tehes ning anda kiiremat tagasisidet pakutavate toodete korrektsuse kohta.

3.5.2 Kasutajaliides levinuimate kliendikonto tegevuste jaoks

Töö käigus koostati teatud hulk funktsioone, mis võimaldasid kliendi kontoga teha testimisel vajalikke toiminguid, nagu näiteks arvete genereerimine ja maksmine, väljamaksete teostamine, kontoga tehtud toimingute "ajas nihutamine" ja palju muud. Kõike seda sai teha kirjutades vajaduse korral koodi või ka läbi klienditeeninduse poolt kasutatava graafilise kasutajaliidese, kuid need meetodid ei olnud kasutamiseks kõige mugavamad: ühekordseks kasutamiseks mõeldud koodi kirjutamine on ikkagi aeglasem kui lihtsalt mõne nupu vajutamine ja olemasolevas liideses oli vaja lisaks testimise jaoks olulistele vaja täita ka täiendavaid välju, et süsteem tegevuse läbi laseks. Tulemusena otsustati teha kasutajaliidese prototüüp, mis võimaldaks kasutada testimiseks mõeldud sagemini kasutatavaid funktsioone.

Kuna testid olid koostatud Java programmeerimiskeelt kasutades ning vormistatud Maven-i projektina, siis oli üsna lihtne genereerida testide projektist Java teek jar-failina, mis sisaldab kogu süsteemiga suhtlemise funktsionaalsust, mida omakorda loodav rakendus kasutada saaks.

Rakenduse prototüüp tehti eraldiseisva veebirakendusena, kasutades backendis Spring raamistikku ning front-endis veidi jQuery teeki, kuna nende vahenditega oli autoril eelnev kokkupuude. Tulemusena saadi kasutajaliides, mis võimaldas valida testkeskkonda, millega rakendus suhtles ning erinevate identifikaatorite põhjal leida vajalik kasutaja. Kasutajaga sai teha konto loomise protsessis vajalikke tegevusi nagu emaili kinnitamine, taotluse kinnitamine, väljamakse tegemine; samuti genereerida arveid ja neid maksta, kas tähtajal või teatud arv päevi enne või peale seda; lisaks mõned muud funktsionaalsused.

Selline kasutajaliides tuli kasuks testide kirjutamise ajal, kui oli tarvis erinevaid stsenaariume läbi proovida, et valida sobilik automaattestide jaoks. See on üsna piiratud kasutusjuht ning teiste, juba tuttavate, alternatiivide olemasolul ei leidnud see rakendus laiemat kasutust,

mistõttu jäigi see esialgu lihtsalt prototüübiks, mis tõestas seda, et testides kasutatud funktsionaalsust on võimalik ka teistes rakendustes kasutada.

3.5.3 Skriptid registreerimisprotsessi läbimiseks

Registreerimisprotsessi läbimiseks läbi kasutajaliidese mõeldud kood moodustas testraamistiku tuuma, kui esines olukordi, eeskätt teatud vigade silumise *(debugging)* protsessis, kus oli vaja teistsugust lähenemist.

Rakenduse *backend* koosnes REST-teenustest, mida kasutajaliides kasutas. Seega sai mõninga eeltöö järel vajalike teenuste poole ka otse pöörduda, mis oli teatud olukordades kasulik. Näiteks kui konto registreerimise protsessi käigus tekkis viga, siis võis selle põhjus olla nii kasutajaliidese koodis kui ka *backend*-i koodis. Sama situatsioon oli ka vigade korral iseteeninduse loogikas. Otse REST-teenuseid kasutades võis vea põhjuse kiiremini leida ning samuti muutis see rakenduse *backend*-i koodist veakoha leidmise lihtsamaks – sel juhul pöördutakse ainult huvipakkuvate teenuste poole, mis toob kaasa vähem kõrvalist müra logides ja samuti kiirema rakenduse töö.

Kuna ei olnud eraldiseisvat koodi, mis süsteemiga läbi REST liidese suhtleks, siis tuli see töö raames ise koostada. Et eesmärgiks ei olnud integratsioon raamistikuga ja võimalikult lihtne teostus, siis valiti programmeerimiskeeleks Python ning HTTP teegiks requests [15]. Nüüd jäi üle ainult aru saada kasutajaliidese REST-liidese kasutamise põhimõttest ja huvipakkuv osa sellest skriptina vormistada.

Esimese asjana tuli läbida sisselogimise või konto registreerimise protsess ning selle käigus saadetud sessioni küpsised *(cookie)* salvestada, et neid järgnevates päringutes kasutada, selleks sai kasutada valitud teegi poolt pakutud *session* objekte [16]. Lisaks tuli lisada päringutele kindlad päised, mida kasutajaliides kasutas. Nüüd sai kasutada iseteeninduse REST-liideste funktsionaalsust. Taotluse esitamise protsessi läbimiseks skripti poolt tuli brauseri konsoolist jälgida tehtud päringuid ja need, koos õige URL-i, päringu tüübi ja sisuga (JSON formaadis) skriptis kirja panna, sisu vastavalt vajadusele muutes.

Tehtud skripti võis kasutada nii testserverite, kui ka lokaalses masinas jookva *backend*-i pihta päringute saatmiseks ja erinevate veaolukordade silumise protsessis.

3.5.4 Dokumentide korrektsuse automatiseeritud testimise prototüüp

Süsteemi töö käigus genereeritakse ja saadetakse erinevaid emaile ja PDF-formaadis dokumente: klientide individuaalsed lepingutingimused, arved ja meeldetuletuskirjad. Kuna dokumentide formaati, kujundust ja sisu aeg-ajalt muudetakse, on vaja veenduda, et muutuste käigus midagi kõrvalist katki ei tehtud. Selle protsessi käigus on abiks eelnevalt loodud skript, mis saadab kõik protsessi käigus tekkinud emailid ja koos PDF-formaadis dokumentidega. Kuid nende sisu ja kujundust peab ikkagi üle vaatama töötaja, kes peab olema kursis dokumentidele esitatud nõuetele ja valdab keelt, mida dokumendid kasutavad. Suurte muudatuste puhul on see mõistlik tegevus, kuid peale igat väiksemat täiendust või veaparandust kõikide dokumentide kontrollimine ilma muude abivahenditeta osutub liiga suureks ajakuluks. Seetõttu otsustati uurida dokumentide kontrollimise automatiseerimist.

Protsess pidi välja nägema järgnev: kasutades süsteemi teste genereeritakse XML-id, mis sisaldavad vajalikke andmeid PDF-ide genereerimiseks. XML-id saadetakse vastavasse teenusesse, mis genereerib andmete põhjal PDF-id. Loodud dokumentide kontroll jäeti autori lahendada.

Lahenduseks pidi valima sobiliku testraamistiku, mis võimaldaks genereerida arusaadavaid testiaruandeid. Dokumentide korrektsuse kontrolliks otsustati kasutada visuaalset võrdlust ehk siis tuli PDF-id pildiformaati teisendada ja kontrollitavat dokumenti võrrelda teise dokumendiga, mille kohta on teada, et see on korrektne. Kuigi PDF-id on võimalik teisendada nende sisu ja kujundust kirjeldavateks XML-failideks, siis saadud failide võrdlus meid ei aita – seose loomine struktuuri ja dokumendi kuvamisel nähtavate elementide vahel on raskendatud.

Loodud prototüübis konvereeriti PDF formaadis dokumendid PNG formaadis piltideks. Kui dokumendis oli mitu lehekülge, siis loodi üks suur pilt. Selleks kasutati Python-i Wand teeki [17], mis võimaldab kasutada osasid funktsioone, mida pakub pilditöötluseks mõeldud ImageMagick tarkvara [18]. PDF-idest vajaliku informatsiooni, nagu näiteks lehekülgede arv, lugemiseks kasutati pdfquery teeki [19], mis võimaldab ka dokumentide sisu kätte saada.

Nüüd oli olemas pildid dokumentidest (korrektsest ja testitavast), mida sai omavahel võrrelda. Selleks kasutati ImageMagick-u võrdlusfunktsiooni, mis tagastab piltide erinevuse valitud numbrilise näitajana ning pildi, millel kuvatakse teise värviga erinevused piltide vahel. Näitaja,

mis kõige paremini iseloomustas piltide erinevust oli erinevate pikslite protsent. Kuna aga võrreldi põhiliselt teksti sisaldavaid pilte, siis isegi oluliste erinevuste korral jäi see protsent väikseks, sest valge taust võtab enda alla tunduvalt rohkem piksleid, kui õhukeses fondis teksti osa. Parema numbrilise tulemuse saavutamiseks oleks pidanud erinevuse pildil kasutama laiendamise *(dilate)* operatsiooni [20], et lähedalasuvad pikslid suuremaks regioonideks muuta ning arvutama nende regioonide osakaalu tulemusena saadud pildil. Lisaks oleks võinud leida leitud regioonide äärejoonte koordinaadid *(contours)* ning neid kasutades erinevustel "ringid ümber tõmmata" [21] [22]. See funktsionaalsus jäeti aga prototüübist välja.

Testraamistiku valikul oli põhiliseks kriteeriumiks see, et see võimaldaks genereerida kergesti mõistetavaid aruandeid ning võimaldaks aruannetele lisada pilte. Kuna projekti käigus sai uuritud erinevaid võimalusi aruannete genereerimiseks ning katsetatud Allure [23] raamistiku võimalusi, otsustati kasutada just seda. Python-i testraamistikest on Allure poolt toetatud pytest [24], mistõttu valiti testide loomiseks just see. Loodi väike näidistest, milles kasutati eespoolt kirjeldatud dokumentide võrdluse koodi, kontrolliti piltide sarnasust ning genereeriti aruanne, millele lisati võrdluses kasutatud ja võrdluse tulemusena saadud pildid.

3.6 Testid CI keskkonnas

Koostatud teste käivitati igal öösel Linux-it kasutavas CI-serveris Jenkins-ist. Kuna automatiseeritud testidega kontrolliti põhiliselt ärireegleid, mitte kasutajaliidest, siis ei olnud vajadust teste mitmes brauseris käivitada.

Serverisse paigaldati Firefox, samuti oli tarvis ka Xvfb [25] programmi, mis võimaldab käivitada graafilisi rakendusi ilma X-serverita arvutis. Teiseks vaadeldud variandiks olid Google Chrome, kuid selle paigaldamine serverisse, millel ei jooksnud kõige viimane distributsiooni väljalase, oleks osutunud keeruliseks. Samuti oli võimaluseks kasutada mõnda *headless*-tüüpi (ei nõua X-serverit) Selenium-i WebDriver Java API-t implementeerivat brauserit. Tehti katsetused Ghost Driver [26] nimelise implementatsiooniga, probleemiks osutus selle veidi erinev lehtede struktuurist arusaamine, mistõttu ei toiminud sellel testid, mis Firefox-il ja Chrome-l korralikult toimisid. Lisaks oli projekti lehel mainitud, et arendus on seisma jäänud.

Testid olid tehtud süsteemi Maven-projekti alamprojektina ning testide käivitamiseks CI-serveris toimis Surefire [27] plugina abil. Et testide käitusele kuluvat aega vähendada, käivitati

teste paralleelselt. Paralleelse töö saavutamiseks pakkus eelnimetatud plugin kaks võimalust: samaaegselt käitavate protsesside abil (fork) või sama protsessi lõimede abil (thread). Samuti on võimalik valida mida paralleelselt käitatakse (klasse, klasside komplekti e. *suite*, meetodeid või nende mingit kombinatsiooni).

Testid olid koostatud nii, et üks klass vastas ühele testloole ning klassis olevad meetodid oli vaja defineeritud järjekorras käivitada. Täiendavalt olid testid grupeeritud komplektidesse (*suite)* testitava funktsionaalsuse järgi, mis võimaldas testide arvu suurenemisel säilitada ülevaate kogu testide kogumist ning vajadusel üksikuid teste sealt eemaldada. Seega oleks olnud kõige parem lahendus määratud testide komplektidest teada saada testiklassid ja neid siis paralleelselt käitada.

Kasutatud lahenduseks kujunes paralleltöö protsesside (fork) abil ning testide käivitamine komplektide kaupa. Server oli 8-tuumaline, seetõttu kasutati testide jaoks 6 protsessi, et süsteemi mitte üle koormata. Lõimede kasutamine ei osutunud võimalikuks: mitme brauseri samaaegne käigus hoidmine samast protsessist ei olnud töökindel. Piirduti komplektide kaupa paralleelse käivitamisega, sest määratud komplektidest kõigi testiklasside otsimiseks ja siis nende paralleelseks käivitamiseks oleks tulnud kirjutada ise JUnit-i raamistiku testide käivitamise loogika, see ei olnud aga antud projekti skoobis.

3.7 Genereeritud testiraportite võrdlus

Kuna teste koostati palju, siis tekkis vajadus arusaadava testiraporti järele, mis annaks ülevaate kogu testide komplektist ja võimaldaks samal ajal konkreetses testides tekkinud veakohti üles leida.

3.7.1 Jenkins-i testiraport

Kuna teste käivitati Jenkins-ist ning testid koostati väga levinud JUnit-i testraamistiku baasil, siis ei tekkinud raskusi testi tulemuste registreerimise ja kuvamisega Jenkins-is: info testide tulemuste kohta lisandus pärast testide käivitamise lõppu ehituse *(build)* juurde automaatselt. Samuti koostati statistika selle kohta, kui palju teste läbi kukkusid *(failed)* ning läbi kukkunud testid grupeeriti klasside ning funktsioonide kaupa, koos vea põhjusega. Kui funktsioonis tekkis mitu viga (testides kasutati `ErrorCollector` funktsionaalsust), siis kõik tekkinud vead olid

välja toodud. Vigade korral logiti vea põhjustanud koht koodis *stacktrace*, samuti tehti võimaluse korral ekraanipilt brauseri aknast. Kõik see võimaldas üsna lihtsalt leida ja aru saada vea põhjusest. Illustreerivaid ekraanipilte saab näha järgnevas viites: [28].

Jenkins-i poolt genereeritud testraportite nõrkuseks on pea puuduv testide grupeerimise või filtreerimise võimalus, näiteks ei ole võimalik kuvada ainult ühe riigi või funktsionaalsuse kohta käivaid teste. Samuti ei näidata edukalt lõppenud teste. Lisaks saab testiraportit genereerida ainult pärast ehituse *(build)* lõppu, mis on probleemiks, kui kõikide testide käivitamiseks kulunud aega saab lugeda tundides. Hoolimata nendest puudustest, kasutati just seda testiraportit projekti käigus kõige enam.

3.7.2 Maven Surefire Report Plugin

Üks hästi toetatud alternatiiv Jenkins-i testiraportile, on Maven-i testide käivitamiseks mõeldud Surefire plugina raporteerimisvõimalusi. Väljund on üsna sarnane eelnevas punktis kirjeldatule: tuuakse välja testide õnnestumise üldine statistika, siis täpsem statistika Java pakettide *(package)*, klasside ning üksikute testfunktsioonide kaupa. Vaikimisi näidatakse ka õnnestunud teste. Seda, kuidas raport välja näeb, saab vaadata plugina dokumentatsioonis [29].

Üldiselt oli antud aruanne arusaadav ning kui Jenkins-it poleks olnud võimalik kasutada, siis oleks tõenäoliselt kasutatud just seda. Kuid esines olulisi probleeme, vähemalt projektis kasutatud versiooni puhul. Tähtsam neist oli see, et vigade korral polnud *stacktrace*-d loetavad, sest ei sisaldanud reavahetusi. Samuti polnud näha testide käigus logitud andmeid: täpsemalt *stdout* ja *stderr* voogusid. Seega oli ainus teade vea korral erandi *(exception)* üherealine sõnum, mis teeb vea põhjuse väljaselgitamise keerulisemaks ning polnud võimalust lisada täiendavat informatsiooni, näiteks ekraanipildi näol. Ning statistika koha pealt ei arvestatud vahele jäetud ehk `@Ignore` annotatsiooniga märgitud teste õnnestunud testide hulka, mis tähendas täiendavat jõupingutust raporti mõistmisel. Kokkuvõttes esines sel viisil genereeritud raportil mitmeid puudusi, mistõttu ei leidnud see eriti kasutust.

3.7.3 Allure raamistik

Kuna kahel eelpool kirjeldatud testide rapordil esines puudusi, siis otsiti neile paremat alternatiivi. Pikemalt peatuti Allure [23] testiraportite raamistikul, mis koosneb front-endist

tulemuste kuvamiseks ning adapteritest erinevate testraamistike jaoks. Protsess on kahejärguline: adapterite abil kogutakse informatsiooni testide kohta ning kogutud informatsiooni põhjal genereeritakse raport. Lisaks testide staatusele on võimalik testiraporti lisada erinevas formaadis manuseid (näiteks pilte), samuti grupeerida testid funktsionaalsuse või kasutuslugude järgi ning lisada viiteid bugidele veahaldussüsteemis.

Et kasutada seda raamistikku JUnit-it kasutavate testide puhul, on vaja täiendada testide käivitamiseks kasutatavat konfiguratsiooni vastavalt juhendile [30]. Seejärel saab hakata kasutama raamistiku poolt pakutavaid võimalusi. Teine meid huvitav keel oli Python, mille puhul oli adapter olemas pytest [24] raamistiku jaoks ning see ei vajanud konfiguratsiooni – piisas vajaliku abiteegi funktsioonide kasutamisest testides. Lisaks sellele toetab Allure paljusid teisi levinud testraamistikke.

Kuna aga antud raamistiku olemasolu avastati alles pärast suure osa testikoodi valmimist, siis selgusid mõned erinevused selle vahel kuidas testid programmeeritud olid ja kuidas Allure-i poolt eeldatakse. Tähtsaim erinevus oli see, et valmis tehtud testides ja abifunktsioonides oli kasutusel JUnit-i `ErrorCollector` funktsionaalsus, mis pole Allure raamistiku poolt toetatud. See võimaldab testi lõpuni läbi teha ka siis, kui vahepeal mingi kontroll ebaõnnestub ning raporteerib vead alles testi lõppedes. Allure raamistiku puhul aga kuvatakse maksimaalselt üks viga, isegi siis kui neid vigu on testi vältel kogunenud mitmeid. See teeb aga vea põhjuse väljaselgitamise raskemaks ning kahandab märkimisväärselt genereeritud testiraportist saadavat kasu, mistõttu ei võetud seda raamistikku põhilise testikomplekti puhul kasutusse.

3.8 Avastatud tehnilised puudujäägid

- Jenkins-is genereeritud testi tulemuste logi pole eriti kasutajasõbralik ja näitab ainult probleemseid teste ehk õnnestunud testide kohta infot ei salvestata. Automatiseeritud testide arendamise käigus on juhtunud, et mõned testid on “katki”, seda tingituna vigadest praeguses süsteemi versioonist ning parandusi on oodata ainult teatud aja pärast. Ehk oli ette teada, et osasid testitulemusi ei ole vaja arvestada. Samas ei võimaldanud kasutuses olevad raportite genereerimise süsteemid sellega arvestada ning näiteks osad testid vaatluse alt välja jätta.

- Kirjeldatult ehitatud süsteemis jooksevad testid aeglaselt: sekundite asemel võivad testid võtta aega minuteid. Seda osalt seetõttu, et kasutatavad SOAP-teenused ja andmebaasipäringud võivad olla aeglased, kuid nende optimeerimine on töö skoobist väljas. Teisest küljest on läbi brauseri kogu protsessi läbikäimine on juba iseloomult aeglane, sest lisaks testi jaoks vajalikule laetakse ning kuvatakse ka kogu muu kasutajaliides ning sellega seonduv. Sellega kaasneb aga täiendav hulk päringuid, mis kõik aega võtavad.

- Testide jaotamine paralleelse käivitamise korral ei ole efektiivne. Erinevate testide käivitamiseks kuluv aeg varieerub mõnekümnest sekundist kuni mitme minutini. See tähendab seda, et kui kõik testid jaotada juhuslikult näiteks 6 protsessi vahel, siis protsessid, mis jooksutasid lühemaid teste, lõpetavad töö vähem ning osad protsessid jäävad kauemaks tööd tegema. Tulemusena on testikomplekti käivitamiseks kuluv aeg parimast võimalikust pikem.

  Et saavutada parem testide jaotus, peaks teostama sobiliku JUnit'i *test runner*-i ehk klassi, mis juhib seda, kuidas teste käivitatakse. Loogika peaks olema järgnev: moodustatakse üks pikk nimekiri testidest, mida tuleb käivitada. Nimekirjast võetakse järjest ja ühe kaupa teste ning suunatakse neid vabadele protsessidele kuni nimekiri jääb tühjaks. Selline loogika võimaldaks teste efektiivsemalt läbi teha, kui lihtsalt juhuslik (testide arvu järgi) jaotamine protsesside vahel.

- Testide osalise käivitamise võimalused on valitud JUnit raamistikus piiratud. Näiteks kui on soov käivitada kõik mingi riigi testid, välja arvatud pooleliolevate funktsionaalsuste omad ning valitud teste ka paralleelselt käivitada, siis ei ole selleks lihtsat lahendust.

  Maven-i Surefire plugin pakub mõningaid võimalusi käivitatavate testide valikuks [31]. Laiendatud võimalused on lisandunud hiljutistes väljalasetes ning võimaldavad näiteks erinevate regulaaravaldiste abil määrata, milliseid teste käivitada ning milliseid mitte.

- Pikkade stsenaariumite automatiseerimine osutus eeldatust ajamahukamaks. Oma osa mängisid aeglaselt töötavad teenused, mida testides kasutada tuli ja üksikud mahukamad päringud, mida korduvalt välja kutsuda tuli. Nende jõudluse parandamine ei olnud arenduse poolelt vaadates prioriteetne ning osade teenustele tuli antud töö autoril käigus täiendusi teha, et neid oleks võimalik testimise jaoks kasutada. Samuti oli

süsteem üles ehitatud selliselt, et eelnevad kasutajakontoga tehtud tegevused mõjutasid oluliselt järgnevalt võimaldatud tegevusi. Seetõttu oli kindla stsenaariumi läbimiseks vaja väga täpselt läbi mõelda mida ja kuidas tehakse: mõne asja tegemine või tegemata jätmine võis stsenaariumi täielikku läbimise võimatuks muuta. Kui tekkis mingi viga automatiseeritud kasutusloo keskel, siis tuli vea jälile jõudmiseks üldjuhul automatiseeritud kasutuslugu algusest peale käivitada, mis jällegi võttis minuteid aega.

- Töös kirjeldatud ülesehitusega testide töökorras hoidmine kujunes eeldatust ajamahukamaks. Süsteemis olevaid tooted ja ärireeglid muutusid aja jooksul, see tingis vajaduse testide korrigeerimise järele. Kui muutusid pakutavad tooted (koos nende toodetele kvalifitseerumiseks vajaliku krediidiskooriga), siis tuli üle vaadata pakutavaid tooteid kontrolliv testkood, kui ka teha muudatusi ka paljude testlugude testimiseks kirjutatud koodi. Sama tuli teha ka siis kui muudeti ärireegleid või krediidiskoori arvutuskäiku. Ka väike muudatus sellises süsteemi koodis võis avaldada mõju paljudele testidele.

  Näiteks testlugu võib ette näha, et kasutaja suurendab oma krediidikonto limiiti kaks korda. Selleks, et iga kord limiiti suurendada peavad iga kord olema täidetud kindlad tingimused: muuhulgas peab krediidiskoor iga kord olema piisavalt suur, kasutaja peab olema kontot piisavalt aktiivselt kasutanud ning eelmisest limiidi muutmisest peab olema möödunud piisavalt aega. Testiloo vältel kontrolliti erinevates sammudes krediidiskoori, valitud tooteid ning maksti arveid nii, et mööduks õige hulk aega enne järgmist tegevust. Ärireeglite muutumise korral ei olnud testi tulemus positiivne ning teha testis parandusi ning jälgida, et testloo algne mõte kaduma ei läheks. Sellele võis järgneda mitu parandamise-käivitamise üritust, sest muudatus võis kaasa tuua mingi järgneva tingimuse mittetäitmise, millele tuli ka lahendust otsida. Kui tegemist oli muudatustest tingitud veaga, siis tuli see raporteerida.

  Kuna teste oli kirjutatud palju ja muutused süsteemis olid pidevad, siis kulus projekti vältel suhteliselt palju aega, et testid töökorras hoida. Samas võimaldasid eelnevalt kirjutatud testid leida regressioone, ehk siis muudatuse käigus tekkinud vigu funktsionaalsusest, mida otseselt ei muudetud.

4. Testlood

Järgnevalt on välja toodud ühe testitava riigi ühe brändi infosüsteemi jaoks loodud testide kirjeldused. Samuti osade testlugude puhul näidiskood, mis näitab kuidas ühe või teise stsenaariumi automatiseerimine on üldjoontes teostatud.

4.1 Automatiseeritavate testilugude valiku põhimõtted

Järgnevad testlood ei olnud koostatud korraga, pigem jälgiti muudatusi süsteemis ning võeti testimisse mingi kindel arenduses olev funktsionaalsus. Vaadati seda, et automatiseeritud testide koostamine oleks mõistlik lähenemine ehk siis seda, et automatiseerimisega hoitaks aega kokku ning seda, et koostatavad automaattestid ka aja jooksul toimima jääksid.

Näiteks ei peetud mõistlikuks automatiseerida teatud süsteemi poolt pakutavaid ühekordseid lepingumuudatusi, mida tehti lepingutingimuste muutunud seadustega kooskõlla viimiseks. Nende puhul tuli järgida, et leping oleks seotud kindlate laenutoodetega ning tehtud kindlal ajavahemikul, samuti kontrollida kasutajaliideses, et uued pakutavad tingimused on korrektsed. Kui lepingumuudatus on juba tehtud, siis käitub süsteem endist viisi. Selliseid süsteemi ajutisi erinevusi käitumises saab kiiremini ja täpsemalt testida manuaalselt.

Samuti ei automatiseeritud erinevaid vähem tähtsamaid tegevusi kliendi iseteeninduses, nagu näiteks kliendi andmete uuendamine või arvete maksetähtaja muutmine. Need on programmi seisukohalt lihtsad tegevused, mille koodi aja jooksul peaaegu ei muudeta. Samas kuluks automatiseeritud testide kirjutamiseks oluliselt aega, kuna peaks läbi tegema pikemaid protsesse ja automatiseerides kokku puutuma täiendavate vormide ja elementidega; samuti peaks erinevate riikide iseärasustega (kliendi poolt muudetav informatsioon on riigiti erinev). Sellest protsessist saadav kasu on liiga väike, et kaaluda üles täiendavat ajakulu.

4.2 Testlugude kirjeldus

Testid on jaotatud gruppidesse funktsionaalsuse järgi, mida kontrollitakse. Samamoodi on testid struktureeritud ka koodi tasemel, grupeerimine on toimunud kasutades Java *package* mehhanismi. Testide koostamisel alustati lihtsamatest testidest ühe riigi jaoks, mille käigus kirjutati ka vajalikud abifunktsioonid. Järgnevalt tehti sama funktsionaalsuse testid teist riiki

teenindava süsteemi jaoks, üritades nii palju koodi taaskasutada kui võimalik. Üldjuhul oli vaja kohaldada või juurde teha abifunktsioonid, sest süsteemi käitumine on veidi erinev riikide lõikes. Kui teatud funktsionaalsus oli testidega kaetud, liiguti järjest keerulisemate ja pikemate testideni, lähtudes eeltoodud loogikast.

Järgnevalt on välja toodud koostatud testid koos kirjeldusega, kindla riigi ühe brändi jaoks. Suurem osa nendest testidest on kasutusel mitme riigi-brändi kombinatsiooni jaoks, arvestades sealjuures spetsiifiliste erinevustega.

4.2.1 Uue kliendina erinevate laenude taotlemine

Tehakse läbi laenu taotluse protsess ja kontrollitakse, et taotlemine õnnestus. Süsteemis saab taotleda väikelaenu, avada krediidikonto koos väljamaksega või avada krediidikonto ilma algse väljamakseta. Tehakse läbi kõik eeltoodud variandid, mida brändi süsteem võimaldab. Need testid annavad kindluse, et kõige tähtsam funktsionaalsus toimib.

Üldiselt on selline funktsionaalsus vajalik pea kõigis testides, seetõttu on koostatud funktsioonid, kuhu saab ette anda parameetrina kasutaja identifikaatorid ning soovitava laenutoote ja siis testi koodis ühe rea lisamisega läbida terve taotlusprotsess. Kuid koodi paremaks mõistmiseks on parem jagada tegevused väiksemateks sammudeks, seetõttu näide laenu taotlemise testikoodist, mida ei ole eraldiseisvaks funktsiooniks tehtud.

1) Alguses määratakse või genereeritakse kasutaja identifikaatorid, nagu kindla riigi isikukood ja mobiili number, mida siis testides kasutakse. Testi alguses kustutakse selliste identifikaatoritega kasutaja(d) testandmebaasist ära:

   ```
   deleteUser("FI", ssn);
   deleteUserWithMobile("FI", mobileNumber);
   ```

   Seda käivitab läbi POST päringu saatmise teenuse *(service)*, mis kustutab kasutaja testandmebaasist.

2) Järgmisena luuakse kasutajakonto ning täidetakse taotlusvormi testikoodis määratud vaikimisi väärtusega.

```
ApplicationForm_FI applicationForm_fi = TestHelpersFI
        .signUpAndFillUsingDefaultsFI(env, ssn, mobileNumber);
```

See funktsioon avab brauseris testimiseks mõeldus *dummy* sisse logimise lehe, mille abil saab etteantud isikukoodiga kasutajana süsteemi sisse logida. Ehk siis tehakse järgnevat:

a) Leitakse valitud keskkonnale vastav URL, mis brauseris Selenide raamistiku `open` funktsiooni abil avatakse ning oodatakse kuni leht on laetud. Lisa 6 toodud näites on kommenteeritud `init` funktsiooni ja `LoginPage` klassi tööpõhimõtet. Iga tegevus, mida on vaja veebilehel teha, on vormistatud vastava klassi funktsioonina, et mujal poleks vaja meeles pidada veebilehe elementide identifikaatoreid.

```
Selenide.open(env.getBasePath() + "/dummy_login/");
LoginPage loginPage = LoginPage.init();
```

b) Logitakse sisse:

```
DummyBankLoginPage dummyBankLoginPage = loginPage
        .loginUsingDummyBankNoPassword_fi();
ApplicationForm_FI applicationForm_fi = dummyBankLoginPage
        .enterSSN_fi(ssn).submit_fi();
```

c) Ja täidetakse ära vorm vaikimisi väärtustega:

```
applicationForm_fi
        .enterEmail(TEST_EMAIL)
        .selectOccupationType("EMPLOYEE")
        .enterSalary("1000")
```

3) Nüüd oleme loonud kindla profiiliga kasutada, kelle pakutakse vastavalt ärireeglitele kindlaid tooteid. Kontrollime, et krediidireiting ja pakutavad tooted oleksid need mida me eeldame.

```
// Kontrollime, et tootevaliku leht avanes ja krediidireiting oleks õige
CreditLineForm creditLineForm = productSelectForm.getCreditLineForm();
collector.checkThat("Credit score", getCreditScore(ssn), is(...));

// Kontrollime, et kasutajaliideses kuvatud tooted vastaksid eeldatavatele
List<Integer> offeredProducts = creditLineForm.getAvailableProductsAsInts();
collector.checkThat("Available products", offeredProducts, contains(750, 500));
```

4) Järgmisena on vajalik kasutajaliideses taotleda määratud toode, koos kindla väljamaksega nõustuda lepingutingimustega. Seejärel tuleb süsteemis taotlus ära kinnitada.

Selleks on meil kirjutatud abifunktsioon, mis initsialiseerib vajaliku veebilehe elementi kontrolliva klassi; seejärel valib vajaliku toote ja summa, mida soovime kohe välja võtta. Lõpuks kinnitame veebilehel lepingutingimused ja kontrollime, et jõudsime taotlusprotsessi lõppu.

```
CreditLineForm creditLineForm = productSelectForm.getCreditLineForm();
creditLineForm.selectProduct(500);

CL_Slider clSlider = creditLineForm.getCL_Slider();
clSlider.setDrawAmount(100);

productSelectForm.checkoutProduct(ProductSelectForm.Product.CREDIT_LINE);
```

5) Nüüd on vaja taotlus ära kinnitada ja kinnitada väljamakse, päris klientide puhul tegeleb sellega klienditeenindus. Selleks kasutame andmebaasist päringute tegemist ja suhtleme SOAP-liidesega. Näited nende funktsioonide ülesehitusest vaata: Lisa 2 ja Lisa 3.

```
Long customerId = env.db.getCustomerId(country, ssn);

Long caseId = env.db.getLatestCustomerCaseId(customerId);
env.soap.closeCase(country, caseId);

Long drawdownCaseId = env.db.getLatestDrawdownCase(country, ssn);
env.soap.closeCase(country, drawdownCaseId);

collector.checkThat("Case should be approved",
        env.db.getLatestCreditApplicationState(customerId), is("APPROVED"));
```

6) Lõpuks kontrollime, et kasutajaliideses näidataks iseteenindust.

```
TestHelpersFI.login(env, ssn);
AioCockpit aioCockpit = AioCockpit.init();
logout();
```

Eelnev funktsionaalsus on sammude kaupa jaotatud eraldiseisvateks funktsioonideks, mis siis näiteks kinnitavad taotluse või taotlevad määratud suurusega laenu. Ning kuna laenu taotlemist on vaja enamustes testides, siis on sammud kokku pandud üheks suuremaks funktsiooniks, mida testides kutsutakse esile järgnevalt:

```
int product = 500;
int drawAmount = 100;
applyForCL(ssn, mobileNumber, product, drawAmount);
```

Kuna tuleb arvestada erinevate laenu tüüpidega ning eri riikide iseärasustega, siis on selliseid taotlemise funktsioone mitmeid, kuid sisuliselt teevad nad läbi eelpool kirjeldatud protsessi.

4.2.2 Uue kliendina lepingu tingimuste muutmine

Tehakse läbi laenusumma suurendamine koos väljamaksega väikelaenu puhul ning limiidi suurendamine ja vähendamine krediidikonto puhul. Krediidikonto puhul proovitakse ka väljamaksete tegemist. Nende testidega kontrollitakse, et kliendile pakutaks ärireeglites nõutud lisateenuseid ning seda, et põhilised lepingu muutmise funktsioonid toimiksid.

Testides kasutatakse eelmises punktis kirjeldatud taotluse funktsiooni, lisatud on abifunktsioonid kasutajaliidese juhtimiseks ning mõned andmebaasist andmeid lugevad funktsioonid operatsioonide korrektsuse kontrolliks.

Järgnevalt näide ühest lihtsamast sellisest testist:

```
@Test
public void test4_upgrade() {
    login(ssn);

    SelfService selfService = SelfService.init();
    selfService.upgradeCL_To(1000);

    Long customerId = env.db.getCustomerId(country, ssn);
    DBUtils.ProductDTO productDTO = env.db.getCurrentProduct(customerId);
    collector.checkThat("Product principal", productDTO.principal, is(1000));

    logout();
}
```

Põhilise töö teeb ära `upgradeCL_To` funktsioon, mis kasutaliideses läbib limiidi suurendamise protsessi. `$` on Selenide raamistiku poolt implementeeritud funktsioon elementide valimiseks:

```java
public void upgradeCL_To(Integer amount)  {
    openCL_UpgradePopup();

    $("div.ChangeCreditlinePopup.upgrade li label input + span")
            .waitUntil(visible, 4000);
    ElementsCollection amountLabels =
            $$("div.ChangeCreditlinePopup.upgrade li label input + span");
    List<Integer> amounts = getCL_UpgradeDowngradeAmounts(amountLabels);

    if (amounts.indexOf(amount) == -1) {
        throw new Error("List " + amounts + " does not contain: " + amount);
    }

    amountLabels.get(amounts.indexOf(amount)).click();

    // suurenda limiiti
    $("div.ChangeCreditlinePopup.upgrade div.buttons button.confirm").click();

    // nõustu tingimusega
    this.confirmTermsChange();
    SelfService.init();
}
```

Tingimustega nõustumiseks tuleb samamoodi läbida teatud protsess kasutajaliideses, mida tehakse sarnaselt eeltoodud näitele. Nagu näha, on testi koodi üritatud hoida võimalikult lihtsa ja lühikesena ning vajalike tegevuste jaoks vajalik tegevuste jada vormistada eraldi funktsioonina.

4.2.3 Uue kliendi perioodi lõppemine

Ärireeglite kohaselt ei kasutata uue kliendi reegleid pärast teatud tingimuste täitumist. Näiteks kui on möödunud teatud hulk aega laenu taotlemisest või esimesest arvest. Testides tehakse läbi kindla laenu taotlemine, kontrollitakse, et lisateenused oleks pakutud. Seejärel viiakse kasutajakonto erinevatesse seisunditesse, kus talle ei tohiks enam uue kliendi lisateenuseid pakkuda ning kontrollitakse, et eelnevalt pakutud teenuseid poleks enam saadaval. Igas testis kontrollitakse ühte tingimust, mis peaks lisateenused välja lülitama.

Kuna antud testides on üldjuhul läbi teha mingi sündmuste jada, siis kasutatakse siin kliendi konto "ajas nihutamist" ehk uuendatakse kõiki kontoga seotud andmebaasiobjekte, määrates nende loomise, muutmise või muid aegu nii, et need peegeldaksid etteantud arv päevi vana klienti. Ning seda tihti korduvalt. Näiteks võib klient esimese arve saada 3 nädalat pärast lepingu sõlmimist ja maksetähtaeg on 1 nädal pärast arve genereerimist. Et seda olukorda saavutada, tehakse uue kliendina laenutaotlus, nihutatakse klient 3 nädalat ajas tagasi,

genereeritakse arve, nihutatakse veel 1 nädal tagasi ja kontrollitakse, et kliendile ei pakutaks kuni arve tasumiseni lisateenuseid.

Kuna oli palju andmebaasiobjekte, mida tuli uuendada ja vahepeal võis midagi valesti minna, siis tuli siinkohal kasuks transaktsioonide tugi kasutatavas andmebaasiteegis. Lihtsustatult nägi selline „ajas nihutamise“ funktsioon välja järgnev:

```
public void shiftBackInTime(final Long customerId, final Integer days) {
    final List<String> sqlStatements = new ArrayList<String>();
    sqlStatements.addAll( Arrays.asList(
        "update CreditApplication set created = (DATE_SUB(created,INTERVAL :days
day)) where customer_ID = :customerId;",
        "update Contract set created = (DATE_SUB(created,INTERVAL :days day))
where customer_ID = :customerId;"
        // [..]
    ));

    Handle h = getDBI().open();
    h.inTransaction(new TransactionCallback<Void>() {
        @Override
        public Void inTransaction(Handle h, TransactionStatus transactionStatus)
                throws Exception {
            for (String sql : sqlStatements) {
                h.createStatement(sql)
                        .bind("customerId", customerId)
                        .bind("days", days)
                        .execute();
            }
            return null;
        };
    });
    h.close();
}
```

4.2.4 Kliendile pakutavad tooted

Kasutaja sisestatud andmete ning väliste registrite päringute põhjal arvutatakse kasutaja krediidireiting ja selle põhjal pakutakse kasutaja teatud valikut süsteemis olevatest toodetest. Iga toodet pakutakse alatest teatud krediidireitingust ning testimise jaoks moodustatakse vahemikud krediidireitingute kaupa.

Näiteks kui on kaks laenu: 500 eurot ja 700 eurot, mis nõuavad krediidireitingut vastavalt 150 ja 170 punkti, siis moodustatakse 2 vahemikku: [150; 170) ja [170; ∞) ning kontrollitakse, et seal pakutaks nõutud tooteid. Lisaks võib moodustada täiendava vahemiku [0; 150) ning kontrollida, et kasutajale ei pakutaks üldse tooteid. Vahemikke kasutatakse seetõttu, et pole alati võimalik konkreetse skooriga (nt. 150) taotlust teha, et täpselt piiril olevate andmetega

(*boundary value*) testida. Sealjuures eelistati vahemiku alumisele väärtusele lähedasemaid väärtusi ning üritati kasutada võimalikult palju erinevaid krediidireitingut kujundavaid muutujaid.

Eelneva meetodi abil valitakse krediidireitingut kujundavad muutujad nii, et moodustuv reiting oleks nõutud vahemikus. Seda tehakse iga vajaliku vahemiku jaoks. Tulemuseks saadakse testlood, milles on välja toodud sisendid (milliseid andmeid tuleb laenu taotlemisel valida) ja oodatavad väljundid (krediidireiting, pakutavad tooted). Nende loodud testlugude põhjal koostati automatiseeritud testid, pärast mille käivitamist veenduti pakutavate toodete ja muude vaadeldud väljundite õigsuses. Samuti tulid välja vead, kus mingile sisendparameetrile vastav väärtus arvutati valesti.

Kuna krediidireitingut arvutatakse iga riigi jaoks erineval meetodil ning muudetakse tihti ka pakutavaid tooteid, siis selliste testlugude käsitsi haldamine võttis liialt palju ressurssi. Seetõttu loodi töö käigus skript vajalike testilugude ja vastava testkoodi genereerimiseks.

Et selliseid teste oli palju ning vajalikud sisendid ja väljundid olid ette teada, siis testide mõistetavuse huvides tehti abiklass ja funktsioon andmete sisestamiseks ning tulemuste kontrolliks. Lõpptulemusena nägi test välja järgmine:

```
@Test
public void test_10() {
    AffordabilityTestData testData = new AffordabilityTestData()
            .setAge(30)
            .setEducation(SECONDARY)
            .setMaritalStatus(DIVORCED)
            .setResidenceOwnership(RENT)

            .setSalaryRegistry(526)
            .setAppNetIncome(653)
            .setOtherObligationsAppForm(80)
            .setOtherObligationsRegistry(67)

            .setExpectedAppRejected(false)
            .setExpectedCreditScore(...)
            .setExpectedProducts(..., ...)
            .setExpectedTier(...);

    String ssn = TestHelpersLT.getAndPrintNonUsedSSN_LT(env, testData.getAge());

    checkResults(ssn, testData);
}
```

`AffordabilityTestData` oli lihtsalt klass andmete hoidmiseks, sisulise töö tegi ära `checkResults` funktsioon, mis lõi testkonto täitis ära vormi väljad nõutud informatsiooniga,

lisas vajaliku info testimiseks mõeldud registritesse, vormi ära saatmise järel kontrollis krediidireitingut ja pakutud tooteid. Kui mingid tulemused ei vastanud eeldatutele, siis testi lõpuks tuli see välja. Väljavõte `checkResults` funktsioonist on toodud järgnevalt.

```
// [..]
if (testData.getExpectedAppRejected()) {
    collector.checkThat("Application rejected",
            creditLineForm.isApplicationRejected(), is(true));
    TestHelpersLT.logout();
} else {
    collector.checkThat("Application not rejected",
            creditLineForm.isApplicationRejected(), is(false));

    if (testData.getExpectedProducts().size() > 0) {
        collector.checkThat("Selected product",
                creditLineForm.getSelectedProductAsInt(),
                is(testData.getExpectedProducts().get(0)));
    } else {
        collector.checkThat("Selected product",
                creditLineForm.getSelectedProductAsInt(),
                is(nullValue()));
    }
} // [..]
```

Nagu näha, kui me ootame, et taotlus lükataks tagasi, siis ei kontrollita pakutavaid tooteid. Samuti näiteks osade vormiväärtuste valikul võivad lisanduda täiendavad kohustuslikud vormiväljad. Sellise abifunktsiooni puudumisel tuleks selle kõigega testikoodi kirjutamisel või genereerimisel arvestada ehk kirjutada funktsioonide lõikes erinevaid kontrolle ja arvestada erinevate väljade täitmise vajadusega. Kirjeldatud vahekiht võimaldab aga keskenduda ainult olulistele parameetritele.

4.2.5 Täiendav toodete filtreerimine

Kliendile toodete pakkumisel vaadatakse lisaks krediidireitingule ka mõningaid teisi andmeid, näiteks palk ja olemasolevad finantskohustused. Ühe näitena ei pakuta kliendile tooteid, mille kuumakse ületab teatud piirmäära. Testides tuli kontrollida seda, et selline filtreerimine korrektselt töötaks.

Selle jaoks võeti aluseks konkreetse krediidireitinguga testlugu ja muudeti sisendeid (näiteks palk), nii et korrektse arvutuskäigu korral jäetaks mingid tooted kliendile pakkumata. Ilma palga kitsenduseta testist oli teada, et ainult krediidireitingut arvesse võttes pakutakse kliendile

õigeid tooteid. Nüüd kontrolliti, et lisades täiendava kitsenduse liialt madala sissetuleku näol, jätab süsteem osad tooted kliendile pakkumata.

Kuna ka need testid sõltuvad süsteemi sisestatud toodetest, mida tihti muudetakse, siis oli ka siin abiks eelnevalt loodud skript testkoodi genereerimiseks. Nende testide jaoks lisati sinna funktsionaalsus, mille abil valiti välja toodete nimekirjast „lõikekoht“, mis välistas osade toodete pakkumise ning arvutati sellele „lõikekohale“ vastavad muutujate väärtused, näiteks palk. Arvutatud väärtusi kasutati sisendina ning kontrolliti, et süsteemi poolt pakutavad tooted oleksid samad, mis skriptis arvutatud.

4.2.6 Aktiivse lepinguga kliendi lisateenused

Aktiivse lepinguga kliendi lisateenuste testid. Kui klient pole enam uue kliendi staatuses, see tähendab, et tema leping on aktiivne ning sõlmitud rohkem kui ärireeglites ette nähtud aja eest, siis hakkavad kliendile kehtima veidi muudetud reeglid. See tähendab ka muutust süsteemi käitumises, mida omakorda tuleb testida.

Testides tehti läbi lisalaenu taotlemist erinevatel tingimustel ning kontrolliti, et pakutav laenusumma oleks ärireeglitega vastavuses ning et seda pakutaks vaid kindla ajavahemiku sees. Sama põhimõttega testid koostati krediidilimiidi suurendamise ja vähendamise funktsionaalsuse jaoks. Kasutajaliidese poole pealt sai kasutada teiste testide kirjutatud funktsioone lisateenuste kasutamiseks ning süsteemi poolelt samuti juba valmis koodijuppe näiteks arvete genereerimiseks ja maksmiseks.

4.2.7 Krediidikonto lisateenused erinevate kliendigruppide jaoks

Krediidikonto puhul on tavaline, et klient kasutab seda pikema aja vältel, tehes vajadusel väljamakseid ning sooritades tagasimakseid oma äranägemise järgi. Aktiivse kontoga klientidele pakutakse teatud lisateenuseid, näiteks limiidi suurendamine ja vähendamine, kuid seda ettenähtud reeglite kohaselt. Projekti käigus lisati süsteemi funktsionaalsus, mis võimaldas jaotada kliente mingi parameetri abil gruppidesse ning kohaldada igale grupile veidi erinevaid reegleid. Erinevused võisid seisneda limiidi suurendamisel pakutavate toodete osas või selles, kui tihti lubatakse limiiti muuta.

Selle funktsionaalsuse testimiseks tehti läbi tavalisest pikem testilugu, milles taotleti uue kasutajana laenu, genereeriti ja maksti arveid, võeti täiendavalt raha välja, tehti läbi mitu korda limiidi suurendamine (kontrollides sealjuures, et seda pakutakse vaid määratud juhtudel) ning limiidi vähendamine. Test koostati algselt ühe riigi teatud kliendigrupi jaoks ning kohaldati seda erinevate kliendi gruppide ja riikide jaoks.

4.2.8 Aktiivse lepinguga kliendi krediidireitingu arvutamine

Projekti käigus uuendati osaliselt krediidireitingu arvutamise reegleid aktiivse lepinguga klientide jaoks. Muude funktsionaalsuste testimise käigus tulid välja ebakõlad osade muutujate arvutuskäigus, mistõttu tekkis vajadus neid põhjalikumalt testida. Aktiivse lepinguga klientide krediidireitingu arvutamisel võetakse arvesse erinevaid, põhiliselt maksekäitumisega seotud muutujaid. Teste koostades oli oluline, et võimalikult palju muutujate sisend- ja väljundväärtusi oleks kaetud. Näiteks maksti osadel juhtudel arved õigel ajal, mõnedel juhtudel maksetähtajast hiljem ning kontrolliti, et arvutatud krediidireiting oleks nõuetele vastav. Muutujaid oli ligikaudu kümne ringis, igal oli mitu sisendväärtuste vahemikku, mis tuli testidega ära katta.

4.2.9 Kliendile saadetavate meilide kontroll

Laenu taotlemise protsessis ja teatud teiste tingimuste korral saadetakse kliendile automaatselt kas teatud meil või kiri. Näiteks saadetakse kliendile automaatselt arveid, erinevaid meeldetuletuskirju (kui arvega maksmisega on hilinetud) ja teisi olulisi teateid. Projekti vältel oli vaja kontrollida, et vastavaid kirju saadetakse ning tehakse seda õigel ajal, samuti et kirjade sisu ning kirjadele lisatud failid oleksid korrektsed. Selle jaoks koostati eraldi testikomplekt, milles tehti testkonto, viidi see erinevatesse olekutesse ning käivitati kirjade saatmine. Erinevates riikides on kirjade tüübid ja saatmise ajastused erinevad, mis tingis vajaduse testi kohaldamise järgi. Tulemust ehk seda kas kirjad saadetakse õigel ajal ning kirjade sisu ja nendele lisatud dokumente kontrolliti testija poolt manuaalselt. Automatiseeritud osa võimaldas kokku hoida aega, mis oleks muidu kulunud testkonto tegemiseks ning selle vajalikesse olekutesse muutmist.

4.2.10 Infolehtedel näidatavad laenukalkulaatorid

Enne taotluse saab klient veebilehel tutvuda pakutavate toodetega ning teada saada krediidi orienteeruva kogumaksumuse ning igakuise tagasimakse. Seda võimaldavad laenukalkulaatorid, mille toimimist mõjutavad andmed saadakse andmebaasist. Testidega kontrollitakse, et need kalkulaatorid toimiksid ja näitaks korrektseid summasid.

4.2.11 Kliendi käest vajalike dokumentide nõudmine

Laenude väljastamisel on teatud tingimustel vajalik kliendi käest teatud dokumente nõuda, näiteks pangakonto väljavõtet, ning teatud tingimustel pole seda vaja teha. Täpsed nõuded on riigiti erinevad. Testides läbi laenu taotlemise protsess ning kontrolliti andmebaasis ning kasutajaliideses seda, et vajalikel juhtudel küsitaks dokumente enne laenu väljastamist ning teistel juhtudel ei küsitaks.

4.2.12 Tootegruppide valiku korrektsus

Tootegruppide valiku korrektsuse testid. Mõnede riikide puhul on kliendile pakutavad laenud jaotatud erinevatesse gruppidesse. See tähendab seda, et on mitu sama suuruse ja tagasimakseajaga toodet, mille mõned parameetrid mõnevõrra erinevad, näiteks intress. See võimaldab madalama riskiga klientidele pakutakse madalama intressiga laenu. Erinevate lepingumuudatuste tegemisel (näiteks lisalaenu võtmisel või krediidilimiidi suurendamisel), peab uus toode üldjuhul olema samast grupist, mis esialgne. Testidega tehakse läbi erinevaid lepingumuudatusi ja kontrollitakse, et tootegruppidega seotud tingimused oleks täidetud.

4.2.13 Maksed läbi veebiliidese

Mõnedele riikidele suunatud süsteemide puhul on lisatud võimalus makseid teha veebiliidesesse integreeritud välise komponendi abil, mis töötab ka testkeskkonnas. Funktsionaalsuse olemasolul tehakse arvete maksmine läbi ka selle komponendi abil.

4.3 Välise krediidireitingu mooduli vastuvõtutestimine

Algselt testimise alla kuulunud süsteemides arvutati kliendi krediidireiting süsteemis, siis uut riiki teenindava süsteemi jaoks võeti vastu otsus kasutada valitud teenusepakkuja poolt pakutavat välist süsteemi. Kasutusele võetav süsteem pidi arvutama kliendi krediidiskoori ja vastavalt reeglitele tagastama kliendile pakutavad tooted. Selline lahenduse eesmärk oli põhisüsteemi keerukust vähendada ning tagama süsteemi kiirema valmimise.

Süsteemi kasutamine toimus läbi SOAP-liidese, milles seisnes põhiline erinevus “sisemise” krediidiskoori mooduli testimise vahel. Kui viimase jaoks oli tarvis läbida testkasutajana erinevaid stsenaariume brauseri juhtimise ja erinevate teenuste kasutamise läbi, siis nüüd piisas õiges formaadis päringu saatmisest ning vastuse kontrollist.

Välise süsteemi SOAP-liidese loogika oli koostatud sama teenusepakkuja visuaalset programmeerimiskeelt kasutades. Programmile anti sisendina ette nõutud formaadis XML dokument ning väljundina väljastas see samuti XML dokument. Programmi kasutamisel läbi SOAP liidese lisandusid dokumendile ka päised.

Selle mooduli testimisel oli tähtis võimalikult lühikese aja jooksul kontrollida selle toimimist ja tulemuste korrektsust, protsessi täielik automatiseerimine ei olnud tähtis. Testide läbiviimiseks valiti rakendus SoapUI, mis on mõeldud sarnaste liideste testimiseks.

Selle abil koostati testide kogum, mis kattis enamusi süsteemi poolt aktsepteeritavaid sisendeid. Kui mingi sisendi puhul andsid samasse gruppi kuuluvad väärtused sama tulemuse (nt. vanus 30-40 aastat: 10 punkti), siis valiti grupist üks suvaline väärtus. Tulemuseks saadi ligikaudu 10 testi, mis katsid kõik erinevad sisendväärtused.

Täiendavalt loodi testid spetsiifiliste ärireeglite jaoks ja korduvatele klientidele pakutavate laenupakkumiste jaoks.

Sellisele testimisele kulus kokku ligikaudu 2 nädalat, millest esimese jooksul tuli süsteemist aru saada ja põhifunktsionaalsuse jaoks testid koostada. Teise nädala jooksul tehti süsteemi arendaja poolt vajalike muudatusi ja täiendusi (sealhulgas muutus mõnevõrra andmete formaat), mis tuli üle testida ning tekkinud vigadest teada anda. Samuti tuli jälgida seda, et olemasolev funktsionaalsus endiselt korrektselt töötaks.

Kuna testimise käigus tuli ümber käia suhteliselt mahukate XML failidega, mille struktuur tihti muutus (seda nii sisendite kui ka väljundite osas) ning testimisele kulunud aeg pidi olema võimalikult väike (et arendajatele pidevalt ja kiiresti tagasisidet anda), siis oli SoapUI programmi valik testide koostamise ja halduse jaoks mõistlik valik. Saadetavate XML-ide koostamine ja muutmine tekstiredaktoris ei põhjustanud eriti suurt ajakulu. Samuti oli võimalik päringute vastuste korrektsust kontrollida koostades XPath päringuid ja saadud tulemust eeldatuga võrrelda, mida oli võimalik teha korraga kõigi koostatud testide jaoks.

Samaväärsete testide teostamine Java koodis ja kasutades projekti käigus loodud raamistikku oleks tähendanud täiendavat tööd ja ajakulu: oleks olnud vajalik teha malli põhjal XML dokumentide genereerimine ning arvestada sealjuures sellega, et päringute ning vastuste formaat on pidevas muutumises. See oleks lisanud testidele ka ühe täiendava vahekihi: XML-i genereerimise, mis oleks olnud potentsiaalne vigade allikas.

Kuigi loodud testide integratsioon JUnit testraamistikuga on võimalik [32] ning võimaldab koostatud teste koos teiste testidega käivitada, ei rakendatud antud projekti raames seda võimalust.

5. Testide automatiseerimine projekti vaatepunktist

Testide automatiseerimine ei olnud projekti vaates eraldiseisev eesmärk, vaid pigem üks vahenditest, mida kasutati testimisprotsessi paremaks ja efektiivsemaks tegemiseks. Kui lühidalt kokku võtta, siis võimaldas see läbi teha keerukamaid protsesse kiiremini ja põhjalikumalt, kui manuaalne testimine ning, pikemat ajavahemikku silmas pidades, avastada regressioone, mis muidu oleksid jäänud märkamata.

5.1 Muutuvad nõuded

Töö käigus on olnud vaja teste ümber teha, arvestada uute tekkinud nõuetega ja pidevalt teha väiksemaid parandusi, et testid ilma põhjuseta läbi ei kukuks.

Eeldatust tihedamini esines selliseid olukordi, kus testikomplektis olid paljud testid läbikukkunud *(failed)* staatuses. Põhjuseks tavaliselt erinevate asjaolude kokkulangemine, näiteks: arendajad on mingi funktsionaalsuse valmis saanud, saatnud selle testimisse, testimiseks on automatiseeritud eeldatav stsenaarium, mis langeb vastuollu süsteemi tegeliku käitumisega. Seejärel kulub aega vearaportite koostamisele, võimalik et ka nõuete täpsustamisele (mõnel juhul ka muutmisele) ja tõenäoliselt on arendajad mõneks ajaks mõne kõrgema prioriteediga ülesandega hõivatud. Enne kui testid kõik edukalt läbitakse, on vähemalt mõned päevad möödunud. Sarnaseid olukordi on tulnud ette korduvalt.

5.2 Protsess

Projektis oli üks liige testijuhi rollis ja tegeles testide planeerimise ning uue/muutuva funktsionaalsuse manuaalse testimisega. Automatiseerimisele kuulusid testilood, mis olid liiga ajamahukad, et neid manuaalselt testida ning mille puhul oli alust arvata, et testitav funktsionaalsus jääb pikema perioodi vältel samaks.

Kui näiteks mingit uus kasutajaliides oli veel arendusjärgus (vormid ja nendel kasutatud väärtused muutusid tihti), siis tuli juba koostatud teste sageli kohaldada, mis ei ole eriti efektiine ajakasutus. Samas äriloogika lisandumisel või muutumisel oli vajalik muudatused võimalikult kiiresti läbi testida, et arendajatele tagasisidet anda. Seetõttu hoiti planeeritud muudatustel silma peal ja üritati teha võimalikult palju eeltööd ära teha: näiteks kui täienduse testimiseks oli

vajalik täiendava veebiteenuse või uue kasutajaliidese elemendi kontrollimine, siis kirjutati vastav kood enne valmis, kui kogu protsess lõplikult valmis tehti. Siinkohal tuli kasuks erinevate testkeskkondade olemasolu: üldjuhul oli võimalik osa testide funktsionaalsust valmis programmeerida ja katsetada mitu korda päevas uuendatava arenduskeskkonna põhjal. Testimise sai lõpule viia siis, kui muudatused olid jõudnud stabiilsemasse, järgmist väljalaset *(release)* peegeldavasse testkeskkonda.

Üldiselt tuli automatiseerimise seisukohalt jälgida seda, et tehtud muudatusi oleks võimalik koheselt testima hakata ehk eelnevalt ärireeglite kontrollimiseks vajalikud funktsionaalsused raamistikus teostada. Kui seda mitte teha, siis on oht, et osa uut funktsionaalsust ei jõuta enne järgmist väljalaset põhjalikult läbi testida ning jäävad sisse vead.

5.3 Automatiseeritud testidest saadud kasu

Suurem osa vigadest avastati testide koostamise ajal. Regressioone, kus funktsionaalsus katki läks või loogika valeks muutus, esines suhteliselt vähe. Need said lühikese aja jooksul raporteeritud ja arendustiimi poolt parandatud. Eelnevalt kirjutatud testide käivitamisega oli võimalik ka lihtsalt ning kiiresti kontrollida seda, et viga on korrektselt parandatud. Niisiis oli testidest kasu pikemat ajavahemikku silmas pidades: uue funktsionaalsuse arendamise käigus tekkinud vead tulid testide käivitamise käigus välja, ka sellistes kohtades, kust neid manuaalse testimise käigus ei otsitaks. Samuti sai olemasolevat testikomplekti kasutada kontrollimaks, et süsteemi komponentide uuendamine ei teinud midagi katki: manuaalse testimise korral oleks läbi tehtud vaid väikese osa kõikidest võimalikest testidest. Automatiseeritud testide käivitamine ja seejärel tulemuste analüüs on aga tunduvalt efektiivsem, kui piiratud ulatuses manuaalne testimine.

Teisest küljest olid aga näiteks kasutajaliideses tehtavad muudatused sagedased, seega ei olnud mõistlik kasutajaliidese täpset toimimist testide abil kontrollida. Samuti on automatiseeritud testidega raske kontrollida seda, kas vormi elemente näidatakse korrektselt ning et kasutatud tekstid oleksid õiged ning õigesti kuvatud. Automatiseeritud testides tulid kasutajaliidese koha pealt välja ainult tõsisemad vead, mis takistasid mingi sammu läbimist (nt. vormi saatmine ei õnnestunud).

6. Kokkuvõte

Töö eesmärgiks oli kokku panna raamistik töös vaadeldud süsteemi ärireeglite testimiseks, lähtudes üldistest nõuetest ning ärireeglitest formuleerida täpsemad testlood ning koostada automatiseeritud testid testlugude jaoks.

Töö käigus valmis testiraamistik, mille aluseks oli süsteemi erinevate liidestega suhtlev kood, mille põhjale oli koostatud funktsionaalsus erinevate äriprotsessi osade läbimiseks. Raamistiku arenduse käigus loodi ka konfiguratsioon testide käivitamiseks CI-serveris. Raamistiku kasutavate testidega kontrolliti süsteemi käitumise vastavust ärireeglitele, sealjuures tuli testlugude ja testide koostamisel arvestada riigi-spetsiifiliste erisustega. Lisaks loodi täiendavad abivahendid testimise hõlbustamiseks ning prototüübid spetsiifilise funktsionaalsuse testimiseks.

Automatiseeritud testide koostamise käigus leiti vigu, mis raporteeriti arendajatele; testide igapäevase käivitamisega CI-serveris ja raportite analüüsiga leiti regressioone, mis samuti raporteeriti, ennetades nende sattumist süsteemi *live* versiooni. Koostatud testraamistik ja teostatud täiendav abifunktsionaalsus olid testimisprotsessis kasulikeks töövahenditeks uue ja muutunud funktsionaalsuse testimiseks. Seetõttu saab väita, et töös vaadeldud süsteemile automatiseeritud testide koostamine oli projektis kasulik ja andis panuse tarkvara kvaliteedi parandamisse.

Protsess võttis eeldatust mõnevõrra rohkem aega, kuna testide koostamise käigus tuli aru saada erinevatest ärireeglitest ja nende omavahelistest seostest. Lisaks testide koostamisele tuli arvestada ka muutuvate nõuetega ning teste vastavalt kohaldada.

Testimise automatiseerimise koha pealt võib järgmisena ette võtta *backend*-i poolt pakutavad tähtsamad REST-teenused ja läbida ainult neid kasutades (ehk ilma kasutajaliideseta) lihtsamad testlood. Sellega loodaks kiire suitsutesti kompleks, mis *backend*-i tähtsamaid funktsioone väljastpoolt kontrolliks. Samuti võiks erinevaid testimise käigus loodud prototüüpe edasi arendada, lahendades enne seda probleemid, mis nende kasutuselevõttu takistavad.

Kasutatud kirjandus

[1] „Selenium - Web Browser Automation,“ [Võrgumaterjal]. http://www.seleniumhq.org/. [Kasutatud 08 02 2016].

[2] A. Solntsev, „Selenide: concise UI tests in Java,“ [Võrgumaterjal]. http://selenide.org/. [Kasutatud 08 02 2016].

[3] „Selenium - Page Objects,“ [Võrgumaterjal]. https://code.google.com/p/selenium/wiki/PageObjects. [Kasutatud 08 02 2016].

[4] M. Fowler, „PageObject,“ [Võrgumaterjal]. http://martinfowler.com/bliki/PageObject.html. [Kasutatud 08 02 2016].

[5] B. McCallister, „JDBI: DBI, Handles, and SQL Statements,“ [Võrgumaterjal]. http://jdbi.org/dbi_handle_and_statement/. [Kasutatud 08 02 2016].

[6] „wsimport,“ [Võrgumaterjal]. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/wsimport.html. [Kasutatud 11 02 2016].

[7] „Apache CXF -- WSDL to Java,“ [Võrgumaterjal]. http://cxf.apache.org/docs/wsdl-to-java.html. [Kasutatud 12 02 2016].

[8] „maven-jaxb2-plugin,“ [Võrgumaterjal]. https://github.com/highsource/maven-jaxb2-plugin. [Kasutatud 11 02 2016].

[9] reficio, „soap-ws - Java library, based on Spring-WS, that enables handling SOAP on a purely XML level,“ [Võrgumaterjal]. https://github.com/reficio/soap-ws. [Kasutatud 26 02 2016].

[10] „jOOX - The Power of jQuery Applied to W3C DOM,“ [Võrgumaterjal]. https://github.com/jOOQ/jOOX. [Kasutatud 26 02 2016].

[11] Apache Software Foundation, „Apache HttpClient Fluent API,“ [Võrgumaterjal]. https://hc.apache.org/httpcomponents-client-ga/tutorial/html/fluent.html. [Kasutatud 26 02 2016].

[12] G. Niemeyer, „python-constraint - Python module for handling Constraint Solving Problems,“ [Võrgumaterjal]. https://labix.org/python-constraint. [Kasutatud 23 03 2016].

[13] „Template strings - Python 2.7.11 documentation,“ [Võrgumaterjal]. https://docs.python.org/2/library/string.html#template-strings. [Kasutatud 25 03 2016].

[14] „scipy.optimize.bisect - SciPy v0.14.0 Reference Guide,“ [Võrgumaterjal]. http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.bisect.html. [Kasutatud 25 03 2016].

[15] „Requests: HTTP for Humans,“ [Võrgumaterjal]. http://docs.python-requests.org/en/master/. [Kasutatud 28 03 2016].

[16] „Session Objects - Advanced Usage - Requests 2.9.1 documentation,“ [Võrgumaterjal]. http://docs.python-requests.org/en/master/user/advanced/#session-objects. [Kasutatud 28 03 2016].

[17] „Wand - a ctypes-based simple ImageMagick binding for Python,“ [Võrgumaterjal]. http://docs.wand-py.org/en/0.4.2/. [Kasutatud 29 02 2016].

[18] „ImageMagick: Convert, Edit, Or Compose Bitmap Images,“ [Võrgumaterjal]. http://www.imagemagick.org/script/index.php. [Kasutatud 29 03 2016].

[19] J. Cushman, „PDFQuery - A fast and friendly PDF scraping library," [Võrgumaterjal]. https://github.com/jcushman/pdfquery. [Kasutatud 29 03 2016].

[20] „Dilate image - MATLAB imdilate," [Võrgumaterjal]. http://se.mathworks.com/help/images/ref/imdilate.html. [Kasutatud 29 03 2016].

[21] „Contour finding - skimage v0.13dev docs," [Võrgumaterjal]. http://scikit-image.org/docs/dev/auto_examples/edges/plot_contours.html. [Kasutatud 30 03 2016].

[22] „OpenCV: Contours : Getting Started," [Võrgumaterjal]. http://docs.opencv.org/trunk/d4/d73/tutorial_py_contours_begin.html#gsc.tab=0. [Kasutatud 30 03 2016].

[23] „Allure | Test report and framework for writing self-documented tests," [Võrgumaterjal]. http://allure.qatools.ru/. [Kasutatud 07 04 2016].

[24] „pytest: helps you write better programs," [Võrgumaterjal]. http://pytest.org/latest/. [Kasutatud 07 04 2016].

[25] „XVFB," [Võrgumaterjal]. http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml. [Kasutatud 15 02 2016].

[26] „Ghost Driver is an implementation of the Remote WebDriver Wire protocol, using PhantomJS as back-end," [Võrgumaterjal]. https://github.com/detro/ghostdriver. [Kasutatud 28 03 2016].

[27] „Maven Surefire Plugin," [Võrgumaterjal]. https://maven.apache.org/surefire/maven-surefire-plugin/. [Kasutatud 15 02 2016].

[28] N. Wells, „How Jenkins CI parses and displays jUnit output," [Võrgumaterjal]. http://nelsonwells.net/2012/09/how-jenkins-ci-parses-and-displays-junit-output/. [Kasutatud 24 04 2016].

[29] „Maven Surefire Report Plugin - Usage," [Võrgumaterjal]. http://maven.apache.org/surefire/maven-surefire-report-plugin/usage.html. [Kasutatud 24 04 2016].

[30] „JUnit · allure-framework/allure-core Wiki," [Võrgumaterjal]. https://github.com/allure-framework/allure-core/wiki/JUnit. [Kasutatud 25 04 2016].

[31] „Inclusions and Exclusions of Tests - Maven Surefire Plugin," [Võrgumaterjal]. http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html. [Kasutatud 24 04 2016].

[32] „SoapUI: Integrating with JUnit," [Võrgumaterjal]. https://www.soapui.org/test-automation/junit/junit-integration.html. [Kasutatud 09 02 2016].

Lisa 1 – Funktsioonide vaikimisi parameetrid

Näide vaikimisi (default parameters) ja võtmesõna parameetrite (keyword parameters) kasutamisest Python programeerimiskeeles:

```
def fn(param1=False, param2=10, param3="something"):
    print("param1=%s; param2=%s; param3=%s" % (param1, param2, param3))

fn()
# >>> param1=False; param2=10; param3=something
fn(param3="changed")
# >>> param1=False; param2=10; param3=changed
```

Java-s on üks võimalus inimeerida vaikimisi parameetreid kasutades funktsioonide ülelaadimist, mis lahendab osa probleemist: vähendab funktsioonile etteantavate parameetrite arvu. Võtmesõna parameetrite imiteerimiseks võiks kasutada Map andmestruuri või konkreetset ainult muutujate edasiandmiseks mõeldud andmeklassi (builder pattern).

```
public void fn(Boolean param1, Integer param2, String param3) {
    System.out.println(
            String.format("param1=%s; param2=%s; param3=%s",
                    param1, param2, param3)
    );
}

public void fn() {
    fn(false, 10, "something");
}

public void fn(String param3) {
    fn(false, 10, param3);
}


fn();
// param1=false; param2=10; param3=something

fn(false, 10, "changed");
// param1=false; param2=10; param3=changed
```

Lisa 2 – Andmebaasiga suhtleva funktsiooni näide

Andmebaasiga suhtleva funktsiooni koodinäide, antud juhul antakse funktsioonile ette riik ja isikukood ning funktsioon tagastab kliendiga seotud lepingute identifikaatorid.

```
public List<Long> getCustomerContractIds(String country, String ssn) {
    Handle h = getDBI().open();
    String sql = "select co.ID from Contract co\n" +
            " left join Customer cu on co.customer_ID = cu.ID\n" +
            " where cu.identifier = :ssn and cu.country = :country;";

    List<Long> contractIds = h.createQuery(sql)
            .bind("ssn", ssn).bind("country", country)
            .map(LongMapper.FIRST).list();

    h.close();

    return contractIds;
}
```

Lisa 3 – SOAP-liidesega suhtleva funktsiooni näide

SOAP-liidesega suhtleva funktsiooni koodinäide. Antud juhul hakatakse töötlema kindla identifikaatoriga makset: see tähendab jaotakse kliendi poolt makstud summa ära laenu põhiosa tagasimakseks, intresside tagasimakseks ja vajaduse korral ka muude kulude vahel (nt. leppetrahvid).

```
public void allocateBatch(Country country, String batchId) {
    // päritakse SOAP-liideselt meid huvitava funktsiooni definitsioon ..
    Wsdl wsdl = Wsdl.parse(getEndpoint() + "/PaymentService?wsdl");
    SoapBuilder builder = wsdl.binding()
            .localPart("PaymentWSImplServiceSoapBinding").find();

    SoapOperation operation = builder.operation()
            .soapAction("http://www.example.com/payments/allocateBatch")
            .name("allocateBatch").find();

    // .. koostatakse sellele vastav päring koos dummy andmetega
    String requestDummy = builder.buildInputMessage(operation);
    Document document = $(requestDummy).document();

    // muudame genereeritud XML-is vajalikud väljad ära
    $(document)
            .namespace("soapenv", "http://schemas.xmlsoap.org/soap/envelope/")
            .namespace("com", "http://www.example.com/schemas/common")

            .xpath("//com:sfRequest")
            .attr("country", country.name())
            .attr("user", "regression-test-auto")

            .xpath("//com:idReference")
            .attr("id", batchId);

    String request = $(document).content();

    // saadame XML formaadis SOAP päringu
    soapPost(getEndpoint() + "/PaymentService",
            "http://www.example.com/payments/allocateBatch", request);
}
```

Lisa 4 – Äriprotsessi testi kommenteeritud koodinäide

Järgneb ühe äriprotsessi läbiva testi kommenteeritud koodinäide.

```
// Laiendame klassi, milles on riigi-spetsiifilised funktsioonid
public class TestUpsell_Tiers_Months_3_4 extends LV_Tier_TestBase {
    // Testid nimetame nii, et nende eesmärk oleks nimest aru saada:
    // Antud juhul kontrollitakse, et 3 või 4 kuuks sõlmitud
    // laenulepingule oleks võimalik pärast 2 arve tasumist teha
    // lepingumuudatus ja täiendavat raha välja võtta.
    @Test
    public void test2_Upsell_Tier1_To_Tier3_Months_3_4_v2() {
        // Genereerime juhusliku isikukoodi, mis vastab 22 aastat vanale isikule
        // ja mida testandmebaasis ei ole; samuti veel kasutamata telefoninumbri.
        String ssn = TestHelpersLV.getAndPrintNonUsedSSN(env, 22);
        String mobileNumber = TestHelpersLV.getAndPrintNonUsedMobile(env);

        // Taoleme 100EUR laenu 4-ks kuuks: selle jaoks loome konto, täidame vormi
        // nii, et kasutajale pakutaks nõutud tootegrupi tooteid, valime õige
        // toote ja kinnitame laenutaotluse.
        //
        // Protsessi käigus tehakse erinevaid kontrolle, nii et kui midagi
        // valesti läheb, tekib erand (exception) või vähem tähtsamate vigade
        // korral saame testi lõpuks need teada (Kasutused on ErrorCollector).
        applyAsTier1(ssn, mobileNumber, 100, 4);

        // Genereerime ja maksame vajalikul hulgal arveid,
        // parameetritest aitab aru saada IDE.
        generateAndPayIL_Invoices(ssn,
                // mitu päeva tähtajast varem/hiljem arveid maksta
                Arrays.asList(10, -12),
                // kas pärast makset peaks lepingu muudatus võimalik olema
                Arrays.asList(false, true),
                // mitu senti arvel nõutud summast rohkem/vähem maksta
                Arrays.asList(300, 0)
        );

        // Taotleme lisalaenu 200 EUR ning kontrollime, et laenu andmise aluseks
        // olev krediidireiting oleks korrektne. Jällegi tehakse protsessi käigus
        // erinevaid kontrolle, seetõttu saame vigadest teada.
        Integer expectedCreditScore = ...;
        applyForUpsell(ssn, expectedCreditScore, null, 200);

        // Kontrollime andmebaasist, et pärast täiendava laenu taotlemist oleks
        // kliendi laenutoote grupp korrektne.
        collector.checkThat("Upsell product group",
                env.db.getCustomerProductGroupName("LV", ssn), is(...));

        // Kustutame kasutaja ainult siis, kui ei testi käigus ei tekkinud
        // erandit (exception), nii saame vajadusel andmebaasist kontoga seotud
        // andmeid kontrollida.
        deleteUser(ssn, mobileNumber);
    }
}
```

Lisa 5 – Tootevaliku testi näide

Tootevaliku testi kommenteeritud koodinäide.

```
@Test
public void test_PS_375() {
    // Registeerib konto ja täidab vormi vaikimisi väärtustega
    ApplicationForm applicationForm = signUpAndFillUsingDefaults();

    // Kirjutame üle osad vaikimisi väärtused: mõned võivad omada mõju
    // pakutavatele toodetele, mõned on lisatud kontrollimaks, et vajalikud
    // kasutajaliidese elemendid oleksid olemas ja töötaksid.
    applicationForm
            .selectEducation(ApplicationForm.Education.UNIVERSITY)
            .selectOccupation("PART_TIME")
            .selectOccupationType("OFFICE_CLERK")
            .selectEmploymentDuration("YEARS_3_5")
            .enterNetIncome("1000")
            .enterTotalMonthlyObligations("100")

            .setLeasing(ApplicationForm.LoanSelect.YES);

    ProductSelectForm productSelectForm = applicationForm.submit();

    try {
        // Kontrollime, et spetsifikatsioonis olevad tingimused oleksid
        // täidetud.
        collector.checkThat("Credit score", getCreditScore(ssn), is(...));

        checkProductForm(collector, productSelectForm)
                .check_CL_DisplayedByDefault()

                .check_IL_And_CL_Offered()

                .check_CL_OfferedProducts(1500, 1000, 500)

                .check_IL_MaxAmount(1500)
                .check_IL_MaxMaturityOfSelectedAmount(36)
                .check_IL_MaxProduct_MonthlyInterest(...);
    } finally {
        page(Sidebar.class).logout();
    }
}
```

Lisa 6 – Page Objects muster

Kasutajaliidese elementide kontrollimiseks kasutati Page Objects [4] mustrit, lisaks tuli lehe laadimise vigade puhul ekraanipildi saamiseks pilte tegev funktsioon ise välja kutsuda. Järgneb näide eelpool mainitud mustri realisatsiooni kohta.

```
public class LoginPage {
    private LoginPage() {
    }

    public static void checkLoaded() {
        try {
            // Ootame maksimaalselt 25 sekundit, et meie poolt oodatava lehe
            // element tuleks nähtavale. Makstimaalselt nii kaua, selleks et
            // vältida juhuslikke probleeme testserveri jõudlusega.
            $("a.register").waitUntil(visible, 25000);
        } catch (UIAssertionError e) {
            // Kui elementi ei leitud, teeme brauseris olevast lehest pilti ja
            // alles seejärel anname erandi.
            TestHelpers.screenshot();
            throw e;
        }
    }

    public static LoginPage init() {
        checkLoaded();
        return new LoginPage();
    }
    // [..]

    public DummyBankLoginPage loginUsingDummyBank() {
        $("li.dummy_no_password").click();
        // Ootame kuni uus leht/element on laetud ning tagastame selle
        // juhtimist implementeeriva klassi.
        return DummyBankLoginPage.init();
    }
    // [..]
}
```

Funktsiooni `loginUsingDummyBank` väljakutsumisel vajutatakse brauseris määratud elemendile ning initsialiseeritakse järgmist lehte või elementi juhtida võimaldav klass, meie näites:

```
public class DummyBankLoginPage {
    // [..]
    public DummyBankLoginPage enterSSN(String ssn) {
        $(By.xpath("//input[@name='CSSN']")).setValue(ssn);
        return this;
    }

    public ApplicationForm submit() {
        $(By.xpath("//input[@type='submit']")).click();
        return ApplicationForm.init();
    }
    // [..]
}
```