TARTU ÜLIKOOL
Arvutiteaduse instituut
Infotehnoloogia mitteinformaatikutele õppekava

Janne Sokk

Geeniandmete infosüsteemi testimine Cypress raamistikus

Magistritöö (15 EAP)

Juhendaja: Sulev Reisberg, PhD

Tartu 2021

Geeniandmete infosüsteemi testimine Cypress raamistikus

Lühikokkuvõte:

Töös antakse ülevaade arendamisel olevast riiklikust geeniandmete infosüsteemist ja tarkvara testimisest. Esitatakse plaan geeniandmete infosüsteemi testimise korraldamiseks, luuakse selle alusel X-tee liidese automaattestid manuaalse testimise vähendamiseks ja integreeritakse GitLab arenduskeskkonda. Töös kirjeldatakse ja kasutatakse Cypress raamistikku. Lisaks sisaldab töö analüüsi tehtud tööst ning arutelu edasiste tegevuste üle.

Võtmesõnad:

Geeniandmete infosüsteem, Cypress raamistik, automaattestimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Testing Genetic Data Information System in Cypress Framework

Abstract:

In the work, an overview is provided of the state genetic data information system and software testing being developed. A plan is submitted to organize the testing of the genetic data information system. On the basis of the latter, automated tests of X-tee to lessen manual testing will be created, and all of it will be integrated with the GitLab environment. The framework of Cypress is used in the work. Moreover, the work includes an analysis of the work done and a discussion about future actions.

Keywords:

Genetic data information system, Cypress framework, automated testing

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus 5
2. Mõisted ja terminid 7
3. Taust 8
   - 3.1. Geeniandmete infosüsteemi tutvustus 8
     - 3.1.1. Paiknemine personaalmeditsiini rakendamise protsessis 8
     - 3.1.2. Geeniandmete infosüsteemi arhitektuur 9
     - 3.1.3. Geeniandmete kogumine 11
     - 3.1.4. Geneetilise testi tellimine 11
     - 3.1.5. Geneetilise testi tegemine 12
     - 3.1.6. Geneetilise testi kasutamine 13
     - 3.1.7. Geeniandmete infosüsteemi testimise vajadus 13
   - 3.2. Tarkvara testimise olulisus 14
   - 3.3. Tarkvara testimine detailsete nõuete puudumisel 14
   - 3.4. Automaattestimine 15
     - 3.4.1. Milleks automatiseerida? 15
     - 3.4.2. Automaattestide liigitus 16
   - 3.5. Testimissüsteem GAISis enne käesolevat tööd 18
   - 3.6. Käesoleva töö eesmärk 19
4. GAISi testimise planeerimine 20
   - 4.1. Üldise skoobi määramine 20
   - 4.2. Testimise liikide planeerimine 20
   - 4.3. Testiideed ning käesoleva töö skoobi määramine 21
   - 4.4. Cypress raamistik testimise tööriistana 25
5. Automaattestid Cypress raamistikus 27
   - 5.1. Testkeskkonna struktuur 27
   - 5.2. Loodud automaattestid 27
   - 5.3. Automaattestide käivitamine 29
   - 5.4. Automaattestide integreerimine 31
6. Arutelu 33
7. Kokkuvõte 36

8. Viidatud kirjandus 37
Lisad 39
I Deklaratsiooni kinnitamise testilood 39
II Litsents 41

1. Sissejuhatus

Paljud haigused on teatud geneetilise eelsoodumusega, kuid kuni praeguse hetkeni ei tegele Eesti arstid igapäevaselt nende haiguste geneetilise testimisega. Seda aga tahetakse muuta, kuna loodud on riiklik Geenivaramu biopank, kuhu on geeniandmeid kogutud 200 000 elaniku kohta [1]. Soovitakse laiapõhjalist geneetiliste testide kasutamist Eestis, mis peab olema kompleksne ning koosnema sellistest komponentidest nagu nõustamine enne ja pärast testimist, geneetiline testimine ning testi tulemuste alusel ennetus ja ravi [2]. Käesolev töö keskendub just teisele komponendile, milleks on geneetiline testimine – selle tehniline teostus ja tõlgendamine.

Kui Geenivaramu on teaduskasutuseks mõeldud andmekogu, siis kliiniliseks kasutuseks on GenMed projektis[1] loomisel uus andmekogu geeniandmete infosüsteem (GAIS), milleks on vaja infotehnoloogilist lahendust (tarkvara). Selleks, et tagada kvaliteetne toode, on iga tarkvara arendamise elutsüklis oluline roll testimisel, kuna rike tarkvaras maksab organisatsioonidele nii aega kui ka raha ning lisaks võib kahju teha mainele [3], terviseandmete puhul võib see tekitada otsest kahju ka patsiendile. Kuna automatiseeritud protsessid nõuavad vähem raha, mis on IT sektori võtmetegureid [4], siis on tarkvara automaattestimine tänapäeval juba väga levinud ning ootused tarkvara testijale kui automaattestide loojale kasvanud. Lisaks on viimastel aastatel suurenenud erinevate automatiseerimise tööriistade kasutamine, nende valik on kasvanud ning need muutuvad aina paremaks. World Quality Report 2020-21 [5] toob välja, et automaattestimise juures on murekohtadeks oskus valida sobivat tehnoloogiat (rakendust, raamistikku) ja testimismetoodikat ning agiilsetes meeskondades puuduvad vajalikud testimisoskused.

Töö kirjutamise hetkel ei ole GAISi arendamise projektis eraldi testijat, kuid kuna tarkvara arendamine toimub kiires tempos, siis on vaja testimisega järele jõuda. Seega on vaja luua teste juba olemasolevatele funktsionaalsustele ning samas on oluline kohe testida ka uusi osasid. Seetõttu osutub ülioluliseks välja selgitada, kuidas testimisega alustada – millised on kõige olulisemad osad mida testida ning mis järjekorras seda tegema peaks. Sellest tulenevalt on käesoleva töö eesmärgiks GAISi testimise planeerimine võttes arvesse nii riske kui ka asjaolu, et tegemist on meditsiinisüsteemis kasutatava tarkvaraga. Vastavalt selle tulemusele

[1] https://sisu.ut.ee/genmed/

alustatakse kõige olulisemate testide loomist. Lisaks on eesmärk vähendada manuaalset testimist.

Töö põhiosa on jagatud neljaks peatükiks. Esimeses peatükis kirjeldatakse GAISi arhitektuuri ja hetkeolukorda ning antakse ülevaade testimisest. Teine ja kolmas peatükk on töö praktilised osad, kus teises peatükis planeeritakse testimist, täpsustatakse skoop ja valitakse testimistööriistad ning kolmandas peatükis antakse ülevaade valminud automaattestidest ja nende käivitamisest. Viimases peatükis arutletakse edasiste tegevuste üle ja analüüsitakse tehtud tööd ning töö lõppeb kokkuvõttega.

2. Mõisted ja terminid

GAIS	Geeniandmete infosüsteem
JavaScript	Interpreteeritav objektorienteeritud programmeerimiskeel, mida kasutatakse peamiselt dünaamilise veebilehtede loomiseks
Kasutajaliides	Ühenduslüli programmi ja kasutaja vahel
Testiidee	Lühike kirjeldus, mida peaks/võiks süsteemis testida
Testilugu	Detailne kirjeldus, kuidas kontrollida süsteemi vastavust nõuetele
Testikomplekt	Testilugude kogum
Rakendusliides	Võimaldab luua ühenduse erinevate programmide vahel
VCF	*Variant Call Format*, geeniandmete peamine failiformaat
X-tee	Infosüsteemide andmevahetuskiht, mis võimaldab turvalist internetipõhist andmevahetust Eesti riigiasutuste ja erasektorite vahel

3. Taust

Selles peatükis antakse ülevaade geeniandmete infosüsteemist ja käsitletakse testimise olulisust. Selgitatakse geeniandmete infosüsteemi vajalikkust ja kirjeldatakse süsteemi arhitektuuri ning peamisi protsesse. Lisaks on ära toodud testimise olulisemad aspektid ning selle roll geeniandmete infosüsteemis. Peatüki lõpus sõnastatakse ka käesoleva töö konkreetsem eesmärk.

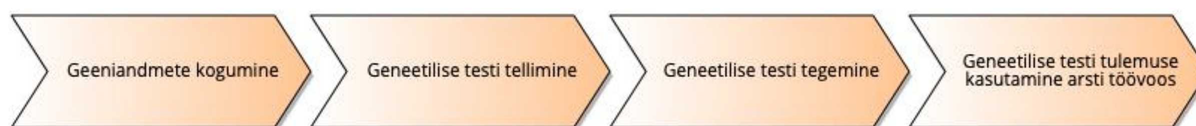
3.1. Geeniandmete infosüsteemi tutvustus

Geeniandmete infosüsteemist ülevaate saamiseks tutvuti projektimeeskonna poolt loodud materjalidega ning osaleti vastavates aruteludes.

3.1.1. Paiknemine personaalmeditsiini rakendamise protsessis

Personaalmeditsiin toimib põhimõttel, et igale inimesele leitakse talle sobiv ennetus- ja/või raviplaan. Tervise Arengu Instituudi andmetel [6] analüüsitakse selleks eelkõige indiviidi tervise-, aga ka keskkonnaandmeid ja käitumisharjumusi. Terviseandmete puhul on oluline just geneetiline eelsoodumus. Kuna üheks põhimõtteks on prognoositavus, siis on oluline kasutada IT-lahendusi, mille abil on võimalik suurte andmehulkade alusel ennustada haigusriske, ravi efektiivsust ja sobivust ning ka optimaalseid koguseid ja kõrvaltoimeid. Selliseid geeniandmeid analüüsivaid tarkvaralisi mudeleid/programme nimetatakse geneetilisteks riskimudeliteks.

Personaalmeditsiini rutiinseks rakendamiseks ja geneetilise info kasutamine ehk geenitestide teostamine eeldab nii keskse geeniandmete infosüsteemi (GAIS) loomist kui ka selle kasutuselevõttu (geeniandmete kogumine, geenitestide tegemiseks kasutamine) meditsiinilaborite poolt. Üldine testide tellimise ja tegemise protsess peaks olema võimalikult sarnane muudele meditsiinilabori protsessidele (joonis 1).

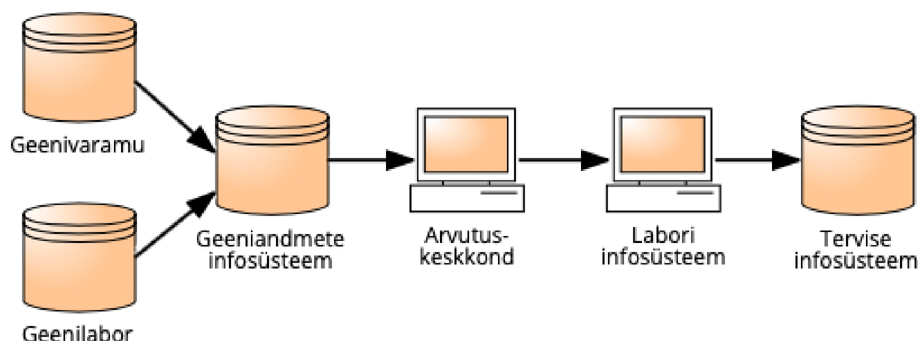


Joonis 1. Geeniandmete infosüsteemi üldine protsessi väärtuste ahel

Käesolev töö keskendub geeniandmete kogumise ning geneetilise testi tegemise protsessidele. Teisi protsesse töös ei käsitleta, kuna nii geneetilise testi tellimise kui ka tulemuste kasutamise protsesse GAISi projekt ei muuda.

3.1.2. Geeniandmete infosüsteemi arhitektuur

GAISi loomise üldeesmärgiks on geeniinfo turvaline edastamine arstideni nii, et see ei tekitaks arstidele lisakoormust, vaid toetaks nende tööd. Selleks on vaja saada patsiendi geneetilised andmed DNA laboritest ja/või geenivaramust, peab olema võimalik kasutada patsiendi geneetiliste andmete põhjal loodud riskimudeleid ning lisaks peab olema võimalik selle põhjal koostada riskihindamise raport juba olemasolevasse tervise infosüsteemi, mis tagab ligipääsu nii arstidele kui ka patsientidele. Kogu infosüsteemi arhitektuuri kirjeldab joonis 2.



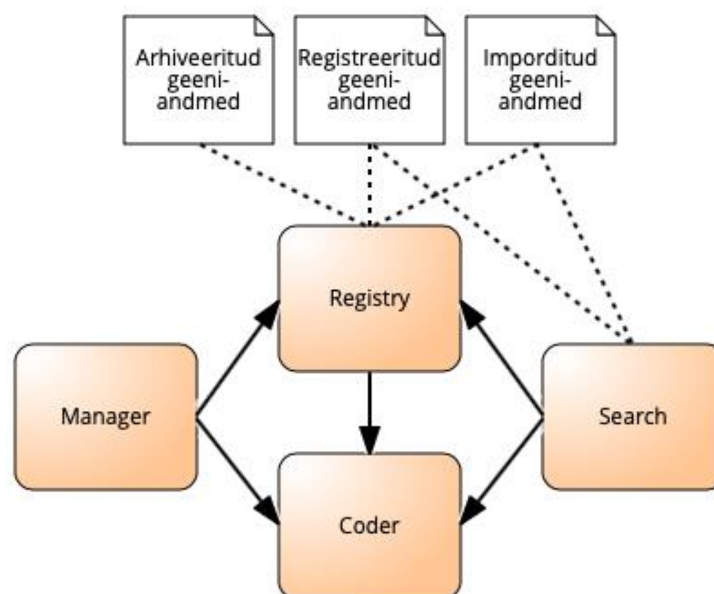
Joonis 2. Infosüsteemi üldine arhitektuur

Infrastruktuur peab tagama järgneva: rakendusliidese kindlate DNA positsioonide pärimiseks ning tarkvara, mis võimaldab riskimudeli kasutamist indiviidi kohta, et kuvada tema riskiraport, mille alusel koostatakse meditsiiniline dokument. Infrastruktuuri praegu arendamisel olevad osad on geneetiliste andmete haldamise infosüsteem (edaspidi geeniandmete infosüsteem või selle lühend GAIS) ning riskimudelite haldamise ja arvutamise infosüsteem (edaspidi arvutuskeskkond), ülejäänud osad on juba olemas.

Kuigi geeniandmeid juba kogutakse, on endiselt puudu riiklik infosüsteem nende haldamiseks. GAISi funktsioon ongi nende andmete haldamine ja kasutamise võimaldamine suures mahus. Andmeid hakatakse sinna saatma geeniandmete pakkujate poolt ning rakendus hakkab monitoorima nende kasutamist. GAISi põhifunktsioonid on geeniandmete importimine, imporditud andmete kuvamine, andmete otsing, isikukoodi lisamine ja eemaldamine, andmete arhiveerimine, andmete uuesti aktiivseks muutmine, andmetele märkmete lisamine. GAIS on omakorda jagatud neljaks komponendiks, kus igal komponendil on oma eesmärk. Sellise jaotamisega vähendatakse ka turvariske, kuna ühe komponendi kompromiteerimine ei võimaldaks kahjustada kogu süsteemi. GAISi komponendid on järgmised (joonis 3):

1) *manager* – peakasutajale mõeldud komponent (kasutajaliides), mille kaudu on võimalik süsteemile käske anda ning seda jälgida;

2) *registry* – sisemine komponent, mis tegeleb saabunud geeniandmete registreerimise, importimise/kustutamise ja arhiveerimisega. Lisaks on sellel komponendil andmefailide liigutamise/salvestamise (ehk lugemise-kirjutamise) õigused;
3) *search* – GAISi X-tee teenuseid pakkuv komponent, mis tegeleb geeniandmete kasutamise deklaratsioonidega (täpsemalt kirjeldatud ptk 3.1.5), isiku andmete otsimise/allalaadimisega, samuti geeniandmete edastamisega *registry* komponenti. Lisaks saab see komponent lugeda nii registreeritud kui ka imporditud geeniandmete andmefaile;
4) *coder* – eelkõige turvakaalutlustel selgelt eristatav sisemine komponent, mis haldab seoseid isikukoodi ja geeniandmete vahel. Lisaks salvestatakse siin ka isiku andmekasutuse keeld/luba.



Joonis 3. GAISi sisemine arhitektuur. Noole suund näitab, millise komponendi poole saab pöörduda.

Kõik kolm joonisel märgitud andmefaili (arhiveeritud, registreeritud ja imporditud geeniandmed) on VCF (*Variant Call Format)* formaadis, mis on geeniandmete peamine standard ning need peavad toetama suuri andmemahte ja ka kiiret andmete lugemist.

Arvutuskeskkond võimaldab hinnata indiviidipõhiselt geneetilisi riske, kasutades arvutuslikke mudeleid. Andmed selleks saadakse GAISi rakendusest kasutades X-tee protokolli. Arvutuskeskkonna põhifunktsioonid on geeniandmete kasutamise deklaratsiooni koostamine ja esitamine, mudelite filtreerimine nime ja/või staatuste järgi, mudelite staatuste, kokkuvõtte

ja andmekasutusdeklaratsiooni kuvamine, uute mudelite lisamine ning mudeli testimine konkreetse inimese isikukoodi kasutades.

3.1.3. Geeniandmete kogumine

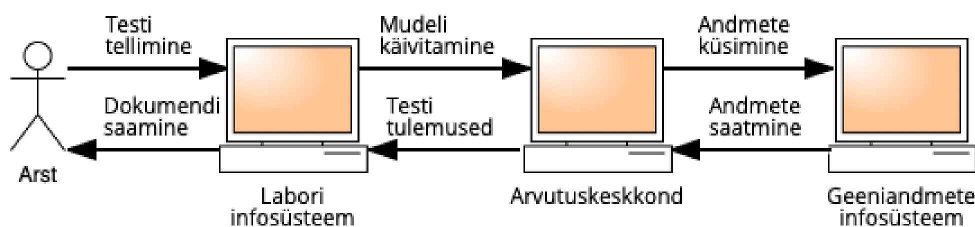
Väga oluliseks GAISi protsessiks on geeniandmete kogumine geeniandmete ühtsesse infosüsteemi (joonis 4).

Joonis 4. Geeniandmete kogumise üldine protsess

Geeniandmete kogumine algab geeniandmete pakkuja poolse ettevalmistamisega, kus peab teadma, milliseid geeniandmeid võib ja saab edastada. Lisaks on kokkulepitud failiformaat (VCF) ning selle sisu peab vastama GAISi nõuetele, mis võimaldab andmed kopeerida ettemääratud import kausta, kust geeniandmete infosüsteem need sisse laadib. Ühes failis hoitakse ühe isiku geeniandmeid. Seejärel kontrollib infosüsteemi peakasutaja imporditavaid andmekomplekte (formaati ja ulatust) ning kas impordib esitatud andmekomplektid andmebaasi või vigaste andmete korral need kustutab. Andmete haldamiseks kasutatakse relatsioonilist andmebaasi PostgreSQL.

3.1.4. Geneetilise testi tellimine

Geneetilise testi tellimise üldine protsess on seotud andmete kättesaamisega riiklikust geeniandmete infosüsteemist ning nendega arvutuste tegemisega (joonis 5).



Joonis 5. Geneetilise testi teostamise üldine protsess ja komponendid

Põhilisteks osapoolteks on keskne geeniandmete infosüsteem ja labor, kellel on eelnevalt olemas labori infosüsteem ning lisaks peab paigaldama arvutuskeskkonna rakenduse. Protsess algab arstipoolse testi tellimisega, mis jõuab labori infosüsteemi. Tellimuse saabumisel käivitab labori töötaja riskimudeli. Mudel vajab arvutuste tegemiseks geeniandmeid, mis päritakse GAISist. Pärast arvutuste tegemist tagastab arvutuskeskkond tulemused, mille alusel

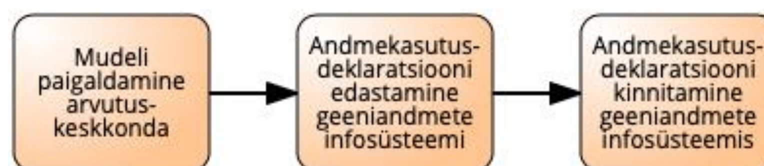
koostab labori töötaja meditsiinilise dokumendi, mis lõpuks väljastatakse testi tellijale (arstile) ning saadetakse tervise infosüsteemi. Arvutuskeskkonda on selles protsessis vaja turvalisuse tagamiseks – kuna geeniandmed on erakordselt tundlikud, on turvalise arvutuskeskkonna kasutamine GAISi andmete kasutamiseks kohustuslik ja see väldib geenialgandmete lekkimist välistesse süsteemidesse.

3.1.5. Geneetilise testi tegemine

Selleks, et geneetilist testi teostada, on vaja teatavat ettevalmistust (joonis 6). Esiteks on testi(de) teostamiseks vajamineva mudeli kasutamiseks vaja laboril eelnevalt paigaldada enda süsteemi arvutuskeskkond, samuti on vaja testi(de) tegemiseks vajaminevaid riskimudeleid. Kui labor otsustab ükskord testi pakkumise lõpetada, tuleb riskimudelid kasutusest eemaldada/kustutada.

Joonis 6. Tegevused geneetilise testi teostamiseks

Nimetatud moodulite paigaldamisest ei piisa veel geneetilise testi teostamiseks, vaid enne mudeli kasutuselevõttu tuleb selle kasutamiseks saada GAISi peakasutajalt luba (joonis 7). Kuna geeniandmete näol on tegemist ülimalt delikaatse infoga, siis tuleb kontrollida, miks ja kuidas neid kasutatakse. Selleks tuleb esitada GAISile andmekasutusdeklaratsioon, kus on detailselt kirjas, kes ja mille jaoks ning milliseid geeniandmeid GAISist soovitakse küsida.



Joonis 7. Riskimudeli kasutusele võtmise protsess

Geeniandmete juurdepääsu taotlemine algab laboripoolse geneetilise riskimudeli ja geeniandmete kasutuse deklaratsiooni sisestamisega arvutuskeskkonda, misjärel see edastatakse GAISi ning taotlemine lõpeb GAISi peakasutaja poolt selle kinnitamise või tagasilükkamisega. Alles andmekasutusdeklaratsiooni kinnitamise järel on laboril võimalik teostada GAISist vastavat isikupõhist geeniandmete päringut. Kui riskimudeli kasutamine on lõpetatud (labor ei paku enam sellist testi), siis vastav riskimudel koos selle deklaratsiooniga eemaldatakse kasutusest/arhiveeritakse.

3.1.6. Geneetilise testi kasutamine

Pärast geneetilise testi teostamist väljastatakse tulemus testi tellijale (arstile). Konkreetne protsess on töö kirjutamise hetkel veel täpselt määratlemata, kuid eelduste kohaselt saadab arvutuskeskkond tulemused labori infosüsteemi, misjärel need edastatakse tervise infosüsteemi, kus neile on ligipääs autoriseeritud osapooltel. See võimaldab mugavalt tulemusi arstil oma töös arvesse võtta nii ravimite soovituste tegemisel kui ka ravi määramisel.

3.1.7. Geeniandmete infosüsteemi testimise vajadus

Testimisel on testitava tarkvara olemus määrava tähtsusega – nt e-kaubanduse rakendust testitakse erinevalt võrreldes ohutuskriitilise süsteemiga. GAISi puhul on tegemist personaalmeditsiini rakendamiseks väga olulise komponendiga. Süsteemi funktsionaalsus on võrdlemisi keerukas, selles toimub isikuandmete töötlemine ning sellest saab riiklik andmekogu. Lisaks peab süsteemi kvaliteet olema väga kõrge ning see peab olema usaldusväärne selleks, et tagada võimalikult õige riskiraport arstile, mis võimaldaks tal teha õigeid otsuseid inimese raviplaani koostamiseks. Vead süsteemis võivad pidurdada arsti tööd ning aeglustada ka muid protsesse, mis võib lõpuks patsiendile kahju teha.

Kuigi GAIS ei liigitu tarkvaraliseks meditsiiniseadmeks, kuna see ei anna geeniandmetele täiendavat interpretatsiooni ega anna ka arstile soovitusi, võib selle arendamisel siiski arvestada meditsiinilise tarkvara loomise ja testimise nõudeid. Eestis peavad riiklikud andmekogud vastama kas ISKE (infosüsteemide kolmeastmeline etalonturbe süsteem) nõuetele või ISO27001 infoturbesüsteemi standardile [7], kus eesmärgiks on tagada piisava tasemega turvalisus infosüsteemides töödeldavatele andmetele. ISO27001 standard põhineb riskihaldusel ning keskendub riskide maandamisele. Mõlemas dokumendis on suunised ka testimisele, kuid kuna GAISi arendamisel ei ole töö kirjutamise hetkel nimetatud standarditest rangelt lähtutud, ei juhinduta neist ka käesoleva töö testide planeerimisel.

Käesolevas töös on lähtutud tarkvaraliste meditsiiniseadmete standardist IEC 62304:2006 [8], mille alusel tuleb meditsiiniseadmete tarkvara elutsükli protsessidesse kindlasti planeerida ka testimine ja seda nii ühiku, integratsiooni kui ka süsteemi tasemetel (vt ptk 3.4.2). Kindlasti tuleb rõhku panna just regressiooni testimisele (samuti ptk 3.4.2), mille eesmärk on tagada kontroll ka juba varasemalt süsteemi integreeritud osade toimimise üle. Integratsiooni testimise puhul tuuakse standardis välja, et kuigi kõiki tarkvara osasid tuleb testida, tuleb detailsemalt ja põhjalikumalt testida just turvalisust mõjutavaid osasid.

3.2. Tarkvara testimise olulisus

Tarkvara kvaliteet on mõiste, mida on raske üheselt defineerida. Adzic jt [9] kirjeldavad, et olenevalt huvigrupist võidakse seda näha erinevalt. Näiteks tarkvara kasutajad vaatavad enamasti selle kiirust ja kasutatavust, arendajad hindavad koodi struktuuri ning äripoole inimesed hindavad tarkvara tõhusust finantsiliselt. Seega on erinevad perspektiivid, mis võivad viia erimeelsusteni. Lisaks on veel vea olulisuse hindamise erinevused, kus üks osapool võib teatud viga pidada kriitiliseks, kuid teise osapoole jaoks ei kategoriseeru see üldse veakski. Sel põhjusel võib juhtuda näiteks olukord, kus kasutajale kriitiline viga võib tarkvaras olla mitu kuud. See võib viia arusaamatuseni, kus erinevad osapooled üksteist süüdistavad, kuna puudub ülevaade suurest pildist. Testijad jäävad aga nende osapoolte ja hinnangute vahele ning peavad välja tooma kõik tarkvaras leitud vead. Selles rollis tuleb samastuda kasutajaga ning samas tuleb mõista ka kliendi äri, et valmiks kõikidele nõuetele ja ootustele vastav süsteem.

Seega on tarkvara testimisel väga oluline roll, mille peamiseks eesmärgiks on kvaliteedi tagamine. Crispin jt [10] kirjutavad, et testija ülesandeks on tagada, et tarkvara vastab kasutajate ja kliendi ootustele ning et lahendused vastaksid nõuetele. Lisaks peaks testimine tuvastama kriitilised vead süsteemis ning võiks moodustada dokumendi nõuetest.

Tarkvara testimisega tuleks alustada koheselt kui loodav rakendus seda võimaldab. Sellisel juhul on võimalik leida ja parandada vead süsteemis arenduse varajases etapis, millega vähendatakse hilisemat lisatööd, kus viga võib mõjutada juba suuremaid osasid. Kui näiteks viga nõuetes avastatakse alles pärast seda, kui toode on kasutusse antud (*released*), siis see võib maksma minna 10-100 korda rohkem võrreldes sellega, kui viga oleks avastatud juba nõuete ülevaatamisel [11]. Seega vigade avastamine ja nende parandamine varajases faasis on märgatavalt ökonoomsem.

3.3. Tarkvara testimine detailsete nõuete puudumisel

Nõuded on kirjeldus/spetsifikatsioon, millele tarkvara peaks vastama. Tarkvara testimist käsitlevas materjalis [12] tuuakse välja, et nõuded peaksid olema üheselt arusaadavad kõikidele osapooltele. Mitmetimõistmise vältimiseks aitab testilugude koostamine arenduse algfaasis, millega on võimalik avastada puudused enne, kui hakatakse vastavat osa arendama.

Alati aga ei ole projektidel detailseid tarkvara nõuete spetsifikatsioone, mille alusel arendada ja testida, kuid see ei vähenda testimise olulisust. Sellisel juhul ei ole võimalik järgida standardset testimise praktikat ja lähenema peab loovamalt. Rani [13] ja Desyatnikov [14]

toovad oma artiklites välja, et sellise tarkvara puhul tuleb nõuete ja informatsiooni kogumiseks tihedalt suhelda projektimeeskonnaga, et vajadusel õigelt inimeselt vastuseid saada. Lisaks tasub uurida muid olemasolevaid dokumente (protsessijoonised, kasutuslood vms). Testimise planeerimiseks tasub koostada nimekiri süsteemi komponentidest ja funktsionaalsustest ning hea oleks, kui projektis on testija, kellel on kogemused ja teadmised sarnasest süsteemist. Veel tuuakse artiklites välja, et nõuete puudumisel tuleb abiks uuriv testimine (*exploratory testing*), kus ei määrata ette konkreetseid testimise samme ja testilugusid ei dokumenteerita üldse või ei dokumenteerita neid täies ulatuses ette, vaid täiendatakse jooksvalt.

Ghazi jt [15] artiklis tuuakse välja, et uuriva testimise puhul õpib testija süsteemi tundma testimise käigus ning seeläbi omandatud teadmiste põhjal otsustab jooksvalt, kuidas ja mida testida. Seejuures mängivad olulist rolli ka testija varasemad teadmised ja kogemused. Eelnevalt võib kirja panna testiideed, mis kirjeldavad lühidalt ja üldisemalt, mida peaks süsteemis testima ning seda võib vaadata kui väga üldist testiplaani [12].

3.4. Automaattestimine

Suurte süsteemide testimine võib olla väga ajamahukas. Lisaks võib väikseimgi muudatus mõjutada süsteemi mitmeid osasid ja nende toimimist. Sellisel puhul peaks testija käima kõik süsteemi osad käsitsi läbi, et kontrollida nende jätkuvat funktsioneerimist. Siin tasubki kasutada automaattestimist.

3.4.1. Milleks automatiseerida?

Oliinyk jt [4] selgitavad, et automatiseeritud testimine on tarkvara testimise meetod, mis hõlmab spetsiaalsete tarkvaraliste tööriistade kasutamist testide käivitamiseks ning tulemuste kontrollimiseks, kus tegelikke testitulemusi võrreldakse prognoositud tulemustega. Need tegevused sooritatakse ilma või vähese sekkumisega testijate poolt. Automaattestijad kirjutavad skripte (automatiseeritud testilugusid), mis sisaldavad komplekti tegevustest ja kontrollidest.

Crispin jt [10] toovad välja, et automaattestimine aitab tõsta testimise efektiivsust, vähendab testimise ajakulu, võimaldab sagedasemat testimist ning võimaldab katta selliseid osasid, mida manuaalse testimisega ei ole võimalik teha. Samuti on automaattestimine veakindlam – vead avastatakse varem ning tekib vähem inimlikke vigu, mis võivad olla põhjustatud nt rutiinsest tööst. Lisaks annab see ka kindlustunde, et pärast muudatuste või uuenduste tegemist süsteemis jäid olemasolevad funktsionaalsused tööle.

Automaattestimisel on ka mitmed piirangud. Näiteks selle juurutamine tähendab täiendavaid kulusid ning lisab projektile keerukust. Automaattestid kontrollivad ainult etteantud kohtasid ning seega kui midagi jäi teste koostades märkamata, ei pruugi see ka hiljem välja tulla. Kõike ei ole ka mõtet automatiseerida – mõned testid tuleb siiski teha manuaalselt. Lisaks võivad hägustuda testimise prioriteedid.

Automatiseerimisel tagab edu läbimõeldud strateegia, õige ja sobiva raamistiku valimine ning testide hallatavus. Testide kirjutamisel tuleb lähtuda põhimõttest, et mida lihtsam seda parem. See aga ei tähenda, et testid peaksid olema oma sisult lihtsad – testid peavad olema ennekõike vajalikud, sõltumatud (ei sõltu teistest testidest), korratavad (käituvad alati samamoodi), jälgitavad ning lihtsasti hooldatavad ja hallatavad. Lisaks peavad testid andma vajalikku tagasisidet arusaadavalt ja kiirelt.

Mida aga tasub automatiseerida? Kuna rakendus areneb ja muutub elutsükli jooksul pidevalt, siis on oluline kontrollida olemasolevate funktsionaalsuste säilimist. Seega iga kord, kui lisatakse uus funktsionaalsus, peab kontrollima, kas programmi uuendused või lisatud muudatused pole katki teinud programmi varem toimivaid osasid (regressiooni testimine) [16]. Just siin tuleb eriti kasuks testide automatiseerimine, mis võimaldab korduvat testimist. Lisaks enne igasugust testimist tuleb veenduda, kas programmi põhifunktsioonid ikka töötavad, mida kutsutakse suitsutestimiseks – seega võiks ka suitsutestimine olla üks esimesi samme testimise automatiseerimiseks [12].

3.4.2. Automaattestide liigitus

Testide automatiseerimine võimaldab testida kiiresti ja seda juba arendamise varajastes etappides. Selleks, et saavutada tarkvara kõrge kvaliteet, on järjepidev automaattestimine oluline järgmistel tasemetel: ühiktestimine, integratsioonitestimine, süsteemitestimine (tabel 1) [17].

Tabel 1. Automaattestimise tasemed

Testimise tase	Olulisus	Hind	Vastutus	Eesmärk	Testimise tüüp
Ühiktestid	Kõrge	Madal	Arendajad	Meetodite, klasside, komponentide ja moodulite testimine.	Funktsionaalne testimine Mitte-funktsionaalne testimine Regressiooni-testimine*
Integratsiooni-testid	Kõrge	Keskmine	Arendajad/ testijad	Rakenduse erinevad moodulid/ komponendid ja/või teenused töötavad koos.	
Süsteemitestid	Kõrge	Kõrge	Testijad	Süsteemi kui terviku töötamine. Siia kuulub ka vastuvõtu-testimine.	

* regressioonitestimine toimub igakord kui automaattestid käivitatakse

Naik jt [18] selgitavad, et ühiktestimise ehk komponenttestimise tasemel testivad arendajad programmi väiksemaid eraldiseisvaid komponente (nt meetodit, klassi, paketti) kas isolatsioonis või asendades liidestuvad komponendid ootuspäraselt käituvate simuleeritud objektidega. Kui ühiku tasemel kõik töötab rahuldaval määral, siis liigutakse integratsiooni testimise juurde, kus eesmärgiks on testida komponentide vahelisi liideseid. Siin tasemel testivad enamasti nii arendajad kui ka testijad. Integratsiooni testimisele järgneb süsteemitestimine, mida teostavad enamasti testijad ning eesmärgiks on süsteemi testimine tervikuna lähtudes äriprotsessidest, kasutuslugudest või muust kirja pandud nõuetest ning see hõlmab laia testimisspektrit, kuhu alla kuuluvad nt turvalisuse, koormuse, stressi jms testimine. Sellel tasemel testimisel on oluline, et testkeskkond oleks stabiilne ning sarnane reaalsele (*live*) keskkonnale. Soovituslik on, et umbes 50% testidest tehakse ühiktestimise tasemel, 30% integratsioonitestimise tasemel ning 20% süsteemitestimise tasemel [19].

Kõikide tasemete testid võivad katta nii funktsionaalseid (mida tarkvara peab pakkuma) kui ka mittefunktsonaalseid (kuidas tarkvara võimalusi kasutamiseks ja haldamiseks pakkuma peab) nõudeid ning sobivad regressiooni testimiseks [20]. Regressioonitestimine on tarkvara muudatustest lähtuv testimine, kus uuritakse ega muudatused ei ole negatiivselt mõjutanud süsteemi varem toiminud osasid. Kuna automaattestide puhul käivitatakse alati ka varasemalt loodud testid, siis sellega toimubki regressioonitestimine [21]. Ühiktestimise tase on kõige

lähemal programmikoodile ning nende kirjutamine ja jooksutamine on testimise tasemetest kõige kiirem. Integratsiooni ehk teenuste testimise tasemel kontrollitakse rakendusliideseid (API-sid) ning nende jooksutamine on küll aeglasem kui ühiktestimisel, kuid kiirem kui süsteemitestimisel, kuna siin ei testita läbi kasutajaliidese. Eristatakse komponendi integratsiooni testimist ja süsteemi integratsiooni testimist [20]. Süsteemi tasemel on kaasatud ka kasutajaliides ning selle puhul tuleb automatiseerida ainult kõige olulisem, kuna nende jooksutamine on kõige aeganõudvam ning seetõttu on see ka kõige kulukam.

Testija rollist lähtudes tuleb keskenduda just integratsiooni- ja süsteemitestimisele. Lisaks funktsionaalsuste testimisele tuleb kvaliteedinäitajatena arvesse võtta ka mittefunktsionaalsete nõuete testimine, mis on välja toodud EVS-ISO/IEC 9126-1:2003 standardis [8] ja meditsiini- ning tervisetehnoloogia edendamiseks loodud assotsiatsiooni AdvaMedi juhendis [22]: jõudlustestimine (tavaolukorras süsteemi toimimise kiirus), stresstestimine (koormus, mille juures süsteem ei suuda korrektselt toimida), turvalisuse testimine (ligipääs ainult volitatud isikutel, tagatud on konfidentsiaalsus, ligipääs täielikule ja õigele infole), taastuvuse testimine (süsteemi taastumine pärast ülekoormust), kasutatavuse testimine (süsteemi arusaadavus ja kasutusmugavus), koormustestimine (süsteemi töötamine maksimaalse võimaliku koormusega) ja robustsuse testimine (süsteemi võime taluda ootamatusi ning sellest taastumine).

3.5. Testimissüsteem GAISis enne käesolevat tööd

Enne käesoleva töö teostamist oli GAISis läbi viidud ühiktestimist ning sisemiste komponentide integratsiooni testimist. Nii kasutaja- kui ka rakendusliidese testimiseks olid tehtud suitsutestid. Kasutajaliidese puhul oli selleks testidega kaetud sisse- ja väljalogimine, deklaratsiooni kinnitamine ja selle tagasi lükkamine ning päringute teostamine. Rakendusliidese puhul olid tehtud positiivsed stsenaariumid deklaratsiooni loomiseks, staatuse kontrollimiseks ja selle arhiveerimiseks ning sertifikaatide uuendamine ja geneetiliste andmete allalaadimine.

Kasutajaliidese testid olid kirjutatud kasutades Cypress raamistikku ning rakendusliideste testimiseks Postmani rakendust, mis võimaldab teha HTTP päringuid. Testimine ei olnud pideva integratsiooni osa, vaid teste käivitati manuaalselt vastavalt vajadusele.

Samas puudus GAISil terviklik testimisplaan, mistõttu polnud projektimeeskonnal ülevaadet, kui suur osa funktsionaalsusest on testidega kaetud, milline osa on katmata, kui hästi tarkvara töötab, ja seda oli raske kommunikeerida ka projekti juhtidele.

3.6. Käesoleva töö eesmärk

Käesoleva töö eesmärgiks on planeerida ja dokumenteerida GAISi testimist ja konkreetse implementatsioonina luua automaattestid GAISi X-tee liideste testimiseks. Esimene on kirjeldatud peatükis 4 ja teine peatükis 5.

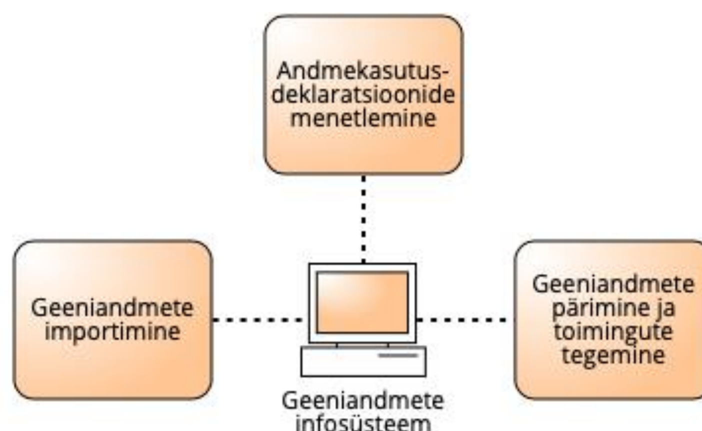
4. GAISi testimise planeerimine

Testimise planeerimise käigus määratakse testimistegevuste põhimõtted, skoop (ulatus) ja testiideed, mille alusel tekib testide planeerimise protsessi dokument. Kuna GAISi kohta puudub konkreetne dokumentatsioon tarkvaranõuetest, siis lähtutakse testimisel arendusmeeskonna juhistest ning töö autori isiklikest kogemustest.

4.1. Üldise skoobi määramine

Geeniandmete infosüsteemi üldine testimise skoop tuleneb selle kolmest põhifunktsioonist (joonis 8), milleks on:

1) sobivate andmekomplektide importimine;
2) andmekasutusdeklaratsioonide menetlemine – staatuste muutmine, kinnitamine ja tagasi lükkamine;
3) geeniandmete pärimine ja kasutamine.



Joonis 8. Testimise skoop

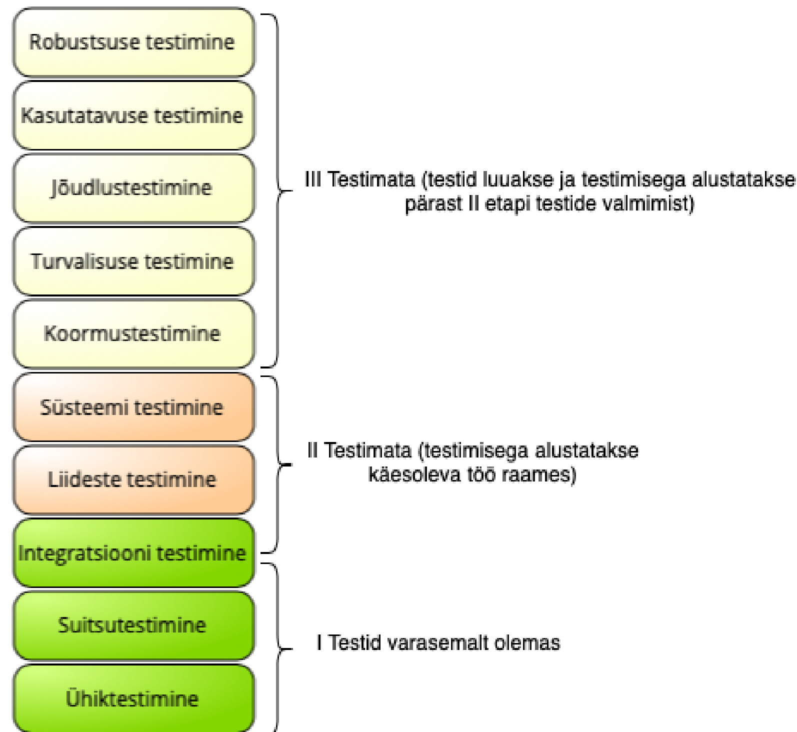
Üldise skoobi määratlemine on oluline, et paika panna konkreetsem töö skoop ning testimise liigid. Lisaks aitab see kategoriseerida testiideid loogiliste üksustena.

4.2. Testimise liikide planeerimine

Kui enamasti tarkvara tootjaid survestatakse toodet kiiresti arendama ja sellega turule tulema, siis on selge, et meditsiinis kasutatava tarkvara puhul ei ole võimalik seda põhjalikult testimata käiku lasta. Seega sujuvaks testimisprotsessiks soovitab ScienceSoft [23] kombineerida liideste testimist ja süsteemitestimist, millele järgneb põhjalik regressioonitestimine. Meditsiinis kasutatava tarkvara puhul on ka oluline valideerida mittefunktsionaalseid nõudeid. Seega soovitatakse lisaks funktsionaalsuse testimisele ka jõudlustestimist ja kasutatavuse ning turvalisuse testimist. Tervise ja Heaolu Infosüsteemide Keskus, kes arendab ja haldab Eestis

teisi keskseid terviseandmekogusid, toob oma tellitavate arenduste nõuetes välja, et minimaalseks testimise skoobiks on ühiktestid, integratsioonitestid, liideste testid, süsteemitestid ning koormustestid ja robustsuse testid [24].

GAISi testimiseks tuleb erinevaid tehnikaid kombineerida ning arvesse tuleb võtta, millist osa ja mis tüüpi testimist on juba teostatud ning tuleb planeerida, millises järjekorras testimisega jätkata (joonis 9).



Joonis 9. Testide planeerimine. Käesoleva töö skoobis on II etapp.

Kuna ühiktestimine toimub samaaegselt süsteemi arendamisega arendajate poolt ning ka sisemiste komponentide integratsiooni testimine, mis käivitatakse enamasti koos ühiktestidega, kuuluvad GAISis arendajate vastutusalasse, siis sellest tulenevalt alustatakse käesolevas töös testide planeerimist integratsiooni, süsteemi ja rakendusliideste testimisest. Pärast käesoleva töö valmimist tuleb kontrollida mittefunktsionaalsetele nõuetele vastavust: koormustestimine, turvalisuse testimine, jõudlustestimine, kasutatavuse ja robustsuse testimine.

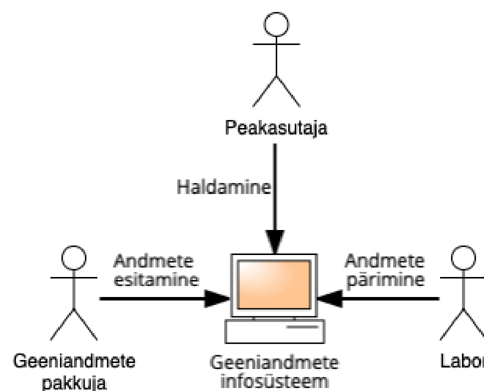
4.3. Testiideed ning käesoleva töö skoobi määramine

Automatiseerimiseks on valitud tegevused, mida tehakse süsteemis sagedasti ja korduvalt ning mis mõjutavad ka süsteemi teisi osasid. Käesolevas töös ei järgita standardset testilugude kirjutamise praktikat, kuna konkreetsete nõuete puudumisel on seda keeruline teha, vaid kirjeldatakse testiideid (vt ptk 3.3), millest tuletatakse testilood. Siinkohal tasub meeles pidada,

et kõiki testilugusid ei ole võimalik kohe ette näha ning neid tuleb täiendada ja täpsustada pidevalt projekti lõpuni [25].

Testiideede koostamisel lähtutakse süsteemi peamistest kasutajagruppidest, kuna just vastavalt nende vajadustest ja tegevustest on arendatud funktsionaalsused (joonis 10). GAISi kasutajagrupid on järgmised:

1) peakasutaja, kes haldab süsteemi;
2) geeniandmete pakkuja, kes esitab geeniandmeid;
3) labor, mis esitab geeniandmete kasutamise deklaratsioone, küsib geeniandmeid isikukoodi alusel ning lõpus ka arhiveerib deklaratsiooni.

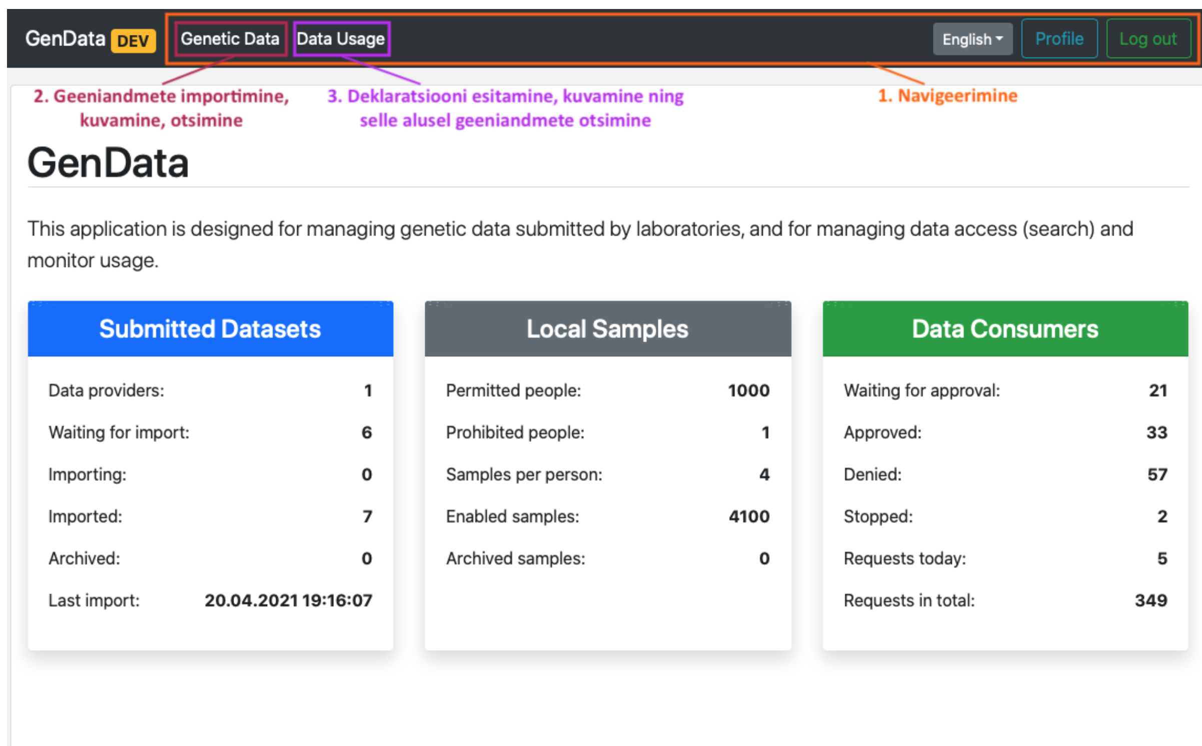


Joonis 10. Geeniandmete haldamise süsteemi peamised kasutajagrupid

Nendest tulenevad järgmised funktsionaalsed eesmärgid:

- sisselogimine;
- geeniandmete edastamine üle X-tee;
- geeniandmete importimine, kustutamine, arhiveerimine;
- andmekasutusdeklaratsioonide kinnitamine, tagasilükkamine, arhiveerimine;
- geeniandmete otsing, arhiveerimine, taastamine;
- andmekasutuse jälgimine.

Seega keskenduvad testiideed peamiselt geeniandmete ja geeniandmete kasutamise deklaratsioonide loomisele ja menetlemisele. Tabel 2 annab ülevaate testiideedest ning need on jagatud selguse mõttes kolme gruppi vastavalt rakenduse vaadetele (joonis 11).



Joonis 11. GAISi rakenduse peavaade

Testide kirjutamisel jagatakse võimalusel kirjeldatud testiideed mitmeks üksteisest sõltumatuks testilooks. Lisaks on tabelis välja toodud, mis eesmärgiga testid on planeeritud – kasutajaliidese testimist tähistatakse UI (*User Interface)* ning rakendusliidese ja veebiteenuste testimist API (*Application Programming Interface).*

Tabel 2. Geeniandmete ja deklaratsioonide haldamise testiideed

Kirjeldus	UI	API	Skoobis
1. Navigeerimine:			
○ Kasutaja logib sisse	x	x	ei
○ Kasutaja navigeerib pealehele, geeniandmete vaatesse ja andmekasutuse vaatesse	x		ei
○ Kasutaja valib rakenduse keele	x		ei
○ Kasutaja vaatab profiiliandmeid	x	x	ei
○ Kasutaja logib välja	x	x	ei
2. Geeniandmete importimine, kuvamine, otsimine:			
○ Geenilabor saadab X-teed kasutades andmed VCF formaadis GAISi (salvestatakse import kausta)		x	ei
○ Esitatud geeniandmed kuvatakse koos detailse infoga (algallikas, märkused, importimise logid, VCF faili päis, kasutamise ajalugu, tegevused)	x	x	ei

○ Peakasutaja a) impordib andmed (salvestatakse registreeritud kausta) b) kustutab andmed (kustutatud import kaustast)	x	x	ei
○ Imporditud geeniandmete kuvatakse koos detailse infoga (algallikas, märkused, importimise logid, VCF faili päis, kasutamise ajalugu, arhiveerimise info)	x	x	ei
○ Teostatakse geeniandmete otsing, kus imporditud geeniandmete viitenumber on otsingu parameetriks	x	x	ei
○ Muudetakse kasutusõigusi vaikimisi staatusest „lubatud“ staatusesse „keelatud“	x	x	ei
○ Teostatakse otsing kasutusõiguste kontrollimiseks, kus parameetriks on isikukood	x	x	ei
○ Arhiveeritakse imporditud geeniandmed (salvestatakse arhiveeritud kausta) ning kontrollitakse, et neid ei ole enam võimalik otsida	x	x	ei
○ Taastatakse arhiveeritud geeniandmed (salvestatakse registreeritud kausta) ning kontrollitakse, et neid on võimalik otsida	x	x	ei
3. Deklaratsiooni esitamine, kuvamine ning selle alusel geeniandmete otsimine:			
○ Koostatakse ja edastatakse deklaratsioon		x	jah
○ Peakasutaja a) Kinnitab deklaratsiooni b) Lükkab deklaratsiooni tagasi	 x x	 x x	 jah* jah*
○ Kinnitatud deklaratsioon kuvatakse koos detailse infoga (ülevaade, märkused, sertifikaat, ajalugu, tegevused)	x	x	jah*
○ Kontrollitakse deklaratsiooni staatust	x	x	jah*
○ Teostatakse deklaratsiooni otsing viitenumbri järgi	x	x	jah*
○ Peatatakse deklaratsioon	x	x	jah*
○ Aktiveeritakse deklaratsioon	x	x	jah*
○ Arhiveeritakse deklaratsioon	x	x	jah*
○ Kontrollitakse deklaratsiooni sertifikaati		x	jah
○ Uuendatakse deklaratsiooni sertifikaati		x	jah
○ Teostatakse isikupõhine otsing, kus parameetriks on isikukood	x	x	jah*

* käesoleva töö skoobis on ainult API testid

Kuna testiideedest tuleneb palju testilugusid ja kõikidele automaattestide kirjutamine on küllaltki ajamahukas ning ületaks antud töö mahu, siis alustatakse testide kirjutamist kõige olulisemale funktsionaalsusele. Sellest tulenevalt luuakse käesoleva töö praktilises osas X-tee teenuste automaattestid, kuna just siin toimub suhtlus väliste osapooltega, mistõttu on väga oluline, et see toimiks korrektselt. X-tee teenused on peamiselt seotud andmekasutusdeklaratsioonidega ning geeniandmete importimise, kuvamise ja otsimisega.

Kuna töö kirjutamise hetkel on X-tee liides geeniandmete importimiseks alles arendamisel, siis automatiseeritakse ainult andmekasutusdeklaratsioonidega seotud tegevused, kuid tulevikus on oluline katta testidega ka geeniandmete importimine, kuvamine ja otsimine.

4.4. Cypress raamistik testimise tööriistana

Automaattestimise raamistik on juhiste kogum testilugude loomiseks ja kujundamiseks, mis võimaldab struktrueeritud ja modulaarset lähenemist ning võimaldab komponente taaskasutada [26]. Käesoleva töö autor ei osalenud tööriista valimisel, vaid see oli eelnevalt välja valitud GAISi arendajate poolt. Planeerides kasutajaliidese testimist oldi valiku eest, kas võtta kasutusele laialt levinud Seleniumi-põhine raamistik või Cypress. Valik langes Cypressi kasuks, kuna tegemist on kaasaegsema ja väidetavalt kiirema lahendusega [27]. Lisaks pakkus huvi Cypressi poolt pakutav graafiline kasutajaliides ning selle muud lisavõimalused.

Kuigi rakendusliideste testimist oldi projektis alustatud Postmaniga, oli siiski soov neid edasi arendada samuti Cypressiga, kuna tundus mõistlik, et kogu testimine on tehtud ühes raamistikus, mis võimaldab testandmeid jagada kahe kihi vahel ning neid ka taaskasutada. Seetõttu võib ka eeldada, et testide haldamine on edaspidi kiirem ja mugavam.

Cypress.io organisatsioon loodi 2015. aastal ning see arendab kasutajaliidese automaattestimise rakendust eesmärgiga jooksutada veebilehitsejas nii ühik- kui ka integratsiooniteste [28]. Cypressi testimise raamistik loodi esialgu arendajatele ühiktestimiseks, kuid peagi tekkis selle vastu huvi ka laiemalt [29]. 2017. aastal eemaldati selle kasutamise piirangud – see sai kättesaadavaks kõigile soovijatele ning loodi ka dokumentatsioon ja juhendid [30].

JavaScript & Friends konverentsil tehtud ettekandes [31] anti ülevaade, et Cypress on avatud lähtekoodiga testimise raamistik, milles on olemas kõik testimiseks vajaminevad teegid ning kasutaja ei pea ise otsustama ega valima, millist raamistikku, validatsioonide teeke ja täiendavaid teeke ta kasutama peaks ning ei pea allalaadima vastava veebilehitseja faile ega erinevaid tööriistu. Cypressi erinevus võrreldes teiste testimise raamistikega on see, et sellel ei ole Seleniumi-põhine arhitektuur ja ei oma seetõttu samu probleeme. Võrreldes Seleniumiga on see kiirem, stabiilsem ning seda on lihtsam kasutada. Lisaks on see mõeldud nii arendajatele kui ka testijatele, kuna selle loomisel oli üheks eesmärgiks ka testimisel põhineva arenduse võimaldamine just kasutajaliidese jaoks [17].

Cypressi graafilise kasutajaliidese omadused [32]:

- võimaldab ajarännet ja vigade avastamist (*debugging*), tehes ekraanitõmmiseid igast sammust koos logiga ning info kuvatakse ka konsoolis;
- reaalajas lehe laadimine, mis tähendab, et teste jooksutatakse muudatuste tegemise järgselt automaatselt;
- automaatne ootamine – teab, millal võrgupäring tagasi tuleb ning ootab kuni element on visualiseeritud, mistõttu ei ole vaja lisada erinevaid ootamise tingimusi;
- võrguliikluse kontrollimine, mis võimaldab tagastada fiktiivseid (*mock, stub*) vastuseid ilma rakendusserveri poole reaalselt pöördumata.

Lisaks salvestab Cypress automaatselt iga ebaõnnestunud testi kohta ekraanitõmmise ning viimati käivitatud testi kohta ka video, mis võimaldab vaadata, mis täpsemalt tehti ning vea korral selle kiiremat tuvastamist. Sellega välditakse olukorda, kus ebaõnnestunud testi põhjuse tuvastamiseks peaks teste uuesti käivitama.

Cypressil on ka mitmed piirangud. See sobib ainult veebirakenduste testimiseks, kuna skripte ei saa käivitada väljaspool veebilehitsejat [33]. Cypress ei toeta mitut vahelehte ega suuda avada samal ajal mitut veebilehitseja akent, ei saa ühe testi ajal külastada erinevaid domeene (ainult alamdomeene), toetab ainult JavaScripti ning veebilehitsejatest ei ole hetkel veel toetatud Safari, Opera ega Internet Explorer [32]. Toetatud ei ole veel ka faili üleslaadimine, veebilehe kerimine ja ülehõljumine (*hovering*) [34]. Kuna tegemist on suhteliselt uue raamistikuga, siis ei leidu selle kohta ka nii palju materjale kui näiteks Seleniumi kohta.

Kuigi Cypressi kodulehel [35] on välja toodud, et tegemist ei ole üldise automatiseerimise raamistikuga ning nad on spetsialiseerunud veebirakenduste läbivtestimisele (*end-to-end testing, E2E testing*), leiab allikaid selle kohta, et tegelikult sobib see hästi ka rakendusliideste testimiseks – ka kasutajaliidese läbivtestimiseks on vaja kasutada rakendusliideseid ning Cypress võimaldab teha HTTP päringuid.

5. Automaattestid Cypress raamistikus

5.1. Testkeskkonna struktuur

Testkeskkonna struktuuri loomisel ja testide kirjutamisel on püütud järgida Cypressi dokumentatsiooni [36]. Struktuur on järgmine:

- cypress.json – konfiguratsioonifail testide jooksutamiseks;
- package.json – fail, kus on teekide sõltuvused ja käsklused testide jooksutamiseks;
- cypress/integration – kaust, kuhu on lisatud testid;
- cypress/fixtures – kaust, kuhu on lisatud testandmed;
- cypress/support – kaust, kuhu on lisatud käsklused, mida kasutatakse korduvalt ja mitmes testis.
- cypress/screenshots – kaust, kuhu salvestatakse ekraanitõmmised ebaõnnestunud testidest;
- cypress/videos – kaust, kuhu salvestatakse viimati käivitatud testikomplektide videod;

Testilood on jagatud loogiliselt tegevuste järgi ning koosnevad enamasti kolmest faasist:

1) iga testi alguses saadetakse geeniandmete kasutamise deklaratsioon(id);
2) tehakse testimiseks vajalikud toimingud ning toimub valideerimine;
3) testi alguses saadetud deklaratsioon(id) arhiveeritakse.

5.2. Loodud automaattestid

Automaatsete testilugude loomisel võeti aluseks ptk 4.3 tabelis 2 kirjeldatud testiideed. Alustati kõige olulisemast, milleks oli deklaratsiooni loomine ning selle arhiveerimine, mille toimimine on oluline ka kõigi järgnevate testide juures, kuna ilma eduka deklaratsiooni saatmiseta ei ole võimalik järgnevaid samme teha. Seejärel kirjutati testid deklaratsiooni staatuste muutmiseks ning kontrolliti deklaratsiooni alusel geneetiliste andmete tagastust. Lisaks valideeriti, et on võimalik kontrollida ja muuta sertifikaate. Tabel 3 annab ülevaate, millise testikomplektiga on kaetud eelnevalt kirja pandud testiidee.

Tabel 3. Testiidee ning sellele vastav testikomplekt

Testiidee	Testikomplekt
o Koostatakse ja edastatakse deklaratsioon	send-declaration.spec.js
o Peakasutaja a) Kinnitab deklaratsiooni b) Lükkab deklaratsiooni tagasi	 approve-declaration.spec.js deny-declaration.spec.js

○ Kinnitatud deklaratsioon kuvatakse koos detailse infoga (ülevaade, märkused, sertifikaat, ajalugu, tegevused)	get-declaration.spec.js
○ Kontrollitakse deklaratsiooni staatust	send-declaration.spec.js
○ Teostatakse deklaratsiooni otsing viitenumbri järgi	search-declaration.spec.js
○ Peatatakse deklaratsioon	stop-declaration.spec.js
○ Aktiveeritakse deklaratsioon	resume-declaration.spec.js
○ Arhiveeritakse deklaratsioon	archive-declaration.spec.js
○ Kontrollitakse deklaratsiooni sertifikaati	check-certificate.spec.js
○ Uuendatakse deklaratsiooni sertifikaati	update-certificate.spec.js
○ Teostatakse isikupõhine otsing, kus parameetriks on isikukood	get-genetic-data.spec.js

Testiideest on loodud eraldi testikomplektid, mille all on nii positiivse kui ka negatiivse stsenaariumiga testilood. Käesolevas töös loodi 11 testikomplekti, milles on kokku 57 testilugu. Kuna puudub detailne dokumentatsioon nõuetest rakendusele, siis on testide valideerimisel oodatud tulemus paika pandud automaattestide kirjutamise hetkel koostöös arendajatega, kellel on need teadmised olemas.

Joonisel 12 on toodud näide ühe testikomplekti (approve-declaration.spec.js) struktuurist. Lisas 1 on näidisena välja toodud selle testikomplekti testilood täies mahus.

```
describe("Approving declaration", () => {

    before('Send declaration', function() {
        cy.sendDeclaration()
    })

    beforeEach('Login', function() {
        cy.login(Cypress.env('keycloakLoginUsername'), Cypress.env('keycloakLoginPassword'))
    })

    after('Archive declaration', function() {
        cy.archiveDeclaration(varDeclarationRef)
    })

    it('should approve declaration', () =>  {. . .})
    it('should return approved status', () => {. . .})
    it('should return declaration reference in approved declarations list', () =>  {. . .})
    it('should not approve declaration if already approved', () =>  {. . .})
    it('should not approve declaration if invalid declaration reference', () =>  {. . .})
    it('should not approve declaration if non-existent declaration reference', () =>  {. . .})
})
```

Joonis 12. Testikomplekt deklaratsiooni kinnitamiseks

Iga testikomplekt koosneb ühest JavaScript (*.js) failist ja algab *describe* meetodiga, mis kirjeldab üldiselt kõiki selles komplektis olevaid testilugusid. Seejärel käivitatakse *before* meetod, kus enne kõikide testilugude käivitamist saadetakse deklaratsioon(id). Sellele järgneb *beforeEach* meetod, mis käivitatakse enne igat testilugu. Seejärel käivitatakse *it* meetodiga testid sellises järjestuses, nagu need testikomplektis on, kus esimeseks argumendiks on testiloo nimi, mis kirjeldab, mida antud testiga kontrollitakse ning teine argument on testimise sammud ja valideerimine (joonis 13).

```
it('should return approved status', () => {
    cy.readFile('cypress/fixtures/declaration_status.xml')
        .then(text => text.replaceAll('$varDeclarationRef', varDeclarationRef))
        .then(text => cy.fetchXML(text))
        .then((response) => {
              expect(response.status).to.eq(200);
              expect(response.body).to.include('APPROVED');
       })
})
```

Joonis 13. Testilugu deklaratsiooni staatuse „kinnitatud“ kontrollimiseks

Kui kõik testilood on lõppenud, siis käivitatakse *after* meetod, mis arhiveerib *before* meetodis loodud deklaratsiooni(d).

Kõik käesolevas töös loodud automaattestid on kättesaadavad repositooriumis Github järgmiselt lingilt: https://github.com/jansok/janne_thesis.

5.3. Automaattestide käivitamine

Kuigi käesolevas töös kirjutatud teste ei ole võimalik tavakasutajal käivitada, kuna geeniandmete infosüsteemi programmikood ei ole töö kirjutamise ajal avalikult kättesaadav, antakse töös siiski ülevaade sellest, kuidas Cypressi paigaldada ning teste käivitada ja tulemusi kontrollida.

Süsteemi seadistamiseks on kõigepealt vaja allalaadida Node.js ja npm (*Node Package Manager*), mille kohta leiab juhised ja täpsemat informatsiooni nende kodulehelt[2]. Lisaks on soovitatav kasutada IDEt (*integrated development environment*, integreeritud programmeerimiskeskkond), kus on mugav teste kirjutada, kuna pakub erinevaid abivahendeid (nt koodi värvimine ja treppimine, funktsioonide loetelu).

Kuna *package.json* failis on Cypress sõltuvusena (*dependency*) lisatud, siis piisab, kui kirjutada Cypressi paigaldamiseks käsureal:

[2] https://www.npmjs.com/get-npm

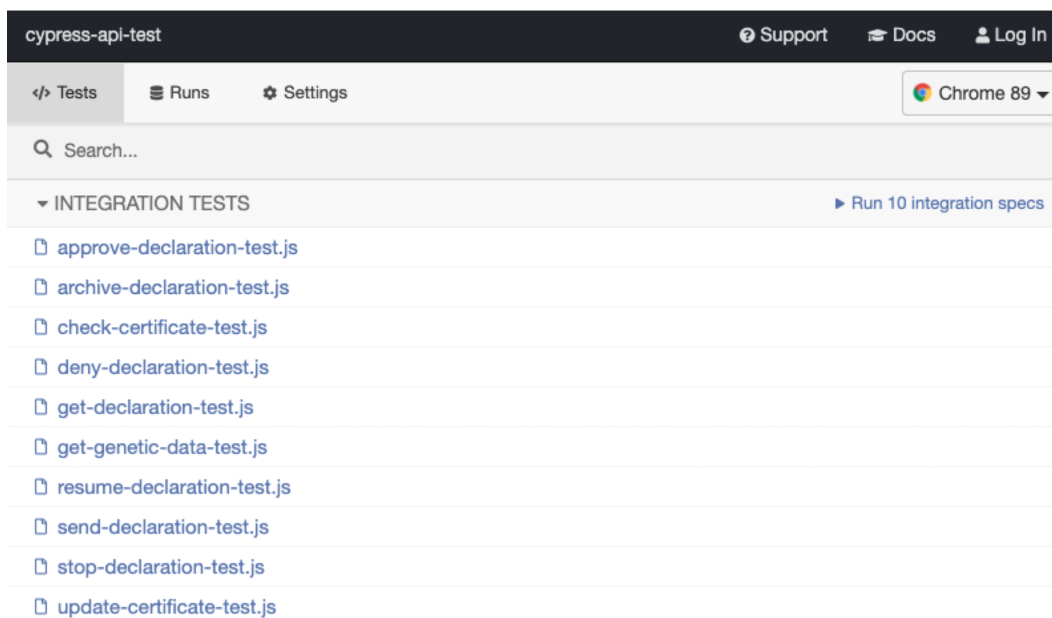
```
npm install
```

Testide käivitamiseks tuleb navigeerida juurkausta ning kirjutada käsureal emb-kumb eelnevalt *package.json* failis määratud käsklus:

`npm test` käivitatakse ilma kasutajaliideseta (*headless*)

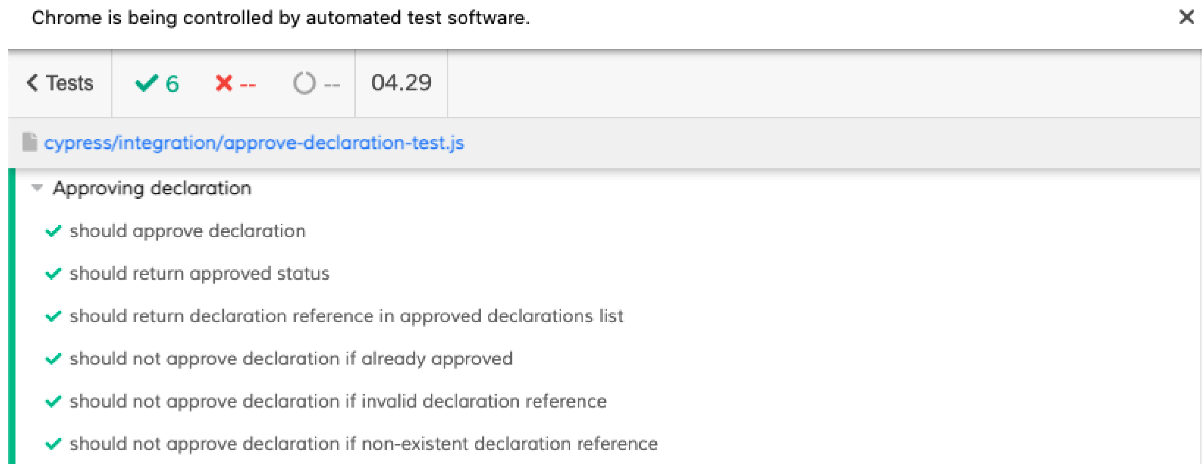
`npm run cypress:open` käivitatakse kasutajaliidesega

Viimane käsk avab Cypressi testide jooksutamise akna (joonis 14), kus kuvatakse loodud testid, mis asuvad cypress/integration kaustas ning mis on .js laiendiga. Aknas on võimalik käivitada kõik testid korraga või valida teste ükshaaval. Lisaks on võimalik valida, millises veebilehitsejas seda tehakse.

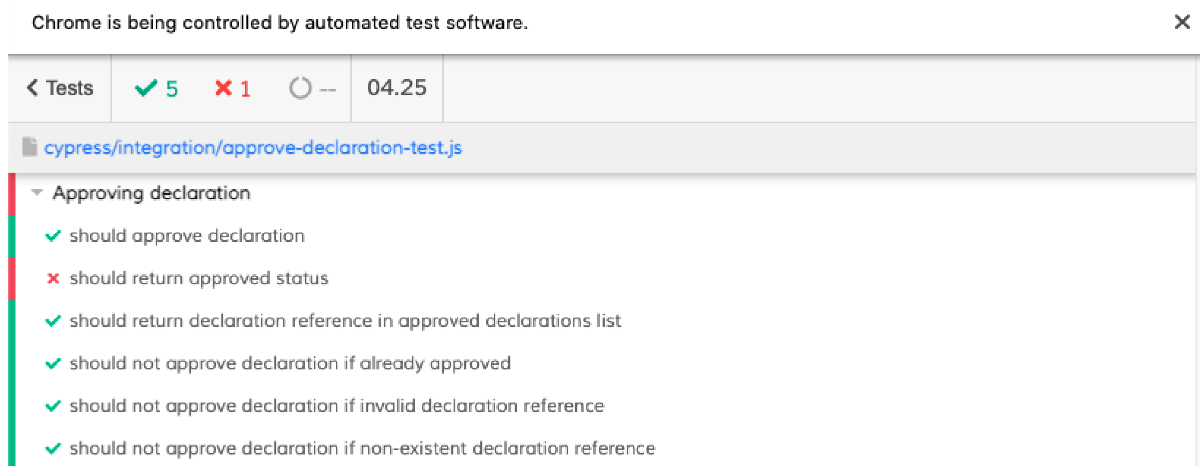


Joonis 14. Cypressi testide käivitamise vaade

Testide käivitamisel avatakse veebilehitseja, mis kuvab info testide õnnestumise (joonis 15) või ebaõnnestumise (joonis 16) kohta. Lisaks kuvatakse testide jooksutamisele kulunud aeg ning õnnestunud ja ebaõnnestunud testide arv.



Joonis 15. Õnnestunud testide vaade



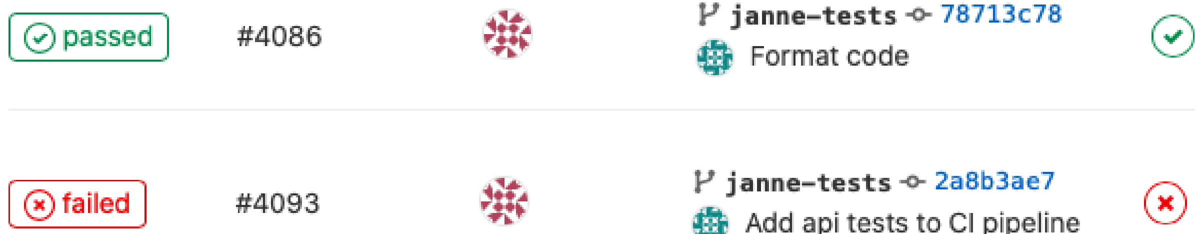
Joonis 16. Ebaõnnestunud testi vaade

Igat testilugu on võimalik samas veebilehitsejas avada, mis kuvab detailset infot tehtud sammudest ning testi ebaõnnestumise korral näitab, millises sammus mis viga tekkis.

Testimise peatamiseks võib veebilehitseja kas sulgeda või vajutada testide käivitamise aknas "Stop". Cypressi sulgemiseks tuleb sisestada käsureal *Ctrl+C* või kinni panna testide käivitamise aken, mis automaatselt sulgeb ka veebilehitseja.

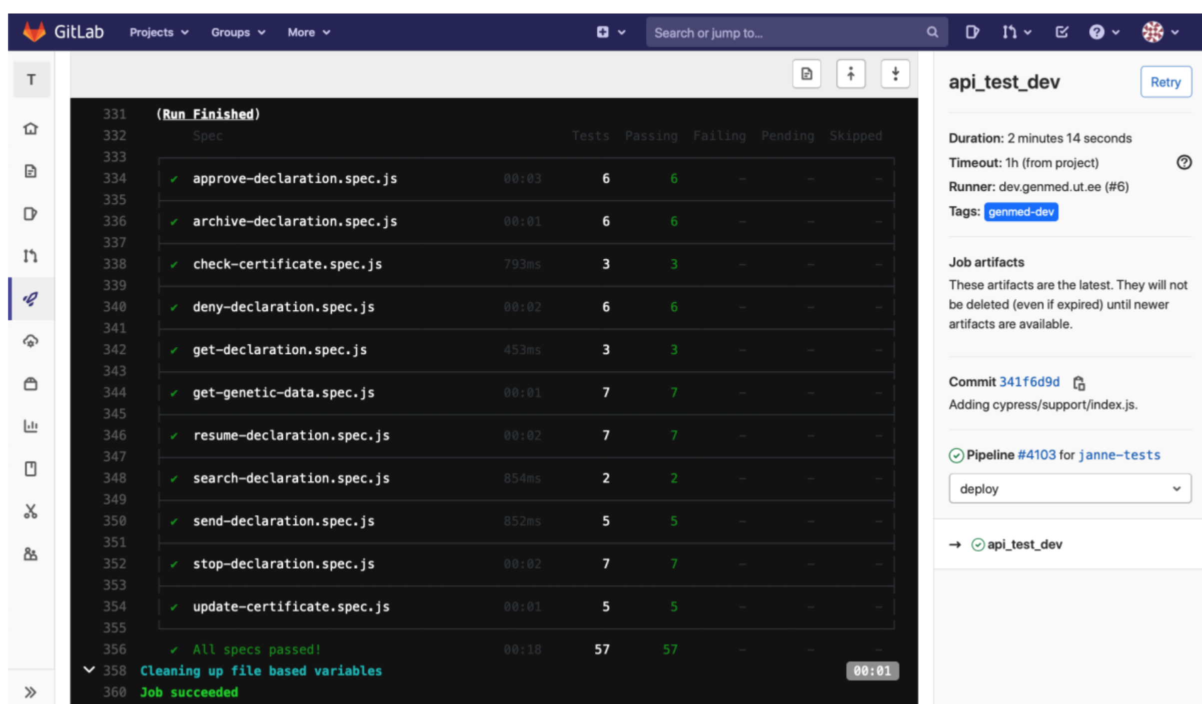
5.4. Automaattestide integreerimine

Pideva integratsiooni puhul migreeritakse pidevalt kood versioonihaldusesse, misjärel käivitatakse rakenduse ehitamine (*build*) ning automaattestid ning tagastatakse info õnnestumise või ebaõnnestumise kohta (joonis 17). Ebaõnnestumise puhul on võimalik ka detailsemalt näha, mis vea põhjustas. Selline lähenemine võimaldab varakult avastada vead süsteemis ning need koheselt kõrvaldada [37].



Joonis 17. Õnnestunud ja ebaõnnestunud töövoog

Kuna GAISi projektis on kasutusel versioonihalduskeskkond GitLab, siis lisati ka töös loodud automaattestid GitLabi pideva integratsiooni voosse, mille õnnestumist on näha joonisel 18.



Joonis 18. Detailne ülevaade õnnestunud töövoost GitLabis

Joonisel on näha, et käivitati 11 testikomplekti ning kõik 57 testilugu on edukalt läbitud. Lisaks kuvatakse testide jooksutamisele kulunud aeg.

6. Arutelu

Käesolev töö valmis vajadusest testida GAISi, milleks aga oli vaja süsteemi eelnevalt tundma õppida. Selleks uuriti olemasolevat dokumentatsiooni, mille alusel koostati joonised ning mis aitasid aru saada süsteemi eesmärkidest, arhitektuurist ning selle funktsionaalsustest. Koostatud dokument andis tervikliku ülevaate, mille alusel sai planeerida testimist.

Testimise planeerimisele lisas keerukust detailsete nõuete puudumine süsteemile, kuna enamasti on tarkvara testimise eesmärk just nõuetele vastavuse kontrollimine. GAISi projektis tuli testijana näha natuke rohkem vaeva, et leida ja koguda testimise planeerimiseks ja testilugude koostamiseks vajaminev informatsioon.

Töö tulemusena valmis GAISi testimise planeerimise dokument ning alustati Cypress raamistikku kasutades rakendusliideste testimisest. Sellega on kaetud osa integratsiooni ja süsteemi testidest. Koostatud automaattestid moodustavad ka dokumendi nõuetest. Lisaks võimaldavad loodud automaattestid hõlbustada regressioonitestimist vähendades sellega manuaalse testimise osakaalu projektis. Loodud testid lülitati ka GitLabi pideva integratsiooni voosse, mis võimaldab saada kohest tagasisidet, kui töövoos peaks mõni etapp ebaõnnestuma.

Varasemalt olid osaliselt Postmanis loodud testid X-tee liidestuste otspunktidele, kuid kuna nende eesmärgiks oli suitsutestimine, siis ei olnud need piisavad veendumaks, et kõik toimib korrektselt, kuna testides kontrolliti ainult veebipäringu staatuskoodi ning vastuse sisutüüpi (*content-type*). Käesoleva töö raames aga tehti põhjalikum kontroll, kus lisaks eelnevale kontrolliti ka vastuse sisu ning lisati ka negatiivseid stsenaariume.

Kas on mõistlik kasutada Cypressi rakendusliidese testimiseks? See on hea kombinatsioon, kui on vaja testida nii kasutaja- kui ka rakendusliidest, kuna võimaldab mugavalt ettevalmistada testandmeid. Selline lähenemine teeb kindlasti kasutajaliidese testimise kiiremaks. Rakendusliidese testimisel on võõras, et testide käivitamisel kasutatakse veebilehitsejat, kuid samas lisab mugavust ligipääs selle arendusvahenditele (*developer tools*), mis võimaldab veelgi mugavamalt tuvastada vigu ning saada infot veebiserveritele tehtud päringute ja nendele saadud vastuste kohta. Cypress laadib automaatselt testid uuesti, kui testskriptides tehakse muudatus, mis on ühest küljest postitiivne, kuna ei pea manuaalselt pidevalt teste uuesti käivitama, kuid sellel on ka negatiivne pool. Kuna käesolevas töös algab iga testikomplekt vähemalt ühe andmekasutusdeklaratsiooni saatmisega, siis pärast igat muudatust ei taha uuesti deklaratsiooni saata, kuna see tekitab andmebaasi otstarbetult palju testandmeid. Selle vältimiseks pidi vahepeal testide jooksutamise aknas testid peatama, tegema

testskriptides vajaliku muudatused ning alles siis uuesti testid käivitama. Seega GAISi projekti rakendusliideste testimise puhul ei olnud sellest Cypressi funktsionaalsusest väga palju kasu ning testide käivitamine toimus siiski pigem manuaalselt.

Testide kirjutamist raskendas asjaolu, et Cypress raamistikku on rakendusliidese testimiseks kasutatud suhteliselt vähe ning seetõttu ei leia selle kohta väga palju praktilisi näiteid ega abistavaid materjale. Seega ei saanud kindel olla, kas see raamistik on sobiv X-tee teenuste testimiseks, kuid testide kirjutamine osutus küllaltki mugavaks ning võib arvata, et skriptid on loetavad ja arusaadavad ka mittetehnilise taustaga inimestele. Siinkohal tasub aga mainida, et Cypressi poolt loodud dokumentatsioon on väga põhjalik ning hea ja palju aitasid ka Cypressi allalaadimisel kaasatulevad näitetestid.

Koostatud testide põhjal võib öelda, et testide jooksutamine Cypressiga on piisavalt kiire, kuid kuna käesoleva töö valmimise hetkel on automaatteste veel suhteliselt vähe, siis on raske hinnata, kui palju aega võiks Cypressil kuluda kogu süsteemi testide jooksutamisele. Kui nende jooksutamine võtab väga kaua aega, siis tekib oht, et teste ei taheta käivitada nii sageli kui võiks. Siinkohal tasuks kindlasti mõelda testide käivitamisele paralleelselt, mis võimaldab kokku hoida testide jooksutamisele kuluvat aega või jooksutada osasid teste koodi salvestamisest (*commit*) sõltumatult. Samas peaksid need testid kindlasti olema jooksutatud enne kui kood arendus- või testkeskkonnast töökeskkonda jõuab.

Edasise tegevusena võiks planeerida testide jooksutamist sõltumatult arendus- või testkeskkonnast. Pärast igat testikomplekti jooksutamist võiks andmebaasi seisu taastada esialgsesse seisu, mis suurendaks testide sõltumatust üksteisest. Kuigi on hea, kui on palju andmeid, kuna see võimaldab näha süsteemi käitumist suuremate andmemahtudega, võiks pigem luua eraldi testid, kus saadetaksegi korraga suurem hulk andmeid.

Kuna loodud on automaattestimissüsteem, mis koosneb nii kasutaja- kui ka rakendusliideste testides, siis on lihtsam jätkata ka uute testide kirjutamisega. Seega tasuks jätkata testide implementeerimist projektis ja seda nii rakendus- kui ka kasutajaliidese poole pealt – esialgu funktsionaalsuse testid ning seejärel teostada ka koormustestimine, turvalisuse testimine, jõudlustestimine, kasutatavuse ja robustsuse testimine. Vältimaks testimise unarusse jätmist, võiks projektile lisada testide katvuse kontrolli koos täitmist nõudvate tingimustega (nt x% funktsionaalsustest peab olema automaattestidega kaetud).

Kindlasti tuleb olemasolevaid teste hooldada ja parandada, kui tarkvaras peaks toimuma muudatused. Samuti võib lisada uusi testiideid või koostada enne automatiseerimist

testilugusid, kuna GAISi arendus jätkub ning nõuded võivad veel täpsustuda, muutuda või juurde tulla. Lisaks tuleb tähele panna, et kui tulevikus peaks rakenduses toimuma suuremad muudatused/uuendused või hakatakse rakendust kohaldama näiteks vastavalt ISO27001 standardile, siis võib juhtuda, et testimine nõuab suuremaid muudatusi. Testimise koormuse hajutamiseks võiks kaaluda testija palkamist.

Käesolev töö pakkus autorile mitmeski mõttes väljakutseid. Varasemalt on testimist planeeritud ning testilugusid koostatud ainult algfaasis arendustele. GAISi rakenduse puhul oli aga testimise planeerimisel eriti oluline osa, kuna see ei olnud enam arenduse algfaasis ning automaattestide loomisel tuli hinnata, millises järjekorras ja millest alustada, kuna olemas oli juba küllaltki palju funktsionaalsust. Samas tuli aga testimise planeerimisele läheneda veidi teistmoodi kui seda harilikult tehakse, kuna puudusid kirjapandud spetsiifilised nõuded tarkvarale. Lisaks oli varasem kogemus teostada automaattestimist Cypressiga minimaalne ning sedagi ainult kasutajaliidese testimisel. Samuti oli vähene kokkupuude programmeerimiskeelega JavaScript ning X-tee andmevahetuskihiga. Testijana aga pakkus Cypress huvi, kuna selle kasutamine on kasvava trendiga ning igasuguse uue testimise raamistiku või rakenduse kasutamise oskus tuleb hilisemas karjääris kasuks.

7. Kokkuvõte

Geeniandmete infosüsteem toetab Eesti personaalmeditsiini programmi tegevusi, mis saab toimida ainult kvaliteetse infosüsteemi kaasabil. Antud töö eesmärgiks oli riiklikuks kasutamiseks mõeldud geeniandmete infosüsteemi testimise planeerimine ning esmaste automaattestide loomine. Töös tutvustati geeniandmete infosüsteemi eesmärki ja funktsionaalsust, samuti testimise vajalikkust ja automaattestimise aluseid. Kuna käesolevas töös kasutati testimiseks Cypress raamistikku, siis anti ülevaade selle tööpõhimõtetest ning kasutamisest.

Töö tulemusena valmis geeniandmete infosüsteemi tervikliku testimise planeerimise dokument, mis võiks olla aluseks selle süsteemi edasise testimise korraldamisel. Samuti loodi rakendusliideste automaattestid X-teed kasutavatele osadele, millega sai kaetud osa nii integratsiooni kui ka süsteemi testidest ning mis lihtsustasid regressioonitestimist. Loodud testilugudega dokumenteeriti ka detailsemad nõuded, mis polnud projektis eelnevalt kirja pandud.

Edasiste tegevustena tasuks jätkata testide kirjutamist projektis nii rakendus- kui ka kasutajaliidese poole pealt vastavalt planeeritule ning seejärel teostada ka koormustestimine, turvalisuse testimine, jõudlustestimine, kasutatavuse ja robustsuse testimine.

8. Viidatud kirjandus

[1] Sotsiaalministeerium. Geeniprojekt personaalmeditsiini arendamiseks Eestis. https://www.sm.ee/et/geeniprojekt-personaalmeditsiini-arendamiseks-eestis (20.02.2021).

[2] Õunap K. Geneetikast ja geneetilisest testimisest meditsiinigeneetiku pilgu läbi. *Eesti Arst*, 2015, nr 94, lk 211–216.

[3] Dahiya O., Solanki K., Dhankhar A. Risk-Based Testing: Identifying, Assessing, Mitigating; Managing Risks Efficiently In Software Testing. *International Journal of Advanced Research in Engineering and Technology*, 2020, nr 11, lk 192-203.

[4] Oliinyk B., Oleksiuk V. Automation in software testing, can we automate anything we want? 2019. http://ceur-ws.org/Vol-2546/paper16.pdf (24.04.2021).

[5] Capgemini, Sogeti, Micro Focus. World Quality Report 2020-21. 2020.

[6] Tervise Arengu Instituut. Personaalmeditsiin. https://www.tai.ee/et/tegevused/personaalmeditsiin (04.02.2021).

[7] Riigi Teataja. Infosüsteemide turvameetmete süsteem. https://www.riigiteataja.ee/akt/13125331 (25.04.2021).

[8] Eesti Standardikeskus. Tarkvaratehnika toote kvaliteet. Osa 1: Kvaliteedimudel EVS-ISO/IEC 9126-1:2003. 2003.

[9] Adzic G., Evans D., Roden T. Fifty Quick Ideas To Improve Your Tests. United Kingdom: Neuri Consulting LLP.On see 2015.

[10] Crispin L., Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams. USA: Pearson Education, Inc. 2009.

[11] Kumar D. P., Syed K. Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications*, 2010, nr 6, lk 7-10.

[12] Markvardt M. Tarkvara testimist käsitlev juhendmaterjal. ASA Quality Services OÜ. 2006.

[13] Rani P. Testing without Requirements or Functional Requirements Document. 2018. https://qa.world/testing-without-requirements (26.04.2021).

[14] Desyatnikov R. How to Test Without Requirements. https://softwaretesting.cioreview.com/cxoinsight/how-to-test-without-requirements-nid-13375-cid-112.html (26.04.2021).

[15] Ghazi A. N., Petersen K., Bjarnason E., Runeson P. Levels of Exploration in Exploratory Testing: From Freestyle to Fully Scripted. 2018. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8357560 (04.05.2021)

[16] Mayfield D. When To Use Automated Regression Testing. 2019. https://testlio.com/blog/regression-testing-automated (04.05.2021).

[17] Pittet S. The different types of testing in Software. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (08.03.2021).

[18] Naik K., Tripathy P. Software Testing and Quality Assurance: Theory and Practice. John Wiley & Sons. 2011.

[19] Das J. Test Automation Strategies. ResearchGate. 2019.

[20] Certified Tester Foundation Level Syllabus. ISTQB. 2018.

[21] Testim. Automated Regression Testing: Everything You Need To Know. 2019. https://www.testim.io/blog/automated-regression-testing (04.05.2021).

[22] Jensen J., Sandoval-Watt I. Software in Medical Devices. AdvaMed.

[23] Mikhailau A. What You Should Know About Testing Software for Medical Devices. 2020. https://www.scnsoft.com/blog/medical-device-software-testing (14.03.2021).

[24] Tervise- ja Heaolu Infosüsteemide Keskus. Automaattestide kasutamise nõuded. https://www.tehik.ee/meist/meistnouded-arendustele/meistnouded-arendusteleautomaattestide-kasutamise-nouded (15.03.2021).

[25] Software Testing Classes. How to Write Good Test Cases? 2013. https://www.softwaretestingclass.com/how-to-write-good-test-cases (06.03.2021).

[26] Test Automation Framework: What is, Architecture & Types. https://www.guru99.com/test-automation-framework.html (18.04.2021).

[27] Cypress Documentation. Why Cypress? https://docs.cypress.io/guides/overview/why-cypress (02.05.2021).

[28] Crunchbase Company Profile & Funding. https://www.crunchbase.com/organization/cypress-io (18.04.2021).

[29] Bhalaik K. Building a Test Automation Framework using Cypress.io - The Intro (Part 1). 2020. https://medium.com/omnius/building-a-test-automation-framework-using-cypress-io-the-intro-part-1-10887855a9f0 (18.04.2021).

[30] Cypress Documentation. Changelog. https://docs.cypress.io/guides/references/changelog (18.04.2021).

[31] Pluralsight. Automated Testing All the Things with Cypress. https://app.pluralsight.com/course-player?courseId=6506f819-af49-4b7e-a5a1-0c3e0940e560 (16.04.2021)

[32] Pluralsight. End to End Testing with Cypress: CodeMash. https://app.pluralsight.com/course-player?courseId=d3324477-3780-4dac-9dbb-1b8361f3e4ef (16.04.2021)

[33] Unadkat J. Cypress vs Selenium: Key Differences. https://www.browserstack.com/guide/cypress-vs-selenium (18.04.2021).

[34] Antweiler J., Hyvo P. Pros and Cons of Cypress. 2020. https://blog.testery.io/pros-and-cons-of-cypress (18.04.2021)

[35] End to End Testing Framework. https://www.cypress.io/how-it-works (17.04.2021).

[36] Cypress Documentation. Writing and Organizing Tests. https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests (18.04.2021).

[37] ThoughtWorks. Continuous integration. https://www.thoughtworks.com/continuous-integration (31.01.2021).

Lisad

I Deklaratsiooni kinnitamise testilood

```
describe('Approving declaration', () => {
    let varDeclarationRef = ''

    before('Send declaration', function() {
      cy.sendDeclaration()
        .then((returned_value) => {
              varDeclarationRef = returned_value
        })
    })

    beforeEach('Login', function() {
      cy.login(Cypress.env('keycloakLoginUsername'), Cypress.env('keycloakLoginPassword'))
    })

    after('Archive declaration', function() {
      cy.archiveDeclaration(varDeclarationRef)
    })

    it('should approve declaration', () =>  {
        cy.request({
                url: 'http://dev.genmed.ut.ee:8010/declarations/' + varDeclarationRef + '/approve',
                method: 'POST'
        })
        .then((response) => {
            expect(response.status).to.eq(204);
        })
    })

    it('should return approved status', () => {
        cy.readFile('cypress/fixtures/declaration_status.xml')
          .then(text => text.replaceAll('$varDeclarationRef', varDeclarationRef))
          .then(text => cy.fetchXML(text))
          .then((response) => {
                const xml = Cypress.$.parseXML(response.body)
                Cypress.$(xml).each(function() {
                    expect(Cypress.$(this).find('declarationRef').text()).to.eq(varDeclarationRef);
                    expect(Cypress.$(this).find('status').text()).to.eq('APPROVED');
                    expect(response.status).to.eq(200);
                })
        })
    })

    it('should return declaration reference in approved declarations list', () =>  {
        cy.request({
                url: 'http://dev.genmed.ut.ee:8010/declarations/EE:COM:CYPRESS:LOCAL/approved',
                method: 'GET'
        })
        .then((response) => {
            expect(response.status).to.eq(200);
            expect(response.body).to.include(varDeclarationRef);
        })
    })
```

```
    it('should not approve declaration if already approved', () =>  {
        cy.request({
                url: 'http://dev.genmed.ut.ee:8010/declarations/' + varDeclarationRef + '/approve',
                failOnStatusCode: false,
                method: 'POST'
        })
        .then((response) => {
            expect(response.status).to.eq(404);
            expect(response.body).to.not.be.null;
            expect(response).property('body').to.contain({
                  title:
                  'Not found: the requested data-usage declaration in
                    REQUESTED state was not found on the server.'
            })
        })
    })

    it('should not approve declaration if invalid declaration reference', () =>  {
        cy.request({
                url: 'http://dev.genmed.ut.ee:8010/declarations/INVALID/approve',
                failOnStatusCode: false,
                method: 'POST'
            })
        .then((response) => {
            expect(response.body).to.not.be.null;
            expect(response.status).to.eq(404);
            expect(response).property('body').to.contain({
                  title: 'Not found: the requested page was not found on the server.'
            })
        })
    })

    it('should not approve declaration if non-existent declaration reference', () =>  {
        cy.request({
                url:
                'http://dev.genmed.ut.ee:8010/declarations/00000000-abcd-123b-456c-123a456b789/approve',
                failOnStatusCode: false,
                method: 'POST'
        })
        .then((response) => {
            expect(response.status).to.eq(404);
            expect(response.body).property('title').to.contain('Not found');
        })
    })
})
```

II Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Janne Sokk**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **„Geeniandmete infosüsteemi testimine Cypress raamistikus“,** mille juhendaja on **Sulev Reisberg PhD**, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Janne Sokk

13.05.2021