TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduste instituut

German Mumma 152889IABM

AUTOMAATTESTIMISE PROJEKTI GENERAATOR VEEBIRAKENDUSTE VASTUVÕTUTESTIMISEKS

magistritöö

Juhendaja: Maili Markvardt
Magistrikraad

Tallinn 2018

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: German Mumma

03.01.2018

Annotatsioon

Käesoleva töö eesmärk oli analüüsida automaattestimise projektiga alustamise probleeme ning pakkuda välja lahendus. Välja pakutud lahendus on automaattestimise projekti generaator, mis põhineb Yeoman generaatoril. Automaattestimise projekti generaator on käsurea rakendus, mis võimaldab generaatori kasutajal koostada automaattestimise projekt erinevas programmeerimiskeeles, kasutades erinevaid automaattestimise vahendeid. Täiendavalt lisab generaator näidistestid ja abiklass, millest kasutaja saab juhinduda oma automaattestide kirjutamisel.

Töö käigus arendati välja ka prototüüp, mille kasulikkust hinnati A/B testimise meetodiga. Meetod näitas, et prototüübi kasutamisel väheneb automaattestimise projekti koostamise ajakulu umbes 60%. Vaatamata sellele, vajab prototüüp edasi arendust, et sellest saaks terviklik lahendus, mida saab kasutada mitmete erinevate tarkvara testimise ja automaattestimise nõudmiste rahuldamiseks.

Lõplik lahendus avalikustatakse GPL (General Public License) litsensi all (versioon 3 või hilisem versioon).

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti viiskümne viiel leheküljel, seitse peatükki, üheksa joonist, üksteist tabelit.

Abstract

Software test automation project generator for web application user acceptance testing

The purpose of this thesis was to analyze the main problems that occur when getting started with software test automation projects and to propose a possible solution. During the thesis, multiple existing tools were researched and analyzed to understand what were the benefits of using them. A conclusion was made that currently no single tool exists that could meet the diverse software testing and test automation needs of a development project.

To address the issue, a solution was proposed in the form of a test automation project generator. The generator was developed using JavaScript programming language and was based on the Yeoman generator API. Using the rich running context built inside of the Yeoman generator, a test automation project generator could be developed which integrates user interaction for selecting required tools for software automation, install and configure the selected tool set, generate the project structure and additional files. Additional files are example tests and a helper class which together can be used as guidelines for developing custom automation tests using the selected tools for the test automation project.

To verify the usefulness of the generator a prototype was developed with reduced functionality- the prototype supported only Java based automation tools and some checks like validating whether the web browser automation tool works with the latest web browser version, were not implement. The verification method used was the A/B testing method where a group of people with IT experience were asked to create a test automation project with a certain tool set and automate a use case. Half of the participants were instructed to use the prototype to generate the project and the other half had to create the project manually. The result showed that participants who used the prototype spent 60% less time on creating working test automation project with a passing automated test.

Although the prototype has proven its usefulness it needs further development for the tool to meet the diverse needs of software testing and test automation. Support for programming languages like JavaScript and Python has to be added and automation tools which are based on those programming languages has to be added. Version management for updating the tools used in a generated project, example files and the generator itself. A website where all the information about the generator can be displayed is also needed.

The thesis is in Estonian and contains fifty-five pages of text, seven chapters, nine figures, eleven tables.

The finished solution will be released under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

Lühendite ja mõistete sõnastik

ISTQB	International Software Testing Qualifications Board.
Testilugu (ingl. keeles *test case*)	Tingimuste ja muutujate komplekt, millega kontrollitakse, et testitav süsteem vastab nõuetele.
Testimeetod (ingl. keeles *test method*)	Ühte meetodisse kuuluv programmi kood. See on protseduur, mis toob esile teatud testi tulemust.
WebDriver/Driver	Programmeerimise liides, mille kaudu saab kaugjuhtimisega mõjutada veebilehitseja käitumist.
JSON	JavaScript Object Notation.
Selenium	Avatud lähtekoodiga projekt veebilehitsejate automatiseerimise rakenduste arendamiseks.
NPM	Node Package Manager.
API	Application Programming Interface

Sisukord

1 Sissejuhatus 11
2 Taust 13
2.1 Tarkvara testimine 13
2.2 Tarkvara testimise protsess 13
2.3 Automaattestimine 15
2.4 Automaattestimine kvaliteedi tagamisel 16
2.5 Automaattestimine kasutajaliidesega 18
2.6 Automaattestimise protsess 18
2.6.1 Skoop 19
2.6.2 Disain 19
2.6.3 Implementatsioon 20
2.6.4 Teostamine 21
2.6.5 Valideerimine 21
2.6.6 Raporteerimine 21
2.7 Levinud probleemid automaattestimisega alustamisel 22
2.7.1 Disain 22
2.7.2 Implementatsioon 23
2.7.3 Teostamine 26
2.7.4 Valideerimine 27
2.7.5 Raporteerimine 28
2.7.6 Robustse automaattestimise raamistiku arendamine 28
3 Automaattestimise projekti generaator 29
3.1 Lahenduse kirjeldus 30
3.1.1 Generaatori töö põhimõte 33
3.1.2 Eeltingimuste kontrollimine 36
3.1.3 Automaattestimise vahendite kontroll 37
3.1.4 Kasutaja sisendi küsimine 38
3.1.5 Konfiguratsioonifaili koostamine 39

3.1.6 Projekti struktuuri loomine 41
3.1.7 Vahendite seadistamine 42
3.1.8 Abiklassi genereerimine 42
3.1.9 Näidistestide genereerimine 45
4 Prototüüp 46
4.1 Prototüübi piiratud funktsionaalsusega etapid 46
4.2 Prototüübi piiratud automaattestimise vahendite nimekiri 49
5 Automaattestimise projekti generaatori prototüübi testimine 52
5.1 Prototüübi testimise põhimõte 52
5.2 Prototüübi testimise tulemused 54
5.3 Automaattestimise projekti generaatori prototüübi kasulikkuse analüüs 56
6 Edasised tegevused 58
6.1 Generaatori versioonihaldus 58
6.2 Programmeerimiskeele toe täiendamine 60
6.3 Ühtne mall näidisfailide genereerimiseks 60
6.4 Projekti koduleht dokumentatsiooniga 61
7 Kokkuvõte 62
Kasutatud kirjandus 64
Lisa 1. Kasutajaga sisendi küsimiseks kasutatud küsimuste tüübid 67
Lisa 2. Generaatori küsimused koos osade valikvastustega 68
Lisa 3. A/B testimise ülesanne 69

Jooniste loetelu

Joonis 1. Automaattestimise projekti genereerimise protsessidiagramm. 32
Joonis 2. Automaattestimise projekti generaatori klassidiagramm. 34
Joonis 3. Automaattesti klassi ja abiklassi protsessidiagramm. 44
Joonis 4. Prototüübi automaattestimise projekti genereerimise protsessidiagramm. 49
Joonis 5. A/B testimise tulemused.. 56
Joonis 6. Generaatori versioonihalduse protsessidiagramm. 59

Tabelite loetelu

Tabel 1. Põhimõtteline tarkvara testimise protsess. 14
Tabel 2. Automaattestimise raamistikud ja nende täiendatud võimekused. 24
Tabel 3. Automaattestimise projekt generaatori eesmärgid. 29
Tabel 4. Automaattestimise generaatori projekti genereerimise protsessi etapid. 30
Tabel 5. Automaattestimise projekti generaatori tegevused Yeoman generaatori kontekstis. 33
Tabel 6. Automaattestimise projekti generaatori abifailid ja nende kirjeldus. 33
Tabel 7. Automaattestimise projekti generaatori küsimused ja nende tüüp. 38
Tabel 8. Konfiguratsioonifaili parameetrid, nende tüüp ning kasutus. 39
Tabel 9. A/B testimises osalejate tulemused variandiga A. 55
Tabel 10. A/B testimises osalejate tulemused variandiga B. 55

1 Sissejuhatus

Tarkvara automaattestimine on viimastel aastatel muutunud populaarsemaks ning selle tähtsus tarkvara kvaliteedi tagamisel on suurem kui kunagi varem. [15] Uute veebiarenduse tehnoloogiate valguses muutub automaattestimine keerulisemaks, sest veebirakenduste automatiseerimise vahenditelt oodatakse, et nad oleksid suutelised uute tehnoloogiatega sammu pidada. Viimastel aastatel on turule ilmunud mitmeid veebirakenduste automatiseerimise raamistikke, mis püüavad lihtsustada tarkvara automaattestimist. Reaalsus näitab, et projekti nõudmised võivad olla liiga keerulised, et neid katta ühe automaattestimise raamistikuga või avastatakse, et projekti alguses valitud raamistik vajab liiga palju erilahenduste arendamist ning otstarbekam oleks minna üle mõne teise raamistiku peale.

Automaattestimise projekti koostamisel kulub suur osa ressurssidest sobivate vahendite uurimisele, valitud vahenditega tutvumisele ning vahendite seadistamisele. Eriti palju aega kulub seadistamisele, sest iga vahend vajab eelkõige individuaalset tähelepanu ning seejärel tuleb need vahendid omavahel toimima saada.

Käesoleva töö käigus analüüsiti veebirakenduste automaattestimise projektis esinevaid probleeme ja uuriti mitmeid erinevaid automaattestimise vahendeid (veebirakenduse automatiseerimise, ühiktestimise, valideerimise ja raporteerimise raamistikud) ning kuidas neid vahendeid kombineerides saab automaattestimisega seotud probleeme leevendada. Sellest tulenevalt oli töö eesmärgiks koostada automaattestimise projekti generaatori, mis võimaldaks:

- ühendada automaattestimiseks vajalikud vahendid ja raamistikud
- automatiseerida automaattestimiseks vajalike vahendite ja raamistike ning vajalike lisavahendite paigaldamist ja seadistamist
- integreerida automaattestimise parimad praktikad projekti algusest peale

Tegemist on käsirearakendusega, mis on arendatud programmeerimiskeeles JavaScript. Rakendus küsib kasutajalt rida automaattestimist puudutavaid küsimusi, mille järel

genereeritakse kasutajale valmis automaattestimise projekt kasutaja valitud automaattestimise vahenditega. Töö tulemusena arendati valmis automaattestimise projekti generaatori prototüüp veebirakenduste testimiseks. Prototüüp toetab Java-põhiseid automaattestimise raamistikke ning prototüübi kasulikkuse hindamiseks viidi läbi A/B testimine.

Töö on jaotatud kuueks peatükiks. Esimeses peatükis antakse ülevaade kogu tööst ning püstitakse töö eesmärk. Teises kirjeldatakse töö taust: seatakse kontekst, tutvustatakse automaattestimise valdkond ning uuritakse levinud probleemid automaattestimisega alustamisel. Kolmandas peatükis pakutakse välja võimalik lahendus teises peatükis käsitletud probleemidele. Neljas peatükk annab ülevaate töö käigus arendatud prototüübist ja viiendas peatükis kirjeldatakse läbi viidud eksperiment prototüübi kasulikkuse hindamiseks. Kuuendas peatükis seatakse projekti edasised sammud ehk pannakse paika tegevuskava, kuidas jõuda prototüübis tervikliku lahenduseni. Viimases peatükis võetakse töö kokku.

2 Taust

2.1 Tarkvara testimine

Tarkvara kvaliteedi tagamine on tarkvara loomise alamvaldkond, mis mõõdab kui hästi tarkvara on disainitud ning kui hästi on tarkvara kooskõlas selle disainiga. Kvaliteet on tase, mis näitab komponendi, süsteemi või protsessi kooskõlastatust kindlalt paika pandud nõudmistega ja/või kasutaja/kliendi vajaduste ning ootustega [1]. Tarkvara kvaliteeti aitab tagada tarkvara testimise distsipliin.

Tarkvara testimine panustab tarkvara kvaliteedi tõstmisesse läbi defektide tuvastamise ning tuvastatud defektide raporteerimise [2]. Tarkavara testimine on tarkvara kasutamise protsess tuvastamaks selle teadaolevad või veel avastamata defektid.

Avastamata defektid on tarkvara vead või kõrvalekalded nõudmistest, mis ei ole veel tarkvara testimise käigus avastatud ja raporteeritud.

Teadaolevate defektide testimise eesmärk on kliendi või testija poolt raporteeritud defekti reprodutseerimine eesmärgiga see klassifitseerida näiteks riski või prioriteedi järgi, või määrata defekti taastekitamise sammud.

Selleks, et tarkvara testimine oleks tõhus eelnevalt kirjeldatud defektide käsitlemises, on soovitatav tarkvara testimisele läheneda struktuurselt ja sihipäraselt, millest räägitakse lähemalt järgnevas peatükis.

2.2 Tarkvara testimise protsess

Tarkvara testimise kõige nähtavam osa on testi käivitamine. Tarkvarale rakendatakse käsklused ning tulemus on inimsilmale nähtav. Selleks, et tarkvara testimine oleks ka efektiivne ja tõhus, tuleb testimise plaani koostamisel arvestada ajaga, mis kulub:

- testide planeerimisele
- testilugude disainimisele

- testide käivitamise planeerimisele
- testi käivitamise tulemuste hindamisele

ISTQB järgi efektiivsust ja tõhusust tõstab struktuurne ning sihipärane lähenemine tarkvara testimisele ehk testimise protsess. ISTQB põhimõtteline testimise protsess koos selle viie põhitegevusega on ära toodud tabelis 1. [1]

Tabel 1. Põhimõtteline tarkvara testimise protsess.

Testimise protsessi tegevus	**Testimise protsessi tegevuse kirjeldus**
Testi planeerimine ja järelevalve	Planeerimise käigus tehakse kindlaks, et kliendi ja projekti eesmärgid ning testimisega seotud riskid on selged.
Testi analüüs ja disain	Üldised testimise eesmärgid muudetakse mõõdetavateks testi tingimusteks ning disainideks.
Testi implementeerimine ja käivitamine	Testi tingimused muudetakse testilugudeks ja -varaks. Seadistatakse ning seatakse üles testkeskkond.
Lõpetamiskriteeriumide hindamine ja raporteerimine	Testide jooksutamine ja selle tulemused hinnatakse vastavalt defineeritud eesmärkide ja mõõdikutele ning koostatakse kokkuvõttev raport projekti huviisikutele.
Testimise lõpetamise tegevused	Lõpetatud testimise tegevuste kohta andmete kogumine kogemuste konsolideerimiseks.

Nimetatud tegevused on loogilises järjestuses, kuid sõltuvalt projektist võivad tabelis välja toodud tegevused seguneda, toimuda samaaegselt ja isegi korduda.

Tarkvara testimine peab olema efektiivne vigade ja rikete tuvastamises, aga ka tõhus testide teostamises võimalikult kiiresti ja odavalt. Tarkvara testimist võib ka nimetada üheks kulukamaks tegevuseks tarkvaraarenduse projektis. Uuringud näitavad, et tarkvara testimise faasi kulud moodustasid üle 50% kogu projekti ressurssidest [3]. Lisaks otsestele kuludele, on tarkvara testimine oluliselt seotud halva kvaliteedi kuludega, kuna talitlushäiretega programm ja tarkvara vead ning nendega tegelemine nõuab olulisi lisakulutusi tarkvara tootjale [4]. Seega tarkvara kvaliteedi ja testimise protsessi efektiivsuse parendamist võib vaadelda kui efektiivset meedet vähendamaks

tarkvara kulud pikemas perspektiivis nii tarkvara arendajatele kui ka selle lõppkasutajatele.

Automaattestimise rakendamine erinevatele tarkvara testimise protsessi tegevustele on üks võimalik lahendus tõstmaks tarkvara testimise efektiivsust. Automaattestimisega saavad testijad pöörata rohkem tähelepanu testitava tarkvara kriitilisele funktsionaalsusele või keerulisematele testilugudele, jättes korduvad või rutiinsed tegevused automaattestimise hoolde.

2.3 Automaattestimine

Automaattestimine on tänaseks omandanud mitmeid tähendusi ja võib sisaldada erinevaid tegevusi. ISTQB järgi võib automaattestimise alla kuuluda järgmised tegevused: [5]

- kasutades otstarbekohaselt ehitatud tarkvara vahendeid testi eeltingimuste kontrollimiseks ja seadistamiseks
- testide käivitamine
- oodatavate tulemuste võrdlemine tegelike tulemustega

Ühelt poolt võib automaattestimine olla nii lihtne kui ühe konkreetse skripti käivitamine ning selle käivituse tagajärgede valideerimine. Teisest küljest võib automaattestimine olla, nagu eelnevalt sai mainitud, nii keeruline kui automaatne keskkonna seadistamine, mingis konkreetses programmeerimiskeeles implementeeritud testilugude komplekti automaatne käivitamine, käivituse vahe- ja lõpptulemuste salvestamine ning eelnevalt defineeritud oodatavate tulemuste vastu valideerimine ja automaatselt koostatud raporti üleslaadimine projektihalduskeskkonda.

Automaattestimine võib märgatavalt vähendada tööd, mis on vajalik saavutamaks adekvaatset testimist. Samuti võib oluliselt suurendada testimise kogust, mida saab läbi viia piiratud ajaga. Testid, mis manuaalselt tehes võtavad minuteid või isegi tunde, saab läbi viia minutite ja isegi sekunditega. [6]

Automaattestimise juurutamine võimaldab vähendada manuaalse testimise kulud koguni 80%. Organisatsioonid, milledel ei õnnestunud automaattestimisega kulusid või tööd

vähendada, on automaattestimine neile võimaldanud toota parema kvaliteediga tarkvara vähema ajaga, kui manuaalse testimisega üksi. [7]

Hästi läbi mõeldud ja juurutatud automaattestimise protsess võimaldab testida tarkvara sisuliselt ühe nupu vajutusega ja nii, et testid käivitatakse automaatselt öösel, kasutades ära perioodi, milles masinad seisavad tegevusetult ja süsteemi aktiivselt ei arendata. Automaattestid on korratavad, kasutades täpselt samad sisendid täpselt samas järjekorras- protseduur, mida manuaalse testimisega ei ole võimalik garanteerida. Automaattestimine võimaldab testida süsteemi täies mahus minimaalse vaevaga ka kõige triviaalsemad süsteemi hooldusega seotud muudatusi.

2.4 Automaattestimine kvaliteedi tagamisel

Kvaliteedi tagamise meeskonna (eriti testija) jaoks üheks probleemiks on arendatava tarkvara pidev muutumine. Tarkvara muutmist ajendavad põhjused võib kokku võtta järgmise kolme juhuga.

Esiteks, tarkvara disaini meeskond, koostöös projekti võtmeisikutega, leiavad, et tarkvara kasutajaliidest on vaja ümber teha, et see vastaks ärivajadustele või kliendi nõudmistele. Teine realistlik põhjus muutusteks tarkvaras on vajadus muuta tarkvara arhitektuuri. Arhitektuuri muudatus võib tuleneda vajadusest vahetada mõne süsteemi komponendi versioon või see täielikult eemaldada või asendada teise komponendiga. Ja kolmandaks, arendusest on tulnud uus tarkvara versioon, mis võib sisaldada: uut funktsionaalsust, ümber tehtud olemasolevad funktsionaalsused ja defektide parandamiseks mõeldud arendused.

Iga kord, kui uus muudatus on implementeeritud, tuleb veenduda, et 1) muudatus töötab nii nagu peab ilma kõrvalnähtudeta ning 2) muudatus pole negatiivselt mõjunud olemasolevatele funktsionaalsustele. Automaattestimine, kui manuaalse testimise abivahendina toob kõige suuremat kasu just viimasena mainitud olukordade testimisel. [9]

Testimine, mille käigus veendutakse, et testitava tarkvarasse lisatud uuendused ja/või muudatused ei ole negatiivselt mõjunud süsteemile terviklikult ja sellest tulenevalt varem implementeeritud ja testitud funktsionaalsuste kvaliteet pole langenud. [1]

Regressioonitestimine võib toimuda kahes tarkvara arenduse faasis: tarkvara arenduses ja tarkvara vastuvõtutestimises. [10]

Agiilse metoodika järgi, toimub arenduses regressioonitestimine enne igat koodi lisamist testkeskkonda. See tähendab, et enne, kui arendaja saab lisada uut koodi kesksesse koodibaasi, peab tema kood läbima nii tema kui ka teiste arendajate kirjutatud ühik- ja integratsioonitestid. Kui kood läbib edukalt nimetatud testid toimub tarkvara versiooni uuendus testkeskkonnas. Siin jõuab järjekord testijani, kes viib läbi vastuvõtutestimist.

Vastuvõtutestimise vältel testitakse lisatud muudatuse vastavust tellitud funktsionaalsuse kirjeldusele, analüüsile, kasutusjuhtudele või muu kliendiga kokkulepitud funktsionaalsuse täpsustus. Uuendus või muudatus on testitud, kui see vastab nõudmistele ning seejärel läbib testija poolt läbiviidud regressioonitestimist. Selles etapis peab testija läbi viima testilugusid, mis on seotud uue funktsionaalsustega, veendumaks, et nende kvaliteet pole langenud. See tähendab, et testija käib läbi varem loodud testid.

Käesolev töö fookus on testija poolt läbi viidud regressioonitestimisele, sest see on põhiline testimise protsessi osa, mille puhul automaattestimine toob kõige rohkem kasu. [11] Automatiseerides testilood, mis kuuluvad regressioonitestide komplekti, on võimalik regressioonile kulutada vähem ressurssi, sest regressioonitestid saab käivitada igal hetkel sisuliselt ühe nupu vajutusega [12] ning tekitades testijale rohkem aega, et teostada põhjalikum testimine süsteemile. Agiilsetes meeskondades ei ole tihti aega kas põhjalikuks regressioonitestimiseks või erinevate keerulisemate testilugude testimiseks. [10] Kui viia läbi vähemas mahus regressioonitestimine võib kliendile jõuda funktsionaalsus, mis varem töötas aga uue muudatusega enam ei tööta. Kui piirneda lihtsate testilugudega (suitsutestimine) ja jätta keerulisemad juhud välja, võib kliendini jõuda defektidega funktsionaalsus. Mõlemad juhud suurendavad tarkvaraarenduse kulusid, sest tuleb viia läbi täiendav testimine.

Käesolevas peatükis toodi välja tarkvaraarenduse ja -testimise protsessi etapid, mille tegevuste automatiseerimine vähendab kulusid tarkvarale või tõstab selle kvaliteeti. Kui tarkvaraarenduse meeskond on jõudnud järeldusele, et automaattestimine on järgmine loogiline samm tarkvara kvaliteedi tõstmiseks, tuleb sellega algus teha.

2.5 Automaattestimine kasutajaliidesega

Käesolev töö kontsentreerub vastuvõtutestimisele, mida viiakse läbi kasutajaliidese kaudu. Tarkvara testimiseks ei ole ilmtingimata vajalik kasutajaliidese olemasolu. [1] Sellele vaatamata on kasutajaliides vajalik, et saada objektiive hinnang vastuvõtutestimise põhjalikkusele, sest kasutajaliides kood võib moodustada 45-60% kogu testitava tarkvara koodist. [14] Kasutajaliidestele mõeldud testide automatiseerimise olulisus on pidevalt kasvanud ning sellega seoses on kasvanud ka vastavad kulud. [15]

Kasutajaliideste testide automatiseerimine on kõige kulukam protsess, mida juurutada ja hallata. Kõrged kulud tulenevalt sellest, et kasutajaliidese testid on haprad. [16] Iga muudatus kasutajaliideses võib negatiivselt mõjuda kõikidele automaattestidele, mis katavad nimetatud muudatust. Automaattestide hallatavus on automaattestimise valdkonna kitsaskoht. Selleks, et automaattestide hallatavusega seotud kulud kontrolli all hoida, tuleb õigesti valida vahend, millega kasutajaliidese testid automatiseeritakse.[6] Käesolev töö ei kata, kuidas kasutajaliideste automatiseerimine täpselt käib, vaid hõlmab vahendeid, millega veebirakenduse kasutajaliidese automatiseerimine viiakse läbi.

Peatükis 2.6 kirjeldatakse põhilised automaattestimisega alustamise etapid, tüüpilised probleemid, millega kohtutakse ning kuidas neid probleeme on varem käsitletud.

2.6 Automaattestimise protsess

Nagu eelnevas peatükis mainiti, saab automaattestimist rakendada nii üksiku testiloo automatiseerimisega (skripti loomisega) kui ka terve protsessi automatiseerimisega. Käesolevas peatükis käsitletakse testimise protsessi tegevusi, mida on võimalik katta automaattestimisega. Kui need automatiseeritavad tegevused grupeerida, siis ilmneb, et tegemist on sisuliselt testilugu etappidega: [6]

1. Skoobi määramine
2. Disainimine
3. Implementeerimine
4. Teostamine

5. Valideerimine
6. Raporteerimine

2.6.1 Skoop

Reaalsus näitab, et kõike automatiseerida on peaaegu võimatu või ei ole majanduslikult ja ka ressursi poolest otstarbekas. [13] Automaattestimiseks head testilood on need, mida on vaja käivitada tihti ja on üksluised (regressioonitestimine) või testilood, mida käsitsi on keeruline läbi viia [6]. Näiteks kasutajaliideses toimingu teostamise tulemusel andmebaasis või teises süsteemis tekkinud muudatuse valideerimine. Sellest tulenevalt on testijal tähtis osata valida, mida automatiseerida ja seejärel valim prioriteedi järgi seada.

2.6.2 Disain

Kui sobilik testilugu on valitud, tuleb otsustada, kuidas seda kõige adekvaatsemalt testida. Automaattestimise mõistes tähendab see vastuse leidmist järgnevatele automaattestimist mõjutavatele küsimustele:

- Mis programmeerimiskeeles testilugu realiseerida?
- Mis lehitsejatega tuleb testida?
- Mis kasutajaliidese automatiseerimise vahendit kasutada?

Programmeerimiskeele valik võib sõltuda nii lihtsast asjaolust kui kogemused. Mida rohkem on testijal kogemust ühe programmeerimiskeelega, seda lihtsam on tal testilugusid realiseerida ning on suurem tõenäosus, et testid on vastupidavamad, hallatavamad ja paremini loetavad.

Veebirakenduste puhul võib olla üks kliendi nõue, et rakendus töötaks kõike populaarsete veebilehitsejate uuemate versioonidega. Organisatsioonilistel põhjustel võib nõue olla ka kindla versiooniga üks veebilehitseja. Sellest küsimusest sõltub nii testilugude hulk, mida vaja realiseerida, aga ka konkreetsete vahendite valik.

Programmeerimiskeele oskus ja vahendi olemasolu ei garanteeri efektiivset automaattestimist. Oluline on osata vahendit õigesti ja õigel eesmärgil kasutada. [21]

2.6.3 Implementatsioon

Järgnevalt on vaja defineeritud testilugu implementeerida valitud vahendiga ja programmeerimiskeelega. Käesolevas etapis tuleb implementeerida järgmised osad, mis on sisuliselt ettevalmistatavad tegevused, mida saab ära teha enne, kui testilugu reaalselt teostatakse: [6]

1. Valida või vajadusel genereerida testandmed
2. Eeltingimused, mis on vajalikud testilugu edukaks automatiseerimiseks
3. Protseduur ehk testi sammud
4. Järeltingimused, mis kontrollivad, et automaattestimise toimingud on korrektselt lõpetatud
5. Oodatavad tulemused

Testandmed kasutatakse veebilehitseja suhtlemisel ja kontrollimisel läbi veebilehitseja automatiseerimise vahendi. Testandmed võivad olla näiteks kasutajanimi ja parool, mida vahendit kasutades sisestatakse sisselogimise vormil. Testandmed võivad tulla failist aga ka andmebaasist. [6]

Eeltingimused võivad kontrollida, et vajalikud testandmete failid on olemas või andmebaasi ühendus on loodud. Samuti võivad eeltingimusteks olla vahendi seadistamine kasutamaks konkreetse lehitseja konkreetset versiooni. [17]

Protseduur on testilugu, mis on jaotatud konkreetseteks käsklusteks, mida veebilehitseja automatiseerimise vahend järgib, kui ta opereerib veebilehitsejaga. Sõltuvalt valitud vahendist võib käskluste kuju olla erinev. Lisaks sellele võivad vahendi käsklused olla üpris tehnilised ning nendest arusaamine vajab kas kogemusi sarnaste vahenditega või vahendi tehnilise dokumentatsiooni tundma õppimist. [17]

Järeltingimuste juures kontrollitakse näiteks, et veebilehitseja sessioon oleks korrektselt lõpetatud, et mitte kulutada liigset protsessori ressurssi. [17]

Oodatavad tulemused kasutatakse protseduuri täitmise järgse süsteemi oleku valideerimiseks. Oodatav tulemus on olukord, mis peab olema peale testiloo täitmist ning tegelik olukord on süsteemi või tarkvara seis peale protseduuri täitmist.

Kui testilugu on implementeeritud ehk see, mis varem teostati manuaalselt on lahti kirjutatud koodis, tuleb valida viis või vahend, kuidas implementeeritud testilugu ehk automaattest käivitada.

2.6.4 Teostamine

Teostamise all on mõeldud automaattesti käivitamist või jooksutamist. Oluline on viis eraldamaks automaattestide kood testitava rakenduse ja automaattestidega otseselt mitte seotud koodist. [17]

Teste saab käivitada üksikult või komplektina, sõltuvalt käivitamise eesmärgist. Automaattestimise vahendi ressursi parema jaotamise eesmärgil võib teste käivitada ka sünkroonselt, asünkroonselt või sõltuvustega teistest testidest.

Lisaks testide eesmärgi pärasele käivitamisele on oluline saada adekvaatne tagasiside testi tulemustest: kas test õnnestus, ebaõnnestus, katkes või jäeti hoopiski vahele puudulike eeltingimuste tõttu.

2.6.5 Valideerimine

Valideerimine sisaldub automaattesti teostamise etapis, võib valideerimise meetodi ja vahendi valik mõjutada nii testide loetavust kui ka validatsioonide efektiivsust. Valideerimise etapis saadakse teada, kas funktsionaalsus on arendatud õigesti ja vastab nõuetele. [17]

Teostamise etapi tulemusel on testitav tarkvara olek muutunud ning see olek tulebki valideerida. Valideerimisel võrreldakse võimalikult täpselt ettevalmistatud või varem defineeritud oodatavad tulemused tegeliku tulemusega testitavas tarkvaras ehk selle muutunud olekuga. Validatsioonide teostamise tulemusel selgub, kas test õnnestub, ebaõnnestub või on testi eeltingimused täitmata ning see test tuleb vahele jätta, sest sellisel juhul ei anna test enam adekvaatset tulemust.

2.6.6 Raporteerimine

Nii nagu manuaalses testimises koostatakse läbiviidud testimise kohta raport, nii on vajalik ka läbiviidud automaattestimise tulemused koguda ja vajadusel ka raporteerida huviisikutele. [1]

Raporteerimine toob läbipaistvust automaattestimisele. Automaattestimise raport annab ülevaadet selle kohta, kui suur osa funktsionaalsusest või testilugudest on kaetud automaattestidega ja millised neist ei vasta nõuetele. Lisaks saab kasutada automaattestimise raportit, et näidata automaattestimise kasulikkust, tuues välja automatiseeritud testilugude arv, testilugude teostamise sagedus ja nende teostamiseks kulunud aeg. [17]

2.7 Levinud probleemid automaattestimisega alustamisel

Käesolevas peatükis vaadeldakse põhilised probleemid, millega puututakse kokku automaattestimisega alustamisel ning uuritakse, kuidas neid probleeme on varem adresseeritud või kuidas automaattestimise kogukond on varem probleemile lähenenud.

Käesolev töö käsitleb probleeme tuginedes automaattestimise protsessi etappidele, mis on kirjeldatud peatükis 2.6, välja arvatud esimene etapp ehk skoobi määramine. Kuigi skoobi määramist on võimalik automatiseerida ja selle kohased uuringud on varem tehtud [33], on autori isiklik arvamus, et automaattestimiseks sobiliku skoobi ehk testilugude valik tuleb läbi viia käsitsi.

Käsitletud probleemid on valitud tuginedes peamiselt autori isiklikele kogemustele ning automaattestimise kogukonnas kogutud infole. [22]

2.7.1 Disain

Programmeerimisekeele osas on soovitatav arendada automaatteste samas programmeerimiskeeles, milles on arendatud testitav tarkvara. [17] Piltlikult öeldes, testid räägivad sama keelt, mida testitav tarkvara räägib. Sellises olukorras võib testija saada tagasisidet testide kohta kogenumatelt arendajatelt ning selle kaudu paraneb ka automaattestide kvaliteet.

Nagu paragrahvis 2.6.2 käsitleti, sõltub vahendi valik programmeerimiskeele ja veebilehitseja valikust. Veebilehitsejate tugi erineb sõltuvalt vahendist. Samamoodi on vahendite programmeerimiskeele tugi varieeruv. [18] Näiteks, ainuüksi JavaScript-il on rohkem kui 10 erinevat veebirakenduse automaattestimise vahendit. [19][20]

Üks põhilisi küsimusi, mida automaattestimisega alustamisel küsitakse ongi: mis vahendit või vahendeid kasutada? Pole olemas vahendit, mis lahendab kõiki probleeme. Seevastu on mitmeid sarnaseid vahendeid, millede kasutamine on põhjendatud teatud olukordades. [12] Vahend, mille kasutamine ebaõnnestus ühes projektis, võib õnnestuda mõnes teises projektis. Põhilisteks probleemideks vahendite ebaõnnestumisel on vahendi piiratud võimekus toimetulemaks kaasaegsetel tehnoloogiatel (nagu NodeJS, AngularJS jms) põhinevate veebirakendustega. [23]

Üks lahendus on kasutada vahendit, millel on sisseehitatud võimekus vastavate tehnoloogiatega töötamiseks nagu Protractor [34] või mõni muu JavaScript-il põhinevat veebirakenduste automaattestimise raamistikku.

Alternatiivne lahendus on valida selline vahend, mida saab kasutada automaattestides, mis on implementeeritud samas programmeerimiskeeles, mis testitav rakendus. See võimaldab kaasata arendajaid samast projektist, et koos luua robustse lahendus, mis sobib konkreetse probleemi ületamiseks [17].

2.7.2 Implementatsioon

Sobiliku veebilehitseja automaattestimise vahendi valikust ainult ei piisa. Nii nagu tarkvaraarenduses on mitmed erinevaid raamistike ja vahendeid (näiteks Java Spring, Maven, Java Development Kit, Angular jms) vajavad mingis ulatuses seadistamist enne, kui neid saab sihipäraselt kasutada, nii vajavad ka automaattestimise vahendid seadistamist. [24]

Mõned seadistused, mida tuleb läbi viia on toodud järgmises listis. List põhineb Selenium WebDriver seadistamise juhendil operatsioonisüsteemis Windows: [24]

- Vahendi allalaadimine kohalikku arenduskeskkonda
- Veebilehitseja spetsiifiliste failide allalaadimine ja vahendi seadistamist kasutamaks soovitud veebilehitsejat testide teostamisel
- Täiendavate teekide allalaadimine (näiteks Selenium Support, mis täiendab Selenium WebDriver raamistikku või Hamcrest inimloetavate validatsioonide kasutamiseks), mis täiendavad automaattestimise vahendit.

Robustsest automaattestimise projektist rääkides tuleb eelnimetatud vahendid ja seadistused hoida tsentraalselt, et neid oleks võimalik hallata (näiteks Apache Maven-i pom.xml fail).

Seega seisneb implementatsiooni etapi keerukus rohketes seadistustest ja täiendavate failide vajadusest. Implementatsiooni etapis tuleb paigaldada ja seadistada ka teostamise, valideerimise ja raporteerimise etappideks vajalikud vahendid.

Tänaseks on turule ilmunud automaattestimise vahendid või raamistikud, mille puhul on aega nõudvad seadistused kasutaja eest ära tehtud või on seadistused tehtud lihtsamaks. Tabelis 2 on toodud kaasaegsed automaattestimise raamistikud ning nende laiendatud võimekused.

Tabel 2. Automaattestimise raamistikud ja nende täiendatud võimekused.

Vahend	**Paigaldamine**	**Lehitsejate seadistus**	**Valideerimine**	**Raporteerimine**
Serenity BDD	Mitmed sõltuvused Maven või Gradle konfiguratsiooni failis	Integreeritud	Läbi täiendavate teekide	Integreeritud
Nightwatch.JS	Allalaaditav Node Package Manager-ist	Integreeritud	Integreeritud	Läbi täiendavate teekide
Protractor	Allalaaditav Node Package Manager-ist	Integreeritud	Integreeritud	Läbi täiendavate teekide
WebDriverManager	Üks sõltuvus Maven või Gradle konfiguratsiooni failis	Integreeritud	Läbi täiendavate teekide	Läbi täiendavate teekide
Conductor	Üks sõltuvus Maven või Gradle konfiguratsiooni failis	Läbi täiendavate teekide	Integreeritud	Läbi täiendavate teekide

Vahend	Paigaldamine	Lehitsejate seadistus	Valideerimine	Raporteerimine
Selenide	Üks sõltuvus Maven või Gradle konfiguratsiooni failis	Läbi täiendavate teekide	Integreeritud	Integreeritud

Tabelis on toodud vaid mõned näited uutest raamistikest. Iga raamistikul on omad tugevused ja nõrkused. [17] Sellest tulenevalt võib neil olla erinev paigaldamise ja seadistamise protsess.

Hallatavus on suurim kulu ja ressursi allikas automaattestimise projektis. [8] [17] [21] Mida tihedamini tuleb automaatteste või projekti seadistusi muuta ja mida keerulisem on neid muudatusi teha, seda keerulisem on automaattestimise projekti hallata.

Hallatavusega seotud automaattestimise projekti aspektid on järgmised:

- Vahend ei tööta enam peale veebilehitseja versiooni uuendamist
- Testitav tarkvara on muutunud ja negatiivse tagajärjena automaattestid ei tööta enam
- Parimate praktikate ja disainimustrite mitte rakendamine

Veebilehitseja versiooni uuendamine ja sellest tulenevalt ka kasutajaliidese automaattestimise vahendi funktsionaalsuste ebatäpsus või nende katki minemine on vältimatu. Selleks, et testid uuesti töötaksid tuleb kas automaattestimise vahend uuendada või seadistada vahend kasutama veebilehitseja spetsiifilist versiooni. Nimetatud probleem ilmneb alles siis, kui testid on teostatud peale veebilehitseja uuendamist. Alles seejärel saab uurima hakata, miks testid katki läksid. Tänapäeval pole olemas vahendit, mis suudaks automaatselt teavitada, et kas vahendi või veebilehitseja uuenduse tulemusel võivad testid katki minna.

Teine probleem on hõlmab kaks viimast aspekti. Nimelt testitava rakenduse muutuste tagajärjel on testid katki läinud ja vajalikud muudatused automaattestides tuleb manuaalselt muuta. Automaattestimise vahendid sõltuvad veebilehtede elementide unikaalsest või võimalikult täpsest identifikaatorist testitavas rakenduses. Sellest tulenevalt, kui testitavat rakendust muudetakse võib see identifikaator muutuja ja

automaattestimise vahend ei pruugi enam opereerida õige elemendiga või läheb üldse katki, sest ta ei leia ühtegi elementi, mis vastaks sellisele identifikaatorile. Vigased identifikaatorid tuleb tuvastada ja parandada kõikides kohtades, kus nad kasutatud on.

Sellise probleem lahendamiseks on välja pakutud mitmeid lähenemisi. Kõige lihtsam oleks lisada kõik identifikaatorid niinimetatud objektide hoidlasse (inglise keeles *Object Repository*). Automaattestid arendatakse nii, et identifikaator küsitakse hoidlast. Selle tulemusel, kui mõni identifikaator peaks muutuma, siis tuleb muudatust teostada vaid ühes kohas. [26]

Kõige populaarsem ja tänaseks välja kujunenud parimaks praktikaks kasutajaliideste automaattestimisel on *Page Object Pattern* disainimustri rakendamine. [17] Disainimustri põhimõte seisneb selles, et testitava tarkavara veebilehed, nendel asuvad väljad ja vormid modelleeritakse eraldi objektidena eelistatud programmeerimiskeeles. Kõik objektidega seonduv nagu identifikaator, alternatiivsed identifikaatorid, veebilehe elemendi tüüp ja muud objektile iseloomulikud omadused on hoitud isoleeritult selles konkreetses objektis. Nii nagu objekt orienteeritud disainimustris modelleeritakse reaalse maailma objekte programmikoodis, nii ka selle lähenemise puhul modelleeritakse reaalset rakendust programmikoodis.

2.7.3 Teostamine

Nii nagu arendustestid käsitletakse ja hoitakse rakenduse koodist eraldi, tuleb ka automaattestide kood hoida ja märgistada. Märgistamisega antakse automaatteste käivitatavale süsteemile teada, et seda koodi osa tuleb käsitleda kui testi. See tähendab, et selles koodis on olemas tingimused, mille täitmisest või mitte täitmisest sõltub, kas test lugeda õnnestunuks või ebaõnnestunuks.

Populaarne viis automaattestide teostamiseks on neid käivitada kasutades ühiktestimise raamistike ehk xUnit raamistike nagu JUnit, NUnit, QUnit jne. Kuigi ühiktestimise raamistikud originaalis kasutatakse arendajate poolt konkreetse rakenduse komponendi või ühiku testimiseks võimaldab automaattestide integreerimine ühiktestimise raamistikuga tekitada lihtsasti käivitatavat automaattestimise komplekti. Lisaks annab see ka parema ülevaate automaattestide käivitamise tulemustest (kas test õnnestus, ebaõnnestus või jäeti vahele). [16]

Ühiktestimise raamistiku valikut teeb keeruliseks nende kogus ning funktsionaalsuste erinevus. Näiteks programmeerimiskeeles Java on üle 10 erineva ühiktestimise raamistiku [28].

Kuigi automaattestimine on oma olemuselt samuti tarkvara arendamine, vajab automaattestimine teistsugust funktsionaalsust. Programmeerimiskeeles Java võimaldavad annotatsioonid kirjutada vähem koodi, et saavutada soovitud tulemus ning üldiselt vähendab seadistamise vajadust. Näiteks annotatsiooniga @Test märgitakse meetod, mis on tegelikkuses test ja seda peab käivitama programmi ülesehitamisel testimise faasis. [29]

2.7.4 Valideerimine

Automaattestide implementeerimisel võib kohati olla keeruline leida õige tasakaal testi detailsuses. Liialt detailsed testid on haprad ja neid on keeruline muuta. Vähese detailsusega testid võivad läbida validatsioone ka juhtudel, kui testitav osa on katki. Selliste olukordade lahendamiseks peab teadma, mis tarkvara aspekti testida ning määrata sobilik täpsusaste, mis sobib selle aspekti käitumisega. [27] Näiteks, kuupäevade valideerimiseks sobib üks lähenemine (kuupäevade formaat, ajatsoon jne.), numbrite jaoks aga teine (komakohtade arvud, negatiivsed numbrid jne.).

Sobiliku lähenemisega validatsioonidega testid ebaõnnestuvad ainult siis, kui testitava tarkvara aspekti käitumine väljub lubatud piirist. Samal ajal, vähese tähtsusega muudatused aspekti käitumises ei mõjuta testi validatsioonide tõhusust ja selle tagajärjel ka testi tulemust.

Tänapäeval on tüüpiline kasutada mõnda valideerimise vahendit või raamistikku toetamaks oodatavate tulemuste võrdlemist tegelike tulemustega testide kirjutamisel. Üks populaarsemaid validatsiooni raamistikke, Hamcrest koosneb mitmetest validatsiooni klassidest, mis lihtsustavad teksti, numbrite ja kuupäevade võrdlemist, erinevate listide võrdlemiseks ja kontrollimiseks mõeldud klassidest ning lisavad loetavust testi koodile. [27] Automaattestide koodi loetavuse tõstmine parendab automaattestide hallatavust. Mida loetavamad on testid, seda lihtsam on nende eesmärgist ja põhimõttest aru saada ning hiljem ka parandada. [17]

2.7.5 Raporteerimine

Raporteerimise tugi sõltub programmeerimiskeelest, ühiktestimise raamistikust, aga ka automaattestimise eesmärgiks: kas eesmärk on tuvastada vigu, anda ülevaadet äripoolele tarkvara kvaliteedist või mõlemad. Viimane sõltuvus määrab ära, kui põhjalik ja äriloetav peab automaattestimise raport olema.

Tänaseks on turul olemas erinevaid raporteerimise vahendeid, mis on integreeritavad automaattestidega ja koostavad inimloetavaid raporteid. Need raporteerimise raamistikud koguvad automaattestide teostamise vältel andmeid testide tulemuste kohta ning teostamise lõpuks genereerivad üleantavat raportit.

2.7.6 Robustse automaattestimise raamistiku arendamine

Robustse automaattestimise raamistiku arendamist peetakse automaattestimise valdkonnas parimaks praktikaks, mis muudab automaattestimise projekti kõrgetasemeliseks ja hallatavaks. [17] Eelnevates peatükkides kirjeldatu, kuidas igas automaattestimise protsessi etapis võib ette tulle probleeme või mõtte kohti, millede lahendamine võib võtta märkimisväärselt aega ja ressurssi. Pürgida tervikliku automaattestimise projekti ehk raamistiku poole tähendab tegelemist kõikide nimetatud probleemidega. Lisaks tuleb asjaolu, et kõik automaattestimise projektis kasutatud vahendid tuleb integreerida omavahel, loomaks terviklikku vahendite ökosüsteemi.

On olemas tasulise litsentsi või teenustasudega vahendeid, millede eesmärk on vähendada erinevate võimalike vahendite olemasolu. Nendes vahendites on olemas samad funktsionaalsused ja seadistamine on sama lihtne nagu tavalise programmi paigaldamine arvutis. Üheks selliseks näiteks on *Unified Functional Testing*, milles on olemas funktsionaalsused katmaks kogu automaattestimise protsessi kasutades ühte automaattestimise keskkonda. [30]

Teisel pool spektrit on tasuta vahendid, millede funktsionaalsused jäävad alla tasulistele vahenditele, kuid integreerides täiendavad vahendid on võimalik need puudused likvideerida. See, aga, nõuab omakorda aega ja oskusi.

3 Automaattestimise projekti generaator

Käesoleva töö autor pakub välja lahenduse, kuidas automatiseerida projekti koostamist veebirakenduste automaattestimiseks. Tänapäevase tehnoloogia ja vahenditega on võimalik integreerida erinevad automaattestimises kasutatud vabavaralised raamistikud, automaattestimise parimad praktikad ja kasutaja juhendamist automaattestimise projekti koostamisel. Lähtuvalt sellest, on käesoleva töö üheks tulemiks kasutajaliidese automaattestimise projekti generaator. Generaatori eesmärgid, eesmärkidega seotud automaattestimise probleemid ning eesmärkide mõõdikud, mille vastu valideeritakse valmis lahendus, on toodud tabelis 4.

Tabel 3. Automaattestimise projekt generaatori eesmärgid.

Eesmärk	**Soetud probleemid**	**Mõõdikud**
Integreerida genereeritavasse projekti automaattestimine.	Automaattestide hallatavus (vaata peatükk 2.7.2)	Saab valida ja kasutada disainimustreid *Object Repository* ja *Page Object Pattern.*
Integreerida levinud kasutajaliidese automatiseerimise raamistikud erinevates programmeerimiskeeltes.	Automaattestide arendamiseks sobiliku kasutajaliidese automatiseerimise raamistiku valik (vaata peatükk 2.7.1)	Saab valida ja kasutada erinevaid raamistikke.
Integreerida levinud ühiktestimise raamistikud erinevates programmeerimiskeeltes.	Automaattestide teostamiseks sobiliku ühiktestimise raamistiku valik (vaata peatükk 2.7.3)	Saab valida ja kasutada erinevaid raamistikke.
Integreerida levinud valideerimise raamistikud erinevates programmeerimiskeeltes.	Automaattestide validatsioonide koostamiseks sobiliku raamistiku valik (vaata peatükk 2.7.4)	Saab valida ja kasutada erinevaid raamistikke.
Projektis kasutatud vahendid on seadistatud projekti genereerimisel.	Automaattestimise projektis kasutatud vahendite seadistamine (vaata peatükk 2.7.2)	Generaator automaatselt seadistab kõige uuemad vahendite versioonid ja vajadusel paigaldab täiendavad ressursid.

Eesmärk	Soetud probleemid	Mõõdikud
Integreerida raporteerimise vahendid erinevates programmeerimiskeeltes.	Automaattestide teostamise tulemuste kogumiseks raporti koostamine (vaata peatükk 2.7.5)	Saab valida ja kasutada erinevaid raamistikke.
Juhendada kasutajat projekti koostamisel.	Tervikliku automaattestimise raamistiku arendamiseks nõutud ajakulu (vaata peatükk 2.7.6)	Generaator kuvab kasutajale vajaliku sisendi kirjeldust ning selle sisendile vastavad valikuvõimalused.

3.1 Lahenduse kirjeldus

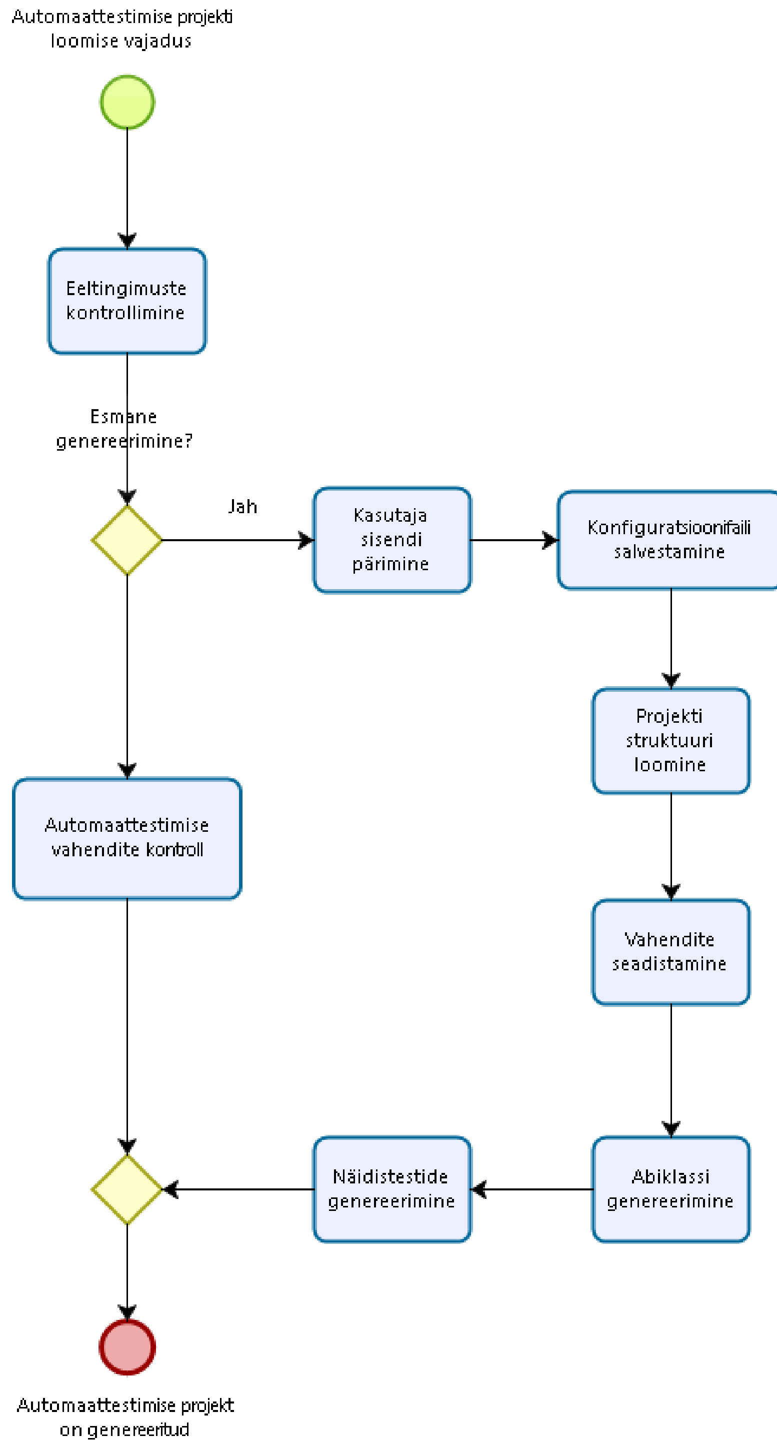
Automaattestimise projekti generaator on oma olemuselt käsurearakendus, mis küsib kasutajalt küsimusi ja võimaldab kasutajal kas sisestada vabatekstiline sisend või valida sobivad sisendid valikvastuste hulgast. Kasutaja sisendid kasutatakse seadistatud automaattestimise vahenditega projekti genereerimiseks.

Lähtuvalt generaatorile püstitatud eesmärkidest ja automaattestimise probleemidest, mida generaator peab katma, on automaattestimise projekti generaatorile defineeritud ülesanded. Ülesanded saab grupeerida etappideks ning seada need loogilisse järjekorda, tekitades automaattestimise projekti genereerimise üldise protsessi. Protsessi etapid koos ülesannetega on toodud tabelis 4. Üldine automaattestimise projekti genereerimise protsess on toodud joonisel 1.

Tabel 4. Automaattestimise generaatori projekti genereerimise protsessi etapid.

Etapp	Ülesanded
Eeltingimuste kontrollimine	Generaatori kasutamiseks vajalike eeltingimuste kontroll (vaata peatükk 3.1.2).
Automaattestimise vahendite kontroll	▪ Kontrollida automaattestimise projektis kasutatud vahendite versioonid ja vajadusel uuendada need. ▪ Kontrollida valitud kasutajaliidese automaattestimise vahendi toimimist kõige uuema veebilehitseja versiooniga.

Etapp	Ülesanded
Kasutaja sisendi küsimine	▪ Küsida kasutajalt automaattestimise projekti puudutavaid küsimusi. ▪ Pakkuda erinevaid vastuse variante koos vaikeväärtustega ehk juhendada kasutajat.
Konfiguratsioonifaili salvestamine	Koostada kasutaja valikute põhjal konfiguratsioonifail.
Projekti struktuuri loomine	Luua projekti kaustade struktuur.
Vahendite seadistamine	Seadistada vahendid ja vajadusel paigaldada täiendavad sõltuvused.
Abiklasside genereerimine	Lisada disainimustritele vastavad abifailid.
Näidisfailide ja –testide genereerimine	Lisada näidisfailid ja –testid.
Genereerimise tulemuste kuvamine	▪ Kuvada kokkuvõte genereeritud projektist koos vahendite ja nende otstarbega ning muu automaattestimist puudutavat informatsiooni ▪ Kuvada genereerimise käigus tekkinud probleemid ja täitmata eeltingimused



Joonis 1. Automaattestimise projekti genereerimise protsessidiagramm.

Yeoman generaatoril on oma kindlalt määratud protsess, mis sisuliselt koosneb erinevatest meetoditest ning igal meetodil on kindel prioriteet. See tähendab, et prioriteetsemad meetodid käivitatakse eelkõige (vaata peatükk 3.1.1).

Automaattestimise projekti generaatori etapid, mis on oma olemuselt samuti JavaScripti meetodid, teostatakse Yeoman generaatori prioriteetidega meetodite sees. Tabelis 5 on välja toodud Yeoman generaatori meetodid koos järjekorraga ning iga meetodi juurde on toodud automaattestimise projekti generaatori meetodid, mis käivitatakse.

Tabel 5. Automaattestimise projekti generaatori tegevused Yeoman generaatori kontekstis.

Meetodi käivitamise järjekord	**Yeoman generaatori meetodid**	**Automaattestimise projekti generaatori meetodid**
1.	Initializing	▪ Eeltingimuste kontrollimine ▪ Automaattestimise vahendite kontroll
2.	Prompting	Kasutaja sisendi küsimine
3.	Configuring	Konfiguratsioonifaili salvestamine
4.	Default	Projekti struktuuri loomine
5.	Writing	▪ Vahendite seadistamine ▪ Abiklassi genereerimine ▪ Näidistestide genereerimine
6.	End	Genereerimise tulemuste (kaasaarvatud tekkinud vigade) kuvamine

3.1.1 Generaatori töö põhimõte

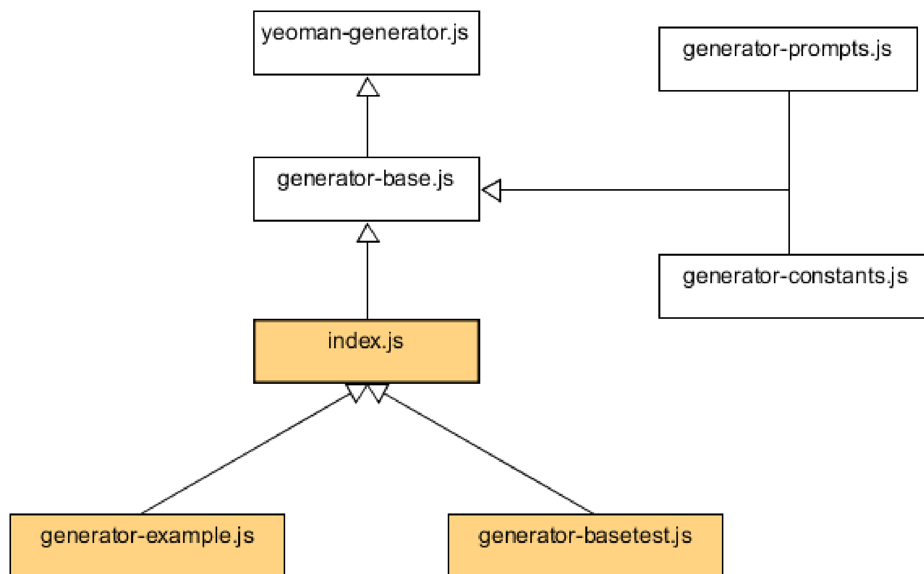
Lahendus põhineb Yeoman generaatoril, mis on vahend sarnaste generaatorite ehitamiseks [31]. Generaator on arendatud programmeerimiskeeles JavaScript ja koosneb peageneraatorist, kahest alamgeneraatorist ning kolmest abifailist. Need failid moodustavad automaattestimise projekti generaatori põhiloogika. Failid koos kirjeldusega on toodud tabelis 6.

Tabel 6. Automaattestimise projekti generaatori abifailid ja nende kirjeldus.

Klass	**Kirjeldus**
index.js	Generaatori põhiklass, mis käivitub generaatori käivitamisega käsurealt. Sisaldab generaatori protsessi etappidele vastavad meetodid. Etappides käivitatavad meetodid päritakse generator-base.js failist.

Klass	Kirjeldus
generator-base.js	Laiendab yeoman-generator.js generaatorit, mille tulemusel pärib generator-base.js standardsed generaatori meetodid ja objektid, mis on vajalikud kohandatud generaatori arendamiseks. Lisaks sisaldab erinevaid abimeetodeid.
generator-basetest.js	Alamgeneraator, mida põhigeneraator kasutab abiklassi genereerimiseks.
generator-example.js	Alamgeneraator, mida põhigeneraator kasutab näidisfailide ja –testide genereerimiseks.
generator-constants.js	Klass, mis hoiab globaalseid objekte ja muutujaid, mida programmi käigus ei muudeta. Sisaldab generaatori poolt toetatud raamistike ja nende paigaldamiseks vajalikud andmed (näiteks Maven puhul *groupId*, *artifactId* ja *version*).
generator-prompts.js	Klass, mis hoiab kasutajaga suhtlemiseks kasutatud küsimusi ja valikvastuseid.

Joonisel 2 on toodud automaattestimise projekti generaatori olulisemad klassid ning nende vahelisi suhteid kirjeldav klassidiagramm.



Joonis 2. Automaattestimise projekti generaatori klassidiagramm.

Automaattestimise projekti generaator on sisuliselt kohandatud rakendus, mis põhineb Yeoman generaatoril. Yeoman generaator ongi mõeldud kohandatud generaatorite arendamiseks ning selle kasutamise eelised on järgmised:

1. Suhtlus kasutajaga
2. Kasutaja konfiguratsiooni salvestamine
3. Suhtlus failisüsteemiga
4. Generaatori protsess on jaotatud kindla prioriteediga etappideks
5. Võimalus kasutada ära olemasolevaid generaatoreid

Suhtlus kasutajaga on oluline kasutajale info (nagu küsimused) ja valikvastuste kuvamiseks ning nende valikute teostamine. Kasutaja poolt tehtud valikud salvestatakse konfiguratsioonifaili ning neid arvestatakse ka peale projekti edukat genereerimist.

Teiseks oluliseks funktsionaalsuseks on suhtlus failisüsteemiga, et luua projekti struktuur, luua soovitud sisuga erineva faililaiendiga faile, käivitada käsurea juhiseid ning vajadusel täiendavate ressursside paigaldamine ja seadistamine.

Yeoman generaatoril on paika pandud etappide järjekord, mis täidetakse prioriteetide järgi. Need etapid koos tegevustega, mida igas etapis teostatakse on [31]:

1. *Initializing* – erinevate eeltingimuste kontrollimiseks, olemasoleva konfiguratsiooni faili lugemine jms.
2. *Prompting* – kasutajale info kuvamine ja sisendi kogumine
3. *Configuring* – konfiguratsiooni salvestamine ja erinevate metaandmete salvestamine
4. *Default* – juurde arendatud meetodid ehk tegevused, mis ei ole osa Yeoman generaatorist käivitatakse selles etapis
5. *Writing* – projekti failide loomine ja salvestamine
6. *Install* – täiendavate ressursside paigaldamine
7. *End* – kasutajale genereerimise protsessi tulemuste ja lõpetuseks info kuvamine

Järgnevates peatükkides kirjeldatakse automaattestimise projekti generaatori genereerimise protsessi tegevused detailsemalt ning märgitakse, mis üleval mainitud etapis need tegevused teostatakse.
Iga generaator Yeoman generaatorite ökosüsteemis on oma olemuselt NodeJS moodul ja käivitatakse käsurealt järgmise käsuga:

```
npm install automation-generator
```

Automaattestimise generaatori käivitamiseks tuleb käsurealt käivitada järgmine käsk, millega käivitub Yeoman generaatori initsialiseerimise etapp:

```
yo automation
```

Märksõna *automation* on automaattestimise projekti generaatori projekti nimetus.

3.1.2 Eeltingimuste kontrollimine

Automaattestimise projekti generaator kontrollib erinevate vajalike tehnoloogiate olemasolu operatsioonisüsteemi tasemel, mis tähendab, et generaator kontrollib nende tehnoloogiate olemasolu kasutaja arvutis. Selle jaoks on kasutatud NPM moodulit *execa*, et käivitada käsurea käske ning töödelda operatsioonisüsteemilt saadud vastust.

Näiteks Java kontrollimiseks käivitab *execa* järgmist käsklus:

```
java –version
```

Operatsioonisüsteemilt saadud vastuses kontrollitakse regulaaravaldisega Java või OpenJDK olemasolu ning selle versioon. Kui Javat ei leita kasutaja arvutis või versioon on vanem kui Java 8, siis kuvatakse kasutajale veateade ja generaator lõpetab oma töö.

Yeoman generaatori initsieerimise etapis kontrollitakse generaatori toimimiseks vajalike tingimuste täitmist. Selleks, et automaattestimise projekti generaator toimiks, peavad olema NodeJS ja NPM paigaldatud ja seadistatud.

Seejärel kontrollib automaattestimise projekti generaator Java ja Python-i olemasolu arvutis. Need on vajalikud, kui generaatori kasutaja eesmärk on kasutada automaattestide arendamiseks vastavalt Java või Python programmeerimiskeelt ja nendel põhinevaid automaattestimise raamistikke. Nende programmeerimiskeelte puudumine süsteemis otseselt generaatori tööd ei takista. Programmeerimiskeele puudumisest teavitatakse kasutajat generaatori töö lõpus.

Lisaks eeltingimustele kontrollitakse ka konfiguratsioonifaili olemasolu. Konfiguratsioonifaili olemasolu tähendab, et tegemist on juba varem valmis genereeritud projektiga. Käivitades generaatorit olemasoleva projekti peal, teostatakse automaattestimise vahendite kontroll. Kui tegemist on aga esmase genereerimisega, siis käivitatakse kasutaja sisendi küsimise etapp.

3.1.3 Automaattestimise vahendite kontroll

Automaattestimise projekti generaatori poolt toetatud vahendite toimimist veebilehitsejaga kontrollitakse pidevintegreerimise vahendiga Jenkins. Automaattestimise projekti generaatori kood valideeritakse erinevate testidega. Ühed sellised testid on toetatud automaattestimise vahendite testid erineva veebilehitsejaga. Kui veebilehitseja uueneb, uuendatakse see ka automaattestimise projekti generaatori koodis. Seejärel käivitatakse testid. Kui testid õnnestuvad, siis tekitatakse generaatori uus versioon. Uus versioon salvestatakse andmebaasi.

Automaattestimise projekti generaatori initsialiseerimise etapis teostatakse generaatori versiooni kontroll. Versiooni kontrollimiseks kasutatakse REST teenust, et tagastada kõige uuem generaatori versioon koos uue versiooniga kaasnevate uuendustega. Generaatori versiooni uuendamisel järgitakse semantilise versiooni halduse põhimõtet. [32]

Järgnevalt on välja toodud erinevad võimalikud uuendused, mis on kajastatud generaatori versiooni uuendamises:

- Kasutajaliidese automatiseerimisvahendi versiooni uuendus
- Automaattestimise abivahendi versiooni uuendus
- Veebilehitseja versiooni uuendus
- Yeoman generaatori uuendus
- Automaattestimise projekti generaatori loogika uuendus:
    - toetatud programmeerimiskeelte täiendamine
    - kasutajaliidese automatiseerimisvahendite täiendamine
    - automaattestimise vahendite täiendamine
    - abiklasside parendamine
    - jõudluse parendus
    - vigade parandus

Versiooni uuenduse käivitamiseks tuleb, olles olemasoleva projekti kõige pealmises kaustas (sama nimega kaust, mis projekt), käivitada uuendamise käsklus, mis paigaldab kõige viimase versiooni:

```
yo automation update
```

Vajadusel võib uuendata generaator kindlale versioonile. See võib olla kasulik, kui automaattestimise projekti genereerimiseks kasutatud generaator on mitu versiooni vana. Selleks, tuleb kasutada järgmist käsklust:

```
yo automation update --target-version=1.2.0
```

Kui initsialiseerimise etapis toimus versiooni uuendus, kuvatakse teostatud uuendused kasutajale generaatori töö lõpus. Kui eeltingimuste valideerimine ja kontrollimine on teostatud algab kasutajalt sisendi küsimine.

3.1.4 Kasutaja sisendi küsimine

Kasutaja küsimise etapis küsitakse kasutajalt rida küsimusi, millele kasutajalt oodatakse kas vabatekstilist sisendit või teatud valikute tegemist. Erinevad lahenduses kasutatud küsimuste tüübid koos näidetega on toodud lisas 1.

Küsimuste progressioon ja valikud võivad sõltuda kasutaja vastustest varem esinenud küsimustele. Generaatori poolt kasutajale esitatud küsimused koos küsimustele vastava sisendid tüübiga on toodud tabelis 7. Tabelis on sisendid toodud eesti keeles, kuid generaatoriga suhtlemine toimub inglise keeles. Kõik generaatoris kasutatud küsimused inglise keeles koos õige järjekorra ja valikvastustega on toodud lisas 2.

Programmeerimiskeele valiku puhul teostab generaator lisaks filtreerimise. Filtreerimise tulemusel kuvatakse kasutajale kõik järgnevad küsimused koos valikvastustega lähtudes valitud programmeerimiskeelest. Näiteks, kui kasutaja valib programmeerimiskeeleks JavaScript, siis kuvatakse kasutajale JavaScript-i põhised ühiktestimise raamistikud (Jasmine, Mocha, Jest, QUnit). Kõikide küsimustele sisendi andmise järgselt salvestab generaator kasutaja sisendid konfiguratsioonifaili.

Tabel 7. Automaattestimise projekti generaatori küsimused ja nende tüüp.

Generaatori poolt esitatud küsimus	**Sisendi tüüp**
Sisestage oma automaattestimise projekti nimi	Vabatekstiline sisend
Vali programmeerimiskeel	Üks kindel sisend
Vali ühiktestimise raamistik	Üks kindel sisend
Vali Selenium-i raamistik	Üks kindel sisend
Vali automaattestimise disainimuster	Üks kindel sisend

Generaatori poolt esitatud küsimus	Sisendi tüüp
Vali valideerimise raamistik	Üks kindel sisend
Vali muud testimise raamistikud, kui on vajadust	Mitu võimalikku sisendit
Vali raporteerimise raamistik	Üks kindel sisend
Vali muud raamistikud, kui on vajadust	Mitu võimalikku sisendit

Kasutajaga suhtlemisel, temalt sisendi saamiseks ja võimalike valikvariantide pakkumisega lisab generaator teatud määral kasutaja juhendamise aspekti. See kompenseerib mõningal määral puuduvad ekspertteadmist ja varasemat kogemust automaattestimisega, vähendades selle kaudu vajalikku ajakulu automaattestimisega alustamisel.

3.1.5 Konfiguratsioonifaili koostamine

Konfiguratsioonifail on sisendiks nii generaatori küsimuste valikvastuste filtreerimiseks kui ka kõikidele automaattestimise genereerimise protsessi etappidele. Tegemist on sisuliselt JSON failiga, kuhu on kogutud kasutaja sisendid ja muud generaatorit huvitavad parameetrid. Kasutaja sisendid salvestatakse konfiguratsioonifaili konkreetse sisendile mõeldud parameetri alla. Generaator salvestab vastava parameetri väärtuse peale seda, kui kasutaja on oma sisendi sisestanud.

JSON kujul konfiguratsioonifail salvestatakse genereeritud projekti kõige tipmise kausta sisse projekti struktuuri loomise etapis. Konfiguratsioonifaili parameetrid, nende tüübid ning mille jaoks generaator neid kasutab projekti genereerimisel on toodud tabelis 8.

Tabel 8. Konfiguratsioonifaili parameetrid, nende tüüp ning kasutus.

Parameetri nimi	Parameetri kirjeldus	Parameetri tüüp	Parameetri kasutus
projectName	Genereeritava automaattestimise projekti nimetus	Tekst	Projekti genereerimisel kasutatakse parameetri väärtust projekti kõige tipmise kataloogi nimetamiseks.

Parameetri nimi	**Parameetri kirjeldus**	**Parameetri tüüp**	**Parameetri kasutus**
browser	Eelistatud veebilehitseja	Tekst	Kasutatakse veebirakenduste automatiseerimise raamistiku seadistamiseks ning selle sama vahendi kontrollimiseks, kui tegemist pole esmase genereerimisega.
programmingLanguage	Eelistatud programmeerimise keel	Tekst	Kasutatakse valikvastustega küsimustes valitud programmeerimiskeelele vastavate vahendite ja raamistike filtreerimiseks.
xUnitFramework	Ühiktestimise raamistik	Tekst	Kasutatakse õige ühiktestimise raamistiku paigaldamiseks ja näidisfailide genereerimiseks.
buildSystem	Programmi koodi kompileerimise ja valmis rakenduseks ehitamise raamistik	Tekst	Kasutatakse projekti metaandmete sisestamiseks ja automaattestimise vahendite sõltuvuste paigaldamiseks (näiteks Maven või Gradle, kui programmeerimiskeeleks on valitud Java).
seleniumFramework	Veebirakenduse automatiseerimiseks kasutatud Selenium-i raamistik	Tekst	Kasutatakse õige raamistiku paigaldamiseks ja vajadusel täiendavate ressursside paigaldamiseks ja näidisfailide genereerimisel.
designPattern	Automaattestimise disainimustrid	Tekst	Kasutatakse abiklasside ja näidisfailide genereerimiseks (loe täpsemalt peatükis 3.1.9).

Parameetri nimi	Parameetri kirjeldus	Parameetri tüüp	Parameetri kasutus
assertionFramework	Valideerimise raamistik	Tekst	Kasutatakse õige raamistiku paigaldamiseks, abiklasside ja näidisfailide genereerimisel (loe täpsemalt peatükis 3.1.9).
otherTestFrameworks	Muud automaattestimise raamistikud	List	Kasutatakse soovi korral täiendavate automaattestimise raamistike paigaldamiseks (näiteks Cucumber testilugude kirjeldamiseks).
reportingFramework	Automaattestimise raporti koostamise raamistik	Tekst	Kasutatakse õige raporteerimise raamistiku paigaldamiseks.
otherFrameworks	Muud kasulikud raamistikud automaattestide arendamiseks	List	Kasutatakse õigete raamistike paigaldamiseks, abiklasside ja näidistestide genereerimiseks.

3.1.6 Projekti struktuuri loomine

Projekti struktuuri loomiseks kasutatakse NPM moodulit *fs-extra*. Selle mooduliga saab generaator suhelda kasutaja failisüsteemiga ning luua sinna vajalike kaustasid ja faile. Millise struktuuri täpsemalt projekt luuakse sõltub teatud konfiguratsiooni parameetritest ehk kasutaja sisenditest.

Genereeritava projekti struktuur sõltub valitud programmeerimiskeelest. Näiteks, kui valitud on programmeerimiskeel Java, siis luuakse standardne Java paketistruktuur [36]. Lisaks, sõltub loodav projekti struktuur kasutatud automaattestimise disainimustrist. Kui valitud on *Page Object Model* disainimuster, siis generaator lisab disainimustrile vastavad kataloogid [37].

Kui konfiguratsioonifail on täidetud ja projekti struktuur paigas ja valmis genereeritud, siis on olemas kõik, et generaator saaks luua õiged failid õigetesse kaustadesse ja hakata seadistama projektis kasutatud vahendid.

3.1.7 Vahendite seadistamine

Automaattestimise projektis vajalike vahendite seadistamine on generaatori üks tähtsamaid etappe, sest vahendite seadistamine on üks ajakulukamaid tegevusi tarkvara automaattestimises.

Automaattestimise projekti generaator kasutab seadistamisel konfiguratsioonifaili, et selgitada välja, mis tüüpi sõltuvused on vaja paigaldada ja seadistada. Kui kasutaja oli valinud programmeerimiskeeleks Java ja projekti kompileerimise raamistikuks Maven, siis paigaldatakse raamistike sõltuvused Maven Repository-st [38].

Teiseks oluliseks sisendiks vahendite seadistamise etapile on konfiguratsioonifaili parameeter *seleniumFramework*. Sõltuvalt sellest, millise Selenium-i raamistiku kasutaja on valinud, tuleb teostada järgmised toimingud:

- Selenium-i raamistiku paigaldamine. Näiteks NightwatchJS allalaadimine NPM-ist
- Täiendavate ressurssside paigaldamine. Näiteks Java puhul Selenium Support raamistiku, mis lisab täiendavad võimalused standardsele Selenium Java raamistikule
- Veebilehitseja Driver-i failide paigaldamine. Näiteks *selenium-binary* mooduli allalaadimine NPM-ist

Samuti paigaldatakse selles etapis ülejäänud valitud raamistikud ühiktestimiseks, valideerimiseks, raporteerimiseks ja muud valitud raamistikud. Selle tõttu on see etapp ka üks aeganõudvamaid, sest kõik failid laetakse füüsiliselt kasutaja arvutisse läbi interneti (kas siis NPM-ist või Maven Repository-st).

3.1.8 Abiklassi genereerimine

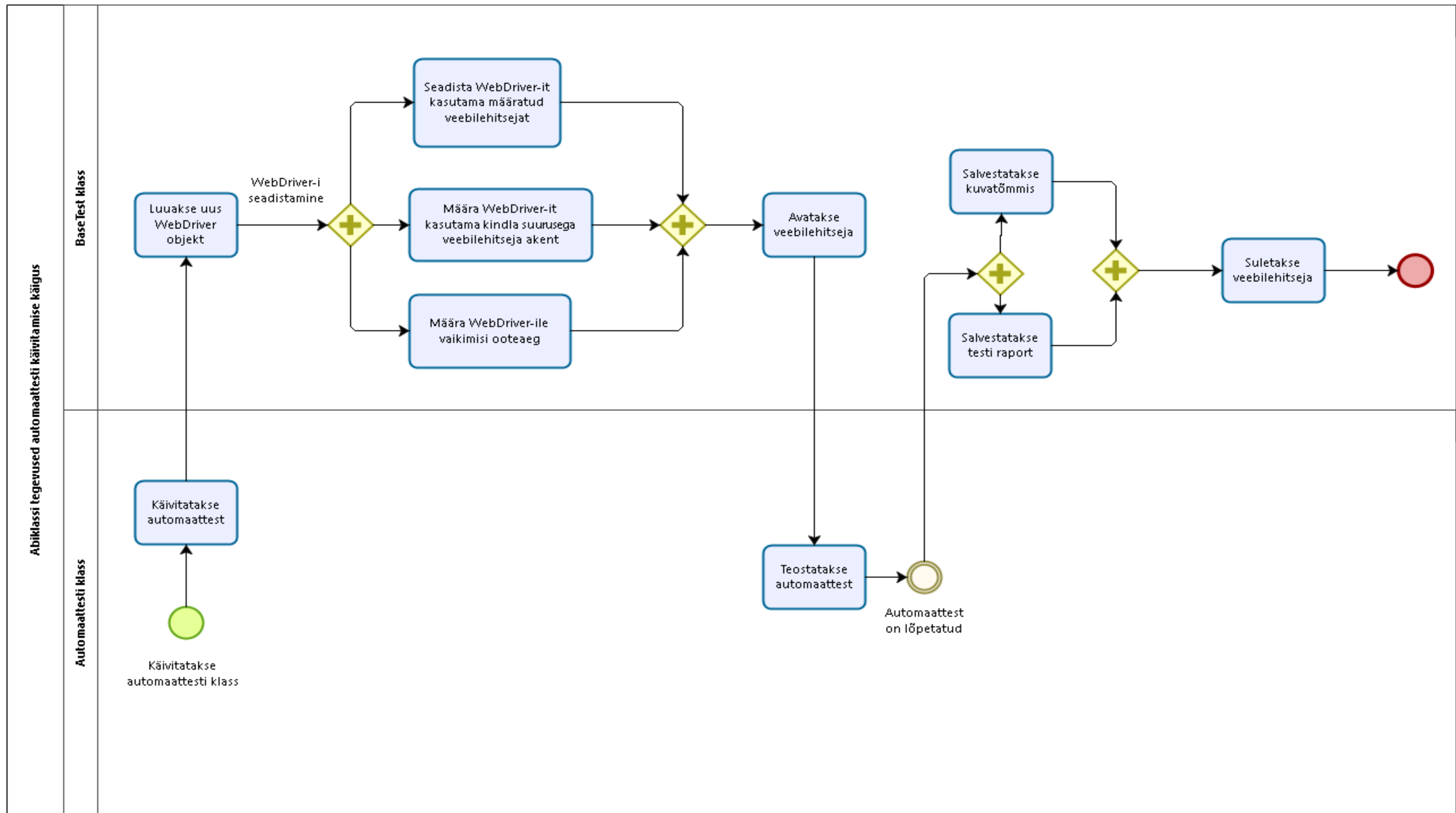
Automaattestide kirjutamise lihtsustamiseks loob generaator eraldi kausta abiklassi *BaseTest*. Abiklass sisaldab meetodeid ja objekte, mida tuleb tihti kasutada automaattestide arendamisel. Kasutaja saab oma testides abiklassi kasutada kas otse

(JavaScript-is mooduli eksportimine [39]) või laiendamise meetodil (Javas pärimine [40]).

Abiklassi eesmärk on automatiseerida teatud tegevused, mida tuleb ära teha sisuliselt iga projekti ja isegi iga testi puhul. Selliste tegevuste kuuluvad näiteks:

- Luua WebDriver-i objekt
- Määrata WebDriver-ile eelistatud veebilehitseja (konfiguratsioonifaili parameeter *browser*)
- Määrata WebDriver-ile vaikimisi ooteaeg, et kompenseerida veebilehe laadimise aega
- Määrata WebDriver-ile veebilehitseja akna suurus
- Avada veebilehitseja
- Sulgeda veebilehitseja
- Iga testi järgselt teha ja salvestada kuvatõmmis
- Iga testi järgselt salvestada testi tulemused raportisse

Abiklassi ja seda kasutava automaattesti klassi protsessidiagramm on toodud joonisel 3.



Joonis 3. Automaattesti klassi ja abiklassi protsessidiagramm.

3.1.9 Näidistestide genereerimine

Näidistestide genereerimise eest vastutab generator-examples.js generaator, mis on arendatud samuti käesoleva töö käigus. Näidistestide eesmärk on näidata kasutajale, kuidas näevad välja valitud automaattestimise raamistikega automaattestid, kuidas valitud veebirakenduste automatiseerimise raamistikku kasutada testides, kuidas valideerimise raamistikku kasutada oodatavate tulemuste võrdlemiseks tegelikega ning samuti ka näidata, kuidas kasutada abiklassi *BaseTest*.

Näidistestid genereeritakse eraldi kataloogi *examples*, mis asub testide kataloogis *tests*. Näidistestides kasutatakse sama veebirakenduse automatiseerimise raamistikku, sama valideerimise raamistikku ning sama raporteerimise raamistikku. Programmeerimisekeel, milles näidistestid on genereeritud vastab konfiguratsioonifailis parameetrile *programmingLanguage* ehk kasutaja valitud programmeerimiskeelele.

Testitavaks rakenduseks on veebipood Amazon. Kokku genereeritakse 3 testi:

1. Veebipoe lehele navigeerimine
2. Toodete otsing
3. Toodete lisamine ostukorvi

Kasutades *Page Object Model* disainimustrit, luuakse testitavale tarkvarale ka näidisfailid, mis vastavad testilugudes kasutatud veebipoe lehtedele ja lehtedel asuvatele elementidele. Näidisfailid salvestatakse eraldi kataloogi *pageobjects*.

Kasutaja saab kasutada näidisteste ja –faile, et arendada oma automaatteste. Need on mõeldud täitma juhendmaterjali rolli. Näidistestid vähendavad aega, mis kulub automaattestimise raamistike tööpõhimõttest ja sisust aru saamisele ning juurutamisele. Lisaks on näidistestid arendatud lähtuvalt automaattestimise disainimustrile. Näidisteste järgides järgib kasutaja ka parimaid praktikaid ning tõstab oma automaattestimise projekti hallatavust.

4 Prototüüp

Automaattestimise projekti generaatori eesmärkide täitmise valideerimiseks ja kasulikkuse hindamiseks on valmis arendatud prototüüp. Prototüüp on piiratud funktsionaalsuse ja automaattestimise vahendite nimekirjaga generaator. Vaatamata sellele on prototüübi tulemiks genereeritud automaattestimise projekt kasutaja poolt valitud vahenditega, mis on automaatselt ära seadistatud.

Põhiline kitsendus, mis prototüübile tehtud, on programmeerimiskeele valiku välja jätmine kasutaja sisenditest. Selle tulemusel kõik kasutajale kuvatud küsimused ja automaattestimise vahendid põhinevad programmeerimiskeelel Java. See tähendab, et antud prototüübi genereeritud automaattestimise projekti saab kasutada automaattestide arendamiseks programmeerimiskeeles Java.

4.1 Prototüübi piiratud funktsionaalsusega etapid

Prototüüp generaatoril on etappide funktsionaalsused (tegevused, mida generaator teostab konkreetse etapi jooksul) piiratud mahuga. Prototüübi automaattestimise projekti genereerimise protsess on toodud joonisel 4.

Eeltingimuste kontrolli etapis viiakse läbi kaks kontrolli: Java, Maven ja NodeJS paigalduste olemasolu kasutaja arvutis. Prototüüp on mõeldud uute automaattestimise projektide genereerimiseks, mille tõttu puudub prototüübil eeltingimuste kontrolli etapis ka automaattestimise vahendite kontroll. Prototüüp generaator paigaldab automaattestimise vahendid käivitamise hetkel kõige uuemate versioonidega.

Kasutaja sisendi küsimise etapis küsitakse vähem küsimusi, kui käesoleva töö lõpptulemuses defineeritud. Prototüüp generaator küsib ja salvestab konfiguratsioonifaili järgmised sisendid:

1. Projekti nimi
2. Ühiktestimise raamistik
3. Selenium-i raamistik

4. Valideerimise raamistik
5. Muud testimise raamistikud
6. Raporteerimise raamistik

Prototüüp generaator salvestab kasutaja sisendid konfiguratsioonifaili, kuid erinevad etapid otseselt ei kasuta konfiguratsioonifaili sisendiks valikvastuste filtreerimiseks. Küll, aga, kasutatakse konfiguratsioonifaili projekti struktuuri loomiseks, et luua projekti põhikataloog sisestatud projekti nimega.

Projekti kataloogide struktuur luuakse vastavalt Java standardsele projekti struktuurile [36]. Selles etapis luuakse ka pom.xml fail projekti põhikataloogi alla.

Prototüüp generaator kasutab konfiguratsioonifaili õigete vahendite sõltuvuste paigaldamiseks. Selleks loetakse konfiguratsioonifaili parameetrite *xUnitFramework, seleniumFramework, assertionFramework, otherTestFrameworks* ja *reportingFramework* väärtused ning nende alusel raamistike sõltuvused lisatakse pom.xml faili. Maven tüüpi sõltuvuste lisamiseks kasutatakse NPM moodulit *xmlbuilder*.

Kui pom.xml fail on edukalt raamistike sõltuvustega täidetud käivitab generaator operatsioonisüsteemi tasemel käsu, millega paigaldatakse vajalikud raamistikud kasutaja projekti jaoks. Vahendite seadistamine jätkub abiklassi genereerimisel.

Peale automaattestimise raamistikkude paigaldamist alustab prototüüp abiklassi *BaseTest* genereerimisega. *BaseTest* on Java klass, mille loomise eest vastutab alamgeneraator generator-basetest.js. Alamgeneraatoril on olemas meetodid erinevate Java-põhiste aspektide loomiseks (näiteks meetodi loomine, meetodi sisu lisamine, muutuja lisamine, muutuja väärtustamine, importide lisamine jms).

BaseTest koosneb ühiktestide meetoditest, mis käivitatakse enne ja pärast iga testi. Näiteks JUnit ühiktestimise raamistiku puhul on nendeks @Before ja @After annotatsioonidega märgitud meetodid. @Before annotatsiooniga meetod käivitatakse enne igat @Test annotatsiooniga meetodit. Seega @Before meetod on hea koht testi käivitamiseks vajalike eeltingimuste täitmiseks. @After meetod käivitatakse peale igat @Test meetodit. Seega on @After hea koht, kus koguda testitulemused ja need salvestada testiraporti faili või salvestada kuvatõmmis.

BaseTest klassis on WebDriver objekt, mida saavad kasutada kõik test klassid, mis pärivad *BaseTest* klassi omadusi. WebDriver objekt on Selenium-i raamistiku objekt veebilehitsejaga suhtlemiseks. See raamistik valitakse lähtuvalt konfiguratsioonifaili parameetrist *seleniumFramework.*

@Before meetodis luuakse uus WebDriver objekti instants ja seadistatakse järgmised parameetrid:

- veebilehitseja Google Chrome
- vaikimisi ooteaeg 1 minut
- veebilehitseja akna suurus
- logimine veebilehe tasemel [40]
- logimine võrgu tasemel [40]

Lisaks, luuakse testi rapordi kanne testi meetodi jaoks ehk see meetod annotatsiooniga @Test enne mida *BaseTest*-i @Before meetod käivitati. Kanne tehakse sama nimega, mis on @Test meetodi nimi.

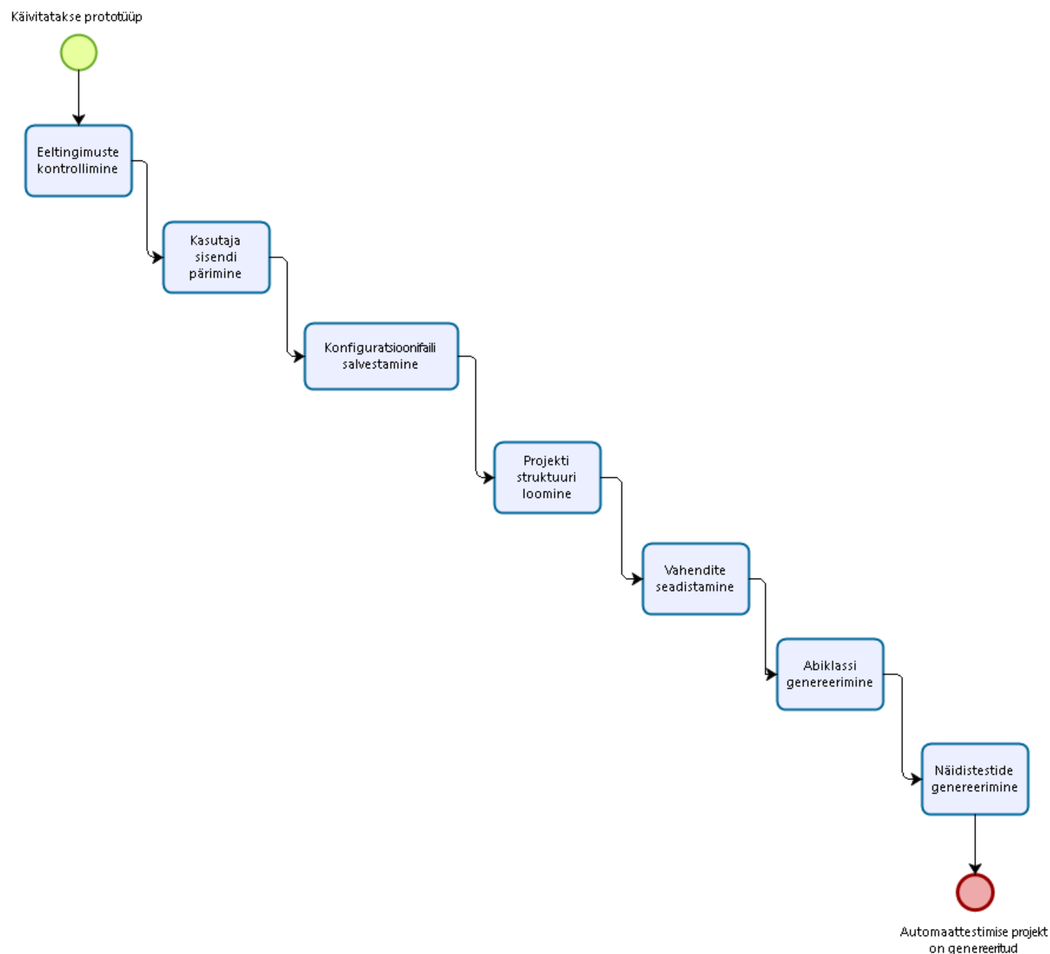
Abiklassi @After meetodis kontrollitakse testi tulemusi. Ühiktestimise raamistiku sündmused on võimalik kinni püüda ning nendel lugeda välja kas @Test meetodi validatsioon õnnestus, ebaõnnestus või tekkis testi teostamisel mõni muu probleem. Sõltuvalt sellest sisendist lisatakse testi raportisse testi tulemus ja kuvatõmmis veebilehitsejast testi lõpetamise hetkeks ning seejärel testi raporti kanne suletakse.

Kui abiklass on genereeritud, siis genereeritakse näidistestid, mille eest vastutab alamgeneraator generator-examples.js. Näidistestide põhimõte sarnaneb peatükis 3.1.9 kirjeldatule selles osas, et testitavaks veebirakenduseks on veebipood Amazon, on 3 näidistesti ning testid vastavad disainimustrile *Page Object Model*, mis tähendab, et luuakse vastavad näidisfailid kataloogi *pageobjects*.

Näidistestides ja –failides kasutatakse konfiguratsioonifailis märgitud raamistikke. Näidisfailid kasutavad parameetriga *seleniumFramework* märgitud Selenium-i raamistikku veebilehitsejale, veebilehele ning veebilehel olevate elementidega suhtlemiseks ning nendelt info kätte saamiseks.

Genereeritavad näidistestid laiendavad abiklassi ning sellega pärivad WebDriver objekti. Seega ei pea näidistestid eraldi WebDriver-i loomise ja seadistamisega

tegelema. Samamoodi kehtivad näidistestidele abiklassi @Before ja @After meetodite funktsionaalsused.



Joonis 4. Prototüübi automaattestimise projekti genereerimise protsessidiagramm.

4.2 Prototüübi piiratud automaattestimise vahendite nimekiri

Prototüüp generaatori toetatud vahendite nimekiri on piiratud. Prototüübi arendamisel on kasutatud vahendeid, mis ei vaja seadistamist täiendavate ressursside paigaldamise näol. Näiteks veebirakenduste automatiseerimise vahendite nimekirjast on välja jäetud originaalne Selenium-i raamistik *selenium-java*, sest erinevate veebilehitsejatega toimimiseks on vajalik paigaldada iga veebilehitseja jaoks eraldi Driver fail ning seejärel paigaldatud failide asukohad seadistada WebDriver objekti seadistamisel. *Selenium-java* kasutamine on põhjendatud siis, kui soovitakse saada võimalikult palju

võimalusi rätseplahenduse loomiseks. Kõik Java-põhised Selenium-i raamistikud on edasiarendused *selenium-java* projektist [17].

Kui rätseplahenduse loomine ei ole vajalik, siis kaasaegsed veebirakenduste automatiseerimise raamistikud on võimelised ise veebilehitseja Driver faile haldama ja seadistama. Sellisel juhul ei ole kasutajal vajalik täiendavaid ressursse paigaldada. WebDriver-i seadistamine, et testides oleks kasutatud soovitud veebilehitseja, tuleb ikka teostada ning selle jaoks on prototüüp generaatoril võimekus olemas.

Prototüübis kasutatud veebilehitseja automatiseerimise raamistikud on:

- WebDriverManager
- Serenity
- Selenide
- FluentLenium
- Conductor

Ühiktestimise raamistike osas on Javas kaks põhilist valikut: TestNG või JUnit. Tarkvara arendajad tihti eelistavad ühik- ja integratsioonitestide jaoks kasutada JUnit raamistikku, sest see on oluliselt vanem ning suurema kogukonnaga raamistik [42]. Tarkvara testimise automatiseerimises, aga, pigem eelistatakse kasutama TestNG raamistikku, sest see võimaldab defineerida erinevad testandmed sisendparameetritena ning kasutada neid sama testi jooksutamiseks erinevate test andmetega [43]. Sellegi poolest on mõlemad raamistikud sobilikud veebirakendustele automaattestide arendamiseks ning selle tõttu võimaldab prototüüp mõlemad raamistikud kasutada.

Valideerimise raamistikest on tänapäeval kasutusel põhiliselt teise, kolmanda ja neljanda põlvkonna raamistikud. Teise põlvkonna raamistikud on sisuliselt ühiktestimise raamistikud, millesse on integreeritud esimese põlvkonna valideerimise võimakus ehk Java sisseehitatud võtmesõna *assert* funktsionaalsus. See tähendab, et piisab TestNG või JUnit raamistikust sooritamaks elementaarseid valideerimisi. Kolmanda põlvkonna raamistikud pürgivad loetavamate validatsioonide poole. Kolmanda põlvkonna raamistike esindajaks on näiteks Hamcrest. [45] Neljanda põlvkonna valideerimise raamistikud on viinud validatsioonide loetavust järgmisele tasemele, tuues sisse ladusat stiili koodi kirjutamiseks. [44]

Prototüübis on esindatud teise, kolmanda ja neljanda põlvkonna valideerimise raamistikud, kuid teise põlvkonna raamistike puhul kuvatakse kasutajale ühiktestimise raamistik, mida kasutaja oli valinud. See tähendab, et kui kasutaja valis ühiktestimise raamistikuks JUnit, siis valideerimise raamistiku valikus on kuvatud JUnit Assert.

Prototüübi poolt toetatud validatsiooni raamistikud on:

- TestNG Assert
- JUnit Assert
- Hamcrest
- AssertJ
- FEST Fluent

Sarnaselt valideerimise raamistikega on ka mitmesuguseid raporteerimise raamistikke ning viise, kuidas neid projektis kasutada. Üks variant kasutada ühiktestimise vahendi integreeritud HTML raportite koostamise võimekust nagu seda on TestNG raamistikul. Teine viis on kasutada Maven-i raporteerimise pluginat Surefire Report, mille lisamise ja seadistamise võimekus on prototüübil olemas. Kolmas variant on kasutada alternatiivseid raporteerimise raamistikke nagu Extent3 ja Allure. Erinevalt Esimesest kahest variandist on kolmanda variandi puhul vajalik seadistamine ka kooditasemel. Täpsemalt abiklassis *BaseTest* tuleb tekitada õige raporteerimise raamistiku objekt ning seadistada see testide andmete ja tulemuste kogumiseks.

Prototüübi poolt toetatud raporteerimise raamistikud on:

- TestNG HTML Report
- Maven Surefire Report
- Extent3
- Allure
- Sahagin

Prototüüp generaatori suurimaks erinevuseks lõpliku lahendusega võrreldes on spetsialiseerumine konkreetsele programmeerimisekeelele. Vaatamata sellele on prototüübile seatud samad eesmärgid ning sellest tulenevalt käsitleb prototüüp ka sarnaseid probleeme automaattestimise valdkonnas. Seetõttu sobib prototüüp esialgse ülevaate saamiseks lahenduse kasulikkusest ning eesmärgipärasuse.

5 Automaattestimise projekti generaatori prototüübi testimine

Automaattestimise projekti generaatori kasulikkusele hinnangu andmise aluseks on võrdlus, mis on sarnane A/B testimise meetodile. A/B testimise käigus teostatakse eksperiment, kus asetatakse vastamisi kaks olukorda või teisendit. Näiteks tarkvara vana versioon ja uus versioon, kaks hüpoteesi või kask erinevat, kuid sama põhimõttega toodet. Võrdluse eesmärgiks on selgitada välja kumb variant, kas A või B täidab püstitatud eesmärgid või muud teatud tingimused.

Käesolevas eksperimendis on kõnealuseks olukorraks automaattestimise projekti seadistamine koos testilugu automatiseerimisega ning selle olukorra kaks varianti on:

- A – Testilugu automatiseerimine koos automaattestimise projekti seadistamisega käsitsi
- B – Testilugu automatiseerimine koos automaattestimise projekti seadistamisega kasutades automaattestimise projekti generaatori prototüüpi

5.1 Prototüübi testimise põhimõte

A ja B variantide võrdluse mõõdikuks on kiirus ehk aeg, mis kulub projekti seadistamisega alustamisest kuni automatiseeritud testiloo eduka teostamiseni. See tähendab, et valituks osutub variant, millele kulub kõige vähem aega, kusjuures automaattest peab edukalt läbi minema. Juhendaja (käesoleva töö autor) kontrollib, et tegemist poleks vale positiivse tulemusega.

Nii A kui ka B variantide puhul kasutatakse automaattesti ja projekti seadistamisel samu raamistikke ning eksperimendi käigus realiseeritakse üks testilugu. Mõlemal variandil realiseeritakse täpselt sama testilugu.

Variandis A täidetakse ülesanne käsitsi. See tähendab, et kõik vajalikud automaattestimise vahendid paigaldatakse ja seadistatakse eksperimendis osaleja poolt.

Eeltingimused, mille täitmist selle variandi puhul ei arvestata ülesande täitmise osana on:

- Java olemasolu arvutis, kaasaarvatud Java Development Kit seadistamine
- Apache Maven ja selle seadistamine
- Mozilla Firefox olemasolu

Variandis B täidetakse ülesanne kasutades automaattestimise projekti generaatori prototüüpi. See tähendab, et osaleja kasutab generaatorit, et koostada projekt vajalike automaattestimise vahenditega. Eeltingimused, mille täitmist selle variandi puhul ei arvestata ülesande täitmise osana on:

- Java olemasolu arvutis, kaasaarvatud Java Development Kit seadistamine
- Apache Maven ja selle seadistamine
- Mozilla Firefox olemasolu
- NodeJS ja NPM olemasolu ja selle seadistamine
- Automaattestimise projekti generaatori paigaldamine

Eksperimendis osales kokku 20 infotehnoloogia taustaga inimest, kes olid varem puutunud kokku programmeerimiskeelega Java. Pooltel osalejatest paluti täita ülesanne variandiga A ja teisel poolel paluti täita sama ülesanne variandiga B.

Eksperimendis osales ka käesoleva töö autor juhendaja ja kontrollija rollis. Juhendamisel anti osalejatele suunavat informatsiooni, et eksperimendis osalejad ei peaks kasutama otsingumootorit. Kontrollimisel veendus autor, et realiseeritud automaattest vastab testiloole ja kõik täiendavad nõudmised on täidetud.

Lisaks õigesti realiseeritud ja töötavale automaattestile, on osalejatele seatud täiendavateks nõudmisteks HTML kujul testiraport, mis sisaldab:

1. Teostatud testi nimi
2. Testi tulemus: kas test õnnestus või ebaõnnestus
3. Kuvatõmmis, kui test ebaõnnestub

Osalejad kasutasid oma eelistatud integreeritud arenduskeskkonda (näiteks IntelliJ IDEA, Eclipse, NetBeans vms). Lisaks, oli osalejatel palutud paigaldada vähemalt Java versioon 8, kaasaarvatud Java Development Kit, ning NodeJS. NodeJS on vajalik

prototüübi paigaldamiseks ja selleks, et see töötaks. Java on vajalik, et automaattestid, mis on realiseeritud programmeerimiskeeles Java, töötaksid. Nii integreeritud arenduskeskkonna valikut ja selle paigaldamist kui ka NodeJS ja Java paigaldamist ning seadistamist ei arvestata A/B testimise teostamiseks vajaliku ajakulu sisse. Praktikas teostatakse sellised seadistused vaid ühe korra.

Eksperimendi mõõdetavaks tulemuseks on ajakulu ehk aeg, mis kulub ülesande lahendamiseks. Iga variandi juures on välja toodud eeltingimused ülesande lahendamiseks, mille täitmine ei lähe aja arvestuse alla. Aeg arvestatakse hetkest, kui osaleja ja juhendaja on kontrollinud, et kõik eeltingimused on täidetud ning osaleja on ülesande läbi lugenud ja sellest aru saanud. Aja arvestamist peatatakse, kui osaleja on veendunud, et automaattest vastab testiloole ja test läbib edukalt.

Ülesanne koos testilooga, mida osalejatel tuleb realiseerida variandiga A või B on toodud lisas 3.

5.2 Prototüübi testimise tulemused

A/B testimise läbiviimise tulemusel koguti kokku 20 osaleja andmed ehk aeg, kui kaua võttis osalejal ülesande täitmise alustamisest kuni testiloo põhjal automaattesti realiseerimiseni. Kogutud andmete põhjal arvutati variandi A ja B kohta keskmised.

Variandi A tulemused on toodud tabelis 9. Tulemused näitasid, et keskmiselt kulus osalejatel ülesande täitmiseks 29.6 minutit. Kõige ajakulukamateks kohtadeks variandiga A ülesande täitmise puhul olid:

1. Veebirakenduse automatiseerimise vahendiga harjumine
2. Testi raporti raamistikuga tutvumine ja seadistamine

Veebirakenduse automatiseerimise vahendi puhul tuli aru saada, kuidas veebirakenduse väljade ja nuppudega opereerida. Näiteks, kuidas sisestada teksti otsinguväljale, vajutada nuppu ning saada kätte teatud tekst, mis oli veebilehel. Testiloos kirjeldatud tegevused veebirakendusega on veebirakenduste automatiseerimise valdkonnas ühed lihtsamate killast ning keerulisemate juhtude puhul oleks osalejatel rohkemgi aega kulunud.

Testi raporti puhul oli väljakutseks aru saada, kuidas seadistada raporteerimise raamistik nii, et testi raport tekiks automaatselt ning sisaldaks nõutud andmeid (testi nimi, testi tulemus ning kuvatõmmis). Kuigi raporteerimise raamistiku dokumentatsioonis olid vastavad koodinäited olemas, tuli osalejaid juhendada, et saavutada nõutud olukord. Tulemuseks oli rapordi loomine enne testi komplekti käivitamist, uue kande loomine enne testi, testi tulemuse ja kuvatõmmise salvestamine ning kande sulgemine.

Tabel 9. A/B testimises osalejate tulemused variandiga A.

Osaleja nr.	Ajakulu (minutites)
1.	31
2.	25
3.	32
4.	27
5.	29
6.	35
7.	27
8.	29
9.	33
10.	28

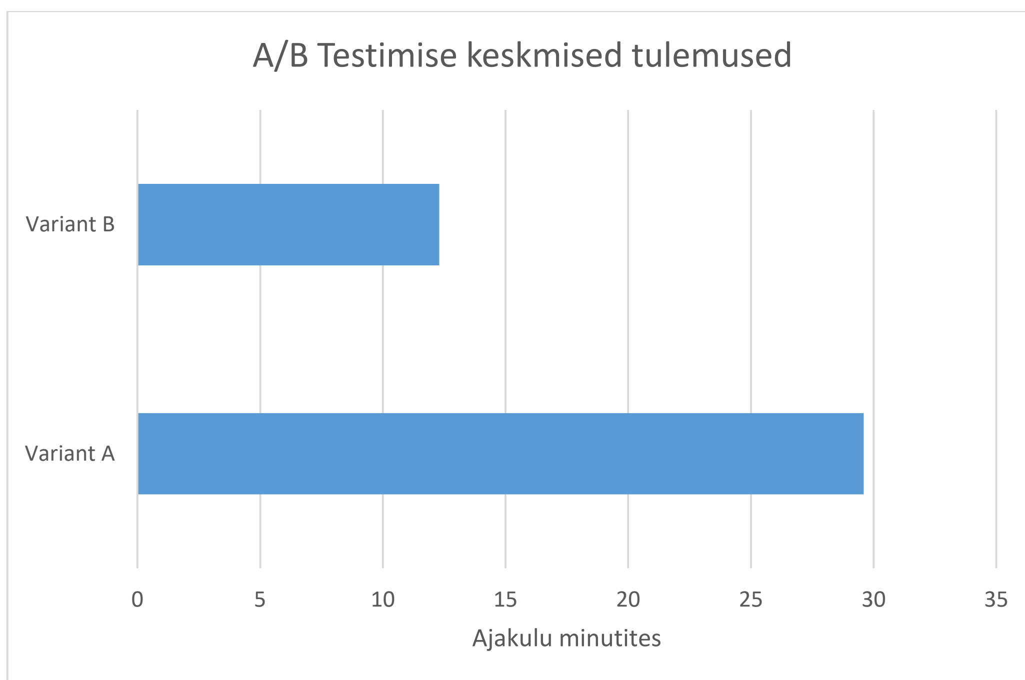
Variandi B tulemused on toodud tabelis 10. Tulemused näitasid, et keskmiselt kulus osalejatel ülesande täitmiseks 12.3 minutit. Kõige ajakulukamaks osaks ülesande täitmisel osutus veebirakenduse elementidele (otsinguväli, otsingu nupp ja veebilehel oleva artikli pealkiri) sobiliku identifikaatori määramine. Seda, kuidas need identifikaatorid tuleb veebirakenduse automatiseerimise vahendile ette anda, said osalejad vaadata näidistestide pealt. Õiged identifikaatorid tuli osalejatel käsitsi tuvastada ja automaattesti lisada.

Tabel 10. A/B testimises osalejate tulemused variandiga B.

Osaleja nr.	Ajakulu (minutites)
1.	13
2.	9
3.	15
4.	12

Osaleja nr.	Ajakulu (minutites)
5.	14
6.	12
7.	15
8.	12
9.	11
10.	10

Kui võrrelda variandi A ja B keskmised ajakulud, siis selgub, et variandil B on ajakulu umbes 60% väiksem kui variandil A. Joonisel 5 on toodud A/B testimise keskmised tulemused tulpdiagrammi näol.



Joonis 5. A/B testimise tulemused.

5.3 Automaattestimise projekti generaatori prototüübi kasulikkuse analüüs

Prototüübi testimiseks läbiviidud A/B testimise tulemusel saadud ajakulude võrdlus ning käesoleva töö autori poolt tehtud vaatlus viitab sellele, et automaattestimise projekti generaatoril on koht veebirakenduste automatiseerimise maailmas.

Prototüüp toetab erinevaid raamistikke ühiktestimiseks, veerikanduste automatiseerimiseks, tulemuste valideerimiseks ning testi tulemuste kohta rapordi genereerimiseks. Lisades lahendusele erinevate programmeerimiskeelte tugi, suureneb toetatavate raamistike hulk veelgi. Vahendite toetamise eeldusteks on abiklassi integratsioon ning näidistestide olemasolu. Abiklassi integratsiooni kaudu tuleb kasutajal kulutada vähem aega veebirakenduse automatiseerimise raamistiku tundma õppimisele. Näidistestide olemasolu annab ettekujutuse, kuidas valitud vahenditega testid võiks välja näha ning probleemide ilmnemisel, võib kasutaja näidistestide kiiremini lahenduseni jõuda, kui otsingumootorit kasutades.

Prototüüp paigaldas ja seadistas automaatselt valitud raamistikud ning vajalikud täiendavad ressursid. A/B testimise vältel oli osalejatel, kes täitsid ülesande variandiga A probleeme raporteerimise vahendi seadistamisega. Tegemist oli vahendiga, mille seadistamine oli keerulisem kui Maven-i sõltuvuse lisamine pom.xml faili. Vahend oli vaja õigesti seadistada ka koodi tasemel, et saavutada soovitud tulemus. Variandiga B osalejad pidid seejuures ainult laiendama abiklassi *BaseTest*, et nende testiklass päriks kõik vajalikud omadused, kaasaarvatud automaatne testi rapordi genereerimine. Lisaks oli prototüüp automaatselt paigaldanud täiendavad ressursid nagu *selenium-support* raamistik, mis lisas täiendavad võimalused veebilehitseja ja veebirakenduste testimiseks.

Prototüübi genereeritud projektiga genereeriti ka näidistestid, mis on arendatud *Page Object Model* disainimustriga. Kõnealust disainimustrit soovitatakse järgi, et tõsta automaattestide hallatavust ja arusaadavust. A/B testimises variandi B osalejad eelistasid järgida sarnast lähenemist automaattestide arendamisel nagu seda oli näidistestides tehtud, sest disainimustri kasutamise eelised olid neile iseenesest mõistetavad.

6 Edasised tegevused

Automaattestimise projekti generaatori prototüüp on kasulik genereerimaks automaattestimise projekte personaalseks otstarbeks. Ettevõtte nõudmised tarkvara automaattestimisele võivad olla palju laiemad. Ühes ettevõttes võib olla meeskondi, mis arendavad erinevates programmeerimiskeeltes. Kui meeskond arendab Java-s, siis on otstarbekas arendada ka automaattestid programmeerimiskeeles Java. Kui aga, meeskond arendab JavaScript-is (näiteks kasutajaliidese arendustega tegelevad meeskonnad), siis ei vasta arendatud prototüüp projekti nõudmistele.

Teiseks kitsaskohaks on veebilehitseja korrasoleku kontrolli ning generaatori versioonihalduse puudumine. Generaatori poolt genereeritava projekti hallatavust tõstab oluliselt võimalus kontrollida automaattestimise projektis kasutatud vahendite töövõimet veebilehitseja uuenduste korral.

Sellest tulenevalt on edasisteks sammudeks:

1. Generaatori versioonihalduse lisamine
2. Programmeerimiskeele toe täiendamine

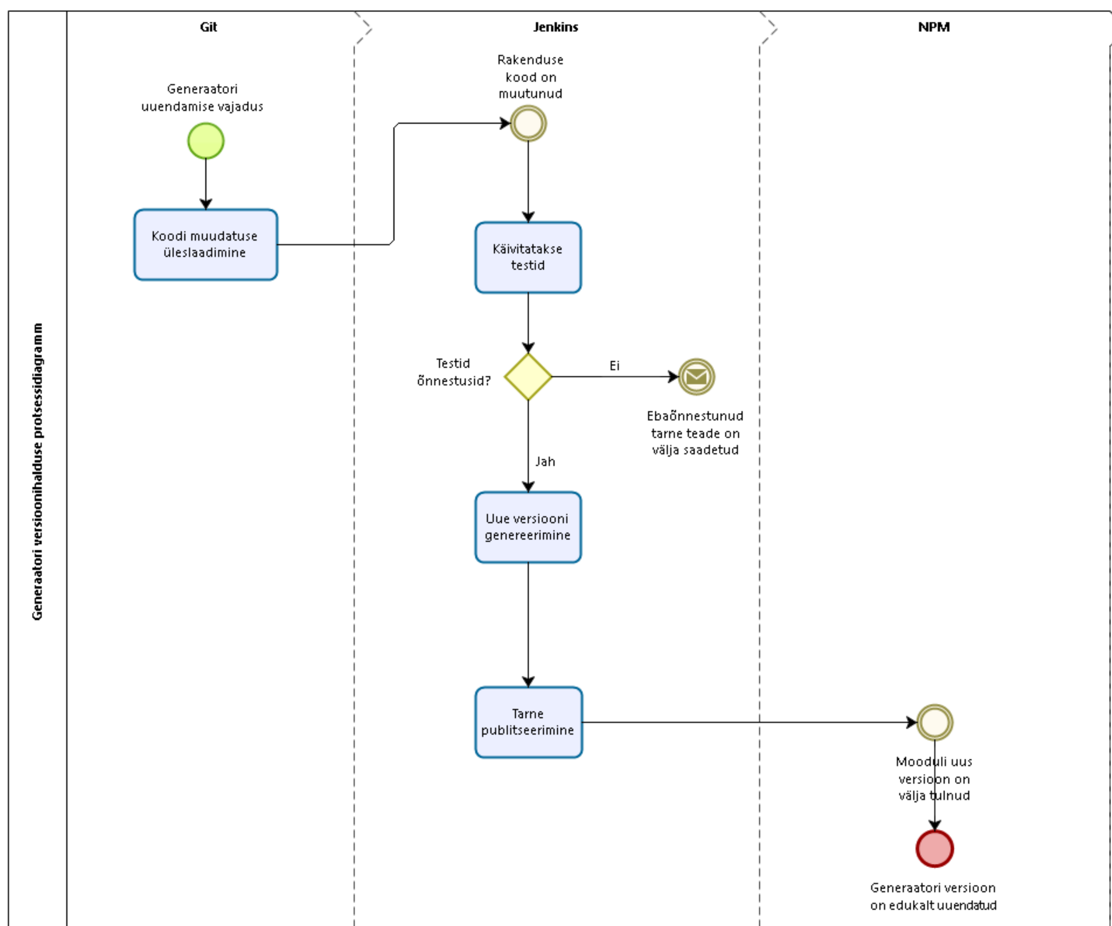
Lisaks sellele on pikemas perspektiivis otstarbekas arendada ühtne mall näidistestide genereerimiseks ning spetsialiseeritud koduleht.

6.1 Generaatori versioonihaldus

Peatükis 3.1.3 kirjeldatud olukorra saavutamiseks tuleb eelkõige seadistada projekt pidevintegreerimise süsteemis Jenkins. Jenkins-is luuakse iga toetatud programmeerimiskeele jaoks eraldi alamprojekt, mille raames jooksutatakse automaatselt iga tunni tagant näidistestid. Lisaks regulaarsele käivitamisele teostatakse iga koodi muudatusega uus projekti tarne (inglise keeles *build*), mille tulemusel uuendatakse generaatori versioon failis *package.json*.

Versioon uuendatakse kasutades NPM moodulit *standard-version*, mis käivitatakse peale igat koodi üleslaadimist versioonihaldussüsteemi Git. See koodi muudatus käivitab automaatselt uue generaatori tarne. Tarne koostamisel käivitatakse kõik testid alamprojektides. Kui testid õnnestuvad, siis tekitatakse uus tarne ning tarne pakk publitseeritakse NPM-i. Uue versiooni ilmumise korral generaatori käivitamine olemasoleva projekti peakaustas (mis on sama nimega, mis projekt), tuvastatakse NPM-is toimunud uuendus ning generaatori versioon uuendatakse. Versiooni uuendamisel teostatavad tegevused on kirjeldatud peatükis 3.1.2.

Kui aga, tarne koostamisel testid ebaõnnestuvad, siis teavitatakse sellest generaatori projekti osalejaid (antud hetkel autorit) automaatselt emailiga. Seejärel uuritakse, mille tagajärjel testid ebaõnnestusid, parandatakse kood ning käivitatakse testid uuesti. See protsess tehakse niikaua kuni testid õnnestuvad ja uus tarne on väljastatud. Generaatori versioonihalduse protsess on toodud joonisel 6.



Joonis 6. Generaatori versioonihalduse protsessidiagramm.

Generaatori versioonihalduse ning peatükis 3.1.2 välja toodud uuenduste läbiviimiseks arendatakse täiendav alamgeneraator *generator-update.js*.

6.2 Programmeerimiskeele toe täiendamine

Suurendamaks automaattestimise projekti generaatori kasutuselevõttu, täiendatakse generaatori poolt toetatud programmeerimiskeeled. Esimesena lisatakse JavaScript-i tugi, sest JavaScript-i põhised veebilehitseja automatiseerimise raamistikud on viimasel ajal kogunud populaarsust ning neil on paremad võimalused uuemate veebiarenduse tehnoloogiatega arendatud veebirakenduste testimiseks. [20]

Programmeerimiskeele toe lisamine tähendab, et generaatori mõistes teostatakse järgmised täiendused:

1. Täiendatakse kasutajale esitatud küsimused, lisades programmeerimiskeele küsimuse ja valikvastused
2. Konfiguratsioonifaili lisatakse uus parameeter *programmingLanguage* seda hakatakse kasutama sisendi küsimise etapis valikvastuste filtreerimiseks
3. Täiendatakse raamistike nimekirja lisatud programmeerimiskeele-põhiste raamistikega
4. Lisatakse näidistestid uute automaattestimise raamistikega
5. Jenkins-is seadistatakse uus alamprojekt, kus käivitatakse lisatud näidistestid

Lisaks Java ja JavaScript-ile on plaanis lisada ka programmeerimiskeele Python tugi, sest Python-it kasutakse tihti programmeerimise algkursustel ning palju populaarseid veebilehitseja automatiseerimise raamistikke on Python-i põhised. [46]

6.3 Ühtne mall näidisfailide genereerimiseks

Prototüübis kasutatud *generator-example.js* ja *generator-basetest.js.* tekitasid genereeritavasse projekti näidistestid ja abiklassi paigaldades vajalikud failid kasutades REST teenust. Selle lahenduse kitsaskohad on internetiühenduse olemasolu vajadus ja jõudlus. Generaatori genereerimise protsess sõltub otseselt internetiühendusest ning failid ei tekitata dünaamiliselt. Selle tagajärjel võivad paigaldatavad failid sisaldada vananenud informatsiooni.

Kasutades ühtset mall näidistestide ja abiklassi genereerimiseks on võimalik tõsta jõudlust (vähendab projekti genereerimisele kuluvat aega) ning võimaldada generaatori näidistestid ja abiklassi genereerida dünaamiliselt ka siis, kui internetiühendust ei ole. Näidistestides kasutatud testilood ja abiklassis *BaseTest* kasutatud meetodid veebirakenduse ja sellel olevate elementidega suhtlemiseks on kõikide programmeerimiskeelte ja vahendite puhul samad. Sellest tulenevalt on ühtse malli loomine võimalik ja otstarbekas.

6.4 Projekti koduleht dokumentatsiooniga

Selleks, et automaattestimise projekti generaator tuua avalikkuse ette, arendatakse välja projekti jaoks koduleht, millelt leiab:

- Kiirjuhend alustamiseks
- Generaatori täielik kasutusjuhend (eeltingimused kasutamiseks, paigaldamine, uuendamine, desinstallimine, käsklused jms)
- Toetatud programmeerimiskeeled ja raamistike nimekiri
- Projekti kontakt selleks, et pakkuda välja täiendused või teavitada avastatud veast
- Versioonihaldust puudutavat informatsiooni

Versioonihaldust puudutavas informatsioonis tuuakse välja, mis on hetkel kehtiv automaattestimise projekti generaatori versioon, millised täiendused ja parandused uue versiooniga paigaldatakse. Samuti oleks olemas ka versioonihalduse ajalugu, et oleks võimalik tutvuda uuenduste ja parandustega, mida lisati vanemate versioonidega.

Lisaks kogu automaattestimise projekti generaatorit puudutava informatsiooni olemasolule, teenib koduleht ka reklaami eesmärki, et muuta vahend laiatarbeliseks. Esialgu hoitakse kogu automaattestimise projekti generaatorit puudutav informatsioon versioonihaldus platvormil GitHub *README.md* failis, kuid pikemas perspektiivis on otstarbekas arendada spetsiaalne veebileht, et esitada informatsioon kasutajasõbralikumal viisil ja teha informatsioon paremini ligipääsetavaks.

7 Kokkuvõte

Käesolevas töös käsitleti probleeme veebirakenduste automaattestimise alustamisega. Fookus oli automaattestimise projektis vajalike vahendite seadistamisel ning kasutajat vajalike abimaterjalidega varustamisel. Töö eesmärk oli pakkuda välja lahendus automaattestimise projekti genereerimiseks, mis ühendaks endas mitmekesist nimekirja automaattestimise vahenditest, valitud vahendite paigaldamist ning seadistamist. Lisaks peaks lahendus varustama kasutajat näidismaterjalidega, mis juhendaksid kasutajat automaattestide kirjutamisel.

Analüüsi osas keskenduti automaattestimise alustamisega seotud probleemidele. Analüüsimiseks kasutati kasutatud kirjandust, autori isiklike kogemusi kui ka automaattestimise foorumeid ja artikleid.

Uuriti kaasaegseid automaattestimise vahendeid, aga eelkõige veebilehitseja automatiseerimise raamistikke. Nende vahendite arv on viimaste aastatega tõusnud märgatavalt ning iga raamistik püüab lahendada automaattestimise valdkonnas leiduvaid probleeme või teha automaattestimist mugavamaks. Antud töö käigus jõuti järeldusele, et hetkel turul saadavatest vabavaralistest vahenditest ei pruugi piisata, et kõiki tarkvara testimise projektis püstitatud nõudmisi täita.

Probleemide lahenduseks pakuti välja Yeoman generaatoril põhinevat käsurea rakendust, mis võimaldab kasutajal ise valida, mis vahendeid automaattestimiseks kasutada. Vahenditest saab kasutaja valida raamistikke veebilehitseja automatiseerimiseks, ühiktestide kirjutamiseks, validatsioonide koostamiseks, testiraportite genereerimiseks aga ka rida muid vahendid, mida võib automaattestide arendamisel vaja minna.

Töös püstitatud eesmärgi saavutamise kontrollimiseks arendati valmis automaattestimise projekti generaatori prototüüp, mis sisaldas osa täiusliku lahenduse funktsionaalsusest. Prototüübi kasulikkuse hindamiseks viidi läbi A/B testimine, kus eksperimendist osalejatel paluti koostada automaattestimise projekt ja automatiseerida üks testilugu. Eksperiment näitas, et osalejatel, kes kasutasid automaattestimise projekti

koostamiseks automaattestimise projekti generaatori prototüüpi oli aja kokkuhoid (ligikaudu 60%) nende osalejatega võrreldes, kes koostasid projekti käsitsi.

Prototüüp sobis näitamaks automaattestimise projekti generaatori vajadust ja kasulikkust. Sellegi poolest vajab prototüüp edasi arendamist, et seda oleks võimalik kasutada laiemalt kui individuaalseks kasutamiseks. Eelkõige vajab prototüüp täiendavate programmeerimiskeelte lisamist (JavaScript, Python jms) ning nendel programmeerimiskeeltel põhinevate automaattestimise vahendite lisamist. Need täiendused võimaldaksid kasutada välja pakutud lahendust laialdasemalt, aga eelkõige ettevõtetes, kus on meeskonnad, mis arendavad erinevates programmeerimiskeeltes ja tehnoloogiad kasutades.

Kasutatud kirjandus

1. Black R., Evans I., Graham D., Van Veenendaal E. Foundations of Software Testing. ISTQB Certification. Reviced Edition. Cengage Learn EMEA, 2008.

2. Spillner A., Linz T., Schoefer H. Software Testing Foundations. A Study Guide for the Certified Tester Exam. Fourth Edition. Rocky Nook Inc, 2014.

3. Kit E. Software Testing in the Real World: Improving the Process. Addison-Wesley, 1995.

4. Tassey G. The economic impacts of inadequate infrastructure of software testing. U.S. National Institute of Standards and Technology. Gaithersburg, USA, 2002.

5. ISTQB Advanced Level Test Automation Working Group. Certified Tester Advanced Level Syllabus. Test Automation Engineer. ISTQB Certification, 2016.

6. Fewster M., Graham D. Software Test Automation. Effective use of test execution tools. Addison-Wesley, 1994.

7. Harold M. J. Testing: A Roadmap. 22th International Conference on Software Engineering. Limerick, Iirimaa, 2000.

8. Ramler R., Wolfmaier K. Economic Perspectives in Test Automation. Balancing Automated and Manual Testing with opportunity Cost. Software Competence Center Hagenberg GmbH.

9. Kasurinen J., Taipale O., Smolander K. Software Test Automation in Practice: Empirical Observations. Lappeenranta university of Technology, 2009.

10. Crispin L., Gregory J. Agile Testing. A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2009.

11. Dustin E., Rashka J., Paul J. Automated Software Testing: Introduction, Management and Performance. Addison-Wesley, 1990.

12. Graham D., Fewster M. Experiences of Test Automation. Case Studies of Software Test Automation. Addison-Wesley, 2012.

13. Sabev P., Grigorova K. Manual to Automated Testing: An Effort-Based Approach for Determining the Priority of Software Test Automation. *International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol:9, No:12,* 2015.

14. Myers B. User interface software tools. *ACM Transactions on Computer-Human Interaction, 2(1), 64-103,* 1995.

15. Capgemini, Sogeti, Micro Focus. *World Quality Report 2017-18*.

16. Malek T., Derezinska A. Experiences in Testing Automation of a Family of Functional- and GUI-similar Programs. Warsaw University of Technology, 2007.

17. Collin M. Mastering Selenium WebDriver. Packt, 2015.

18. Wikipedia. List of web testing tools. [WWW] https://en.wikipedia.org/wiki/List_of_web_testing_tools (04.12.2017).

19. The Top 8 Essential JavaScript Automation Frameworks. [WWW] https://www.joecolantonio.com/2016/06/14/top-8-essential-javascript-automation-frameworks/ (04.12.2017).

20. 10 test automation frameworks for cross-browser testing. A comparison guide for 2017. [WWW] http://info.perfectomobile.com/10_Test_Automation_Frameworks_for_Cross_Browser_Testing.html (04.12.2017)

21. Hendrickson E. The Differences Between Test Automation Success and Failure. *International Conference On Software Testing, Analysis & Review*. 26.-30. oktoober, 1998.

22. StackExchange: Software Quality Assurance & Testing [WWW] https://sqa.stackexchange.com/ (07.12.2017)

23. Pettichord B. Success with Test Automation. *Quality Week,* 2001.

24. Setting up a Selenium WebDriver Project [WWW] http://www.seleniumhq.org/docs/03_webdriver.jsp#setting-up-a-selenium-webdriver-project (07.12.2017)

25. Test Design Considerations [WWW] http://www.seleniumhq.org/docs/06_test_design_considerations.jsp (07.12.2017)

26. Gohare S., Joshi R., Gaigaware D. Analysis and Design of Selenium WebDriver Automation Testing Framework. *2nd Inernation Symposium on Big Data and Cloud Computing*, 2015.

27. The Hamcrest Tutorial [WWW] https://code.google.com/archive/p/hamcrest/wikis/Tutorial.wiki (08.12.2017)

28. List of unit testing frameworks – Java. [WWW] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java (10.12.2017)

29. JUnit 4 vs. TestNG – Comparison. [WWW] https://www.mkyong.com/unittest/junit-4-vs-testng-comparison (10.12.2017)

30. Unified Functional Testing. [WWW] https://software.microfocus.com/en-us/software/uft (10.12.2017)

31. Yeoman – Running context. [WWW] http://yeoman.io/authoring/running-context.html (11.12.2017)

32. Semantic Versioning 2.0.0. [WWW] https://semver.org/ (11.12.2017)

33. Apfelbaum L., Doyle J. Model based Testing. *Software Quality Week Conference*, 1997.

34. Testing AngularJS apps with Protractor. [WWW] https://www.thoughtworks.com/insights/blog/testing-angularjs-apps-protractor (11.12.2017)

35. Most Popular and Influential Programming Languages of 2018. [WWW] https://stackify.com/popular-programming-languages-2018/ (20.12.2017)

36. Introduction to the Standrad Directory Layout. [WWW] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html (20.12.2017)

37. Test automation tips – create the correct project folder structure. [WWW] https://test-able.blogspot.com.ee/2015/06/create-correct-project-folder-structure.html (20.12.2017)

38. Introduction to the Dependency Mechanism. [WWW] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html (20.12.2017)

39. Node.js v9.3.0 Documentation. [WWW] https://nodejs.org/api/modules.html#modules_exports_shortcut (20.12.2017)

40. Inheritance. [WWW] https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html (20.12.2017)

41. Capabilities & ChromeOptions. [WWW] https://sites.google.com/a/chromium.org/chromedriver/capabilities (22.12.2017)

42. JUnit vs TestNG: Which Testing Framework Shoul You Choose? [WWW] https://blog.takipi.com/junit-vs-testng-which-testing-framework-should-you-choose/ (22.12.2017)

43. TestNG Documentation – Parameters with DataProviders. [WWW] http://testng.org/doc/documentation-main.html#parameters-dataproviders (22.12.2017)

44. Wikipedia. Fluent interface. [WWW] https://en.wikipedia.org/wiki/Fluent_interface (22.12.2017)

45. A Brief Overview of Java Assertions. [WWW] https://dzone.com/articles/brief-overview-java-assertions (22.12.2017)

46. Python Wiki. Python Testing Tools Taxonomy [WWW] https://wiki.python.org/moin/PythonTestingToolsTaxonomy (23.12.2017)

Lisa 1. Kasutajaga sisendi küsimiseks kasutatud küsimuste tüübid

Vabatekstilise sisendiga küsimuse näide

```
? Enter your automation project name:
```

Joonis 7. Vabatekstilise sisendiga küsimuse kuvatõmmis.

Ühe kindla sisendiga küsimuse näide

```
? Select programming language: (Use arrow keys)
  java
  js
  py
  c#
```

Joonis 8. Ühte kindlat vastust eeldava küsimuse kuvatõmmis.

Mitme võimaliku sisendiga küsimuse näide

```
? Choose other testing frameworks (if required):
(Press <space> to select, <a> to toggle all, <i>
to inverse selection)
>( ) Gatling
 ( ) Cucumber
 ( ) RestAssured
```

Joonis 9. Mitme valikuga küsimuse kuvatõmmis.

Lisa 2. Generaatori küsimused koos osade valikvastustega

Tabel 11. Generaatori küsimused koos osade valikvastustega.

Küsimuse nr.	**Küsimus**	**Näited valikvastustest**
1.	Enter your automation project name	
2.	Select programming language	▪ Java ▪ Python ▪ JavaScript ▪ C#
3.	Select xUnit framework	▪ TestNG ▪ JUnit
4.	Select Selenium framework	▪ Conductor ▪ WebDriverManager ▪ Selenide ▪ Protractor
5.	Select automation design pattern	▪ Page Object Model ▪ Screenplay ▪ Object Repository
6.	Select assertion framework	▪ Hamcrest ▪ AssertJ ▪ Spock
7.	Choose other testing frameworks (if required)	▪ Cucumber ▪ Gauge ▪ RestAssured
8.	Select reporting framework	▪ Extent2 ▪ Allure
9.	Choose other frameworks (if required)	▪ Lombok ▪ SLFJ4 ▪ Jackson

Lisa 3. A/B testimise ülesanne

Kirjeldus

Koostage automaattest järgmisele allpool olevale testiloole. Ülesande lahendamiseks kasutage oma eelistatud integreeritud arenduskeskkond (IntelliJ IDEA, Eclipse vms.). Veenduge, et Teie arvutis oleks paigaldatud Java 8 (kaasaarvatud JDK) ning Maven.

Automaattestimise projekti jaoks vajalikud vahendid

Testiloo automatiseerimisel kasutage järgnevaid vahendeid:

1. Java + Maven
2. Mozilla Firefox
3. Veebirakenduse automatiseerimiseks WebDriverManager: https://github.com/bonigarcia/webdrivermanager
4. Ühiktestimise raamistikuks JUnit: http://www.vogella.com/tutorials/JUnit/article.html
5. Validatsioonide kontrollimiseks Hamcrest: http://www.vogella.com/tutorials/Hamcrest/article.html
6. Testirapordi koostamiseks Extent Report: http://extentreports.com/docs/versions/3/java

Testilugu

<u>Kirjeldus</u>

Kasutades Mozilla Firefox veebilehitsejat, otsides Wikipedia kodulehel märksõna 'Selenium' avaneb keemilise elemendi Selenium Wikipedia leht.

<u>Stsenaarium</u>

1. Mine lehele https://www.wikipedia.org
2. Sisesta otsingusse märksõna 'Selenium'

3. Vajuta otsingu nuppu

Oodatav tulemus

Kuvatakse keemilise elemendi Selenium Wikipedia leht.

Lisatingimus

Automaattesti käivitamise tulemuste kohta tuleb genereerida HTML raport, kus peab olema näha:

1. Teostatud testi nimi
2. Testi tulemus: Pass/Fail/Skip
3. Kuvatõmmis (Screenshot) rakendusest testi lõpus

Kasulikke materjale

How to Locate Web Elements with Selenium WebDriver - https://loadfocus.com/blog/2013/09/05/how-to-locate-web-elements-with-selenium-webdriver/