TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

Kristo Tšekenjuk 134438IAPB

AUTOMAATTESTIMIS RAAMISTIKE VÕRDLUS JA ANALÜÜS PROFIT SOFTWARE OÜ PÕHJAL

Bakalaureusetöö

Juhendaja: Mart Kiviselg
Kaasjuhendaja: Deniss Kumlander
PhD

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: Kristo Tšekenjuk

10.01.2018

Annotatsioon

Antud töö peamiseks eesmärgiks on välja selgitada Profit Software OÜ põhjal kas Cucumber JVM'i testimisraamistik on parem alternatiiv kasutuses olevale Robot Framework'ile võrreldes nende raamistike erinevaid aspekte.

Töö käigus toon välja testise tähtsuse, võrdlen omavahel Robot Framework'i ja Cucumber JVM'i. Kasutan neid tulemusi, et analüüsida kas Cucumber JVM oleks parem alternatiiv praegusele lahendusele Profit Software OÜ's.

Töö tulemusest selgub, et Cucumber JVM on praegusest lahendusest mitmes aspektis üle. Analüüsi käigus on näha 21% ajavõitu automaattestide loomisel ning hooldamisel. Teiste eeliste hulka kuuluvad kiirem testide läbimise aeg 7,5% võrra, väiksemad arvuti ressursi kulud ning paremad võimalused keerukate testide implementeerimiseks. Kuna uue raamistku kasutusele võtuga kaasnevad täiendavad ajalised kulud, tehes vajalikud arvutused, leidsin, et uus testimis raamistik hakkaks ennast ära tasuma peale kahe ja poolt kuud.

Lõputöö on kirjutatud keeles ning sisaldab teksti 36 leheküljel, 6 peatükki, 14 joonist, 8 tabelit.

Abstract

The main objective of this study is to determine if Cucumber JVM framework is a better alternative for the current testing framework Robot Framework based on the needs of Profit Software OÜ.

Durning this study I will be comparing Cucumber JVM with Robot Framework in terms on functionality, performance, resource usage, usability, test writing and maintenance. Using data gathered through the comparison I will calculate the return of investment in order to help determine whether it would be beneficial to start using Cucumber JVM.

Based on this studies results Cucumber JVM holds the potentisal to be a greate alternative testing framwork for Profit Software. The study shows possible time savings of 21% when it comes to test writing and maintance when compared to the current solution Robot Framework. Other benefits include faster test execution times by 7.5 %, less computer ressource usage and functionalilty to overcome Robot Framework limitations. Time investment costs that come with starting using a new testing frameworks can be covered thanks to faster performance and test writing. Estimated calculations show that to cover the time investment it would take approximately 71 days.

The thesis is in Estonian and contains 36 pages of text, 6 chapters, 14 figures, 8 tables.

Lühendite ja mõistete sõnastik

Käitumispõhine testimine(BDT)	Testimismeetod kus testi sammud pannakse kirja lihttekstina järgides kindlat struktuuri (Given, When, Then).
Robot Framework	Märksõna põhine testimisraamistik
Cucumber JVM	Testimiraamistik mis mõeldud käitumispõhiseks testimiseks.
Gherkin	Cucumber JVM poolt kasutatav keel BDT testi sammude defineerimiseks.
Automaattestimine	Testimisviis kus kasutatakse teste mida saab programmina korduvalt käivitada
Profit Life&Pension	Profit Software'i tarkvara mis on selle töö raames testitavaks veebirakenduseks.
Docker	Vahend riistvara virtualiseerimiseks, sarnane tavapärase virtuaalmasinaga.
Page object teek	Teek mis sisaldab veebirakenduse kõikides veebilehe vaadades olevad elemente millega kasutaja kokku puutuks (lingid, tekstiväljad, nuppud, jne). Sisaldab ka funktsioone nende elelmentidega suhtlemiseks (nuppu vajutus, teksti sisetamine, jne).
ROI	(ing. k.Return of Intrest) investeeringu tasuvus
Regressioonitestimine	Testid mida kasutatakse süsteemi muutuste tõttu tekkinud vigade leidmiseks.
Manuaaltestimine	Testimisviis kus inimene teeb testid käsitsi läbi.
Maven	Java projekti haldus vahend

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Testimisel on oluline roll tarkvara arendamisel, olenevalt projektist võib testimine alata paralleelselt arendamisega ning võib jätkuda peale tarkvara toimetamist kliendile. Tarkvara testimist saab teha manuaalselt ja ka automatiseeritult. Manuaalne testimine kujutab endast sellist protsessi kus testija kontrollib tarkvara käsitsi [1]. Automaattestimise puhul on testija rolliks kirjutada valmis test mida saab korduvalt jooksutada [2]. Mõlemal viisil on omad positiivsed ja negatiivsed küljed ning mõlemad võivad olla kasutuses samas projektis.

Selle töö praktiline osa on seotud Profit Software OÜ poolt loodud kindlustus tarkvara testimisega. Töö eesmärk on uurida kas hetkel kasutuses olev testimisraamistik Robot Framework'ile leidub paremat alternatiivi mis oleks sobilik Profit Software'i projektide raames. Võrdluse alla tuleb Robot Framework ning Cucumber JVM automaattestide raamistikud, uurides nende jõudlust, funktsionaalsust ning kasutusmugavust. Lisaks annaks hinnagu kas testimisraamistiku vahetamine oleks majanduslikult kasulik kasutades investeeringutasuvuse arvutuse kaudu.

1.1 Taust ja probleemid

Profit Software OÜ on tarkvara firma mis on spetsialiseerunud kindlustus tarkavara loomisele. Oma olemuselt on kindlustus tarkvara pidevas muutumises kas seaduste tõttu või siis kindlustusfirmade soovis uue funktsionaalsuse järgi, et olla konkurentsi võimeline. Profit Software pikimad projektid on kestnud rohkem kui kümme aastat kus pidevalt lisatakse uut funktsionaalsust ning muudetakse olemasolevat. Seetõttu on automaattestimisel tähtis koht Profit Software'is kuna projekti maht on aastate jooksul muutunud nii suureks, et manuaalse testimine pole enam praktiline regressioonitestimiseks. Manuaaltestimine on küll kasutusel kuid ainult uue funktsionaalsuse testimisel arenduse käigus. Regressioonitestimine toimub automaat testitega, kuid nende arvukus hakkab juba probleeme tekitama. Et ühe projekti jaoks

kirjutatud testid ühe masina peal jooksutada võtaks see üle 24 tunni aega. Reaalselt jooksutatakse teste paralleelselt mitme masina peal, et aega kokku hoida, kuid siis tõusevad riistvara kulutused. Probleemiks on ka kasutusel oleva testimisraamistiku funktsionaalsed piirangud. Antud töö eesmärgiks on leida kas praegusele testimisraamistikule leiduks paremat alternatiivi tuues võrdluse alla testimisraamistike erinevad aspektid.

Profit Software on plaaninud pikemat aega, et peaks leidma parema testimisraamistiku, kuid kiirete aegade tõttu on seda edasi lükatud. Kuna firmas oli teada, et mul on vaja lõputööd, seetõttu pakuti mulle välja seda uurida oma lõputöö raames. Seega on selle töö tulemus oluline Profit Software'il kuna see on aluseks otsusele kuidas edasine testimine toimuma hakkab. Juhul kui otsustatakse uue testimisraamistiku kasuks, oleks selle töö praktiline osa uue testimise projekti põhjaks.

Pragune testimis lahendus kujutab ennast järgnevat. Robot Framework'i kasutadatakse Java projektina mida hõlmatakse kasutades Maven'it. Testitavaks tarkvaraks on Profit Life&Pension. Seda tarkvara jooksutakse veebirakendusena Linux'i virtuaalmasinas kasutades Docker'it. Arendus keskonnan kasutan Eclipse'i mis jookseb Windowsi peal. Uue raamistiku puhul peavad need jääma need samaks, ainult Robot Framework'i asemel on kasutusel Cucumber JVM.

1.2 Ülesande püstitus

- Annan ülevaate tarkvara testimese olulisusest. Kirjeldan automaattestimise rolli Profit Software projektide näitel. Toon välja praeguse testimisraamistiku head ja vead, et määrata mida alternatiivne lahendus peaks endast kujutama.
- Kirjeldan Cucumber JVM raamistiku ning kontrollin, et see raamistik vastaks miinimum nõuetele Robot Framework'i asendamiseks.
- Võrdlen testimisraamistike erinevaid aspekte: jõudlust, kasutusmugavust, funktsionaalust, ajakulu testide koostamisel. Analüüsin nendest võrdlustest tulenevat informatsiooni, et teha otsus kas raamistiku vahetamine tasuks ennast ära.

1.3 Testimis metoodika

Eesmärgini jõudmiseks võrdlen testimisraamistike omavahel tuginedes peamiselt raamistike dokumentatsioonile, Profit Software testimis nõuetele ning isklikule kogemusele. Jõudluse võrdemiseks loon mõlemas raamistikus mitmed võrdväärsed teste, et teada saada raamistike arvuti ressurside kasutuse ning testide läbimise kiiruse. Võrdlusest saadud tulemuste põhjal arvutan välja ajalise investeering tasuvuse. Neid andmeid kasutades saab teha lõpp otsuse kas raamistiku vahetus tasub ennast ära.

1.4 Ülevaade tööst

Ülesehituselt on töö järgnev, esimeses pooles kirjeldan üldiselt automaattestide tähtsuses Profit Software'is ning tutvustan ka testimisraamistke mis on selle töö raames uurimise all. Töö teises pooles tegelen peamiselt raamistike võrdlusega Lõpetuseks kasutan võrdluses saadud andmeid, arvutan testimisraamistiku vahetusega seoses oleva investeeringutasuvuse kasutades kohandatud ROI valemit [3]. Võrdluse ning investeeringutasuvuse arvutuste põhjal teen järelduse kas raamistiku vahetus tasub ennast ära.

2 Automaattestimise tähtsus Profit Software'is

Tarkvara arenduses on testimisel äärmisel tähtis koht, olenevalt projeti eripäradest ning skoobist võib olla vajalik osa testimis protsessist automatiseerida. Selle peatükiga üritan välja tuua automaattestimise tähtsus Profit Software projektides.
Enamus projektid Profit Software'is on väga suure mahulised, üks suurimatest tarkvara toodetest mida pakutakse on arenduses ning kasutuses olnud juba üle kümne aasta. Nende aastate jooksul lisatakse pidevalt uut funktsionaalsust, muudetakse olemasolevat. Muudatused süsteemis võivad sisse tuua vigu mida oleks võimalikult kiiresti üles leida. Sellega aitab testide automatiseerimine.

Profit Software tegeleb peamiselt kindlust tarkvaraga siis sellele tarkvarale rakendub kindlad seadused mis hõlmavad näiteks kliendi informatsiooni hoidmise nõudeid. Need seadusesed otselt määravad pakutava tarkavara toote mõningad aspektid. Kuna seadusi pidevalt muudetakse ja loodakse uusi, siis tarkvara peab sammuti muutuma. Lisaks sellele on pakutavast tarkvarast mitmeid versioone kuna erinevad kliendid otsutavad erinevatel aegadel millal suuremad uuendused vastu võtta. Arvestades, et projekti versioone on mitmeid, osad muudatused on vajalikud kõikides versioonides ning osad muudatused ja uus funktsionaalsus on kliendi spetsiifiline siis selle kõige haldamine on raskendatud.
Muutused ning uuendused tarkvaras võib sisse tuua ootamatuid käitumisi süsteemis. Nende vigade enneaegseks avastamiseks tulebki mängu automaat testimine. Ühe projekti testide jooksutamine ühe masina peal võtaks aega üle 24 tunni. Paralleelselt testide jooksutamine saab seda aega oluliselt vähendada kuid kogu testide maht on siis meeletult suur, et regressioonitestimine manuaalselt ei ole enam äriliselt kasulik lahendus. Samas automaat testimise protsessi kiirendamine, parandamine ja uuendamine võib tuua ajalist võitu kui ka finants kasu. Kuna automaat testimisel on Profit Software'is nii suur tähtsus siis on vajalik aeg ajalt uurida mis tehnoloogijaid ning meetoteid on saadaval ning kas üleminek tuleks kasuks või mitte.

2.1 Käitumispõhine testimine

Testimisega seotud trend mis on hiljuti firmas kasutusele tulnud on käitumispõhine testimine, mis kujutab endast kindla ülesehitusega testide kirjeldamist. Sisuliselt tuleb

test kirja panna lihtlausetega, esiteks eeltingimused (given), tegevused (when) ning tulemused (then) [4]. Vajadusel saab eeltingimusi, tegevusi ning tulemusi juurde lisada lisades vastav märksõna (and), lisaks kasutatakse tulemustes ka eitavat vormi (but). Tegin näiteks ühe testi kirjelduse kus on kasutusel kõik erinevad lause korrektsed lause algused (Joonis 1).

```
Given regular user is logged in
And home page is displayed
When user navigates to payment tab
Then payment view is displayed
But adding new payments is not available
```

Joonis 1. Näide käitumispõhise testi stuktuurist.

Põhimõtteliselt selline näebki välja tüüpiline testi kirjeldus. Tihtipeale jäetakse testist välja mis pole otseselt testitava funktsionaalsusega seotud näiteks pole mõtet kontrollida navigatsiooni lingi vajutamise järel, et õige leht avaneb kuna navigatsioon on testitud eraldi testides. See nõuab testijalt mõningaid teadmisi testitava tarkvara kohta kuid testi sammud on defineeritud piisava täpsusega, et ei tekiks segadust mida need sammud endast kujutavad. Enne käitumispõhist testimist oli automaat testide kirjutamiseks valmis tehtud Test Case'id, kus oli kõik sammud UI's tehtud sammud nii, et üks Test Case võib koosneda kuni 100+ sammu. Käitumispõhisel testimisel ei keskenduta nii väikestele detailidele, tegevused mis kasutaja peab tegema võetakse terviklikeks tegevusteks kokku, mille tulemusena saab parema ülevaate mida test teeb ning ei ole vaja raisata välja kirjutada liigselt detailseid samme mida testija suudab ise tuletada terviklikest tegevustest. Käitumispõhisel testimisest tuleb ka kasu siis kui testid vajavad muutmist. Kui olemasolevat funktsionaalsust on teadlikult muudetud siis peab uuendama ka automaatteste ning sobiliku koha ülesleidmine on lihtsustatud kuna testidest on selge ülevaade ning selle abil on võimalik leida muutmist vajavad sammud üles. Kui test oleks Test Case'i põhised ning oleks vaja vaadata kogu test läbi, et õige koht üles leida. Testi raportides näeb ka kasu käitumispõhistes testidest kuna testi sammud on kirja pandud kõigile arusaadavalt, tänu millele ei ole testi raportidest kasu ainult mitte testijatel vaid ka projektijuhtidel, ärianalüütikutel, klientidel, jne.

3 Automaattestimise vahendid

Testimisvahendeid on palju ning erinevate suunitluste ja funktsionaalusega. Enamus automaattestimise raamistikke saab jagada kuude kategoriasse [5]. Nendeks kategooriateks on :

- Lineaarne
- Modulaarne
- Arhitektuuril põhinev
- Test andme põhine
- Märksõna põhine
- Hübriid

Viide siia
Kõikidesse kategooriatesse ma süvenema ei hakka vaid kirjeldan ainult neid mis on selle töö raames olulised. Märksõna põhine raamistik võimaldab kirjutada testi sammud lihttekstina mis omakorda kutsuvad välja kas sisse ehitatud funktsioone või väliste teekide meetoditeid. Test andmete põhine raamistikuga saab teste jookutada kergelt erinevate andmetega. Arhitektuuril põhinev raamistik sõltub välistel teekidel mis sisaldaks testide ühiseid funktsioone. Hübriid raamistik võimaldab mitut eriviisi testist.

Robot Framework on hübriid testimisraamistik mille põhirõhk on märksõna põhisel testimisel kuid võimaldab ka test andmete põhis testimist ning saab kasu ka arhitektuuril põhinevast testimisest. Cucumber JVM efektiivseks kasutamiseks tuleks seda raamistiku käsitleda kui arhitektuuril põhinevat testimisraamistut, kuigi toimib piiratud viisil ka märksõna põhisena ja saab kasutada ka test andme põhisena.

3.1 Robot Framwork

Robot Framework on märksõna põhine testimisraamistik mis on implementeeritud Python keeles [6]. Testimisraamistik võimaldab kasutada täiendavaid Python ja Java teeke. Java puhul on vaja kasutada Java Python tõlgendajat.

Märksõnad on sisuliselt Python funktsioonide kutsed mida saab kirja panna liht tekstina. Võimalik on ka luua kõrgema astme märksõnu mis kutsuvad välja teisi märksõnu. Testide failid on lihtsa ehitusega mis koosnevad seadetest, ressursidest, muutujatest, testidest ning märksõnadest (Joonis 2).

```
*** Settings ***

# KOMMENTEERIMISEKS KASUTATAKSE # MÄRKI

Documentation     # Siia lisatakse vajalik info testi kohta, projekti dokumentatsionni lingid
Suite Setup       # Siia lisatakse märksõna  mis käivitud enne kõiki teste(nt. browseri avamine)
Suite Teardown    # Siia lisatakse märksõna  mis käivitud peale kõiki teste (nt. browseri sulgemine)
Test Setup        # Siia lisatakse märksõna  mis käivitud enne igat testi (nt. andmebaasi nullimine, sisselogimne õige kasutajaga)
Test Teardown     # Siia lisatakse märksõna  mis käivitud peale igat testi (nt. välja logimine)
Resource          # Siia lisatakse imporditavate failide asukohad

*** Variables ***

${example_variable}    example value

*** Test Cases ***

Example test
    Examlple test step 1
    Examlple test step 2
    Examlple test step 3
    Examlple test step 4

*** Keywords ***

# Märksõna definitsioon võib koosneda sisse ehitatud märksõnadest,
# Imporditud märksõnadest või madalamadest märksõnadest mis on selle failis defineeritud

Examlple test step 1
    Test step defintions go here

Examlple test step 2
    Test step defintions go here

Examlple test step 3
    Test step defintions go here

Examlple test step 4
    Test step defintions go here
```

Joonis 2. Näidis Robot Framework'i faili ehitusest.

Test Cases alla tuleb kõigepeal testi nimi, järgmisele reale ühe astme võrra trepituna tulevad märksõnad mis hõlmavad testi samme. Kui märksõnad mis ei ole eelnevalt defineeritud tuleb lisada vastavasse faili osasse (Keywords), sinna võib lisada ka abistavaid märksõnu, et testimist lihtsustada. Märksõnad on korduvalt kasutavad ning on võimalik luua märksõnu parameetritega, et mõjutada märksõna funktsionaalsust. Võimalik on defineerida kohustuslike ning mitte kohtuslikke parameetreid ning määrata neile algne väärus kui muud väärtust ei anta. [Pilt märksõna parameetrite loomisest]

Robot Framework võimaldab luua ka eraldi märgsõna faile mis teste ei sisalda. Need on mõeldud enam levinud märksõnade jaoks mida saaks teistesse testidesse vajadusel importida. Kasutades ainult sisse implementeeritud funktsionaalsust siis pole testijal vaja erilist programmeerimisoskust ning süntaks testide kirjutamiseks on üsna lihtne. Keeruliste testide kirjutamiseks on vajalik kas Python'is või Java's kirjutada valmis funktsioonid mida saaks hiljem märksõnade abil välja kutsuda.

Testidele on võimalik lisada silte (tags) mille abil saab teste märgistada [7]. Testide märgistamisega tuleb mitmeid häid aspekte. Sildid võimaldavad kategoriseerida teste kas funktsionaalsuse või mõne muu omaduse põhjal, neid on võimalik kasutada testide käivitamisel, näiteks saab siltide kaudu käivitada kõik testid mis on seatud kindal

funktsionaalsusega kui vastavad sildid on paika pandud. Silte saab ka kasutada ka osade testide ignoreerimiseks kui on näiteks teada testid mis vajavad uuendamist saab neile lisada sildi, käivitades testid sobiva käsuga, et süsteem ignoreeriks vastavat silti siis ei pea aega raiskama testidele mis kindlasti läbi kukuvad ega pea neid ajutiselt süsteemist eemaldama. Tänu siltide kasutamisele saab ka olulist statistilist informatsiooni kuna testi raportides tuuakse testide tulemused välja ka siltide kaupa, sealt saab järeldada millised testitava süsteemi osad töötavad nii nagu peab ning millised osad on testitava süsteemi muutuste suhtes tundlikumad.

3.2 Alternatiivi leidmine

Kõige pealt oleks vajalik paika panna tingmused mille tähelepanu pöörata uue testimisraamistikus mis sobiks Profit Software'ile

Üheks tingimuseks on kindlasti see, et uus raamistik peab olema vähemalt sama võimekas kui Robot Framework ning mingites aspektides parem. Oleks vajalik, et uues raamistik võimaldaks testi siseselt implementeerida keerukat funktsionaalsust ilma suuremate probleemideta. Kuna Robot Framework on sisulistelt liides Python'i ja Java funktsioonidele siis liidese piiratud funktsionaalsus piirab ka kuidas testi siseselt väliseid teeke kasutada saab. Uus testimisraamistik peaks võimaldama seda probleemi vältima.

Profit Software on plaaninud pikemat aega üle minna Java põhisele testimisraamistikule. Kuigi pole eelenevalt põhjalikult uuritud võimalike alternaatiive, on siiski võetud samme ülemineku poole. Nimelt on Javas tehtud enam vähem valmis Page Object teek, kus ideaalis peaks olema defineeritud kõik testitava UI võimalikud tegevused mida kasutaja saab teha, lisaks mõningad täiendused, et lihtsustada testimist. Kuigi see java teek ei hõlma hetkel kõike, on see piisavalt heal tasemel, et seda kasutatakse juba ka Robot Frameworkiga läbi Pythoni tõlgendaja. Seega on firmas tehtud ära otsus, et järgmine testimisraamistik peaks olema Java põhine. Kuna Profit Software'is toimub arendus peamiselt Javas, siis oleks hea kui testimine oleks ka Javas kuna rohkem inimesi saavad vajadusel aidata testimisega.

Üle on mindud käitumispõhisele testimisele mida toetab ka Robot Framework piiratud kujul kuid leidub raamistike mis saavad rohkem kasu käitumispõhisest testimisest. See oleks teine kriteerium uuele testimisraamistikule.

Nende kriteeriumite põhjal jääb silma kaks raamistiku: Jbehave ja Cucumber. Cucumber oli algselt küll Ruby keelne raamistik kuid tuli hiljem välja ka Java versioon Cucumber JVM. Mõlemad raamistikud on spetsiaalselt tehtud käitumispõhiseks testimiseks. Algselt oli plaanis võrrelda neid mõlemaid raamistike Robot Framework'iga kuid uurides neid lähemalt selgus, et Jbehave ja Cucumber on üsna sarnaseid kuid Cucumber on peaaegu kõikide aspektides parem. Jbehave on mõni aasta vanem kui Cucumber kuid Jbehave'i kommuun on väiksem ning vähem aktiivsem ja uuendused on harvemad. Cucumber seevastu on paremas seisus kuna kommuun on aktiivsem, info on kergemini leitav ning google trend järgi on populaarsus kasvav, Jbehave'il kahenev. Kuigi otsinugu populaarsus ei ole kuigi teaduslik meetod raamistike võrdlemiseks, siiski näitab see, et huvi Cucumber'i vastu kasvab ning annab parema perskpektiivi selle edasise arengu suhtes. Mõlemad raamisikud kuuluvad vabavara alla mistõttu nende raamistike edukus sõltub oluliselt just nende kasutajatest ning seda ümbritsevast kommuunist.

Vaadates Cucumber'i ja Jbehave'i sisse ehitatud testide raporteerimis võimekust selgub, et Jbehave jääb selles valdkonnas kõvasti alla. Kuigi mõlemate raamistikele on võimalik lisada liideseid raportide parandamiseks, on ilmselge, et Jbehave'i raportide muutmist sobilikust Profit Software'i projektidesse, on vaja kulutada rohkem aega. Jbehave'il on veel mõningaid puudujääke võrreldes Cucumber'iga kuid ma ei hakka nendele tähelepanu pöörama.

3.3 Cucumber JVM

Cucumber JVM on Java's implementeeritud versioon Cucumber Framework'ist mis on implementeeritud Ruby keeles [4]. Cucumber on raamistik mis on mõeldud käitumispõhiseks testimiseks.

Cucumber kasutab käitumispõhise testi sammude defineerimiseks keelt nimega Gherkin. Gherkin'i eesmärk on tõlgendada käitumispõhise testi sammude lauseid (Given, When, Then, And, But) ja jagad neid stsenaariumiteks. Gerkin'i faile nimetadakse funktsiooni failideks (Feature file) mis sisaldab testitava funtsionaaluse kirjeldust ning stsenaariume ehk eraldi seisvaid teste mis on seotud vastava funtsionaalusega (Joonis 3).

```
Feature: manage pension agreement
    As a back office user
    I want to login into Lbo
    So that i could create and manage pension agreement

Scenario: Create and manage pension agreement
    Given there is a client
    And pension agreement is made
    When initial payment is made
    Then agreement status is changed to "In force"
    When pension time arrives
    Then agreement status is changed to "Claim"

Scenario: Create and manage pension agreement, death case
    Given the correct agreement is imported
    When client is declared dead
    Then agreement status is changed to "Claim, Death case"
```

Joonis 3. Näide Cucumber JVM'i Feature failist.

Testi käivitamiseks on vaja testi jooksutus faili (Runner class) kus saab paika panna funktsiooni faili asukohad, raportide koostamise vormingu, testi sammude defintsioonide asukoha jne (Joonis 4).

```
package feature.payment;

import org.junit.runner.RunWith;

import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;

@RunWith(Cucumber.class)
@CucumberOptions(strict = true, plugin = { "html:target", "json:target/cucumber.json"},
    features =  {"src/test/resources/features"},
    glue = {"./src/test/java/features/"})
public class runTest {

}
```

Joonis 4. Näidis Cucumber JVM'i test runner failist.

Käivades testi ilma testi samme defineerimata kukub test ootuspäraselt läbi, kuid Cucumber genereerib tühjad funktsioonid mis vajavad implemteerimist (Joonis 5). Tänu sellele ei ole Cucumber'i testide keerulisem ehitus segavaks faktoriks testide koostamisel.

```
You can implement missing steps with the snippets below:

@Given("^pension agreement is made$")
public void pension_agreement_is_made() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^initial payment is made$")
public void initial_payment_is_made() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^agreement status is changed to \"([^\"]*)\"$")
public void agreement_status_is_changed_to(String arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^pension time arrives$")
public void pension_time_arrives() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Given("^the correct agreement is imported$")
public void the_correct_agreement_is_imported() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^client is declared dead$")
public void client_is_declared_dead() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

Joonis 5. Näide Cucumber JVM testisammude implementatsiooni genereerimisest.

Sammude definitsioonid tuleb implementeerid Javas, andes testijale rohkem kontrolli mida ja kuidas test asju tegema peaks. Kuna sammud on defineeritud Java, võimaldab see maksimaalselt ära kaustada Page Object teeki. Page Object teek on niimoodi ülesehitatud iga veebilehe vaate kohta on vastav klass teegis olemas. Võtame näiteks avalehe, seal asuvad navigatsiooni lingid mis viivad kasutaja uue vaate juurde. Page object teegis on olemas funktsioonid mis läbi veebidraiveri kutsub esile vajatuse vastavale nupule, tagastades uue vaate klassi mille kaudu saab käivitada funktsioone selles vaates. See võimaldab testijal kiiresti vajalikud funktsioonid ahelasse panna väga ökonoomselt.

Sisuliselt pole Cucumber täisväärtuslik testimisraamistik, pigem raamistik mis võimaldab kirja panna kergelt loetavad testi definitsioonid kuid sel puudub tegelik sisse

ehitatud funktsionaalsus mis oleks testi implemteerimiseks vaja. Raamistikus on küll abistavaid funktsioone mis lihtsustavad testide implemteerimist Javas kuid kõik vajalikud testi sammud tuleb Javas kirja panna. Cucumber sõltub tugevalt eelnevalt valmis tehtud funktsioonidest mida saaks korduvalt kasutada testide implementeerimisel. See tõttu on ka väga olulisel koha ka Page Object teek. Page Object teek on sisuliselt organisseritud kaart testitavast veebilehest. Ideaalist peaks seal olema kõik võimalikud nupud, lingid, tekstiväljad millega tegelik tarkvara kasutaja kokku puutuks ning sisaldab funktsioone mis jäljendab reaalseid tegevusi vastaval lehel näiteks nuppu vajutus, teksti sisestamine jne. Ilma vastava infrastruktuurita oleks Cucumber'i kasutamine testimiseks äärmiselt ebaefektiivne ning tüütu, kuid õige infrastruktuuriga toimib Cucumber nagu täisväärtuslik testimisraamistik.

Cucumber võimaldab ka silte kasutada [8] mis enamjaolt toimivad samamoodi nagu Robot Framwork'is ehk neid saab kasutada testide grupeerimiseks, käivitamiseks ja statistika kogumiseks. Cucumber'is saab kasutada silte ka selleks, et siduda test ning testi eelsed ja järgsed protsessid. See siltide funktsionaalsus on äärmiselt tähtis suurte projektide puhul kuna testi sammude implementatsioonid on globaalsed, seal hulgas ka testi eelsed protseduurid. Need testi eelsed ja järkseid protseduure nimetatakse Cucumber'is konksudeks (hooks) mis on sisuliselt tavalised Java meetodid mille on lisatud vastav annotatsioon: testi eelsete jaoks @Before, testi järgsete jaoks @After. Testi jooksutus fail käivitab kõik nendede annotatsioonidega funktsioonid mis võib tekitada probleeme kui erinevatel testi stenaarumitel on erinevad testimisekeskonna ettevalmistus ning mahavõtmis protseduurid. Ainuke mõistlik lahendus on anda testi stsenaariumitele unikaalne silt, et saaks ka määrata vajaminevad tegevused enne ja pärast testimist, et testid saaksid olla eelnevatest testide tulemustest sõltumatud.

4 Raamistike võrdlus

Raamistike võrreldes arvestan olenevalt vajadusest kas sisse ehitatud funktsionaalusest või mida raamistik võimaldab lisamoodulite ning vajamineva infrastruktuuriga. Oluline oleks paika panna prioriteedid sellest mis omadused on raamistiku puhul Profit Software'is, et hiljem saaks paremini analüüsida raamistiku vahetuse tasuvust. Prioriteedid alates olulisemast:

1. Funktsionaalsus
2. Ajakulu testide kirjutamisel ning hooldamisel
3. Testide läbismise kiirus
4. Testi raportite kasulikkus
5. Ressursi kasutus
6. Kasutusmugavus

Funktsionaalsus on kõige tähtsamal kohal kuna praguse lahenduse funktsionaalsed piirangud on peamiseks põhjuseks uue raamistiku otsimiseks. Ajakulu testimide kirjutamisel ning hooldamisel on äärmisel oluline kuna testimine võib tagasi hoida tarkvara uuenduste jõudmist kliendini. Testide läbimise kiirus muutub tähtsaks tarkvara arenduse vaatepunktist kuna muudatuste tegemisel käivitatakse testid regresioonitestimise eesmärgil. Mida kiiremini läbivad regressiooni testid seda kiiremini saab arendaja vajadusel teha parandusi või alustada uue ülesandega. Ülejäänud raamistiku omadused nagu testi raportide kasulikkus ning kasutusmugavus on proriteedilt madalamal kuna need on seotud teiste omadustega. Kui ajakulu testide kirjutamisel ja hooldamisel on madal siis enamisti see tähendab, et kasutusmugavus ja testi raportid on heal tasemel. Ressursi kasutus on üks viimastest asjadest mida jälgida kuna testimisprojekti arvuti ressurside kasutus on suhtelislt madal võrreldes testitava tarkvara nõuetega. Kulutusi pole prioriteedina välja toodud kuna mõlemad raamistikud kuuluvad vabavara alla.

4.1 Jõudlus

Jõudluse võrdlemiseks kirjutasin mõlema raamistiku jaoks 13 testi mille tegevuskäik on võimalikult sarnane [Lisa 1, Lisa 2]. Cucumber'i puhul pidin ka implementeerima üldkasutatavaid meetodeid, et lihtsustada testi suhtlust veebidraiveriga, ületooma testimiseks vajalikke meetoteid Robot Framework'ist. Teste jooksutades dokumenteerisin testidele kuluva aja, protsessori ja muutmälu kasutust reaalajas kasutades Windows 10 sisse ehitatud Performance Monitor'i. Testi jooksutamist panin

kirja kesmise ressurside kasutuse, et hiljem arvutada resursside kasutuse muutust. Testide raportidese on näha kui kaua testid jooksid kuid seal olev aeg ei ole täiesti täpne kuna ei arvestata aega mis kulub enne esimese testi käivitamist.

Mõlemas raamistikus jooksutasin teste 3 korda, peale igat korda ootasin natuke, et arvuti resursside kasutus stabiliseeruks ning fikseerisin uued keskmised ressursi kasutuse keskmised enne testi (Tabel 1, Tabel 2, Tabel 3).

Tabel 1. Jõudluse testimise esimese jooksutamise tulemused.

Esimene jooksutamine	**Robot Framework**	**Cucumber JVM**
CPU kasutus(enne/ keskmine/max)	4% / 32% / 87% Δ28%	4% / 29% / 84% Δ 25%
Muutmälu kasutus(enne/keskmine/max)	47% /61% / 63% Δ14%	54% / 61%/ 62% Δ7%
Aeg	17 min 21 sek	16 min 44 sek

Tabel 2. Jõudluse testimise teise jookutamise tumlemused.

Teine jookutamine	**Robot Framework**	**Cucumber JVM**
CPU kasutus(enne/ keskmine/max)	4% / 31% / 93% Δ 27%	4% / 28% / 71% Δ 24%
Muutmälu kasutus(enne/keskmine/max)	55% /66% / 68% Δ 11%	56% / 61%/ 62% Δ 5%
Aeg	18 min 19 sek	16 min 46 sek

Tabel 3. Jõudluse testimise kolmanda jookutamise tulemused

Kolmas jookutamine	**Robot Framework**	**Cucumber JVM**
CPU kasutus(enne/ keskmine/max)	4% / 30% / 98% Δ 28%	4% / 28% / 83% Δ 24%
Muutmälu kasutus(enne/keskmine/max)	55% / 64% /66% Δ 9%	56% / 62% / 63% Δ 6%

Aeg	18 min 25 sek	16 min 36sek

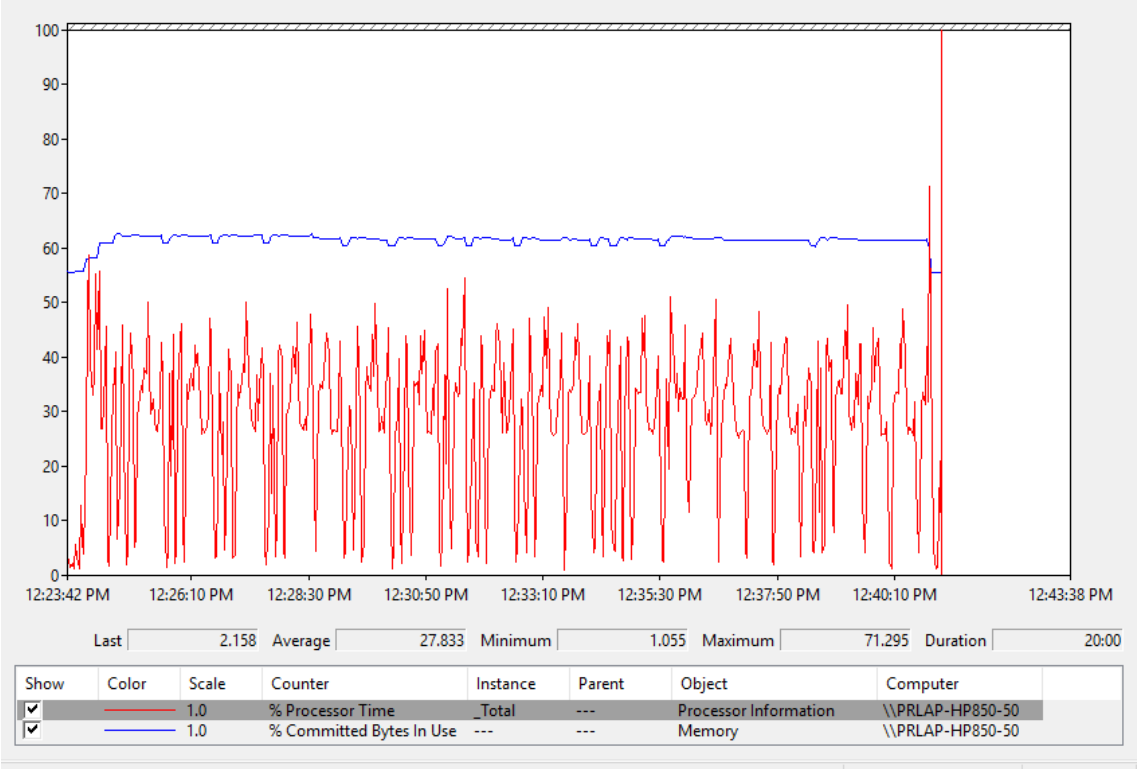
Esimesest Robot Framework'i jooksutuses on muutmälu kasutus oluliselt suurem kui teistel kordadel, see andis põhjust uurida kas tegemist oli anomaaliaga või on tulemused korduvad. Jooksutasin Robot Framework'is veel paar korda kõiki teste kuid kõik ülejäänud tulemused jäi 9% ja 11% vahemikku. Kuid suurema muutmälu kaustusega korral läbisid testid kiiremini ka, ning hakkasin uurima millised testi sammud jooksid kiiremini, et äkki need sammud kasutasid rohkem ressursse kui tavaliselt. Kontrollimiseks võtsin ette vastava testide jookutamise ajal salvestatud resursikasutus graafikud kuid otsest seost ma sealt ei leidnud. Statistika eesmärgil ma panin kirja muutmälu kasutuse enne testi, keskmise väärtuse testi ajal ning maksimaalse väärtuses testi ajal. Kuid mis mul algselt kahe silma vahele jäi, oli see, et peale testi ei taastunud muutmälu kasutus testi eelsele tasemele, erinevus oli umbes 4% mis on võrdne hälvega võrreldes teiste tulemustega. Kuna ma jooksutasin teste üksteise järel, siis järgnevatel kordadel ei olnud seda 4% tõusu enam. Nähtuse kontrollimiseks taas käivitasin arvuti ning proovisin uuesti teste jooksutada ning tulemuses oli jälle näha 4% tõus võrreldes normiga. Cucumber'iga proovis ka selle läbi ning ka siis oli näha seda 4% tõusu. Arvestades, et muutmälu kasutuse tõusu on märgata ainult esimesel testide jookutamisel hoolimata raamistikust, on põhjuseks suure tõenäosusega virtuaalmasin kus jooksis testitav veebirakendus. Ma oletan, et esmasel testide käivitamisel kui veebirakendust avatakse, suureneb virtuaalmasina muutmälu kasutus mida testide lõpetades enam ei vabastata. See selgitaks miks tekkis esimesel korral normist suurem tulemus. Ma teeksin jõudlustestid uuesti kuid ainuke tulemus mis sellest mõjutatud oli ning õnnestus väljaselgitada vea suurus siis otsustasin neid tulemusi keskmise ressursi arvutamiseks kasutada, muutes esimese jookutamise muutmälu kasutuse neljateistkümnelt protsendilt kümnele protsendile. Põhjus miks ma lahutan need 4% maha on see, et see 4%-ine lisa on seotud virtuaalmasinaga mitte testimisraamistikuga. Keskmise arvutamisel ei võta ma arvesse täiendavaid jõudlusteste kuna nende teostamise vahel algsete testidega oli paar päeva. Nende paari päeva jookul toimusid mõningaid tarkvara uuendusi mis võisid mõjutada tulemusi. Keskmised tulemused on tabelis välja toodud (Tabel 4).

Tabel 4. Jõuduse testimise tulemuste keskmised.

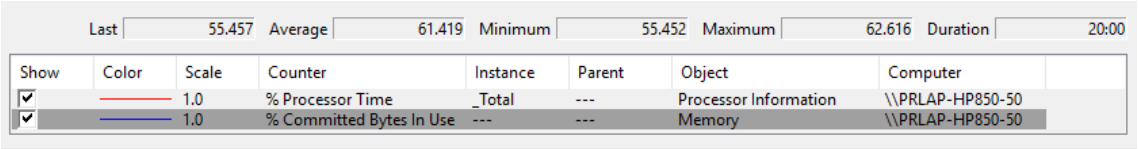
	Robot Framework	**Cucumber JVM**

Keskmine CPU kasutus	27%	24,3%
Keskmine muutmälu kasutus	10%* (Algselt 11.3% enne parandust)	6%
Keskmine aeg	18 min 2 sek	16 min 41 sek

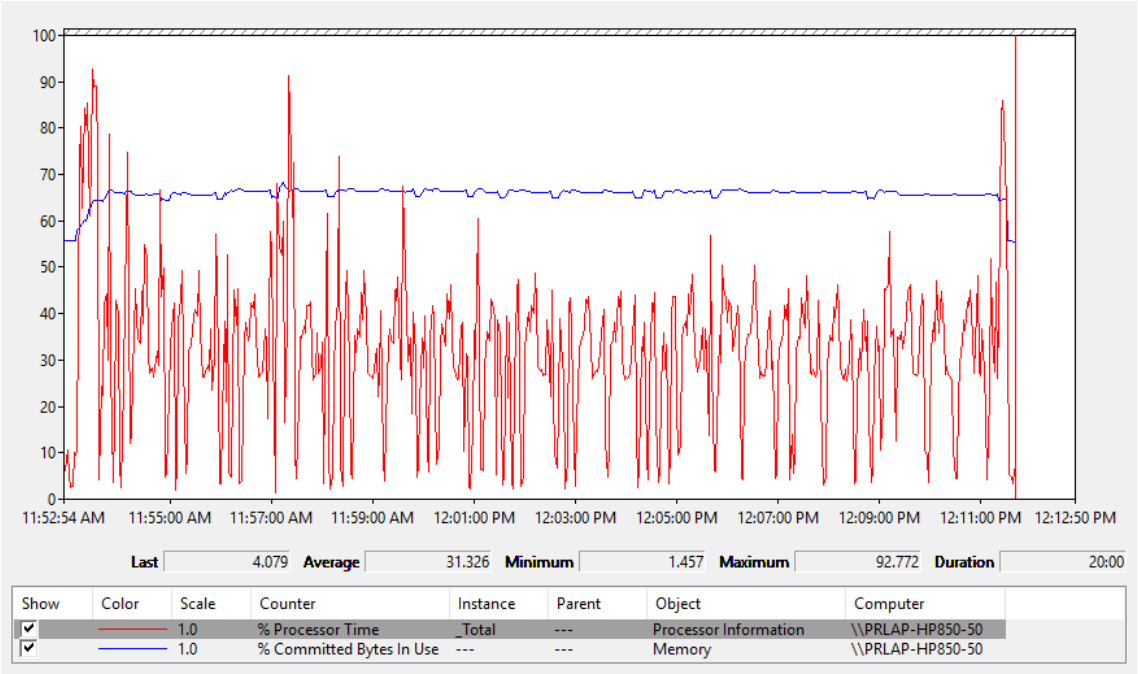
Kuigi katsetuste arv on suhteliselt madal, saab siiski järeldada, et Cucumber JVM kasutab vähem arvuti resursse ning sellest hoolimtata jookseb kiiremini. Antud andmete põhjal võib väita, et Cucumber JVM kasutab keskmiselt 10% vähem protessorit, 40% vähem muutmälu ning testid jooksevad 7.5% kiiremini. Vaadates ka resurssi kasutus graafikuid, leiab veel aspekte mis on Cucumber'i kasuks.



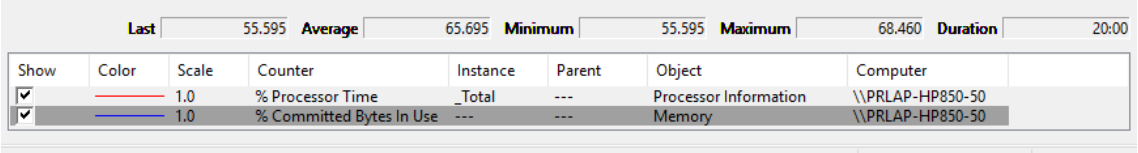
Joonis 6. Cucumber JVM ressursi kasutus.



Joonis 7. Cucumber JVM muutmälu kasutus.



Joonis 8. Robot Framework'i ressursi kasutus.



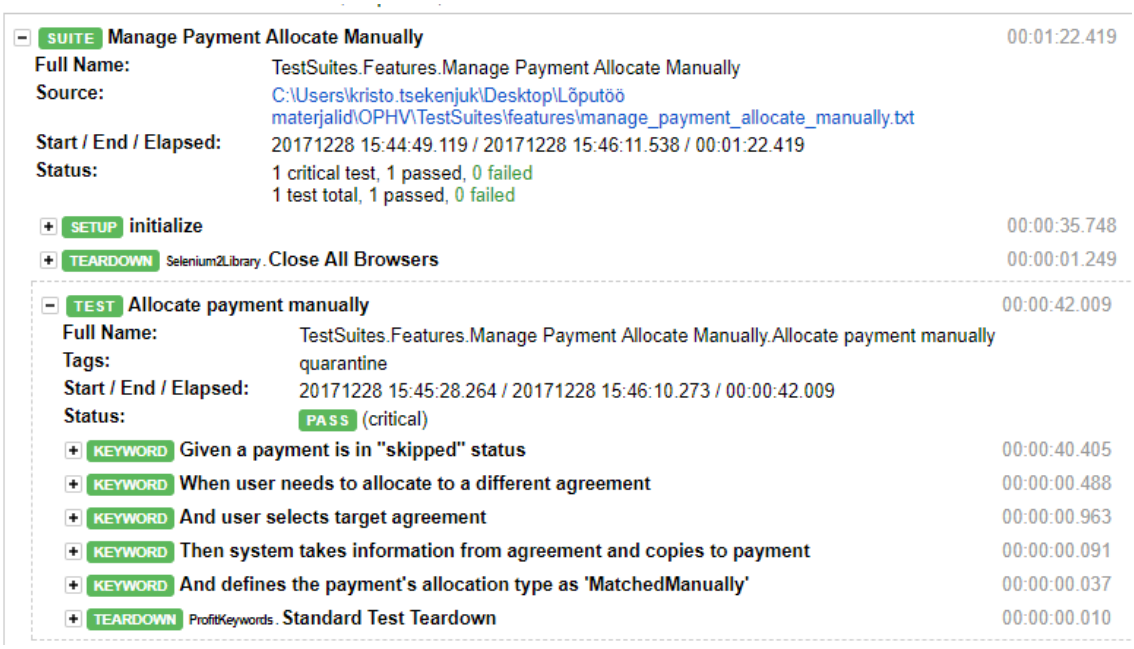
Joonis 9. Robot Framework'i muutmälu kasutus.

Graafikutelt on näha, et Robot Framework'i ressursi kasutus on natuke ebastabiilsem. See ebastabiilsus kajastub ka erinevate jookutamiste tulemustes. Ebastabiilsus väljendub selles kontetekstis peamiselt suurema erinuvuse keskmise ning maksimaalse ressursi kasutuse vahel ning suurem kõikumine protsessori kasutuses testide jookutamisel. Mõnel korral olid muutused väiksemad, mõnel suuremad, kuid üldiselt oli kerge eristada Robot Framework'i ning Cucumber'i tulemusi kasutades ressursikasutus graafikuid.

Cucumber jooksutas kõik testid kokkuvõttes kiiremalt kui Robot Framework, kuid tasuks uurida kas see kehtib ka kõikde üksiktestide kohta ka. Selleks on vaja uurida testide jookutamisel genereeritud logisi ning neid uurida. Mõlema raamistike logides on näha testi sammude kestvusi, kuid Cucumber'i puhul piirduvad need Feature failis defineeritud Given, When, Then väidete kohta. Tegeliku implementeeritud funktsioonide jooksutamise aega ei ole näha. Robot Framework seevastu võimaldab näha iga märksõna poolt väljakutsutud funktsioonide jooksutamisaega.

Scenario: Allocate payment manually			
Passed: 5	Failed: 0	Undefined: 0	Duration: 69.41s
Before			31.30s
Given a payment is in "Skipped" status			35.82s
When user needs to allocate to a different agreement			0.36s
And user selects target agreement			0.81s
Then system takes information from agreement and copies to payment			0.07s
And defines the payment's allocation type as 'MatchedManually'			0.02s
After			1.03s

Joonis 10. Cucumber JVM'i testide logi.



Joonis 11. Robot Framework'i testidelogid.

Võrreldes ühe testi logisi siis on näha, et Cucumber'is jooksevad kõik testi sammud kiiremini. Vaadates ka teiste testi logisis on võimalik leida mõningaid erandeid kuid ajalised erinevused on väiksed. Need sammud kus Robot Framework ühel korral kiiremine töötas ei ole alati kiirem, kuid see on üsna oodatud tulemas kuna statistikast oli ka näha, et Robot Framework'i kiirus on kõigub rohkem.

Suurimaid ajavõite on näha just keerukamatest sammudes mis on loogiline kuna need koosnevad rohkematest alamsammudest, kuid see pole ainuke erinevus. Enne testi vajalike seadistuste tegemiseks ning maksumute tegemiseks on peab raamistik suhtlema ka andmebaasi ning muude teenustega mis ei ole testitavasse veebilehte sisse ehitatud.

Kahjuks ei saa Cucumber'i logidest näha kui kaua jooksevad alamülesanded, mistõttu ei ole näha kas just andmebaasiga suheldes töötab Cucumber kiiremini. Seda oleks võimalik kontrollida kirjutades eralidi testid andmebaasi ja muude teenustega suhtlemiseks. Selle töö raames on siiski piisav, et tavapäraste testide puhul on üldkokkuvõttes Cucumber kiiremin kui Robot Framework.

4.2 Kasutus mugavus

Kasutusmugavus on subjektiivne teema mille kohta on raske midagi konkreetset öelda. Lisaks sellele sõltub testimisraamistiku kasutusmugavas oluliselt infrastruktuurile mis lihtsustab testimist. Cucumber sellest aspektis sõltub rohkem abistavast infrastruktuurist mille alla siis käivad Page Object teek ning muud Java teegid ja funktsioonid mis aitavad testimisel. Cucumber'il puudub sisse ehitatud funktsionaalsus mis otseselt abistaks testide implenteerimisega. Robot Framework'i on sisse ehitatud Pythoni teegid mis sisaldavad funktsioone veebilehega suhtlemiseks, XML failidega ümberkäimiseks ning mõningaid veel. Kui võrrelda testimisraamistike puhtalt selle järgi mis on kohe olemas siis on ilmselge, et Robot Framework on parem. Cucumber'isse on vaja investeerida rohkem aega, et testide kirjutamine oleks ladus ega peaks pidevalt lisama abistavat funktsionaalust mis Robot Framework'is juba olemas. Kuigi ma olen Cucumber'it kasutanud vähem kui Robot Framework'i, olen siiski arvamusel, et Cucumber'il on potentsiaal olla mugav testide kirjutamiseks. Põhjus miks ma nii arvan on seotud Robot Framework'i piiranugutega. Nimelt Robot Framework'is on kohmakas muutujatega ringi käia ning keerukama funktsionaalsuse saavutamiseks piiratud vahenditega on aeganõudev. Nende asjadega Cucumber'is probleemi pole kuna teste saab implementeerida võimekas programmeerimis keele, mitte märksõnade kaudu mis toimivad lisa kihina testija ning tegeliku funktsionaalsuse vahel. Kuigi Robot Framework'is saab ka kasutada Java funtksioone ning teeke on piiravaks faktoriks see kuidas Robot Framework nende väliste teekidega suhtleb. Robot Framework'is saab kasutada ainult muutjatena ainult primitiivseid andmetüüpe ja liste kuid ei saa kasutada Java funktsiooni parameetiks näiteks objekti. See tähedab, et funktsioonid mis on implementeeritud kas Javas või Python'is, et neid kasutada Robot Framework'is peavad need funktsioonid kasutama parameetriteks ainutl primitiivseid andmetüüpe. See muudab nende funktsioonide kasutamise ebamugavaks, sest kõrgemaastme testi

sammude jaoks võib vaja minna rohkelt parameetreid. Cucumber'i korral saab kasutada ka Java objekte, vähendades vajaminevate parameetrite arvu.

4.3 Funktsionaalsus

Antud kontekstis funktsionaalsus viitab sellele mida raamistik võimaldab teha, mitte mis nendes raamistikes on juba juurde implementeeritud kas siis selle töö käigus või firma siseses raamistiku kasutamisel. Funktsionaalsust võrreldes ei hakka ma detailidesse minema vaid toon välja tähtsamad funktsionaalsused ning lisan tabelisse vastav raamistiku juurde + märgu seda toetatakse. Juhul kui mõlemad raamistikud sisaldad sama funktsionaalsust kuid ühes raamistikus on selle implementatsioon parem siis lisan sobivasse lahtrisse täiendava + märgi.

Tabel 5. Funktsionaalsuse koondtabel.

Funktsionaalsus	**Robot Framework**	**Cucucmber**
Java teekide kasutamine	+	++
Tekstiredaktori võimekus, abistav funktsionaalsus	+	++
Sildid(Tags)	+	+
Keerukate andmestruktuuride kasutamine	+	++
Käitumispõhine testimine(BDT)	+	++
Põhjalikud logid ja raportid	+	+
Andmemaht logide genereeimisel (vähem on parem)	+	++
Debugimine (testide samm sammu haava käivitamine)		+

4.4 Testide kirjutamine

Testide kirjutamisel on Robot Framework'i suurimaks miinus tekstiredaktori piiratud funktsionaalsus. Kuigi Robot Framework toetab Java teekide kasutamist, ei ole sealseid funktsioone nähe kui see teek on Maven'i kaudu projekti lisatud. Et testi kirjutamise ajal *auto complete* aitaks nende funktsioonidega tuleb Java teek eraldi projektina importida samasse keskkonda. See omakorda tähendab, et iga Java teegi efektiivseks kasutamiseks kasvab projektide arv mida testija peab uuendama lokaalselt ning on suurem oht kasutada aegunud Java teeke.

Kuigi Java süntaks on palju keerulisem kui Robot Framework'i märksõnade koostamine, siiski teeb Java tekstiredaktori võimekus selle rohkem kui tasa. Kuna Java koodi kirjutamiseks on pakkuda rohkem keskkondi siis testija saab valida endale sobiliku, tõstes produktiivsust. Robot Framework'i kirjutamiseks on põhimõteliselt ainult kaks valikud: Eclipse ja Intellij. Kumbaski neist ei ole Robot Framework ametlikult toetatud vaid sõltub kolmanda osapoole lisandmoodulitest.

Üritades hinnata kuidas erinevad testide kirjutamiseks kuluv aeg nendes kahes raamistikus siis on vaja paar asja paika panna. Hetkel oleks Robot Framework'il eelis kuna väljaspool Java teeke on aastate pikkuse kasutusaja jooksul valmis kirjutatud funktsionaalsust mida Cucumber kasutada ei saaks. Praguse hinnagut andes ei arvesta ma aega mis kuluks lisa funktsionaalsuse implementeerimiseks, seda arvestan hilisemas analüüsis. Kuigi ma olen Robot Framework'iga rohkem kokku puutunud kui Cucumber'iga, võin siiski tõdeda, et Cucumber'iga on võimalik kiiremini teste kirjutada. See tuleneb peamiselt Java tekstiredaktorite võimekusest võrreldes Robot Framework'iga. Sellele järeldusele tulin ma kirjutades lihtsaid võrdväärseid teste mille puhul oli mõlemas raamistikus vastav funktsionaalsus olemas. Kuid kirjutada võrdväärseid teste ühes raamistikus ning siis teises võib mõjutada tulemusi kuna teist korda sama testi kirjutada võib olla lihtsam isegi kui seada teha teises raamistikus. Teaduliku hinnangu andmine testi kirjutamise kiirusele erinevates raamistikes on keeruline kuna see sõltub osaliselt inimfaktoritel, mistõttu ma ei hakka selle töö raames liiga põhjaliku uurimust tegema, vaid annan hinnagu omast kogemusest. Minu seisukohast võimaldab Cucumber kirjutada teste umbes 20% kiiremini kui Robot Framework peamiselt tänu võimekamatele tekstiredaktoritele. Algselt oli plaan jõudlus

testimise käigus mõõta raamisistike aja kulu kuid üsna varakult leidsin probleeme selle otsusega. Ajalise võrdluse eesmärgil peaks kirjutama samu teste erinevates raamistikes kuid luues testi ühes raamistikus mõjutab oluliselt kaua aega peab kulutama teises raamistikus. Põhjus on vägagi lihtne, kui testimise käigus tekib probleeme tuleb need ära lahendada, teist korda sama testi kirjutades samu vigu enam ei tehta. See muudaks praktilise katsetuse tulemused kasutuks.

4.5 Testide hooldamine

Testide kirjutamine ning jooksutamine ei ole ainuke ajakulu mis on seotud automaattestidega. Testid hooldamine on ka testija üks töökohustustest. Kui test ebaõnnestub, võib see tähendada kahte asja: süsteem on vigane ning vajab parandamist mille käigus võib vaja minna ka testi ümberkirjutamist, teine varjant on see,et test ise on vigane ning vajab sammuti ümberkirjutamist.

Testide hooldamise võrdlus on enamjaolt lihtsalt spekulatsioon kuna ainult üks raamistikest on hetkel kasutusel siis ei saa anda täpset hinnagut kuidas erinevad nende raamistikes testide hooldamine. Mis ma saan anda, on oma hinnang sellest kumb raamistik selles valdkonnas peale jääb arvestades just seda kuidas oleks neid kasutada Profit Software'is. Olen arvamusel, et Cucumber on testide hooldamiseks parem, seda järgnevatel põhjustel. Esiteks Cucumber'iga pole veel kasutusele võetud mis ka tähendaks, et see võimaldab paremini organiseerida uusi teste, et testide hooldamine oleks tõhusam ning vajadus väiksem. Robot Framework'is seevastu on aastate jooksul kogunenud rohkelt teste, abistavaid funktsioone mida peaaegu kunagi ei kaustata, rohkelt funktsioone mis on sisult sama kuid erineva nimetusega. Testimise standardid on firmas muutunud mitmeid kordi mistõttu on töös eritüüpi teste, uuenad testid on kirja pantud käitumispõhise testimise pritsiipide järgi, vanemad mitte. See muudab testide hooldamise Robot Framework'is edaspidi raskemaks kuna vanad ja uued standardid kõik segamini ja samaaegselt kasutuses. Teiseks on Cucumber'is vaja kasutada selgemat ärikeelt testi stsenaariumite kirjapanekul vähendades sõltuvust üldkasutatavate funktsioonidega. Kolmandaks sõltub Cucumber'i testid rohkem Java Page Object teegist mis on hästi organiseeritud, lihtsusdades mitte ainult testide kirjutamist vaid ka nende hooldamist. Suurim põhjus miks Cucumber on parem testide hooldamiseks on Java funktsionaalsus mis võimaldab teste samm sammu kaupa jookutada. Robot

Framework'il puudub võimalus teste sammu kaupa debugida. Kui anda arvuline väärtus sellele kui palju aega kulub testide hooldamisel võrreldes Robot Framework'iga siis anna hinnaguks, et ideaalsetes oludes kus Java Page Object teek ning muid abistavaid teeke on piisavalt täiendatud siis Cucumber võimaldab teste hooldada 30% kiiremini, mis tuleb ka osaliselt sellest, et saab alustada testimisega puhtalt lehelt ning vältida olemasolevaid probleeme.

4.6 Testi raportid

Robot Frameworki raportid on väga head, et üles leida miks test läbi kukkub. Kõik vajalik info on enamasti olemas kuid oleneval testsist võib olla üsna tüütu sobilik koht raportis avada kuna kasutusel on mitme tasemelised märksõnad. Võib öelda, et mõningatel juhtudel on raport liiga detailne, mistõttu kannatab raporti kasutavus mugavus. Kuna Robot Framework on märksõnade põhine testimisraamistik mis võimadab kasutada märksõnu teiste märksõnade sees eesmärgiga luua kõrgema taseme märksõnu mis teostavad keerulisemaid protsetuure, siis raportist kätte saada õiget infot võib olla mitetasemeliste märksõnade all. Märksõnad mis testi jooksutades olid edukalt jäävad ette vajaminevale infole. See on üsna tühine probleem ega sega olulislelt töövoolu.

Cucumber võimalab kergelt kohandada raportide sisu ning välimust ning on võimalik defineerida mitu erinevat raporti formaati erinvaks eesmärgiks (analüüsimiseks, kliendile näitamiseks, testide parandamiseks, jne). Lisaks paindlikule raporteerimis süsteemile saab vajadusel genereerida ka graafikuid mis annavad ülevaate testide seisust. Erinevalt Robot Framework'ist saab testi jookusutamise ajal lisada testi raportisse lisada peale teksti, ka muid objekte näiteks ekraani pilte läbikukkunud testi kriitilisest kohast. Testi jookustamise ajal on jooksev stsenaarium ehk test java objectina testi siseselt ligipääsetav andes võimaluse testijal lisada raportisse lisa infot kuna selle stsenaariumi objektide põhjal raport luuaksegi. Robot Framework'is saab ka salvestada ekraani pilte testi läbikukkumise hetkest kuid need ei ole raportidega seotud, muutes nende kasutamise testi hooldamisel äärmiselt ebamugavaks.

Robot Frameworki testide logid ja raportid on väga põhjalikud, kuid rohkete testide korral võib see muutuda problemaatiliseks. Nimelt Robot Framework'i logides on

üksikasjalikult kirjas ka korrektselt läbitud märksõnade alamkomponendid mis otsest väärtust testijale ei anna. Robot Framework genereerib oma testi logid ja raportid XML faili põhjal kus on olemas kogu testide jookustamisel tehtud tegevused. Arvestades, et Profit Software'i projektide peale kokku on üle 1000, muutuvad testideste jooksutamisest genereeritud andmemahud üsna suureks, ainuüksi XML fail on umbes 400 MB. Kuna neid logis ja raporteid on vaja säilitada, kuna nende abil saab parandada ebaõnnestunud teste ning kõiki teste jookutatakse igapäevaselt siis andmemahud kasvavad kiiresti. Cucumber ennetab seda probleemi tänu sellele, et logidesse ei lisata täiendavat infot õnnestunud testi sammudest. Kuna Cucumber'is pole arvuliselt sama palju teste implementeeritud nagu Robot Framework'is siis ma ei saa täpselt väita kui palju vähem andmemahtu Cucumber kasutaks, kuid võrreldes jõudlustestimise käigus loodud logisi siis oodatav andmemaht Cucumber'i puhul on umbes 4 korda väiksem. Tegemist on üsna suure võiduga arvestades, et midaga testimiseks ja testide parandamiseks vajalikku kaduma ei lähe.

5 Analüüs

Jõudluse võrdlemiseks kasutatud testide loomise käigus üritasin tähelepanna mis probleeme võib Cucumber'i kasutamine tuua kui see kasutusele võtta. Selgus, Profit Software'i kontekstis muutub suure mahulise projekti haldamine keerulisemaks võrreldes praguse lahendusega.. Need probleemid on seotud asjaoluga, et Cucumber'is on testi sammu implemetatsioonid on globaalsed ehk kui testi stsenaarium sisaldab sammu mis on muu testi jaoks juba implementeeritud siis uus test kasutab sama implementatsiooni. Lihtne lahendus on uue testi samm kas ümbersõnastada, et see oleks unikaalne või määrata sammule skoop mille abil saab sama sõnastusega samme eristada. Need lahendused on üsna kohmakana, mistõttu hakkasin asja lähemalt uurima. Kuna Cucumber on loodud käitumispõhiste testide loomiseks, tundus mulle imelik, et see raamistik ei sobi kõige paremini hetkel kasutatavate käitumispõhiste testidega. Ma hakkasin vaatama kas teistel Cucumber'i kasutajatel on esinenud sarnaseid probleeme ning mis ma avastasin tõi välja huvitava tõe. Selleks on fakt, et Profit Software'is defineeritakse käitumispõhiseid teste valest. Kui täpsem olla siis mitte täiesti valesti vaid ignoreeritakse ühte olulist aspekti. Eelnevalt olen töös maininud, et käitumispõhise testimise peamine eelis on see, et selle abil luuakse kergemini loetavad testid võrreldes eelnevalt siinses firmas kasutatud võtetega. Kuid hetkel ei jälgita seda, et käitumispõhise testi sammud tuleb kirja panna niimoodi, et need oleks ka üheti mõistetavad. Üritan tuua näite, et seda ilustreerida. Oletame, et testi üheks sammuks mis võib esineda mitemest testis on teha näiteks makse , see näeks välja midagi sellist "When user makes payment", see on liiga üldine, et puhtalt sellest tuletada korrektne implementatsioon. See pole enamasti probleem kuna testija on tavaliselt testitava funtsionaalsusega piisavalt tuttav, et muust dokementatsioonist vajaminev info kätte saada, vajaduse korral saab kontsulteerida ärianalüütikuga. On ilmselge, et üldiste sammude kasutamine raskendab ja aeglustab testide loomist. Samas on konkreetsete testi sammude loomine ka keeruline ning vajab süsteemset lähenemist. Hoolimata sellest kas Cucumber võetakse Profit Software'is kasutusele või mitte, olen selle töö käigus leidnud testimise aspekte mida saaks firmas parandada raamistikust sõltumatult. Põhimõtteliselt oleks vaja täpsustada ärikeelt mida kasutatakse testi sammude defineerimisel. Lisaks sellele on vaja luua keerulistemate testi tegevustele nagu näiteks vormide täitmise, luua kategooriad vastavalt sisule. Tuues uuesti näiteks, et 'Kasutaja

teeb makse’ pole piisavalt konkreetne testi samm kuna selle sisu võib kontkestist muutuda. Oleks vaja luua täpsutavad nimetused erinevatele maksumustele, eesmärgiga, et kui kasutada uues testis sammu mis on mõnes muus testis juba implementeeritud, et see implementatsioon oleks vastavuses ka uue samanimelise testi sammuga. Teisisõnu testi sammu definitsioonid peavad olema üheselt mõistetav. Kui saaks testimise sellele tasemele, et seda tingimust täidetakse siis testi sammude defineerimine lihtustab testimist kuna testija ei pea otsima dokementatsioonist täiendavat infot, seeläbi muutes tesimisprotsessi ökonoomsemaks. Täpsustuseks ütlen, et Profit Software’is automaattestija enamisti ei tegele testi sammude paika panemisega, ainult implementeerimisega. Automaat testi sisu paneb paika enamisti manuaaltestija kes on juba testitava funktsionaalsusega põhjalikumalt tutvunud või hoopis ärianalüütik. Automaattesti sammude defineerimiseks peab nagunii dokumentatsiooni uurima, et test kontrolliks kõiki vajalike ärireegleid mis uus funktsionaalsus nõuab. Kuid osa sellest vajalikust infost läheb kaduma kuna testi sammude definitsioonid on tihti peale liiga üldised. Selle tulemusena peab automaattestija uue funktsionaalsuse dokumentatsioonist välja selgitama mida tegelikult peab testi samm tegema. Võib jääda mulje, et ma süüdistan manuaaltestijaid ja ärianalüütikuid halvasti tehtud töö eest kuid tegelikkuses on probleemiks hoopis see, et pole paika pandud kindlat süsteemi mille järgi testi sammud ühesti mõistetavalt kirja panna.

5.1 Investeeringu tasuvus

Profit Software’i plaan minna millalgi üle Java põhisele testimisraamistikule ei ole majanduslik otsus vaid pigem seotud piiragutega praguses laheduses. Kuid raamistike võrdluse tulemuste põhjal on oodata ka ajalist kasu kasutades Cucumber JVM’i. See annab põhjust arvutada kui suur see ajavõit on pikema ajavahemiku vältel. Kuigi on Cucumber’ist on oodata ajavõitu, ei tohiks unustada, et see raamistik edukalt kasutusele võtta, tuleb kulutada lisa aega projekti ülesseadmiseks, Java teekide täiendamiseks ja nii edasi. Esmalt oleks vaja määrata kui palju aega peab investeerima, et Cucumber’it saaks uute testide kirjutamiseks kasutada. Tuletan veelkord meelde, et algse plaani järgi ei hakata Robot Framework’is olemas olevaid teste ümber tooma Cucumber’isse vaid hakatakse seda raamistiku kasutama vaid uute testide loomisel. Inesteeringu tasuvuse

arvutamiseks kasutan ROI (Return Of Investment) arvutust. ROI'd kasutatakse erinevates ärivaldkondades, et välja arvutada investeeringu tasuvust, enamasti kasutatakse seda rahalise investeeringu kasulikkuse määramiseks kuid saab ka selle abil arvutada ka ajalisi investeeringuid. Testimis valdkonnas kasutakse ROI arvutust tihtipeale automaat testimise kasulikkuse arvutamiseks võrreldes manuaaltestimisega. Arvutuseks kasutatakse tavaliselt lihtsat valemit: säästetud tunnid * tunni tasu. Kuigi selline arvutus tundub loogiline, ei ole see väga täpne kuna automaattestimine ei ole täpses vastavuses manuaaltestimisega [9], teisisõnu automaattestimisega ei saa katta kõike mida manuaalselt tehtakse [kasutatud kirjandus]. Lisaks tuleb sisse muid ajalisi kulusi peale testide kirjutamise millega nendes arvutustes alati ei arvestata. Selle töö raames seda probleemi ei teki kuna kasutan ROI'id mitte automaattestimise ja manuaaltestimise võrdlemiseks vaid kahe testimisraamistiku võrdlemiseks mille katuvus implementeeritavate testide poolest oleks täielik. Arvutuse lihtsustamiseks keskendun puhtalt ajalisele investeeringule, jättes finantsressursid kõrvale. Arvutuse teostamiseks on vaja teha veel täiendavaid lihtsustusi kuid ma toon nad välja siis kui jõuan valemitega vastavasse kohta.

Eesmärgiks on ROI valem viia kujule, et selle abil välja selgitada kas pikema perioodi jookul tasuks ajaliselt ära kulutada ressurse Cucumber'i testimise üleminekule.On vajalik arvutada kui palju teste selle perioodi jookul kirjutakse Robot Framework'is, seejärel arvutada kui kaua läheks Cucumber'is sama arv testide kirjutamiseks ning liita sellele juurde aeg mis on vaja investeerida, et täiendada vajalike Java teeke ning testimis projekt paika panna. Valemi kohandamiseks võtan algse valemi ning toon sisse vajalikud muudatused ja lihtsustused koos selgitustega.

Algne valem:

$$\text{ROI} = \frac{t - b}{b} \qquad (1)$$

ROI- investeeringu tasuvus

t - tulu

b - müüdud kaupade maksumus

Kuna eesmärk on võrrelda Robot Framework'i Cucumber'iga ning tulu ja kulu asemel kasutada kulutatud aega siis valem võtab järgneva kuju:

$$\text{ROI} = \frac{per\ * RF\_res\_jao. - (per * C\_JVM\_res\_jao. + lisakulud)}{per * C\_JVM + lisakulud} \quad (2)$$

kus:

per- perioodi pikkus töötundides

lisa kulud - ajaline kulu Java teekide uuendamisele, muu vajaliku funktsionaalsuse lisamiseks ning projekti paika panemiseks, et saaks Cucumber'it edukalt kasutada.

RF_res_jao - Robot Framework'i ressursi jaotus, sisaldab ajalist osakaalu testide kirjutamiseks, analüüsimiseks, hooldamiseks.

Rfress_jao = (analüüsi_osakaal + testide_kirjutamise_osakaal + testide_hooldamise_osakaal) (3)

C_JVM_res_jao -Cucumber JVM'i ressursi jaotus, sisaldab ajalist osakaalu testide kirjutamiseks, analüüsimiseks, hooldamiseks kuid on arvestatud ka kasuteguritega mille poolest Cucumber võimaldab kiirendada testimis protsessi.

C_JVM_res_jao = (analüüsi_osakaal + testide_kirjutamise_osakaal * testide_kirjutamise_kasutegur + testide_hooldamise_osakaal * testide_hooldamise_kasutegur) (4)

Töötundide perioodi vältel on lihtne arvutada kui on teada perioodi pikkust, testijate arvu ning testija tööpäeva pikkust. Perioodi pikkuse saab ise valida, kuid muude parameetrite määramine sellisel viisil, et see peegeldaks Profit Software'is olevat olukorda on keeruline. Probleemiks on see, et testijate arv ühe projekti raames on pidevas muutuse. Firma siseselt liigutakse projektist projekti olenevalt töökoormusest. Muutuses on ka vajadus uute testide loomise järele kuna uued testid on enamasti seotud muutustega testitavas tarkvaras mida suunab klient uue funktsionaaluste soovidega ning nende uute funktsionaalsuste arendamisega. Tuues sisse töötunnid lihtsustatud kujul ei pruugi anda piisavalt täpset tulemust. Parem oleks arvutada investeeringu tasuvus kasutades ainult ressursi jaotuste protsente ning kasutegureid. Kuna lisa kulud on ka seotud ajalise väärtusega tuleks see ka praguseks valemist kõravle jätta. Hiljem saab seda kasutada, et arvutada kui kaua peaks uut raamistikku kasutama, et selle kasutegurid

kataks ära võimalikud ajakulud mis kaasneksid raamistiku vahetamisega. Tuues konkreetsed arvud valemisse sisse hiljem, saab vajadusel neid väärtusi kergemini muuta.

$$\text{ROI} = \frac{RF\_res\_jao - C\_JVM\_res\_jao}{C\_JVM\_res\_jao} \quad (5)$$

Selle valemi tulemuseks saab protsentuaalse ajalise kasuteguri kasutades Cucumber JVM'i. Et teada saada kaua uut raamistikku kasutada tuleks, et lisakulud kaetud saaks, tuletame vastava valemi. Sellest valemist saadud tulemust saab kasutada hinnangu andmiseks kas raamistiku kasutusele võtt tasub end mõistliku aja perioodi vältel ära.

Lisakulud on kaetud kui:

lisakulud = töötunnid_C * kasutegur (6)

kus
töötunnid_C – töötundi arv mil Cucumber peaks olema kasutuses.

Vajalike Cucumber'i töötundide leidmiseks teisendame valemit:

$$\text{töötunnid\_C} = \frac{lisakulud}{kasutegur} \quad (7)$$

Kogu perioodi pikkuse arvutamiseks mil Cucumber hakkaks ennast ära tasuma, peab Cucumber töötundidele arvestama ka lisakuludega.
per = töötunnid_C + lisakulud (8)

kus-
per – Perioodi pikkus töötundides mille korral Cucumber'i efektiivsus kataks ära lisakulud mis tekivad selle raamistiku kasutusele võtuga.

Parema hinnagu andmiseks oleks kasulik periood teisendada ümber tööpäevadeks.

$$\text{per\_tp} = \frac{per}{töötajate\_arv * tpp} \quad (9)$$

kus –
per_tp – periooodi pikkus tööpäevades
testijate_arv – projektiga tegelevate töötajate arv
tpp – tööpäeva pikkus tundides

Vajalikud andmed arvutusteks:

Tabel 6. Investeeringu tasuvuse arvutuse andmed.

	Väärtus	**Lisa märkmed**
Analüüsi osakaal	20%	
Testide kirjutamise osakaal	65%	
Testide hooldamise osakaal	15%	
Testide kirjutamise kasutegur	80%	Tuleneb võrdluses antud hinnagust, et Cucumber võimaldab teste kirjutada 20% kiiremini
Testide hooldamise kasutegur	70%	Tuleneb võrdluses antud hinnangust, et Cucumber võimaldab teste kirjutada 30% kiiremini
Töötajate arv	6	
Tööpäeva pikkus	8h	
Lisakulud	600h	Tegemist on hinnanguga kus olen arvestanud vajamineva funktsionaaluse mahtu mida peab lisama, muu infrastruktuuri paika panemine, testijate koolitamine uue raamistiku kasutamiseks. Lisa kulude hulka ei arvesta tegevusi mis on seotud lõppotsuse tegemisega kas raamistik võtta kasutsele kuna firmas tegeldakse pidevalt uute tehnoloogiate ja lahenduste uurimisega. Lisa kulusi vähendab ka see, et lõputöö praktilise osa tegemiseks oli vajalik osa infrastruktuurist paika panna ning implementeerida lisa funktsionaalust

Valemi (5) arvutamiseks on vaja valemite (3) ja (4) tulemusi.

Valem (3):
RF ressursi jaotus = (0.2 + 0.65 + 0.15) = 1

Valem (4):
Cucumber JVM ressursi jaotus = (0.2 + 0.65 * 0.8 + 0.15 * 0.7) = 0.825

Valem(5):
ROI = $\frac{1-0.825}{0.825}$ = 0.(21) ≈0.21

Tulemuseks saab, et kasutades Cucumber'it on potentsiaalne ajavõit 21% ehk sama arv testide loomiseks ning hooldamiseks kulub Cucumber'is 79% mis oleks vaja sama paljude testide jaoks Robot Framework'is. Nüüd saab arvutada ka vajaliku kasutusaja, et kasutegur kataks ära lisa kulud.

Valem(7):
töötunnid_C = $\frac{600\ h}{0.(21)}$ ≈ 2828.6 h

Kogu vajaminev aeg, et Cucumber ennast ära tasuks saab arvutada valem(8) järgi:

perioodi töötunnid = 2828.6 h + 600 h = 3428.6 h

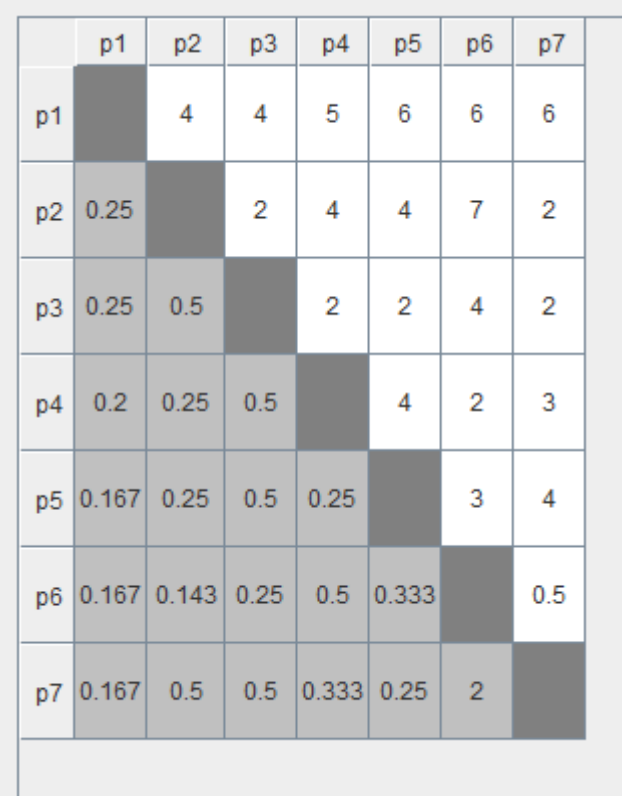
Lõpptulemuseks saame, töötundide arvu 3428.6 mis kujutab endast aega, et Cucumber JVM hakkab ennast ära tasuma ajalises mõttes. Küsimus on selles kui pikaks kujuneb perioodi pikkus nende töötundide täitmiseks. Vastuse leidmiseks kasutame valem(9):

$$\text{Periood} = \frac{3428.6\ h}{6 * 8\ h} \approx 71{,}4 \text{ tööpäeva}$$

Tuletades meelde, et uue raamistiku kasutusele võtu peamiseks eesmärgiks Profit Software'is ei oleks kiirenda testimisprotsessi vaid hoopis üle saada praeguse raamistiku piirangutest. See, et Cucumber näitab potentsiaali testimisprotsessi kiirendamisel on ainult lisa boonus ning aitab kaasa lõpliku otsuse tegemisele kas võtta raamistik kasutusele või mitte. Tulemuseks saadud 71.4 tööpäeva mis oleks vaja, et Cucumber ennast ajaliselt ära tasuks alates hetkest mil alustatakse ettevalmistustega raamistiku kasutusele võtmiseks. Seda aega võib ka tõlgenda kui kauaks testimine jääks graafikust maha. Tegemist on igati mõistlikku tulemusega, ajaline investeering hakkas ennnast suhteliselt kiiresti ära tasuma. Peale seda perioodi oleks edaspidine ajavõit 21% võrreldes praeguse lahendusega.

5.2 Analüütiliste hierarhiate meetod

Objektiivsema otsuse tegemiseks kasutan analüütiliste hierarhiate meetodit [10]. See meetodi kasutamiseks on paika vaja panna võrreldavad objektid, antud juhul Robot Framework ja Cucumber JVM. Seejärel tuleb välja tuua otsuse tegemiseks kriteeriumid. Nendeks kriteeriumiteks on: funktsionaalsus, testide kirjutamisele kuluv aeg, testide hooldamiseks kuluv aeg, testide läbimise kiirus, testi raportite kasulikkus, ressurside kasutus ja kasutusmugavus. Kriteeriumid on vaja panna tabelisse kuhu tuleb lisada nende vahelised prioriteetide kaalud, et määrata nende vahelised tähtsused. Tabeli täitmisel arvestan raamistike võrdluse käigus paika pandud prioriteetidega.



	p1	p2	p3	p4	p5	p6	p7
p1		4	4	5	6	6	6
p2	0.25		2	4	4	7	2
p3	0.25	0.5		2	2	4	2
p4	0.2	0.25	0.5		4	2	3
p5	0.167	0.25	0.5	0.25		3	4
p6	0.167	0.143	0.25	0.5	0.333		0.5
p7	0.167	0.5	0.5	0.333	0.25	2	

Joonis 12. Prioriteetide tabel.

Kus:
P1 – Funktsionaalsus
P2 – Testide kirjutamiseks kuluv aeg
P3 – Testide hooldamiseks kuluv aeg
P4 – Testide läbimise aeg
P5 – Testi raportite kasulikkus
P6 – Ressursi kasutus
P7 – Kasutus mugavus

Tabelis olevad väärtused on paika pandud Saaty fundamentaalskaala põhjal (viide):

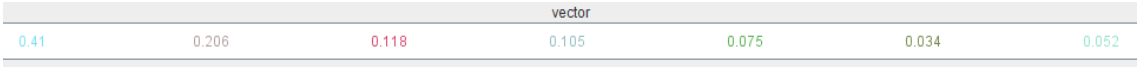
Tabel 7. Saaty undamentaalskaala otsustuste jaoks

Intensiivsus	**Definitsioon**
1	Võrdtähtis
3	Mõõdukas paremus
5	Oluline paremus
7	Väga tugev paremus
9	Ektreemne paremus

Iga kriteeriumi kohta on vaja võrrelda ka testimisraamisike omavahel, kasutades seda sama skaalat. Väärtusi on võimalik näha siit [Lisa 3].

Kasutades neid andmeid on võimalik arvutada kriteeriumite tähtsuste osakaalud, ning nende osakaalude ning raamistke omaduste abil määrata kumb raamistik on kokkuvõttes parem. Arvutuste tegemiseks kasutan programmi nimega PriEsT (Preference Elicitation using Pairwise Comparisons). Ruumi kokkuhoiu huvides toon

väja ainult vastused (Joonis 13, Joonis 14Joonis 13. Kriteeriumite tähtuse osakaalud.) [11].



Joonis 13. Kriteeriumite tähtuse osakaalud.

Vasakult paremale on näha tulemusteks: funktsionaalsus 0.41, testide kirjutamisele kuluv aeg 0.206, testide hooldamiseks kuluv aeg 0.118, testi raportite kasulikkus 0.075, ressursi kasutus 0.034, kasutusmugavus 0.052. Suurem väärtus tähista

Korrutades läbi kõik kriteeriumid testimisraamistike suhtelise võimekusega ning vastavaste raamistike tulemused kokku liita, saame lõpptulemuse (Joonis 14) mis näitab milline raamistik on nende kriteeriumite põhjal objektiivselt parem.

Joonis 14. Analüütilise hierarhia meetodi lõpptulemus(Cucumber vasakul, Robot Framework vasakul).

Analüütilise hierarhia meetodid lõpptulemust saab välja lugeda, et Cucumber JVM on oluliselt parem testimisraamistik eelnevalt paika pandud kriteeriumite põhjal.

6 Kokkuvõte

Antud töö eesmärgiks oli välja selgitada kas Cucumber JVM oleks parem testimisraamistik võrreldes Robot Framework'iga Profit Software OÜ vajadusete põhjal. Eesmärgin jõudmiseks tõin välja praeguse testimisraamistiku head küljed ja piirangud ning võrdlesin raamistike erinevaid aspekte. Kasutasin võrdlusest saaduid tulemusi, et arvutada ajalise investeerinugtasuvust ning teha sellest järeldusi kas raamistiku vahetamine oleks mõistlik.

Raamistike võrdluse käigus selgus, et Cucumber JVM on pragusest lahendusest Robot Framefork'ist mitmest küljest üle:

- Kasutab vähem arvuti ressurse mille alla kuuluvad
    - Protsessori kasutus
    - Muutmälu kasutus
    - Kõvaketta kasutus
- Testid läbivad kiiremini (7.5%)
- Suurem valik arvenduskeskkondi
- Võimekam tekstiredaktor
- Eeldused kiiremaks testide kirjutamiseks (20%) ning hooldamiseks (30 %)
- Võimaldab testi siseselt kasutada keerukaid andmetüüpe
- Debugimine tänu Java'le

Investeeringu arvutusest selgus, et ajakulu mis läheks raamistiku vahetamisele on võimalik suhteliselt kiiresti tasa teha tänu Cucumber JVM'i võimekusele. Otsutasin kasutada ROI arvutust, et määrata perioodi pikkust millal hakkab investeering ennast

ära tasuma. Määrasin raamistku vahetamise kuludeks 600 tundi, töötajate arvuks 6. Nende väärtusete põhjal sai tulemuseks, et investeering hakkab end ajaliselt ära tasuma peale 71.4 tööpäeva. Peale seda perioodi oleks edaspidine ajavõit automaattestimisel 21%. Tulemuse arvutamiseks on lihtsutatud tegelikku olukorda kuid see tulemus näitab siiski potentsaali Cucumber JVM'l olla suurepärane alternatiiv Robot Framework'ile.

Täpsemate tulemuste saamiseks oleksin pidanud vältima üleliigset lihtsutamist valmitest mida kasutasin investeering tasuvuse arvutamiseks. Ma oleks pidanud arvestama ka faktiga, et uut raamistikku hakataks kasutama ainult uute testide loomisel, tegemist ei oleks täieleiku asendusega. Pragused tulemused näitavad võimalike ajavõite vaakumis, ehk ei arvestata teste mis on juba loodud. See mõjutaks testide hooldus kulusi. Kuna uue testimisraamistikus olevate testide arve oleks esialgu väga väike võrreldes olemas olevate testidega siis reaalsete tulemuste nägemiseks võib kuluda palju aega.

Summary

The purpose of this study was to determine whether Cucumber JVM would be a better testing framework compared to Robot Framework based on the needs of Profit Software OÜ. To reach to a conclusion I described the current testing frameworks positives and limitation it has and compared the two frameworks in many aspects. I used the results from the comparrsion to calculate the return of intrest (using time investement) and find an awnwer if swithching frameworks would be a sensible choice.

The framework comparrison showed that Cucumber JVM is superior to the current solution, Robot Framework, in many ways:

- Uses less computer resources which include:
    - Processor usage
    - RAM usage

    - Hard drive usage
- Test execution is faster (7.5%)
- Larger selection of development environments
- More powerful text editors
- Potentsial to faster test writing (20%)
- Potentsial for faster test maintenance (30 %)
- Allows the usage of complex data structures in tests
- Debugging capability thanks to Java

The results of the return of intrest (ROI) show that in order to cover the time investent that would come with switching testing frameworks, it would take relatively little time to cover, thanks to the capabilities of Cucumber JVM. I decided to use ROI calculation to determine the period length when the investment would start paying off. To calculate the result I set the time needed to preparing usage of the new framework to 600 hours and the number of testers to 6. Using these values I got the final result of 71.4 work days would be required to cover the time investement. After this period the time gain for test automation would be 21%. In order to calculate these results I had to simplify the situation, nevertheless the results show that Cucumber JVM has potentsial to be a great alternative for Robot Framework.

To get more accurate results I should have refrained from excessive simplification of the formulas used to calulate return of intrest. I also should taken into account the fact that the new framework would be only used for new tests and not as a complete replacement. The current results shows possible time gains in a vaccum, as in not considering the tests that have been already been implemented. That would affect the costs of test maintance. Since amount of tests in the new framework would start off very small compared to already existing tests, then to see actual results clould take a long time.

Kasutatud kirjandus

[1] Manual testing. [WWW] https://en.wikipedia.org/wiki/Manual_testing

[2] Test automation. [WWW] https://en.wikipedia.org/wiki/Test_automation

[3] How to Calculate ROI for Test Automation. [WWW] https://www.testing-whiz.com/blog/how-to-calculate-roi-for-test, 2011.

[4] Cucumber. [WWW] https://cucumber.io/docs/reference

[5] Kristen Aebersold, Test Automation Frameworks [WWW] https://smartbear.com/learn/automated-testing/test-automation-frameworks/

[6] Robot Framework dokumentatsioon. [WWW] https://cucumber.io/docs/reference

[7] Robot Framework kasutusjuhend. [WWW] http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

[8] Jeb Rose, The Cucumber for Java Book Behaviour-Driven Development for Testers and Developers, 2015.

[9] Linda Hayes, [WWW] https://www.techwell.com/techwell-insights/2015/11/what-roi-test-automation, 2015.

[10] L.Võhandu, Subjektiivsetest hinnangutest objektiivsete tulemusteni : loengukonspekt, 1998

[11] Sajid Siraj, Preference elicitation from comparisons in multi-criteria decison making, 2011

.

Lisa 1 – Testi ’Create and manage pension agreement, death case’ sammude implementatsioonid Cucumber’is ja Robot Framework’is.

Cucumber’i sammud:

```
Scenario: Create and manage pension agreement, death case
      Given the correct agreement is imported
      When client is declared dead
      Then agreement status is changed to "Claim, Death case"
```

Robot Framework’i sammud:

```
Create and manage pension agreement, death case
      Given the correct agreement is imported
      When client is declared dead
      Then agreement status is changed to Claim, Death case
```

Lisa 2 – Testi ’Create and manage pension agreement, death case’ sammude implementatsioonid Cucumber’is ja Robot Framework’is.

Cucumber’i implementatsioon:

```
@Given("^the correct agreement is imported$")
public void the_correct_agreement_is_imported() throws Throwable {
      new TestUtils().importPolicy("policy-102-1017.zip");
}

@When("^client is declared dead$")
public void client_is_declared_dead() throws Throwable {
      new LboFrontPage(browser).openClientsSearch();
      new ClientSearchPage(browser)
            .setId("010159-889B")
            .search()
             .clickEditClient()
```

```
            .openBasicInfoTab()
            .setDeathDate("01/01/2017")
            .setDeathInformDate("01/01/2017");
      page.selectFromListByText(browser, "SubmitForm:status",
      "Deceased");
      new PersonData(browser)
      .saveClientData();
      batch.runJavaBatch("morning batch");
      batch.runJavaBatch("monthly batch");
      batch.runJavaBatch("afternoon batch");
      batch.runJavaBatch("bookkeeping");
}

@Then("^agreement status is changed to \"([^\"]*)\"$")
public void agreement_status_is_changed_to(String status) throws
Throwable {
      goToBackOffice();
      new AgreementSearchPage(browser)
            .setLastName("%%")
            .clickAgreementSearch()
            .openFirstAgreementFromResults();
      page.containsText(browser, status);
}
```

Robot Framework'i implementatsioon:

```
the correct agreement is imported
    Import Policy ${zip}

client is declared dead
    Open Clients
    Input Customer Search Pin    010159-889B
    Click Customer Search Search Btn
    Click Edit Client Btn
    Select Client Status    Deceased
    Profit Click Link    Basic info
    Input Client Death Date    01/01/2017
    Input Client Death Inform Date    01/01/2017
    Click Save And Exit Button
    Open Front Office
    Open Back Office
    Run Web Batch for Docker  Morning Batch
      Run Web Batch for Docker      Monthly batch
      Run Web Batch for Docker      Afternoon batch
      Run Web Batch for Docker      Bookkeeping

agreement status is changed to Claim, Death case
    Open Agreement In Edit Mode    ${agreement_id}
    Page Should Contain    Claim, Death case
    Save And Close Agreement
```

Lisa 3 – Raamistike suhteline võrdlus

Funktsionaalus:

	Robot Framework	Cucumber JVM
Robot Framework		0.2
Cucumber JVM	5	

Testide kirjutamiseks kuluv aeg:

	Robot Framework	Cucumber JVM
Robot Framework		0.333
Cucumber JVM	3	

Tesitde hooldamiseks kuluv aeg:

	Robot Framework	Cucumber JVM
Robot Framework		0.333
Cucumber JVM	3	

Testide läbimise kiirus:

	Robot Framework	Cucumber JVM
Robot Framework		0.5
Cucumber JVM	2	

Testi raportite kasulikkus:

	Robot Framework	Cucumber JVM
Robot Framework		1
Cucumber JVM	1	

Ressurside kasutus:

	Robot Framework	Cucumber JVM
Robot Framework		0.333
Cucumber JVM	3	

Kasutusmugavus:

	Robot Framework	Cucumber JVM
Robot Framework		0.2
Cucumber JVM	5	