TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Tarkvaratehnika õppetool

Java rakenduste ühiktestimise alternatiivraamistike empiiriline analüüs ja võrdlus AHP meetodil

Bakalaureusetöö

Üliõpilane: Kris Tambre
Üliõpilaskood: 112546
Juhendaja: Martin Rebane

Tallinn
2015

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.

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Töö eesmärgiks on anda ülevaade arenenumatest ühiktestimise alternatiivraamistikest ning võrrelda nende võimalusi ning süntaksit JUniti omadega. JUnit on omalaadsete seas kõige laialdaselt levinum ning vanem raamistik, mistõttu püstitati küsimus, et kas on olemas teisi ühiktestimise raamistikke, mis teevad oma tööd paremini, kiiremini, mida on lihtsam kirjutada või millel on rohkem võimalusi.

Töö tulemusena leidsin, et lisaks JUnitile on olemas väga häid alternatiive, mis mitme külje pealt ületavad seda nii kasutusmugavuse, kui ka funktsionaalsuse poolelt.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 52 leheküljel, 8 peatükki, 11 joonist, 6 tabelit.

Abstract

The goal of this thesis is to give an overview of the most advanced alternative frameworks for unit testing and compare their features and syntax with JUnit's. JUnit is the most widely spread and oldest framework among its' kind and for that reason a question was posed as to assert if there are others with similar purpose, which do their job better, faster, which are easier to write or have more functionality.

As a result I found out that in addition to JUnit there are several very good alternatives, which, in many ways, are better in usability and functionality than it.

The thesis is in Estonian and contains 52 pages of text, 8 chapters, 11 figures, 6 tables.

Lühendite ja mõistete sõnastik

API	***Application Programming Interface*** Programmi funktsioonide kirjeldus.
BDD	***Behaviour-Driven Development*** Arenduse meetod, mille kohaselt luuakse testid enne, kui meetodi sisu. Vt peatükk 1.6.
DDT	***Data-driven testing*** Testimise liik, kus andmed on etteantud ning teada.
Helper	Raamistikule lisatav funktsionaalsus, mis ei ole piisavalt mahukas, et seda iseseisvaks lugeda. Näiteks rohkem *assert*'e, et saaks paremini tulemust testida.
IDE	***Interactive Development Environment*** Interaktiivne arenduskeskkond.
Mock	Klassist uue instantsi loomine, mille meetodites ei ole midagi peale enda defineeritud tulemuse.
Stub	Meetod, mille sisu on ainult kohahoidja. Vt Mock.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Kuna inimressurss on kallis, siis järjest arenevas infotehnoloogilises maailmas on üha enam vajalik lihtsat ja otsekohest viisi tarkvaraliste lahenduste arenduse juures võimalikult palju automatiseerida. Ühiktestimine on nendest sammudest üks levinuimaid. Kõige levinum Java ühiktestimise raamistik on JUnit. Samas ei pruugi kõige levinum lahendus olla igaks otstarbeks kõige parem. Selles töös vaadeldakse erinevaid alternatiivseid ühiktestimise raamistikke Java rakenduste testimiseks.

1.1. Taust ja probleem

Oma praeguseks hetkeks küll lühikese töökogemuse juures on autor märganud tendentsi, et osadele arendajatele ei meeldi automaattestide protsess, või täpsemalt siis nende kirjutamine. Erinevate testraamistike kasutamine erinevatel otstarvetel nõuab tihti suuremat pingutust, kui antud funktsionaalsuse lisamine. Ühe selgeks õppimine ei aita, on vaja veel tunda *mock, stub, helper* jt kontseptsioone ning mahu kasvades kasutada skaleerimist, võibolla ka parallelisatsiooni. Hea koodikirjutamise tavasid jälgides ning koodi refaktoreerides oleks hea mõned testid parametriseerida. Seega on hea nii tööandjale kui arendajale, kui arendajal on kasutada talle mõistetav ja kõige tootlikum raamistik. Kuna ühiktestimise raamistikud on pidevas arengus, siis on põhjust rohkem ringi vaadata ning hoida silm peal potentsiaalsetel alternatiividel JUnitile.

1.2. Ülesande püstitus

Leida ja omavahel võrrelda ühte või mitut testimise raamistikku kiiruse, kasutusmugavuse ja võimaluste koha pealt erinevas mahus Java projektidele, mis kataksid funktsionaalsuselt võimalikult palju ning oleksid lihtsalt skaleeritavad.

Leida ja omavahel võrrelda üks või mitu testimise raamistikku erineval tasemel Java arendajatele.

1.3. Metoodika

Kuna töö on tihedalt seotud tarkvaraarendusega, siis lähenesin probleemidele süstemaatiliselt otsides ning kirjutades teste erinevates raamistike süntaksites, erinevas mahus ning erinevate võtetega.

1.4. Ülevaade tööst

Töö koosneb neljast osast.

Esimene osa seisneb süntaksi võrdlemises. Samale lähtekoodile kirjutatud testide koodid on välja toodud koos nende pikkuse ning klasside arvuga, mida saab igaüks ise vaadata.

Teises osas võrreldakse iga raamistiku võimalusi vastu kindlaid etteantud parameetreid. Tulemused esitatakse tabelis ning lõpus on ära toodud koondtabel lihtsaks jälgimiseks.

Kolmas osa on kiirus, kus jooksutatakse teste 1000 korda ning tuuakse tulemused välja samuti tabelikujul. Tehakse ka võimalikult suur optimiseerimine, et näha vahet sama raamistiku funktsionaalsusi maksimaalselt ära kasutades.

Neljas osa on statistika, kus võrreldakse igat parameetrit, antakse nendele kindel kaal ning arvutatakse välja parim valik igas kategoorias kasutades AHP meetodit.

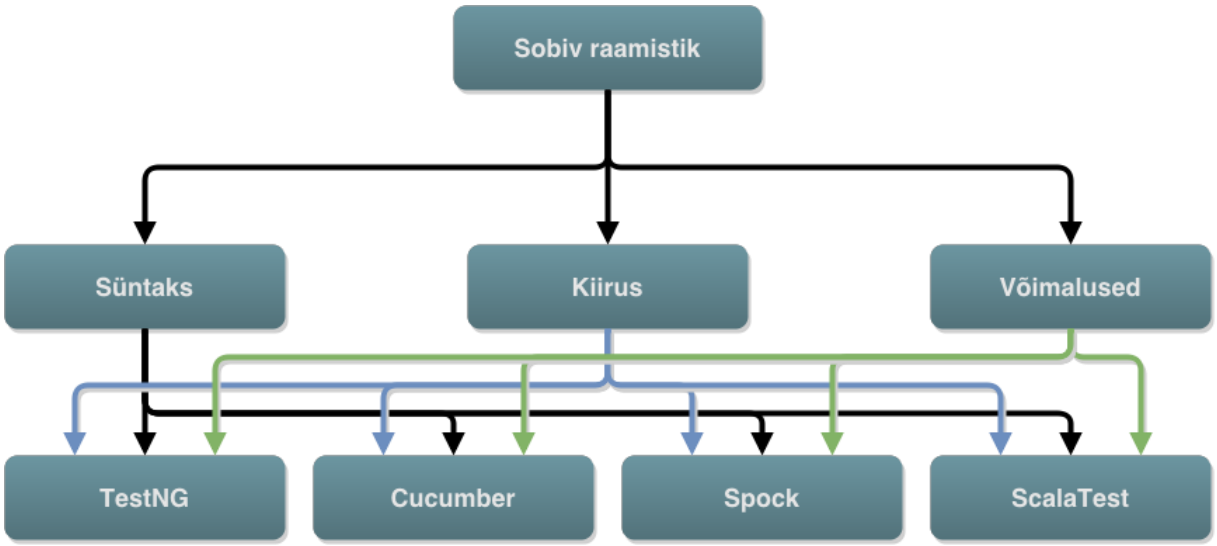
1.5. Analytical Hierarchy Process

Töös kasutatakse sobivaima raamistiku välja valimiseks *Analytical Hierarchy Process*'i. See otsuste langetamise protsess leiutati Thomas Saaty poolt Colorado ülikoolis [1]. Tema teooria kohaselt tuleks probleemidele läheneda järgides nelja sammu:

1. Defineerida probleem ning otsitav teadmine
2. Struktureerida otsuse tegemise hierarhia
3. Konstrueerida otsuse tegemiseks paaridest maatriksid
4. Maatriksite väärtuste saamiseks kasutada ära prioriteete, mida saadakse valikute üksteisega võrdlemisest

Viimases punktis lähtub autor lihtsast skaalast 1-st 9-ni, mis näitab mitu korda on mingi kriteerium tähtsam, kui teine. [1]

Antud töös näeks hierarhiline struktuur välja järgmine:



Joonis 1. AHP struktuur käesoleva töö kontekstis

Maatriksite arvutused ning tulemused asuvad peatükis 5.

1.6. Behaviour-Driven Development

Behaviour-Driven Development on tuletis *Test-Driven Development*'ist, Agile Alliance ülevaates [2] on välja toodud punktid, mida mõlemad arenduse käigus kasutavad:

- iga kasutusjuhu kohta rakendatakse „Viie Miksi" põhimõte, st kui mingi element ei tööta, siis küsitakse „Miks?" 5 korda, kuni jõutakse probleemi algeni
- mõeldakse väljapoolt sisse, mis tähendab et implementeeritakse ainult need lahendused, mis on äriloogika puhul tähtsad
- käitumised kirjeldatakse ühtselt, et kõik oleks arusaadav nii domeenieksperdidele, testijatele, kui arendajatele
- kõik need punktid rakendatakse ka kõige madalamatele tarkvara abstraktsioonidele, pannes lisarõhku käitumisele, et skaleerimine oleks odavam [2]

Ehk siis lihtsamalt öelduna BDD kasutab võimalikult kirjeldavaid meetodi- ja muutujanimesid ning nende jaoks on tavaliselt olemas mingi mall, mille järgi nimetus käib. See kõik on hea sellepärast, et on palju lihtsam tarkvaralisi vigu vältida või parandada, kui kõik on kirjeldatud kui tegevused, mitte kui funktsioonid. Antud töös kasutavad mitmed

raamistikud sellist viisi tegevuste juures (*Given, When* ja *Then* blokid), kus on täpsemalt sellist arendusviisi töös näha.

2. Süntaks

Kõik järgnevad testid on kirjutatud vastu autori loodud Araabia numbrite konverteerimist Rooma numbriteks alljärgneva koodiga:

```
public class Generator {
  public String generate(int k) {
    String val = new String();
    String[] keys = {"M", "D", "C", "L", "X", "V", "I"};
    int[] values = {1000, 500, 100, 50, 10, 5, 1};
    for (int i = 0; i < keys.length; i++) {
      while (k / values[i] > 0) {
        val += keys[i];
        k -= values[i];
      }
      for (int j = keys.length - 1; j >= i + 1; j--) {
        while (k / (values[i] - values[j]) > 0 &&
            values[i] / values[j] != 2 &&
            j <= i + 2 &&
            !(String.valueOf(values[i]).startsWith("5") &&
            String.valueOf(values[j]).startsWith("5"))) {
          val += keys[j] + keys[i];
          k = k - (values[i] - values[j]);
        }
      }
    }
    return val;
  }
}
```

Antud kood valiti eesmärgiga rakendada ühiktestimist praktikas sageli esinevale probleemile, kus testida tuleb juba valmis meetodit, mida kirjutades ei ole tingimata arvestatud ühiktestimise vajadusega.

Süntaksi võrdluses tuuakse välja raamistike keelelised erinevused ja nende loetavus võrreldes JUnitiga.

Testi eesmärk on võtta esimesed 4000 numbrit ja konverteerides need Rooma numbriteks, kinnitada nende õigsust.

Rooma numbrite kirjutamisel tuleb meeles pidada kolme reeglit:

1. Numbrit `I` saab panna `X` ja `V` ette.
2. Numbrit `X` saab panna `L` ja `C` ette.
3. Numbrit `C` saab panna `D` ja `M` ette.

Ülejäänud numbreid millegi teise ette panna ei saa. [3]
Selle kontrollimiseks kasutan regulaaravaldist:
`\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b`
[4]

Vastav test JUnitis näeb välja selline:

```
@Test
public void generatorTest() {
  Generator generator = new Generator();
  String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;
  for (int i = 1; i <= 4000; i++) {
    assertTrue(generator.generate(i).matches(regex));
  }
}
```

2.1. TestNG

Esimesena vaatleme TestNG raamistikku, mis [5] on testimise raamistik, mis on inspireeritud JUnitist ja NUnitist, kuid lisab mõned uued funktsionaalsused, muutes testimist võimsamaks ja lihtsamaks. Raamistiku autorid rõhutavad järgmiseid võimalusi:

- annotatsioonid
- mitmelõimuline testide jooksutamine – iga meetod omas lõimes, iga klass omas lõimes jne
- mitmelõimelisuse toe test sinu lähtekoodile
- paindlik konfiguratsioon
- DDT tugi (kasutades annotatsiooni @DataProvider)
- parameetrite tugi
- mitmete IDE-de ja *plug-in*'ide tugi (Eclipse, Maven jne)
- sõltuvusteta funktsioonid logimise jaoks

TestNG on disainitud toetamaks kõiki testide kategooriaid: ühiktestid, funktsionaalsed, integratsioonitestid jne. [5]

Süntaksi poolelt on TestNG üsnagi sarnane JUnitile, praeguse näite põhjal on vahe sees ainult testi tulemuste kinnituste või „*assertion*"-ite kasutamises. Testkood oli järgmine:

```
@Test
public void generatorTest() {

  Generator generator = new Generator();

 String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"

  for (int i = 1; i <= 4000; i++) {

    assert generator.generate(i).matches(regex);

  }

}
```

2.2. Cucumber

Cucumber on testimise raamistik, mis aitab luua silla arendaja ja ettevõtte või tellija äripoole vahel, muutes testid loetavaks ka ilma testimise alase oskusteabeta inimestele. Testid kirjutatakse tavalises inglise keeles kasutades BDD stiilis *Given, When ja Then* blokke, mis on üldiselt mõistetav. Testjuhud pannakse *features* faili, mis katavad ühe või rohkem teststsenaariumi [6]. Cucumber tõlgendab neid vastavasse programmeerimiskeelde ja vajadusel kasutab Seleniumit, et avada see brauseris. Cucumber kasutab testide käivitamiseks põhjana JUnitit.

BDD süntaks kasutab intuitiivselt *Given, When, Then* blokke. Need elemendid teevad järgmist:

- **Given** annab teststsenaariumile konteksti, mida käivitatakse, nagu näiteks see punkt rakenduses, millal see test esile kutsutakse koos kõigi eelnevalt vaja mineva andmestikuga.
- **When** täpsustab tegevuste hulga, mis testi jooksul käivitatakse, näiteks kasutajate või alamsüsteemide käsud.
- **Then** blokis on oodatav testi tulemus. [6]

Cucumber test näeb välja täiesti teistsugune eelnevatest. See küll kasutab testide jooksutamiseks põhjana JUnit raamistikku, kuid sisuline sarnasus puudub täielikult.

Originaalne testklass näeb välja järgnev:

```
@RunWith(Cucumber.class)

@CucumberOptions(features =
{"C:/Users/kris.BLS/Documents/NetBeansProjects/projekt/org.ttu
.epood/features/roman_number_generator.feature"})

public class GeneratorCucumberTest {

}
```

Ehk testklass kui selline on ise täiesti tühi. Ainukesed asjad, mis seal tuleb ette näidata on millist testide jooksutajat JUnit kasutab (meie näite puhul siis Cucumberi oma) ning kust võib leida *.features* faili. Selle sisu on järgmine:

```
Feature:Roman Number Generator

To generate Arabian numerals to Roman numerals



Scenario:Generate x Arabian numerals to Roman

Given The Roman number generator

When User wants to generate numbers between '1' and '4000' to
Roman

Then The result should match
'M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})'
```

See fail kirjeldab ära õiges inglise keeles mida antud test täpsemalt teeb. Testi loogika ise asub veel kolmandas failis, mille sisu on järgmine:

```
public class GeneratorCucumberSteps {



  Generator g;

  String[] res = new String[4001];



  @Given("The Roman number generator")

  public void generator() {

    g = new Generator();

  }

@When("^User wants to generate numbers between '(\\d+)' and
'(\\d+)' to Roman$")

  public void generating(int from, int to) {

    for (int i = from; i <= to; i++) {

      res[i] = g.generate(i);

    }

  }
```

```
  @Then("The result should match '(.+)'")
  public void checking(String s) {
    for (int i = 1; i <= 4000; i++) {
      assertTrue(res[i].matches(s));
    }
  }
}
```

2.3. Spock

Spock on ühiktestimise raamistik, mis kasutab Groovy-t ning sarnaselt Cucumberile ka *Given, When* ja *Then* blokke. Hoolimata sellest saab seda kasutada ka tavaliste Java klasside testimiseks. Groovy on loodud Google poolt, mis on teadagi suur tehnoloogiafirma ning tänu läbimõeldud disainile on süntaks väga lihtne õppida ja silmale hea vaadata. Testi loomine võtab vähem aega, kui mõnes tema ekvivalendis (JUnit+Mockito näiteks), kuna siia on juba sisse ehitatud erinevad *mock, spy* ja *stub* käsud [7].

Süntaksi poolelt ei ole Spockis ei annotatsioone ega *assertion* API-t [8], vaid tulemuse õigsuses veendumiseks kasutatakse *expect:* blokki. See kõik muudab testkoodi minimalistlikuks – ainult vajalik jääb. Hea on veel see, et testi jooksutamise raamistik on JUnit ning see on juba igas suuremas IDE-s sees. Kui arendaja on sellega enam-vähem tuttav, siis on Spock testide kirjutama hakkamine suuremalt jaolt valutu. Selle töö kirjutamise ajal kahjuks olid erinevad probleemid NetBeansi kasutamisega nii Groovy, kui Spockiga ja sellepärast soovitan mina kasutada kas Eclipse-i või IntelliJ-d.

Spock ise suunab arendajaid rohkem BDD poolele testide kirjutamise osas, kuid see ei ole üldse kohustuslik ning antud on üldiselt vabad käed kuidas midagi teha.

Asjakohane test näeb välja selline:

```
class GeneratorSpockTest extends Specification{

     def "generation"() {

          expect:

          answer ==~
("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b
");



          where:

          i << (1..4000);

          answer = new Generator().generate(i);

     }

}
```

2.4. ScalaTest

ScalaTest on üles ehitatud, nagu nimigi juba näitab, Scalale, kuid see ei tähenda, et sellega Java klasse testida ei saa. Pakkudes täielikku integratsioonivõimalust populaarsete raamistikega, nagu JUnit, TestNG, Ant, Maven, JMock, EasyMock, Mockito, Selenium ja palju muud, võib selle kasutamine osutuda palju otstarbekamaks ning kiiremaks kui teised ekvivalendid. Nagu ka paljud teised osutab ScalaTest rohkem BDD poole, kuid annab ette päris mitu malli ka muudele stiilidele. Antud testis kasutasin ma `FlatSpec`'i [9], kuna selle süntaks tundus mulle teistest loetavam. Positiivne on ka ScalaTesti võimalus juba olemasolevaid JUnit või TestNG teste mitte ümber kirjutada, vaid hoopis jätkata olemasolevaid teste säilitades Scalaga.

Teistest erinev on ka ScalaTestil veel peale teste näidatav info konsoolis. Seal on peale testi jooksmise aja ja (eba)õnnestumise veel ka lihtsalt muudetav info selle kohta, mida antud meetod tegema peaks ning veel jooksutamata testide arv. [10]

ScalaTestiga `FlatSpec` malli kasutades näeb test välja järgmine:

```
class GeneratorScalaTest extends FlatSpec with MustMatchers{

  "generated value" should "match
'\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b'
" in{

    var generator = new Generator();

    for(i <- 1 to 4000){

      generator.generate(i) must fullyMatch
regex("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3
})\\b");

    }

  }

}
```

Ning konsoolis kuvatud info on selline:

```
Run starting. Expected test count is: 1

GeneratorScalaTest:

generated value

- should match
'\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'

Run completed in 328 milliseconds.

Total number of tests run: 1

Suites: completed 1, aborted 0

Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0

All tests passed.
```

3. Tehnilised võimalused

Järgnevalt vaatame kõikide eelmainitud raamistike tehnilisi võimalusi võrreldes JUnitiga ehk mida miski oskab rohkem või vähem teha. Tulemused on esitatud tabelikujul ning testitavateks parameetriteks on:

- testi ignoreerimise võimalus
- testide grupeerimise võimalus
- testide kategoriseerimise võimalus
- võimalus käivitada teste paralleelselt
- võimalus käivitada meetodit enne/pärast igat testi
- võimalus käivitada meetodit enne/pärast testklassi
- võimalus käivitada meetodit enne/pärast testide gruppi
- võimalus käivitada meetodit enne/pärast testide kategooriat
- võimalus testida erindite viskamist
- võimalus luua parameetritega teste
- võimalus luua ajalise piiranguga teste
- võimalus käivitada meetodeid kindlas järjekorras

Valik tulenes nii isiklikust kogemusest, kuna autori ametikohal on kasutusel JUnit kui ka taustauuringu tulemusel. Nimekirja koostades lähtusin eelkõige sellistest kriteeriumitest, millega arendajatel teste kirjutades rohkem probleeme näis olevat või mis puuduolevaid funktsionaalsusi raamistike puhul kõige enam otsiti.

JUniti puhul näeks võimaluste tabel välja alljärgnev:

Võimalus	**Olemasolu**
Testi ignoreerimine	Jah, kasutades annotatsiooni *@Ignore*
Testide grupeerimine	Jah, kasutades annotatsiooni *@Suite*
Testide kategoriseerimine	Jah, kasutades annotatsiooni *@Category*
Testide paralleelselt käivitamise võimalus	Jah, eksperimentaalne võimalus kasutades *ParallelComputer* klassi JUnit4-s. Väga kohmakas. [11]
Meetodi käivitamine enne/pärast igat testi	Jah, kasutades annotatsioone *@Before* ja *@After*
Meetodi käivitamine enne/pärast igat testklassi	Jah, kasutades annotatsioone *@BeforeClass* ja *@AfterClass*
Meetodi käivitamine enne/pärast igat testgruppi	Ei
Meetodi käivitamine enne/pärast igat testkategooriat	Ei
Erindite viskamise testimine	Jah, kasutades annotatsiooni *@Test(expected = Exception.class)*
Parametriseeritud testide loomine	Jah [12]
Ajalise piiranguga testide loomine	Jah, kasutades annotatsiooni *@Test(timeout=<aeg ms>)*
Kindlas järjekorras testide käivitamine	Jah, kasutades annotatsiooni *@FixMethodOrder* [12]

Tabel 1. JUniti võimalused

Nagu näha, siis on JUnit disainitud käivitamaks teste ükshaaval ja maksimaalses isolatsioonis. Testide mingis kindlas järjekorras käivitamine on alates versioonist 4.11 olemas, kuid see kehtib ühes klassis kõikide meetodite kohta. Olles mingis olukorras, kus on vaja käivitada ainult 2 testi kindlas järjekorras, peab nendele tegema oma klassi, näiteks kasutaja loomine ja kasutaja muutmine.

Samuti võime tabelist välja lugeda, et ilma lisavahenditeta on JUnitis mõttetu kategooriate ja gruppide loomine, kuna enne/peale neid ei ole võimalik käivitada mingeid meetodeid.

3.1. TestNG

Võimalus	Olemasolu
Testi ignoreerimine	Jah, kasutades annotatsiooni *@Test(enabled=false)* [13]
Testide grupeerimine	Jah, defineerides grupeeritud klassid XML-iga [13]
Testide kategoriseerimine	Jah, kasutades annotatsiooni *@Test(groups = {'group1', 'group2'})* [13]. Siinkohal on kategooriad ja grupid teistpidi.
Testide paralleelselt käivitamise võimalus	Jah, kas defineerides *thread-count* ja *parallel* parameetrid grupi XML-is või ühe testi puhul kasutades annotatsiooni *@Test(threadPoolSize = <num>, invocationCount = <num>)* [13]
Meetodi käivitamine enne/pärast igat testi	Jah, kasutades annotatsioone *@BeforeTest* ning *@AfterTest*
Meetodi käivitamine enne/pärast igat testklassi	Jah, kasutades annotatsioone *@BeforeClass* ja *@AfterClass*
Meetodi käivitamine enne/pärast igat testgruppi	Jah, kasutades annotatsioone *@BeforeSuite* ja *@AfterSuite* [13]
Meetodi käivitamine enne/pärast igat testkategooriat	Jah, kasutades annotatsioone *@BeforeGroups* ja *@AfterGroups* [13]
Erindite viskamise testimine	Jah, kasutades annotatsiooni *@Test(expectedExceptions = Exception.class)* [13]
Parametriseeritud testide loomine	Jah, kasutades testklassi sees meetodil annotatsiooni *@DataProvider(name =*

	'nimi') või testklassil annotatsiooni *@Parameters({'parameters1'})* ning defineerides need XML-i kaudu. [13]
Ajalise piiranguga testide loomine	Jah, kasutades annotatsiooni *@Test(timeOut=<aeg ms>)* [13]
Kindlas järjekorras testide käivitamine	Jah, defineerides testide jooksmise järjekorra XML-i kaudu [13]

Tabel 2. TestNG võimalused

Lisaks kõigele nendele pakub käesolev raamistik ka võimalust jooksutada JUnit3 ja JUnit4 teste, ehk kui on vajadus või tahtmine JUniti kasutamiselt TestNG kasutamisele üle minna, siis saab seda väga lihtsalt teha. Samuti on väga hea testida veebiteenuseid, kus on vajalik sisselogimine – kasutades *@BeforeSuite* annotatsiooni saab enne sisse logida, seejärel jooksutada kõik testid paralleelselt ning *@AfterSuite* annotatsiooni kasutades pärast uuesti välja logida.

Testide paralleelselt käivitamisel tuleb silmas pidada seda, et ei ole mõtet panna lõimede arvuks rohkem, kui arvutil on füüsilisi protsessoreid või tuumi, kuna peale seda ei muutu testide jooksutamine enam kiiremaks. Sellisel juhul hakkab protsessor võtma *task*'e kordamööda erinevatelt tuumadelt, mitte ei täida ülesannet enam paralleelselt.

3.2. Cucumber

Võimalus	Olemasolu
Testi ignoreerimine	Jah, kasutades *tag*'e [14]
Testide grupeerimine	Jah, kasutades *tag*'e [14]
Testide kategoriseerimine	Ei
Testide paralleelselt käivitamise võimalus	Ei
Meetodi käivitamine enne/pärast igat testi	Ei
Meetodi käivitamine enne/pärast igat testklassi	Jah, kui ühes grupis (*scenario*s) on ainult 1 testklass, muidu ei [15]
Meetodi käivitamine enne/pärast igat testgruppi	Jah, kasutades *Hook*'e [15]
Meetodi käivitamine enne/pärast igat testkategooriat	Ei
Erindite viskamise testimine	Ei
Parametriseeritud testide loomine	Jah, ehitatud üles kohe parameetrite peale
Ajalise piiranguga testide loomine	Jah, kasutades *Hook*'e [15]
Kindlas järjekorras testide käivitamine	Jah, testid käivitatakse *.features* failide järjekorras

Tabel 3. Cucumberi võimalused

Cucumberi võimaluste väheseks jäämise põhjuseks võib pidada seda, et see on saadaval paljudel erinevatel platvormidel [16].

3.3. Spock

Võimalus	Olemasolu
Testi ignoreerimine	Jah, annotatsiooni *@Ignore* [17]
Testide grupeerimine	Tehniliselt võimalik kasutades JUnit4 *@Suite* annotatsiooni, muidu ei
Testide kategoriseerimine	Ei
Testide paralleelselt käivitamise võimalus	Ei
Meetodi käivitamine enne/pärast igat testi	Jah, defineerides bloki *setup()* või *cleanup()* [17]
Meetodi käivitamine enne/pärast igat testklassi	Jah, defineerides bloki *setupSpec()* või *cleanupSpec()* [17]
Meetodi käivitamine enne/pärast igat testgruppi	Ei
Meetodi käivitamine enne/pärast igat testkategooriat	Ei
Erindite viskamise testimine	Jah, kasutades annotatsiooni *@FailsWith* [17]
Parametriseeritud testide loomine	Jah [18]
Ajalise piiranguga testide loomine	Jah, kasutades annotatsiooni *@Timeout* [17]
Kindlas järjekorras testide käivitamine	Jah [19]

Tabel 4. Spocki võimalused

3.4. ScalaTest

Võimalus	Olemasolu
Testi ignoreerimine	Jah, kasutades *tag*'i *ignore* [20]
Testide grupeerimine	Ei
Testide kategoriseerimine	Jah, kasutades ettemääratud *Spec*'e [21]
Testide paralleelselt käivitamise võimalus	Jah, kasutades klassi *ParallelTestExecution* [22]
Meetodi käivitamine enne/pärast igat testi	Jah, kasutades *FunSuite*'i ning defineerides *before{}* või *after{}* [23]
Meetodi käivitamine enne/pärast igat testklassi	Jah [24]
Meetodi käivitamine enne/pärast igat testgruppi	Ei
Meetodi käivitamine enne/pärast igat testkategooriat	Ei
Erindite viskamise testimine	Jah, kasutades lihtsat *try...catch* blokki, muidu ei [25]
Parametriseeritud testide loomine	Ei
Ajalise piiranguga testide loomine	Jah [26]
Kindlas järjekorras testide käivitamine	Ei

Tabel 5. ScalaTesti võimalused

3.5. Kokkuvõte

Kokkuvõttev tabel erinevate testimisraamistike võimaluste kohta on järgmine:

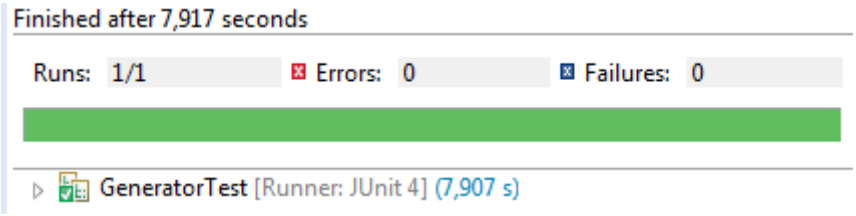
Võimalus	TestNG	Cucumber	Spock	ScalaTest
Testi ignoreerimine	Jah	Jah	Jah	Jah
Testide grupeerimine	Jah	Jah	vt 3.3	Ei
Testide kategoriseerimine	Jah	Ei	Ei	Jah
Testide paralleelselt käivitamise võimalus	Jah	Ei	Ei	Jah
Meetodi käivitamine enne/pärast igat testi	Jah	Ei	Jah	Jah
Meetodi käivitamine enne/pärast igat testklassi	Jah	vt 3.2	Jah	Jah
Meetodi käivitamine enne/pärast igat testgruppi	Jah	Jah	Ei	Ei
Meetodi käivitamine enne/pärast igat testkategooriat	Jah	Ei	Ei	Ei
Erindite viskamise testimine	Jah	Ei	Jah	Jah
Parametriseeritud testide loomine	Jah	Jah	Jah	Ei
Ajalise piiranguga testide loomine	Jah	Jah	Jah	Jah
Kindlas järjekorras testide käivitamine	Jah	Jah	Jah	Ei

Tabel 6. Raamistike võimaluste võrdlus

4. Kiirus

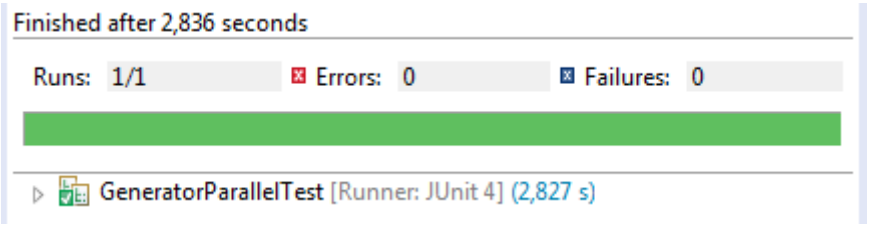
Antud peatükis jooksutatakse teste mitmeid kordi, et luua mingisugune ettekujutus nende jõudluse kohta. Eraldi võrreldakse paralleliseeritud teste, kus see on võimalik. Kasutatud mitmelõimeline testikood asub lisas.

Kõigepealt näitan ära JUniti tulemused. Hetkel puudub antud raamistikul lihtsalt arusaadav võimalus jooksutamaks sama meetodit mitu korda järjest, seega ma tegin lihtsa tsükli.
Jooksutades testi programmaatiliselt 1000 korda saame tulemuseks 7,907 sekundit:



Joonis 2. JUnit 1 lõimega.

Jooksutades testi programmaatiliselt 1000 korda 4 lõimega saame tulemuseks 2,827 sekundit:

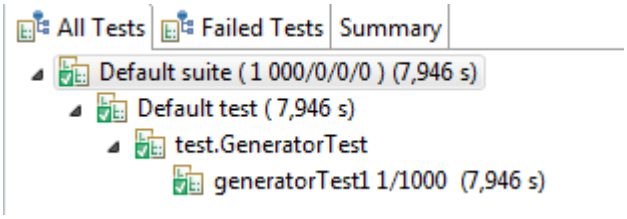


Joonis 3. JUnit 4 lõimega.

Kahjuks võimaldab *ParallelComputer* klass ainult korraga 4 lõime kasutamist, mistõttu võib see jääda pudelikaelaks, kui serveris on rohkem füüsilisi protsessoreid. Nagu näha on tulemus parallelisatsiooni kasutades 2,8 korda parem. Neljakordne vahe on võimatu, kuna JVM-i käivitamine, kompileerimine ning testide ettevalmistus võtavad oma aja. Samamoodi on Javal uute lõimede loomisel tekkiv *overhead* üsna suur. See tähendab seda, et mida vähem teste on, seda rohkem see rolli mängib ja seda suuremaks läheb vahe mahu suurenedes. [27]

4.1. TestNG

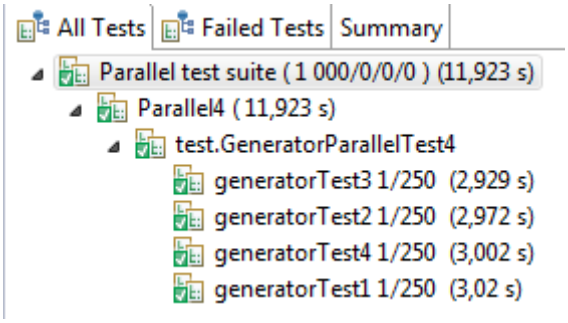
Jooksutades TestNG raamistikul sama testi programmaatiliselt 1000 korda on 1 lõimega tulemus järgmine:



Joonis 4. TestNG 1 lõimega.

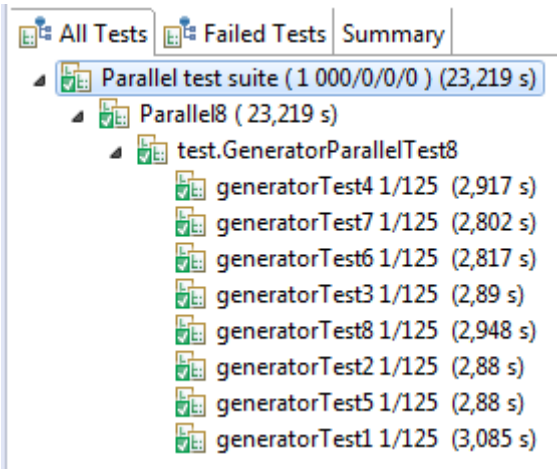
Kõik kokku arvutatuna on tulemuseks 7,946 sekundit. See on umbes sama, kui JUnitil, kuid süntaksi poolelt on siinkohal lihtsam jooksutada ühte testi mitu korda – tuleb lihtsalt kasutada *invocationCount* parameetrit *@Test* annotatsioonis (praegusel juhul *@Test(invocationCount = 1000)*).

Jooksutades nüüd seda sama testi 4 lõime peal, saame tulemuseks:



Joonis 5. TestNG 4 lõimega.

Nüüd on kiirus kõige aeglasema lõime lõpetamise aeg ehk 3,02 sekundit. See on veidi aeglasem, kui JUnitil, kuid ma saan praegu kasutada ka 8-t tuuma. Tulemus on järgmine:



Joonis 6. TestNG 8 lõimega.

TestNG näitab iga testgrupi kohta summeeritud aega, isegi, kui testid on paralleelselt üles seatud. See tekitab veidikene segadust, aga siiski jooksid need siinkohal erinevates lõimedes. Vaadates ja võrreldes aegu saame, et kõrvuti lähevad testid läbi umbes 2,6 korda kiiremini, kui järjest. See number sarnaneb JUniti omaga.

Kuna testiks kasutataval arvutil siiski 8-t füüsilist tuuma ei ole, siis on tulemus sama, mis 4-ga, kuid potentsiaalselt saab serveril paralleelselt teste käivitada täpselt nii palju, kui on olemas protsessoreid. See tagab väga suuremahuliste projektide juures väga suure ajavõidu, kuid vaadates eelnevaid pilte, siis on sealt näha, et kui lõimude arv ületab tuumade arvu, siis ei ole enam vahet kui mitu testi paralleelselt jookseb – tulemus jääb siiski samaks. Siit võibki järeldada, et reaalset ajavõitu saavutatakse ainult siis, kui on olemas füüsiliselt eksisteeriv protsessor, mis on valmis ülesande koheselt vastu võtma.

4.2. Cucumber

Kahjuks Cucumber ise ei toeta ühe *scenario* mitu korda järjest jooksutamist [28].
Käsitsi jooksutamisel kahjuks lõpetatakse JVM-i töö vahepeal ära, mistõttu ei ole tulemused usaldusväärsed, seega peame usaldama siinkohal ainult ühte tulemust, milleks on:

```
1 Scenarios (1 passed)
3 Steps (3 passed)
0m0.099s

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.394 sec

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

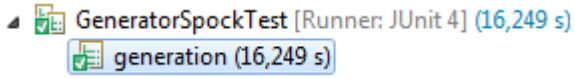
Joonis 7. Cucumber 1 lõimega

See tulemus on 10st käsitsi jooksutatud testist parim. Testi enda jooksutamine võttis aega siis 99ms ehk 1000 testi jaoks läheks vaja 99 sekundit. Cucumberis on kaotatud aja arvelt tehtud tasa süntaksi arusaadavusega, mistõttu saavad teste kirjutada ka arendajad, kellel on kogemust veel vähe. Võib-olla oleks mõeldav selliste testide kasutuselevõtt mõne aine raames.

Testimata jääb ka Cucumberi testide paralleelsus, kuna selle töö kirjutamise ajal puudus vastav tugi.

4.3. Spock

Ühe lõimega Spock testide tulemus on järgmine:



Joonis 8. Spock 1 lõimega.

Hetkene tulemus on aeglasem, kui JUnitil ja TestNG-l. Kahjuks puudub hetkese seisuga ka Spockil iseseisev testide paralleelselt käivitamise võimalus. Samamoodi ei ole Spockil ka lihtsalt arusaadavat testide kordamise süntaksit. Võrreldes neid näitajaid teiste raamistikega, võime selle lugeda kuskile TestNG ja Cucumberi vahepeale testide loetavuse ning kiiruse poolelt.

4.4. ScalaTest

Pärast ScalaTesti 1000 tsükli lõppu oli tulemus järgmine:

```
Run starting. Expected test count is: 1
GeneratorScalaTest:
generated value
- should match '\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'
Run completed in 10 seconds, 574 milliseconds.
Total number of tests run: 1
Suites: completed 1, aborted 0
Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
```

Joonis 9. ScalaTest 1 lõimega

Ka siinkohal ei ole ametlikku tuge ühe testi mitu korda jooksutamisel, seega kasutasin täiesti tavalist for-tsüklit. Kiiruseks näeme, et tuli 10,574 sekundit, mis on suhteliselt hea tulemus. Kasutades 4 lõime saame tulemuseks järgneva:

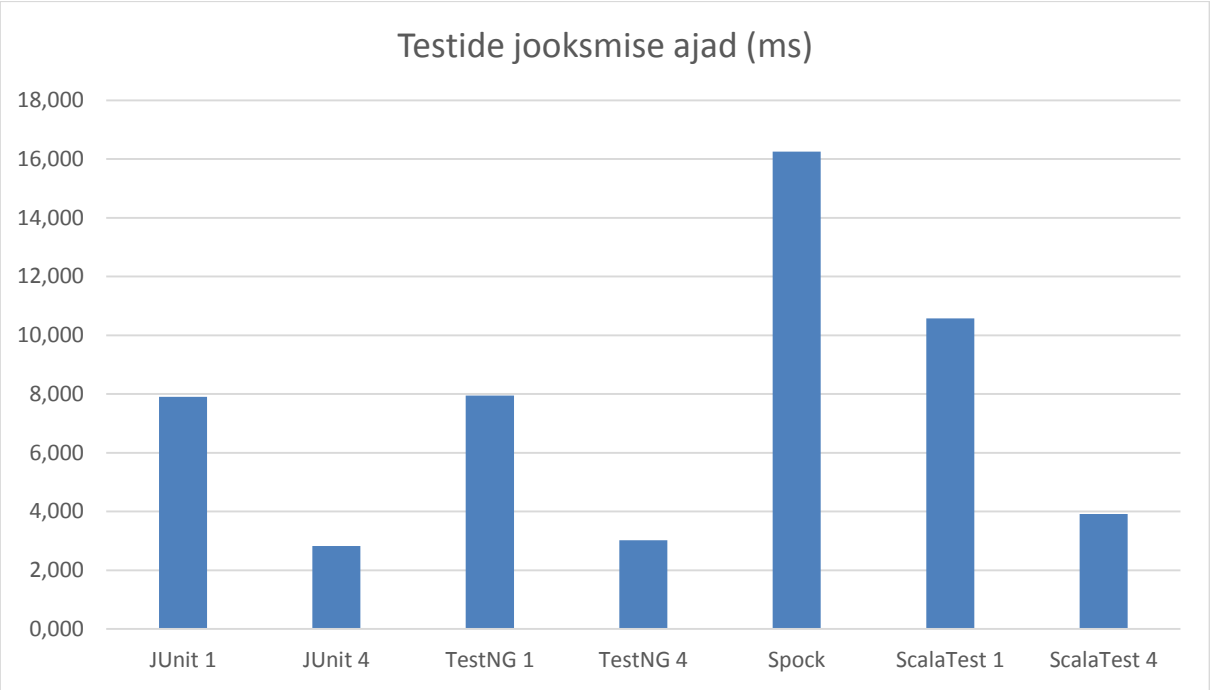
```
Run starting. Expected test count is: 4
GeneratorScalaTest:
1) generated value
2) generated value
- should match '\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'
- should match '\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'
3) generated value
- should match '\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'
4) generated value
- should match '\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\b'
Run completed in 3 seconds, 917 milliseconds.
Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
```

Joonis 10. ScalaTest 4 lõimega

Ajavõit oli peaaegu 2,7-kordne, mis ühtib ka TestNG ja JUnit ajavõidu kordajatega.

4.5. Kokkuvõte

Kõikide raamistike tulemused on võrdlemiseks välja toodud järgneval joonisel.



Joonis 11. Testide täitmise ajad, number raamistiku lõpus näitab kasutatud lõimede arvu

Pidin välja jätma Cucumberi, kuna antud raamistiku aeg oli võrreldes teistega üsna suur. Kiiruse koha pealt on asi üsna selge – kui on tähtis see, et võimalikult kiiresti saaksid testid jooksutatud, siis langeb valik paratamatult JUniti ning TestNG vahele, mis on ka loogiline, kuna nad kasutavad vaikimisi Java süntaksit. Teised nö 'dialektid' peab nende kompilaator ära tõlkima ning alles seejärel saadakse klass kokku pakkida ja siis seda jooksutada, kuid kiirematel jääb see samm vahele. Samuti kasutavad tagumised oma testide jooksutamise mootorina JUnitit, mistõttu ei olegi reaalne, et nad saaksid kuidagi kiiremad olla.

5. Parima raamistiku valimine

5.1. Parameetrite vektorid

Nagu juba sissejuhatuses mainitud, kasutan parima raamistiku välja selgitamisel *Analytical Hierarchy Process*'i (AHP). Veel vajab täpsustamist see, et kõigepealt selgitan ma välja kõigi kolme kriteeriumi parima ning alles peale seda saab üldistava järelduse teha.

AHP eeldab, et iga kriteeriumi jaoks koostatakse kõigepealt maatriksid erinevate võimaluste võrdlemiseks. Maatriksitele väärtuste saamiseks peab otsustama, kui mitu korda on mingi parameeter teisest parem. Kuna parameetritele väärtuse määramine on suhteliselt subjektiivne tegevus, siis selleks, et mitte eelistada ühte parameetrit teisele, kasutan ma kõikide arvutamiseks kolme väärtust: 1, 4 ja 6. Väärtuste valikul on silmas peetud seda, et ühe tähtsama kategooria valikuga kaasnevad võidud ka teises, näiteks kiiruse, peab arendaja ka hästi antud raamistiku API-t tundma. Süntaksi puhul näeks väärtuste järjekord välja selline:

1. Kiiruse võtan põhjaks, mis tähendab, et selle väärtuseks jääb 1. Seda just sellel põhjusel, et kui inglise keelele lähedus on olulisim, siis on ilmselt tegemist vähekogenud arendajaga ning koodi loogika mõistmine on tähtsam, kui maksimaalne optimeeritus.
2. Tehnilisi võimalusi arvestan, kui keskmisest veidi olulisemat, mis viitab väärtusele 4, ehk neli korda tähtsam, kui kiirus. Põhjuseks võib lugeda selle, et raamistiku API tundmine on väga vajalik, ilma selleta ei ole võimalik kirjutada hästi loetavat koodi.
3. Viimasena, süntaks on praegu ilmselgelt tugevalt olulisem kiirusest ning keskmiselt olulisem eelmisest punktist. See viitab väärtusele 6.

Nüüd saame kokku panna maatriksi:

$$S = \begin{vmatrix} 1 & 1{,}5 & 6 \\ 0{,}6667 & 1 & 4 \\ 0{,}1667 & 0{,}25 & 1 \end{vmatrix}$$

Antud maatriksis on esimene rida ja veerg süntaks, teine jäi tehnilistele võimalustele ja kolmas kiirusele. Antud väärtused seal on arvutatud nii, et iga väärtus võrreldes iseendaga on 1, mis teeb peadiagonaalide arvudeks ühed. Seejärel on leitud rea väärtuse ja veeru väärtuse jagatis, näiteks teise rea esimene veerg – seal on rea peal süntaks väärtusega 6 ning tehnilised võimalused väärtusega 4. Veeru jagatisel reaga saame 2/3, mille ümardame nelja komakohani.

Järgnevalt võtame antud maatriksi ruutu ning arvutame esimesed eigenvektorid. Selleks me kõigepealt liidame ridades olevad arvud üksteisega, seejärel liidame kokku tekkinud veeru arvud üksteisega ning lõpus jagame tekkinud veeru arvud kokku liidetud veeru arvuga, saame:

$$S_2 = \begin{vmatrix} 3{,}0003 & 4{,}5 & 18 \\ 2{,}0002 & 3{,}0001 & 12{,}0002 \\ 0{,}5001 & 0{,}7501 & 3{,}0002 \end{vmatrix} \rightarrow \begin{vmatrix} 25{,}5003 \\ 17{,}0005 \\ 4{,}2504 \end{vmatrix} \rightarrow \begin{matrix} 0{,}5454 \\ 0{,}3636 \\ 0{,}0909 \end{matrix}$$

Seda protsessi tuleb korrata nii kaua, kuni Eigenvektor enam väga palju ei muutu. Peale järgmist iteratsiooni on tulemus selline:

$$S_3 = \begin{vmatrix} 27{,}0038 & 40{,}5023 & 162{,}009 \\ 18{,}0028 & 27{,}002 & 108{,}008 \\ 4{,}5009 & 6{,}7508 & 27{,}0033 \end{vmatrix} \rightarrow \begin{matrix} 229{,}5151 \\ 153{,}0128 \\ 38{,}255 \end{matrix} \rightarrow \begin{matrix} 0{,}5454 \\ 0{,}3636 \\ 0{,}0909 \end{matrix}$$

Tulemus on täpselt sama, mis eelnevalt, mis tähendab, et eigenvektorid on leitud.

Tehniliste võimaluste väärtused näevad välja järgmised:

1. Süntaksi väärtuseks jääb seekord 1. Seda sellel põhjusel, et kui on tunda, et testide kirjutamisel jääb puudu just funktsionaalsusest, siis on see märk sellest, et arendaja tunneb ennast antud raamistikus juba mugavalt ning uue valikul tuleks inglise keele lähedusest vähem lähtuda.

2. Kiiruse väärtuseks valisin 4, kuna uue funktsionaalsusega käib tavaliselt kaasas ka vajadus hakata teste optimeerima kiiruse suhtes.

3. Viimasena jääb siis võimaluste väärtuseks 6.

Sellest tulenevalt näeb tema maatriks koos eigenvektoritega välja järgmine:

$$T = \begin{vmatrix} 0{,}1667 & 0{,}25 & 1 \\ 1 & 1{,}5 & 6 \\ 0{,}6667 & 1 & 4 \end{vmatrix} \rightarrow \cdots \rightarrow \begin{matrix} 0{,}0909 \\ 0{,}5454 \\ 0{,}3636 \end{matrix}$$

Ning viimasena kiiruse puhul valisin väärtused järgmisena:

1. Kõige suurema kiiruse saamiseks on vaja kõige paremat optimiseerimist, mida lihtsa süntaksiga raamistikud ei võimalda, seega jääb siinkohal väärtuseks 1.
2. Minimaalse ajakuluga testide jooksutamiseks on vaja teada ka raamistike API-sid üle keskmise taseme, mistõttu jääb tehniliste võimaluste väärtuseks 4
3. Kuna kiirus on siinkohal tähtsaim, siis jääb siia väärtuseks 6.

Antud maatriks koos vektoritega on selline:

$$K = \begin{vmatrix} 0{,}1667 & 0{,}25 & 1 \\ 0{,}6667 & 1 & 4 \\ 1 & 1{,}5 & 6 \end{vmatrix} \rightarrow \cdots \rightarrow \begin{matrix} 0{,}0909 \\ 0{,}3636 \\ 0{,}5454 \end{matrix}$$

5.2. Raamistike maatriks

Nüüd tuleb sama teha läbi raamistikega, aga vastu kõiki kriteeriume. Neid maatrikseid saab kasutada ka teiste parameetrite esikohale paneku puhul.

Süntaksi kohal arvestan ma testi kirjutamisel inglise keele lähedust, mida kõrgem väärtus seda lähedamal ehk seda arusaadavam on kood inimesele, kes ei tea programmeerimisest midagi.

Põhjana kasutasin ma seekord TestNG-d, kuna antud raamistikul on kõige rohkem rõhku pandud funktsionaalsusele ja kiirusele. Selle väärtuseks jääb 1. ScalaTestil on inglise keelele lähedam lõpptulemuse oodatava tulemuse süntaks, aga ülejäänud on loogilises mõttes üsna sarnane, mistõttu saab tema väärtuseks 3. Spock, kasutades Groovy süntaksit ja erinevaid blokke erinevateks tegevusteks saab väärtuseks 5 ning Cucumber tänu oma *features* failidele 6. Maatriksi panen suuruse järjekorras, mis tähendab, et esimene rida ja veerg on Cucumber, teine Spock jne. Eigenvektorid näevad välja järgmised:

$$S = \begin{vmatrix} 1 & 1{,}2 & 2 & 6 \\ 0{,}8333 & 1 & 1{,}6667 & 5 \\ 0{,}5 & 0{,}6 & 1 & 3 \\ 0{,}1667 & 0{,}2 & 0{,}3333 & 1 \end{vmatrix} \rightarrow \cdots \rightarrow \begin{matrix} 0{,}4 \\ 0{,}3333 \\ 0{,}2 \\ 0{,}0667 \end{matrix}$$

Liikudes edasi tehniliste võimaluste juurde, arvestasin punkte nii, et iga puuduva funktsionaalsuse eest sai 1 miinuspunkti. Kui puudus võimalus teha midagi, mis oli mingi teise puuduva võimaluse järeldus, siis selle eest lisamiinust ei saanud, näiteks testide kategoriseerimine ning meetodi käivitamine enne või peale igat testkategooriat. Sellest

tulenevalt tekkis seis, kus Cucumber sai 5 miinuspunkti, Spock 3, ScalaTest 4 ning TestNG 0. Seejärel, et kasutada ära võimalikult palju skaalast, korrutasin vahed läbi 1,5-ga ning ümardasin ülespoole. Peale seda kõike kasutasin Cucumberit põhjana, mis tähendas talle väärtust 1, ScalaTest sai väärtuseks 3, Spock 5 ning TestNG 8. Maatriks on, nagu eelnevategi puhul, suuruse järjekorras ning koos eigenvektoritega näeb välja järgmine:

$$F = \begin{vmatrix} 1 & 1{,}6 & 2{,}6667 & 8 \\ 0{,}625 & 1 & 1{,}6667 & 5 \\ 0{,}375 & 0{,}6 & 1 & 3 \\ 0{,}125 & 0{,}2 & 0{,}3333 & 1 \end{vmatrix} \rightarrow \cdots \rightarrow \begin{matrix} 0{,}4706 \\ 0{,}2941 \\ 0{,}1765 \\ 0{,}0589 \end{matrix}$$

Kiiruse puhul võtsin arvutasin protsentuaalselt, kui palju mingi raamistik kogukiirusest moodustab ning kasutades alusena Cucumberit võrdlesin vahesid. Paralleelsuse võimaluse olemasolul lisasin 1 punkti juurde ning väärtusteks tulid TestNG 8, ScalaTest 7, Spock 6 ning Cucumber 1. Vastav maatriks koos eigenvektoritega tuli selline:

$$K = \begin{vmatrix} 1 & 1{,}1429 & 1{,}3333 & 8 \\ 0{,}875 & 1 & 1{,}1667 & 7 \\ 0{,}75 & 0{,}8571 & 1 & 6 \\ 0{,}125 & 0{,}1429 & 0{,}1667 & 1 \end{vmatrix} \rightarrow \cdots \rightarrow \begin{matrix} 0{,}3636 \\ 0{,}3182 \\ 0{,}2727 \\ 0{,}0455 \end{matrix}$$

Järgmine samm on koostada kokkuvõttev maatriks kõigi kolme parameetri põhjal tehtud arvutustest. Tulemuseks saame siis vastavalt veergudesse parameetrid süntaks, võimalused ja kiirus ning ridadesse TestNG, Cucumber, Spock ja ScalaTest:

$$T = \begin{vmatrix} 0{,}0667 & 0{,}4706 & 0{,}3636 \\ 0{,}4 & 0{,}0589 & 0{,}0455 \\ 0{,}3333 & 0{,}2941 & 0{,}2727 \\ 0{,}2 & 0{,}1765 & 0{,}3182 \end{vmatrix}$$

5.3. Tulemused

Viimase sammuna korrutame otsitava kriteeriumi põhjal koostatud eigenvektorid raamistike kogumaatriksiga, mis annab tulemuseks pingerea, alustades süntaksist:

$$T_S = \begin{vmatrix} 0{,}0667 & 0{,}4706 & 0{,}3636 \\ 0{,}4 & 0{,}0589 & 0{,}0455 \\ 0{,}3333 & 0{,}2941 & 0{,}2727 \\ 0{,}2 & 0{,}1765 & 0{,}3182 \end{vmatrix} * \begin{vmatrix} 0{,}5454 \\ 0{,}3636 \\ 0{,}0909 \end{vmatrix} = \begin{matrix} 0{,}24054 \\ 0{,}243712 \\ 0{,}313505 \\ 0{,}20218 \end{matrix}$$

Kõige suurema väärtusega on kolmas ehk Spock. Siit võime järeldada, et kui on oluline süntaks, siis kõiki parameetreid arvesse võttes oleks kõige parem valik just see, järgnevad TestNG ning Cucumber üsna lähestikku ning kolmandana tuleb ScalaTest.

Tehniliste võimaluste puhul näeb pingerida välja selline:

$$T_T = \begin{vmatrix} 0{,}0667 & 0{,}4706 & 0{,}3636 \\ 0{,}4 & 0{,}0589 & 0{,}0455 \\ 0{,}3333 & 0{,}2941 & 0{,}2727 \\ 0{,}2 & 0{,}1765 & 0{,}3182 \end{vmatrix} * \begin{vmatrix} 0{,}0909 \\ 0{,}5454 \\ 0{,}3636 \end{vmatrix} = \begin{matrix} 0{,}394933 \\ 0{,}0850279 \\ 0{,}289853 \\ 0{,}230141 \end{matrix}$$

Seega on kõige paremate tehniliste võimalustega TestNG, veidi maas on Spock ning ScalaTest ning Cucumber on lõpus. Kiirust arvestades on lõplik tulemus selline:

$$T_K = \begin{vmatrix} 0{,}0667 & 0{,}4706 & 0{,}3636 \\ 0{,}4 & 0{,}0589 & 0{,}0455 \\ 0{,}3333 & 0{,}2941 & 0{,}2727 \\ 0{,}2 & 0{,}1765 & 0{,}3182 \end{vmatrix} * \begin{vmatrix} 0{,}0909 \\ 0{,}3636 \\ 0{,}5454 \end{vmatrix} = \begin{matrix} 0{,}375481 \\ 0{,}0825917 \\ 0{,}285962 \\ 0{,}255902 \end{matrix}$$

Ehk siis suhteliselt sarnane eelnevale, juhib TestNG, teise valikuna võiks võtta ühe Spockist või ScalaTestist ning Cucumber on jällegi viimane.

6. Kokkuvõte

Töö põhieesmärgiks oli uurida olemasolevaid erinevaid ühiktestimise raamistikke ning nende erinevaid võimalusi omavahel võrrelda, lootes leida arendajatele lihtsamaid, kiiremaid ja rohkemate võimalustega lahendusi. Tulemusi vaadates võib väita, et peale ühe ja üldlevinud JUniti on nüüdseks välja töötatud mitmeid teisi, mis on mõnes valdkonnas sellest peajagu üle. Töö käigus tutvusin ka teiste kandidaatidega, kuid erinevatel põhjustel ei sobinud need antud teemasse – paljude raamistike arendus oli lihtsalt seisma jäänud (JDave, easyb) või veel liiga toores, et midagi mõistlikku kasutusse võtta. See on ka põhjus miks on töösse valitud praegu kõige enamkasutatavad.

Hetkel sobiva ühiktestimise tarkvara valikus tuleks lähtuda kolmest punktist: arendaja tasemest, projekti mahust ning võimalustest. Samamoodi on ka antud raamistikega: need on kas kiired, väga hea funktsionaalsusega või väga loetava süntaksiga ja valida saab ainult 2, kuna hetkel veel ei ole olemas kõikehõlmavat universaalselt head lahendust antud probleemile. Kasutades *analytical hierarchy processi*'i, selgitasin välja, et süntaksi poolest jääb peale Spock, kuid Cucumber ja TestNG ei ole kaugel maas, ülejäänud parameetrite koha pealt on TestNG, Spock ning ScalaTest kahjuks Cucumberist üsna palju ees. Siit võib järeldada, et esimesed 3 sobivad tarkvaraliste lahenduste testimiseks hästi, valik oleneb arendaja enda tasemest ning eelistustest, kuid Cucumber on, vähemalt Java platvormil, veel natukene liiga maas. Hetkel oleks sellega võimalik väga hästi ära näidata ülikooli programmeerimisülesannetes testide vajadus ning loogika, kuna viimane on hästi arusaadavalt kirjas ning kood üsna loetav.

Eesmärkide täituvuse võib lugeda positiivseks. Arendaja töö toimub testimisega pidevalt käsikäes ning käesolevad raamistikud võimaldavad muuta automaattestide kirjutamist veidi lihtsamaks ja otsekohesemaks. Loogilised edasiminekud oleksid siinkohal integratsioonitestide samalaadne võrdlus ning seejärel minna juba kasutajaliidese ning koormustestide juurde. Kui viimaseid on suhteliselt lihtne leida, siis ülejäänud raamistike maastik nii kirev ei ole.

7. Summary

The purpose of this thesis was to research different existing unit testing frameworks and to compare their different features to find easier, faster and more functional solutions to software developers. Looking at the results you can say, that besides the most common JUnit there are now several others, some of which are more advanced in certain areas than others. During this thesis I also got acquainted with several other candidates, but for different reasons they did not make it – many of them have stalled or are just too raw for overall usability. This is also the reason why the given ones are the most used right now.

At the moment choosing the most suitable software you should take into consideration three main points: the level of the developer and the size and features of the project. This also applies to the frameworks: they are either fast, have very good functionality or are very readable, but you can only choose 2. When it comes to syntax, Spock is the best choice right now with Cucumber and TestNG following after that, but other parameters' results say that Cucumber is, unfortunately, a little bit too much behind of the other 3 to be used effectively in Java development. At the moment it would be of good use in universities' programming assignments to show the logic behind the tests, because it is written in good English and the code behind it is also understandable.

The goals of this thesis may be considered complete. The work of the developer is constantly tied to testing and the given frameworks will make it easier and more straightforward to write automated tests. Logically, the way to continue from this point on is to compare integration tests similarily to this and then move on to user interface and load testing. The last one is easier to find, but the field of the rest of the frameworks is not quite that colorful.

8. Kasutatud materjal

[1] T. Saaty, 2008. [Võrgumaterjal]. Available: http://www.colorado.edu/geography/leyk/geog_5113/readings/saaty_2008.pdf. [Kasutatud 15 Mai 2015].

[2] Agile Alliance, „Guide to Agile Practices," [Võrgumaterjal]. Available: http://guide.agilealliance.org/guide/bdd.html. [Kasutatud 16 Mai 2015].

[3] M. Stroh, „Trick question: How to spell 1999? Numerals: Maybe the Roman Empire fell because their computers couldn't handle calculations in Latin.," 27 Detsember 1998. [Võrgumaterjal]. Available: http://articles.baltimoresun.com/1998-12-27/news/1998361013_1_roman-numerals-roman-numbers-information-on-roman.

[4] „regex - How do you match only valid roman numerals with a regular expression? - Stack Overflow," Stack Overflow, 6 November 2008. [Võrgumaterjal]. Available: http://stackoverflow.com/a/267405. [Kasutatud 29 Aprill 2015].

[5] „TestNG," [Võrgumaterjal]. Available: http://testng.org/doc/index.html. [Kasutatud 29 Aprill 2015].

[6] A. Bowers ja J. Bell, „Automated testing with Selenium and Cucumber," IBM, 6 August 2013. [Võrgumaterjal]. Available: http://www.ibm.com/developerworks/library/a-automating-ria/. [Kasutatud 29 Aprill 2015].

[7] „spock - the enterprise ready specification framework," [Võrgumaterjal]. Available: https://code.google.com/p/spock/. [Kasutatud 1 Mai 2015].

[8] „WhySpock - spock - Ten reasons why Spock is for you.," [Võrgumaterjal]. Available: https://code.google.com/p/spock/wiki/WhySpock. [Kasutatud 1 Mai 2015].

[9] „org.scalatest.FlatSpec," [Võrgumaterjal]. Available: http://doc.scalatest.org/1.8/org/scalatest/FlatSpec.html. [Kasutatud 3 Mai 2015].

[10] Artima Inc, „ScalaTest," 2009. [Võrgumaterjal]. Available: http://www.scalatest.org/. [Kasutatud 1 Mai 2015].

[11] „junit/ReleaseNotes4.6.md · junit-team/junit · GitHub," 7 Aprill 2009. [Võrgumaterjal]. Available: https://github.com/junit-team/junit/blob/817ea24f69a040d4161525e600b73d7037a3222f/doc/ReleaseNotes4.6.md. [Kasutatud 1 Mai 2015].

[12] „junit/ReleaseNotes4.11.md at master - junit-team/junit - GitHub," 11 November 2012. [Võrgumaterjal]. Available: https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.11.md. [Kasutatud 1 Mai 2015].

[13] „TestNG,“ [Võrgumaterjal]. Available: http://testng.org/doc/documentation-main.html. [Kasutatud 3 Mai 2015].

[14] „Tags · cucumber/cucumber Wiki · GitHub,“ 11 Juuli 2014. [Võrgumaterjal]. Available: https://github.com/cucumber/cucumber/wiki/Tags. [Kasutatud 3 Mai 2015].

[15] „Hooks · cucumber/cucumber Wiki · GitHub,“ 17 September 2014. [Võrgumaterjal]. Available: https://github.com/cucumber/cucumber/wiki/Hooks. [Kasutatud 3 Mai 2015].

[16] „Documentation-Cucumber,“ Cucumber Ltd, 2014. [Võrgumaterjal]. Available: https://cukes.info/docs. [Kasutatud 3 Mai 2015].

[17] „SpockBasics - spock - Anatomy of a Spock specification. - the enterprise ready specification framework - Google Project Hosting,“ 12 Mai 2012. [Võrgumaterjal]. Available: https://code.google.com/p/spock/wiki/SpockBasics. [Kasutatud 3 Mai 2015].

[18] „Parameterizations - spock - Learn how to parameterize your feature methods. - the enterprise ready specification framework - Google Project Hosting,“ 28 August 2011. [Võrgumaterjal]. Available: https://code.google.com/p/spock/wiki/Parameterizations. [Kasutatud 3 Mai 2015].

[19] „spock-example/StepwiseExtensionSpec.groovy at master · spockframework/spock-example · GitHub,“ 3 Märts 2014. [Võrgumaterjal]. Available: https://github.com/spockframework/spock-example/blob/master/src/test/groovy/StepwiseExtensionSpec.groovy. [Kasutatud 3 Mai 2015].

[20] „ScalaTest,“ Artima Inc, [Võrgumaterjal]. Available: http://www.scalatest.org/user_guide/tagging_your_tests. [Kasutatud 3 Mai 2015].

[21] „ScalaTest,“ Artima Inc, [Võrgumaterjal]. Available: http://www.scalatest.org/user_guide/selecting_a_style. [Kasutatud 3 Mai 2015].

[22] „ScalaTest 2.0 - ParallelTestExecution,“ Artima Inc, [Võrgumaterjal]. Available: http://doc.scalatest.org/2.0/index.html#org.scalatest.ParallelTestExecution. [Kasutatud 3 Mai 2015].

[23] „ScalaTest,“ Artima Inc, [Võrgumaterjal]. Available: http://www.scalatest.org/getting_started_with_fun_suite. [Kasutatud 3 Mai 2015].

[24] „trait - BeforeAndAfterAll,“ Artima Inc, [Võrgumaterjal]. Available: http://doc.scalatest.org/1.0/org/scalatest/BeforeAndAfterAll.html. [Kasutatud 3 Mai 2015].

[25] „ScalaTest,“ Artima Inc, [Võrgumaterjal]. Available: http://www.scalatest.org/user_guide/using_assertions. [Kasutatud 3 Mai 2015].

[26] „ScalaTest 1.8 - org.scalatest.concurrent.timeouts,“ Artima Inc, [Võrgumaterjal]. Available: http://doc.scalatest.org/1.8/index.html#org.scalatest.concurrent.Timeouts. [Kasutatud 3 Mai 2015].

[27] P. Lawrey, „java - Why is creating a Thread said to be expensive? - Stack Overflow,“ 30 Märts 2011. [Võrgumaterjal]. Available: http://stackoverflow.com/a/5483467. [Kasutatud 14 Mai 2015].

[28] GitHub, „execute a scenario multiple times · Issue #164 · cucumber/cucumber · GitHub,“ [Võrgumaterjal]. Available: https://github.com/cucumber/cucumber/issues/164. [Kasutatud 13 Mai 2015].

9. Lisa 1

JUnit kiiruse testimise kood:

```
public class GeneratorParallelTest {

     @Test

     public void parallel() {

          Class[] cls = {ParallelTests4.class};

          JUnitCore.runClasses(new ParallelComputer(true,
true), cls);

     }

}
```

```
public class ParallelTests4 {

     public void generatorTest() {

          Generator generator = new Generator();

          String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;

          for (int i = 1; i <= 4000; i++) {


     assertTrue(generator.generate(i).matches(regex));

          }

     }

     @Test

     public void run250Times1() {

          for (int i = 1; i <= 250; i++) {

               generatorTest();

          }

     }

     @Test
```

```
    public void run250Times2() {
        for (int i = 1; i <= 250; i++) {
            generatorTest();
        }
    }
    @Test
    public void run250Times3() {
        for (int i = 1; i <= 250; i++) {
            generatorTest();
        }
    }
    @Test
    public void run250Times4() {
        for (int i = 1; i <= 250; i++) {
            generatorTest();
        }
    }
}
```

TestNG kiiruse testimise kood:

```
public class GeneratorParallelTest4 {

     @Test(invocationCount = 250)

     public void generatorTest1() {

          Generator generator = new Generator();

          String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;

          for (int i = 1; i <= 4000; i++) {

               assert generator.generate(i).matches(regex);

          }

     }


     @Test(invocationCount = 250)

     public void generatorTest2() {

          Generator generator = new Generator();

          String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;

          for (int i = 1; i <= 4000; i++) {

               assert generator.generate(i).matches(regex);

          }

     }


     @Test(invocationCount = 250)

     public void generatorTest3() {

          Generator generator = new Generator();

          String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;
```

```
          for (int i = 1; i <= 4000; i++) {
               assert generator.generate(i).matches(regex);
          }
     }

     @Test(invocationCount = 250)
     public void generatorTest4() {
          Generator generator = new Generator();
          String regex =
"\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b"
;
          for (int i = 1; i <= 4000; i++) {
               assert generator.generate(i).matches(regex);
          }
     }
}
```

```
<suite name="Parallel test suite" parallel="methods" thread-
count="4">
  <test name="Parallel4">
    <classes>
      <class name="test.GeneratorParallelTest4"/>
    </classes>
  </test>
</suite>
```

Ja ScalaTesti kiiruse testimise kood:

```
class GeneratorScalaTest extends FlatSpec with MustMatchers
with ParallelTestExecution {

  "1) generated value" should "match
'\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b'
" in {

    for (j <- 1 to 250) {

      var generator = new Generator();

      for (i <- 1 to 4000) {

        generator.generate(i) must fullyMatch regex
("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b
");

      }

    }

  }



  "2) generated value" should "match
'\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b'
" in {

    for (j <- 1 to 250) {

      var generator = new Generator();

      for (i <- 1 to 4000) {

        generator.generate(i) must fullyMatch regex
("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b
");

      }

    }

  }



  "3) generated value" should "match
'\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b'
" in {
```

```
    for (j <- 1 to 250) {

      var generator = new Generator();

      for (i <- 1 to 4000) {

        generator.generate(i) must fullyMatch regex
("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b
");

      }

    }

  }



  "4) generated value" should "match
'\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b'
" in {

    for (j <- 1 to 250) {

      var generator = new Generator();

      for (i <- 1 to 4000) {

        generator.generate(i) must fullyMatch regex
("\\bM{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})\\b
");

      }

    }

  }

}
```

Siinkohal tuleb meeles pidada seda, et ScalaTesti paralleelsus tuleb käivitada andes talle ette parameetri –P.