

UNIVERSITY OF TARTU  
Institute of Computer Science  
Software Engineering Curriculum

**Ali Belakehal**  
**Test Automation Case Study**  
**Master's Thesis (30 ECTS)**

Supervisor(s): Dietmar Alfred Paul Kurt Pfahl  
Rainer Tikk

Tartu 2017

## **Test Automation Case Study**

### **Abstract:**

Over the last two years, the testing process of one of the software development teams at LHV bank went through several development stages. However, there hasn't been any methodical approach towards validating that evolution. The aim of this thesis is to conduct an investigation of three key periods, and measure the cost and effectiveness of the testing process during each period. A multilevel analysis is then performed in order to identify problematic, as well as improvement patterns, and the factors associated with them. The analysis is concluded with setting the goal of shifting the testing process to a more automated model. Subsequently, the remainder of the thesis undertakes the task of combining a multiplicity of techniques that try to make such model achievable, by automating certain aspects of the test automation process itself. These techniques are articulated as a proposed solution, which is then implemented and validated in the context of this thesis.

### **Keywords:**

Test automation, acceptance testing, regression testing, Fitness, parsing expression grammar, code generation.

**CERCS:** P170

## **Testimise automatiseerimise juhtumiuuring**

### **Lühikokkuvõte:**

Viimase kahe aasta jooksul on LHV panga ühe arendustiimi testimisprotsess läbinud mituarendustsükli. Samas pole seda arengut metoodiliselt valideeritud. Selle töö eesmärk on analüüsida kolme võtmetähtsusega perioodi ning mõõta nende testimisprotsessi maksumust ja efektiivsust. Seejärel viiakse läbi mitmetasandiline analüüs, et tuvastada problemaatilised ja kasulikud mustrid ning nendega seotud tegurid. Analüüsi tulemusel seatakse eesmärgiks muuta testimisprotsess automatiseeritumaks. Sellest tulenevalt tegeleb ülejäänud lõputöö erinevate meetodite kombineerimisega, et muuta selline lähenemine läbi testide automatiseerimise protsessi teatud osade endi automatiseerimise saavutatavaks. Nendest tehnikatest moodustatakse pakutav lahendus, mis seejärel implementeeritakse ja selle lõputöö kontekstis valideeritakse.

### **Võtmesõnad:**

Testimise automatiseerimine, vastuvõtutestimine, regressioontestimine, Fitness, avaldisetuvastuse grammatika, koodi genereerimine.

**CERCS:** P170

## Table of Contents

1	Introduction .....	5
1.1	Aim of the Thesis .....	5
1.2	Selected Project .....	5
2	Terms.....	7
3	Background .....	8
3.1	Fitness .....	8
4	Method .....	10
4.1	Data Gathering Methodology .....	10
	Periods.....	10
	Methodology .....	11
	Reliability .....	14
5	Baseline .....	15
5.1	The Testing Process.....	16
5.2	Measurements.....	17
5.3	Problem Formulation.....	19
	Benefits of Automated Testing .....	19
	Problems with Automated Tests .....	20
	Summary .....	21
6	Improvement proposal .....	22
6.1	Solution Outline.....	22
	Replacing Manual Testing .....	22
	Introducing Test Generation .....	24
6.2	Analysis .....	24
	Test Types .....	25
	Grammar .....	31
6.3	Solution.....	34
	Fixture Generation .....	35
	Test Data Generation.....	41
6.4	Validation .....	46
7	Conclusion.....	50
7.1	Summary.....	50
7.2	Goals.....	50
8	References .....	51
9	Appendix .....	52

9.1	Task Names Mapping .....	52
9.2	Full Parsing Grammar .....	54
	Assumptions .....	54
	The parsing grammar .....	54
9.3	IPM File Content Example .....	58
9.4	Generated File Example .....	60
9.5	License.....	62

# 1 Introduction

## 1.1 Aim of the Thesis

The lack of detailed analysis of the testing process in the project under study led to the rise of many unchecked assumptions, and doubts about how effective test automation is, and what could be done to improve the process. The thesis is set out to tackle two main matters and their underlying components specified as follows:

- Firstly, to provide precise measurements of the factors related to testing, and their respective effects. From there conclusions can be made on what the exact problems are, and what the improvements should be. Thus, the first part of thesis will eliminate the speculative approach, and provide well based conclusions. The sub goals are outlined as follows:
  - Measure the effectiveness and cost of current testing process.
  - Pinpoint the problems with the current testing process.
- Subsequently, the thesis will then attempt to reinforce whichever behaviours that led to improvements, and try to solve the issues that are found to be harmful towards the progress of the testing process. A solution which incorporates these two aspects will be explained, implemented, and finally validated to the best extent possible. The sub goals are outlined as follows:
  - Measure the effectiveness of test automation.
  - Propose and implement a solution to the previously identified problems.
  - Validate the elements of the solution.

In addition to eliminating the speculative approach towards developing the testing process, and offering a solution to increase the effectiveness of said process, achieving those goals will also make a case for testing process changes that could be adopted by other teams inside the company.

## 1.2 Selected Project

The selected project for this case study is the Acquiring system at AS LHV Pank<sup>1</sup>, which handles the card payments and the cash withdrawals that are done using LHV terminals, ATMs, and Ecommerce portals.

This system is chosen for the following reasons:

- It's the system with the most evolved testing process, the other systems' testing approaches are quite primitive.
- I am well acquainted with this system's implementation, and the Business Model.
- I can put the collected data into context because of the history of my involvement in the development and testing of this system.
- I have all access to test any improvements that might result or be part of the thesis.

The acquiring system (ACQ) has been under development since August 2014. It consists of several modules that provide the following services:

- Means for merchants to receive payments by debit and credit cards, through physical terminals or virtual Ecommerce terminals.
- Providing detailed reports about those payments for merchants.
- Communication with Mastercard, Visa, and local banks in order to process those payments.

---

<sup>1</sup> <https://www.lhv.ee/en/business-client>

- Communication with Mastercard, Visa, and local banks in order to process ATM transactions.
- Merchants' management and accounting functionalities for the Back Office.
- Providing custom acquiring services for private clients with special requests, or unconventional business models.

The project is one of the six main projects in LHV that have dedicated teams operating them.

## 2 Terms

This section has some definitions of terms that are used during the thesis, to ensure that they mean what is expressed here and nothing else, because the reader might have a different understanding of them.

**Developer** is a team member who is a Software Developer.

**Tester** is a team member who is a Quality Assurance Specialist, or a developer with knowledge about testing.

**Requirements (specifications)** are a description of what is expected of a process, or a user interface to achieve, given a set of conditions. Ideally, it's a list of independent rules, and restrictions about the input and the output of a process.

**Task** is an implementation of new requirements into the system, or of a modification of a set of requirements, performed by the person assuming the developer role, within the context of a specific task.

**Epic** is a set of tasks which collectively build-up to the same end goal. It can be thought of as a sub-project.

**Test case** is a set of steps that are the result of an analysis of the requirements in question, in the context of a specific task, which is performed by the person assuming the tester role. Those steps are to ensure that the implementation of the requirements meets said requirements. A test case is said to have passed if the implementation meets the requirements, otherwise it is said to have failed.

**Requirements' coverage** is the extent to which a list of requirements is verified by a set of test cases. It's expressed as the ratio of the verified requirements – those covered by test cases – over the total number of requirements, it can also be expressed as a percentage.

**Manual testing** is going through the test cases' steps individually, which requires direct interaction with the system by the tester.

**Automated test** is a scripted version of the test cases that can be executed, or scheduled to execute, after which the test can be deemed as passed or failed, without the need for a manual interaction with the system.

**Regression test**<sup>2</sup> an automated tests with the purpose of ensuring that new system changes are not conflicting existing system behaviours.

**Acceptance test**<sup>3</sup> is an automated test which is created based on the requirements before a task is released into live environment, in order to test a task, and become a regression test after the release of the task.

**Live bug** is an anomaly detected in the live environment, which can be considered as a breach of the requirements, or a deviation from them.

**Hotfix** are process blocking live bugs.

**Cost** is the number of man hours logged by a team member under an analysis, development, or a testing activity.

---

<sup>2</sup> [https://en.wikipedia.org/wiki/Regression\\_testing](https://en.wikipedia.org/wiki/Regression_testing)

<sup>3</sup> <https://www.agilealliance.org/glossary/acceptance/>

### 3 Background

This section will give an overview of Fitnesse, from the perspective of how it is used in our project.

#### 3.1 Fitnesse<sup>4</sup>

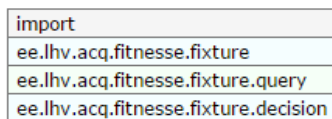
Fitnesse software development tool aimed mainly at automating acceptance tests<sup>5</sup>. Fitnesse tests are text based, written in a form that can be thought of as a more technical form of the requirements. The text document is called a **wiki** page, and is a series of instructions. Those instructions are implemented by – in the case of Java – classes and methods called **fixtures**.

##### *Wiki*

A series of tables of several types, their text form is delimited by the character '|', example:

```
| import |  
| ee.lhv.acq.fitness.fixture |  
| ee.lhv.acq.fitness.fixture.query |  
| ee.lhv.acq.fitness.fixture.decision |
```

For readability purposes, wiki pages are edited in a browser, where a local server formats them and provides facilities to run and debug them. Figure 0 shows how the previous text format of the table is displayed in the browser:



import
ee.lhv.acq.fitness.fixture
ee.lhv.acq.fitness.fixture.query
ee.lhv.acq.fitness.fixture.decision

Figure 0. Formatted wiki table.

For readability purposes, Fitnesse wiki examples will be shown in the HTML format in this thesis document.

##### *Wiki table types*

Excluding the import table, we use three types of tables:

1. Script tables<sup>6</sup>: a flexible construct where every row can be implemented by a fixture method, in the class which corresponds to the header of the table.
2. Decision tables<sup>7</sup>: takes several rows of arguments and performs an action on those arguments and displays the result in the column with a name ending in '?'
3. Query tables<sup>8</sup>: executes a query, which is implemented in the fixture class corresponding to its title, and displays its results in its body.

---

<sup>4</sup> <http://www.fitnesse.org/FrontPage>

<sup>5</sup> <http://www.fitnesse.org/FitNesse.UserGuide.AcceptanceTests>

<sup>6</sup> <http://www.fitnesse.org/FitNesse.FullReferenceGuide.UserGuide.WritingAcceptanceTests.SliM.ScriptTable>

<sup>7</sup> <http://www.fitnesse.org/FitNesse.FullReferenceGuide.UserGuide.WritingAcceptanceTests.SliM.DecisionTable>

<sup>8</sup> <http://www.fitnesse.org/FitNesse.FullReferenceGuide.UserGuide.WritingAcceptanceTests.SliM.QueryTable>



The description of the tables is left vague on purpose at this point, as the usage of the tables will become clearer with examples in Section 6.

### ***Fixtures***

These are the implementations of the instructions found in the wiki tables. For example the script table:

| script | Journal Job Fixture |

| prepare graph for types | CALC\_CLAIM\_ISSUER\_MASTERCARD\_FEE |

Means that there's a Java class named JournalJobFixture, which contains a method named prepareGraphForTypes(), and which takes one argument of type "string"; prepareGraphForTypes(String type).

Again, this will become clearer when the grammar is discussed at length in Section 6.

## 4 Method

The case under study is the testing process of the acquiring project described earlier.

In order to reach the goals mentioned in the introduction section, I will gather data as will be described in Sections 4.1. The measurements are then aggregated into a result that can be analysed, and from there problems will be highlighted as part of establishing the baseline.

Using the conclusions from the previous step, an experiment will be conducted in order to validate those conclusions as well as the newly proposed testing model. Measurements will be taken the same way as done for the baseline, and then compared against it.

### 4.1 Data Gathering Methodology

The data gathered in the context of this thesis is from three distinct periods, each represents a period of time characterized mainly by the state of the automated tests.

The data is focused on two dimensions along which conclusions can be drawn in the baseline and improvement sections, the first is **effectiveness**, which is has everything that relates to requirements coverage, and resulting live bugs. The second covers timeliness, and testing **cost**.

#### Periods

Every period is defined by a stretch of time during which an epic, or two related epics were developed, tested, and covered by automated tests. The three periods are consecutive and are the most recent time sections where test automation was starting to be used. All three periods lasted around four months – not necessarily consecutive months-, and are roughly of the same size and complexity. The first two periods will be used as a baseline for the current testing process, and the third period will be used as a model for the improved process.

#### *Period 1 (P1)*

Lasted from mid-December 2015 until mid-April 2016, this period represents an early stage of the automated testing endeavour. This period is characterized by:

- A significant deficiency in requirements' coverage.
- The most live bugs and hotfixes.

#### *Period 2 (P2)*

Lasted from the end of April 2016 until end of August 2016, excluding most of July because of overlapping vacation times of team members. This period is characterized by:

- A chaotic testing process, and frequent retesting.
- An average level of requirements' coverage.
- An improved test automation level compared to the previous period.
- Less bugs and hotfixes.

#### *Period 3 (P3)*

Lasted from the end of October 2016 until the end of March 2017, with fewer team members than in the other two periods. This period can be considered as the ideal for our testing process, and is characterized by:

- Mature testing and test automation processes, in comparison to the other periods.
- An optimal level of testing with automated tests as opposed to manual testing.

- A very good requirements' coverage percentage.
- An acceptable testing cost, given the benefits.
- Less bugs and hotfixes.

This period is distinct from the preceding ones, in the sense that due to some special circumstances within the team, I as a developer had to take a testing role, and that was the perfect opportunity for me to conduct an experiment in the context of this thesis, which answer the following questions:

- **Q1:** Can testing in our project be replaced by automated testing, in most cases at least?
- **Q2:** Can the tests be structured in a way that makes it possible to generate them?
- **Q3:** What would be the outcome of the desired level of test automation?
- **Q4:** Is the desired level of test automation reasonable considering the testers' technical skill level, if not, then would the time saved by test generation compensate it?

Note: This period is the most recent, but the developed functionality is heavily used, and by now all the scenarios which are described by the requirements have played out in the live environment, and it's safe to claim that the live bugs and hotfixes count will remain as it is henceforth.

## **Methodology**

### ***Task Relevance***

From each period, the **core** tasks from the involved epics are chosen and measured for:

- Development cost
- Testing cost
- Time spent on automated tests
- Requirements' coverage

Where a **core** task is characterized by:

- Being essential to the achievement of its epic's goal, as opposed to accessory or auxiliary. An example of such distinction would be the difference between a calculation engine and a user interface enhancement.
- Eligible to test automation, in the sense that the task is not a one time job, for example an SQL script that performs data manipulation in a very specific context that doesn't reproduce (or does so rarely), rather, automated tests for this task would serve as acceptance, or at least as regression tests for the future.

### ***Development Cost***

Includes Analysis, requirements' adjustment, and code development and fixes. Excludes the time taken to write automated tests.

### ***Testing Cost***

Includes both manual testing, and the time spent writing automated tests. The time spent writing automated tests is not exact, but reliably approximate.

The inaccuracy is inevitable because the tests are written by different team members, at different times, within different tasks, for example the positive case test is usually written by the task developer, and logged under development time, then the task tester would add negative cases tests, and that time would be logged under testing time. Then due to time

constraints the task is released to the live environment with a less than optimal requirements' coverage by the automated tests, and the coverage is rectified within a new task.

The confidence that the approximated measured time is reliable comes from following the git commits related to the automated tests written for a specific task, and finding consistent patterns:

- Considering the test complexity and length, the time taken to write it by a specific team member is consistent with the skill level of that team member, and with the time taken to write other tests of similar length and complexity.
- Considering the skill levels within the team, the time taken to write a certain type of automated test is consistent with the time taken for a similar skill level team member to do a test of the same type. Where the type is dependent on the process that's being tested, examples are: parsing, importing files, calculation engines, crosschecks, etc.
- Considering the tool used to write an automated test, the time spent on a test for a specific team member is consistent with times from other tests using that tool, across tasks.

After the separate time measurements are validated along the previously mentioned patterns or factors, they are aggregated by task, then that task is assigned a test automation time measurement, and the development time is adjusted for those tasks where it included time for writing automated tests.

### Initial Test Automation Cost

Refers to the percentage of the testing cost that was spent on test automation just before the development done in an epic was released into the live environment.

### Final Test Automation Cost

Refers to the percentage of the testing cost that was measured just after the last automated test related to a specific period was added. In other words, it's the initial test automation cost plus the cost of all the added automated tests after the epic in question was released into the live environment. These late automated tests are added either because of live bugs discovered after the release, or they couldn't be fit into the original release cycle due to time constraints.

### ***Requirements' Coverage Calculation***

The coverage is a percentage that represents the ratio of those requirements that have automated tests supporting them, over the total number of requirements for a specific task.

The requirements are not always well articulated, therefore making it hard to calculate the coverage methodically. An initial idea was to weigh the requirements by importance, but after testing and consideration, the weights seemed arbitrary and susceptible to bias, as what I consider important might not be what the product owner considers important. Alternatively, to ensure that the requirements are covered fairly, they are reformatted into a logically separate list of rules, where more complex or nuanced requirements would have clear and distinct sub-rules, and thus the disambiguation of the requirements spares a needless, and potentially corrupt weighting system.

An example of a well formatted, and logically distinct list of rules, which could be used for coverage calculation:

1. Job runs on the second period open of the month before fee calculation jobs

2. The job checks that the currency rate of the last day of the month is available (for previous month), if not, an error message will be shown
3. Revenue month range is between the second day of the previous month and the first day of the current month
4. Reset all active or future tier based pricing contracts to no active tiers
5. Find the first terminal installation date for every merchant
6. Calculation:
  1. Set previous month revenue if the first terminal installation for a merchant is in the month before that (current month - 2 or earlier)
  2. Merchant revenue is calculated as the sum of all revenue claim entries with direction credit minus all revenue claim entries with direction debit
  3. In case the revenue is null then the estimated monthly revenue will be used, if both are null than the zero tier will be chosen
7. Set the active tier with the closest tier minimum amount value to the revenue, which is smaller than the revenue
8. The job expects at least one merchant has a revenue claim entry in the past month, otherwise an error message will be shown
9. The job expects the currency exchange rate to be available for every revenue claim entry date in the past month, even EUR

The next step is to go through the contents of the set of automated tests that cover a certain process, and determine which requirements are covered, and from there the requirement coverage percentage results.

To continue with the same example, the following are the tests which are supposed to cover the process which the previously mentioned requirements describe, along with the rules they cover:

Table 1: Requirements' coverage calculation example

Test name	Requirements covered
Calc Previous Month Revenue Multiple Currencies Success	3; 4; 5; 6.1; 6.2; 7;
Calc Previous Month Revenue No Currency Rate	2; 9
Calc Previous Month Revenue Total Negative Revenue Success	6.3
<b>Coverage</b>	9/11
<b>Percentage</b>	81%

#### Initial Requirements' Coverage

The requirements' coverage calculated at the time of putting the epic to the live environment, before any live bugs are discovered.

#### Final Requirements' Coverage

The coverage calculated after the last time an automated test was added, which is usually after most live bugs have been solved.

### ***Live Bugs and Hotfixes***

Linking live bugs to specific period requires analysing the effect of a bug and determining the exact causes, the process can be summarized as follows:

1. Query from Jira<sup>9</sup> all the live bugs which were created after day zero; day zero being the date when the first task from P1 was released into the live environment.
2. Go through the queried list and eliminate all those bugs which were caused by technical issues, such as a misconfiguration, or performance issues, such as unoptimized or missing database indexes.
3. Classify the live bugs into one of the three periods which define the context of this study, and dispose of those which are linked to other periods.
4. For every period's live bug list, eliminate those which can be traced to a fault of an external dependency, such as faulty files, and those which can be traced to business errors, such as the lack of requirements. What should remain at this point are only the live bugs which can be directly traced to development errors, and were not discovered due to a lack of requirements' coverage, whether from an incomplete manual testing round, or an insufficient number of automated tests.

The final step is to summarize the number of live bugs for each period.

Hotfixes are the live bugs which blocked a time-critical part of the system, required immediate attention, and had negative consequences that in some cases are limited only to the loss of development time, and in others affect the business and the product owner.

### **Reliability**

For the sake of precision, all the measurements were taken from three iterations of the methods described above. Those results which proved consistent across iterations were kept, and on rare instances, when an inconsistency appears, a recalculation was performed.

For verification purposes, Appendix 9.1 contains the mapping of the task codes used when presenting the measurements to the real task codes in our task management system.

---

<sup>9</sup> <https://www.atlassian.com/software/jira>

## 5 **Baseline**

This chapter will first describe the current testing process in order to lay down the context. Subsequently, the cost and efficiency measurements described in the previous chapter will be presented. Finally, the problems with this version of the process will be discussed in preparation for the solution chapter.

## 5.1 The Testing Process

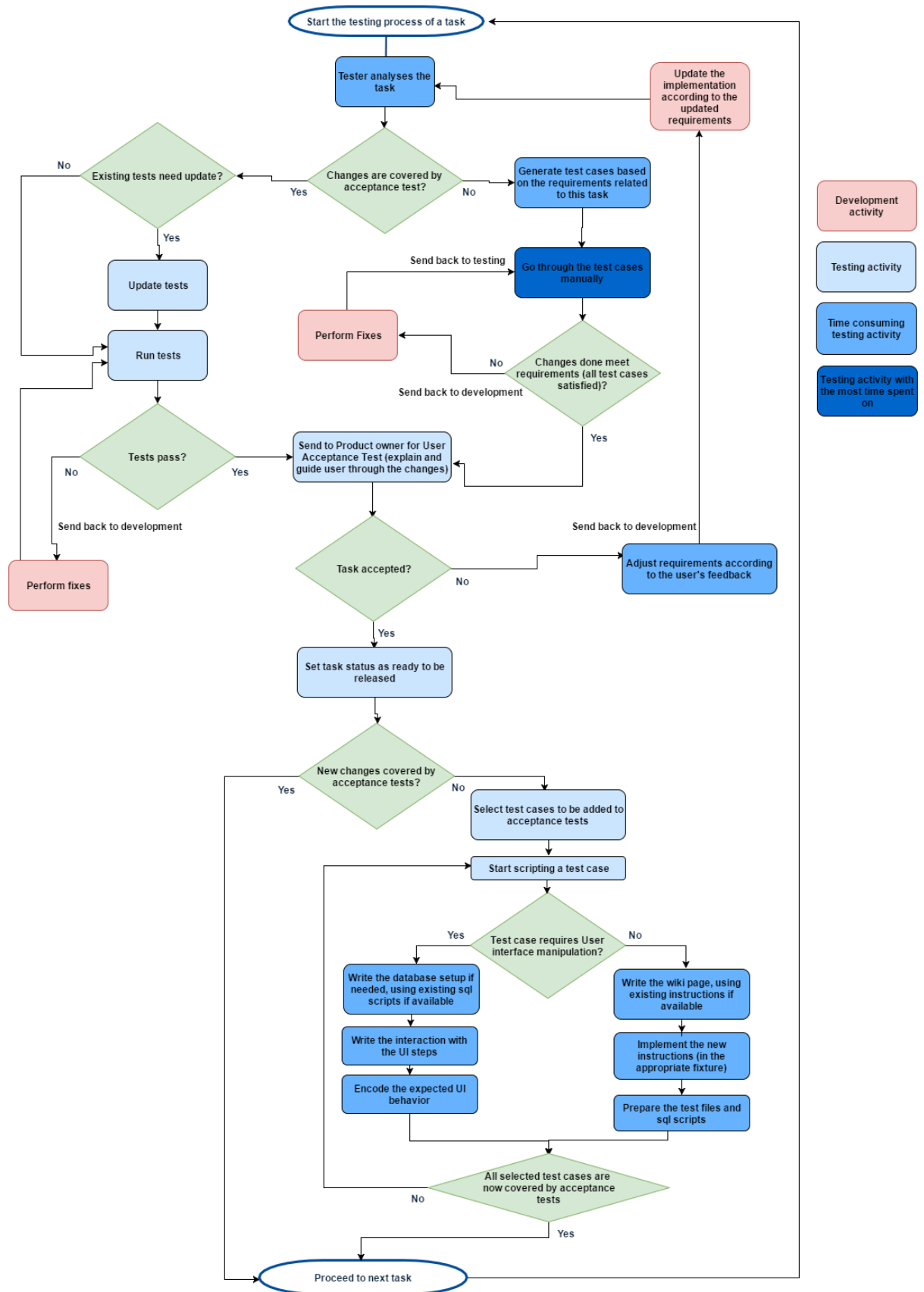


Figure 1. The testing process.



The diagram in Figure 1, describes this testing process, which has been the main methodology of testing for about two years.

Whenever a task is ready to be tested, a tester analyses the task in question, and decides whether it's covered by automated tests that need to be updated, or it should be manually tested. In the latter case test cases are made based on the requirements, then the task is manually verified based on those test cases. The task is sent back to development whenever requirements are not met, or anomalies are detected, and after the fixes the task is manually verified again, until all test cases pass.

Then a second round of user testing is done by the product owner to determine that the requirements were understood correctly, and are adequate. In case they are not, they will be adjusted, then tester analysis of the new or modified requirements and verification is done again.

Finally, the test cases are prioritized by the tester, and as many of them are made into automated tests as time allows, before the task is deployed to the live environment.

## 5.2 Measurements

The following two tables show the task list of a specific period, along with the measured attributes (development cost, testing cost, test automation cost, and requirements' coverage). The rows highlighted in blue are for the tasks which increased the requirements' coverage after the release, i.e. these tasks are excluded from the initial requirements' coverage and test automation cost calculations.

Task	Development cost (h)	Testing cost (h)	Automated Tests' cost(h)	Requirements' Coverage
P1T1	40,50	34,50	3,90	15%
P1T2	14,75	14,47	4,73	70%
P1T3	48,82	37,50	4,25	15%
P1T4	40,32	45,83	13,72	66%
P1T5	13,32	7,07	3,43	70%
P1T6	5,00	18,55	3,33	50%
P1T7	3,33	3,75	1,12	40%
P1T8	7,13	6,12	1,37	10%
P1T9	0,25	5,12	0	0%
P1T10	0,33	4	0	0%
P1T11	6,33	3,38	0	0%
P1T12	5,00	2,80	0	0%

<b>P1T13</b>	0,00	10	10	100%
<b>P1T14</b>	0,00	26,63	26,63	100%
<b>P1T15</b>	0,00	36,22	36,22	100%
<b>P1T16</b>	6,75	5,58	2,25	75%
<b>P1T17</b>	19,85	16,70	3,80	40%
<b>P1T18</b>	15,10	37,52	1,02	10%
<b>P1T19</b>	19,50	53,12	6,48	33%
<b>P1T20</b>	6,35	19,60	2,10	50%
<b>P1T21</b>	18,38	31,77	6,12	33%
<b>P1T22</b>	0,00	16	16	100%
<b>P1T23</b>	0,00	56	56	100%

Table 2: Period 1 measurements.

<b>Task</b>	<b>Development cost (h)</b>	<b>Testing cost (h)</b>	<b>Automated Tests' cost(h)</b>	<b>Requirements' Coverage</b>
<b>P2T1</b>	32,22	131,27	17,35	15%
<b>P2T2</b>	92,53	84,99	56,86	65%
<b>P2T3</b>	32,18	43,66	17,33	67%
<b>P2T4</b>	24,38	50,13	13,13	50%
<b>P2T5</b>	2,60	28,73	1,40	100%
<b>P2T6</b>	59,37	55,47	31,97	81%
<b>P2T7</b>	16,25	8,75	8,75	80%
<b>P2T8</b>	0,65	12,68	0,35	0%
<b>P2T9</b>	16,90	15,60	9,10	70%
<b>P2T10</b>	64,32	145,63	34,63	100%
<b>P2T11</b>	24,70	26,30	13,30	70%
<b>P2T12</b>	1,63	4,21	0,88	20%

<b>P2T13</b>	7,64	35,61	4,11	66%
<b>P2T14</b>	12,68	6,83	6,83	80%
<b>P2T15</b>	25,35	25,15	4,60	46%
<b>P2T16</b>	0,00	13	13	100%
<b>P2T17</b>	0,00	30	30	100%
<b>P2T18</b>	0,00	35	35	100%
<b>P2T19</b>	0,00	24	24	100%

Table 3: Period 2 measurements.

Table 4 presents the final step of the aggregation as a summary of the two previous tables, along with the effectiveness measurements (number of live bugs, and the number hotfixes).

<b>Period</b>	<b>P1</b>	<b>P2</b>
<b>Development cost (h)</b>	271,02	413,37
<b>Testing cost (h)</b>	460,22	777,50
<b>Initial requirements' coverage</b>	32%	58%
<b>Initial test automation cost</b>	13%	28%
<b>Final requirements' coverage</b>	48%	70%
<b>Final test automation cost</b>	46%	41%
<b>Number of Live bugs</b>	26	16
<b>Number of hotfixes</b>	18	10

Table 4: Baseline data gathering results

### 5.3 Problem Formulation

This section will discuss measurement results, highlighting the aspects that are relevant to this thesis.

#### Benefits of Automated Testing

##### *Prevent Bugs*

It's needless to remind the importance of regression testing, the benefits have been well established by now [1], and having those regression tests ready to be executed saves lots of

testing time, and eliminates the possibilities of human errors occurring, where if those automated tests were not present, a tester would have to go through and test all related parts of the system related to the changes done in the task being tested, relying on memory, and this model is time consuming, and error prone, which make it unsustainable and unreliable.

From the results presented in the previous chapter, it's clear that as the requirements' coverage increases, the number of live bugs and hotfixes decreases. Taking difference in coverage between **P1** and **P2**, and comparing the number of live bugs and hotfixes presents enough evidence to claim that preventing bugs is correlational with the requirements' coverage.

### ***Prevents Retesting Manually***

In **P2** changing the design of the solution was a prominent problem, that's the reason for the incoherent development cost in comparison with **P1** and **P2**. The testing cost also suffered as a consequence. On three occasions the retesting of the whole epic had to be done because the implementation had to be changed. The key point here is that even though the implementation went through several changes, the outcome of that implementation was the same. In other words, if automated tests were prepared for the first iteration of testing, there would have been no need to retest manually, instead, the prepared test would require slight modifications and execution only. Additionally, the developers would have a more technical and interactive form of the requirements to develop against.

Considering the time taken to finalize the test coverage in **P2**, if those tests were prepared in the earliest iteration as acceptance tests, the Testing cost would have been two thirds of what is resulted to be. So, a saved cost of 259.17 hours would have been the benefit of preventing manual retesting by an early test automation decision.

## **Problems with Automated Tests**

### ***Cost***

Writing automated tests is a time consuming task for testers to perform, due to it being demanding in technical skill. For the same reason it's also frustrating, and out of the comfort zone of a tester, and that makes it a demoralizing process, compared to the manual testing that they are adept at.

From analysing the data that led to time spent on test automation for **P1**, **P2**, testers take from three to five times the length of time that a developer would take to write a test of the same length and complexity roughly.

### ***Non-covered Requirements***

Nearly all tasks which have been covered by automated tests, the underlying requirements are not fully covered, and for all those that were created during **P1**, **P2**, or prior, the coverage is less than desirable, or in other words, it does not even contain the essential requirements, and that poses the following issues:

- The non-covered requirements are forgotten, and with passing time, the team memory of which requirements are covered and which are not becomes blurry, and unreliable.

- Leaving uncovered requirements is error prone. Every time regression tests are considered passed, there's an implicit assumption that the new changes are safely integrated with existing parts of the system, where in reality they are well integrated with only the parts of the system which are covered by automated tests, since automated tests are the only mean used for regression testing.
- The value of the automated tests is not well revealed to the product owner and the business side, when bugs still appear, even though time was taken to make automated tests, time which from their perspective could have been spent on developing and testing new functionality.

## Summary

The problems with the current testing process can be stated as follows:

- **Prob1:** Low requirements' coverage by automated tests causes several sub-problems.
- **Prob2:** Retesting manually of tasks after logical, architectural changes in the code.
- **Prob3:** Writing cost automated tests is a costly process for testers to do.

## 6 Improvement proposal

This chapter will explore a proposed solution to the problems discussed in the previous section.

### 6.1 Solution Outline

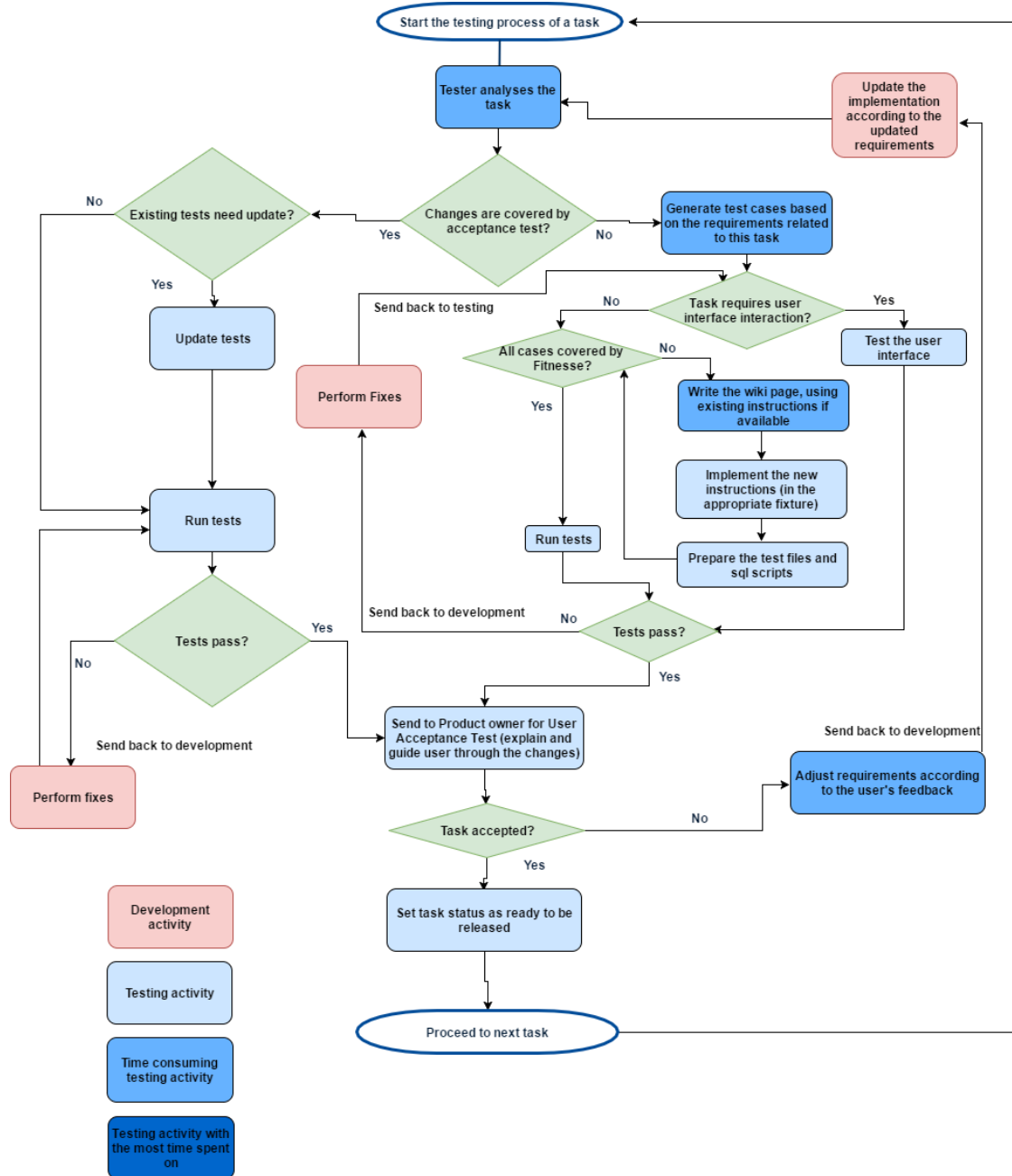


Figure 2. Improved testing process.

Figure 2 shows the testing process described earlier, with one omission that will be justified in the next section, and two main modifications:

#### Replacing Manual Testing

The first modification is moving the test automation where the manual testing used to be, in order to directly attack the first two problems (**Prob1**, and **Prob2**):

- Retesting won't be an issue, as the automated tests are proactively created, and development fixes can be done against those prepared tests.
- Requirement coverage will be forced to be no less than ideal, since the task can't be marked as ready to be released until all requirements are verified, and in this new model, verification of the requirements means that they will be covered by automated tests.

There has been an attempt to adopt this model in the past, but it failed due to the lack of experience partly, but mainly due to the time consuming aspect of writing those tasks that caused the tasks requested by the business to be late, and releases were postponed. However, as the result from **P3** will show, if testing time remains around double the time required for development, that is; testing cost = 2 \* development cost ( $\pm$  development cost/10), then this model is possible.

### ***The Experiment***

During **P3** (details in Section 4.1) the testing was done according to the improved process presented in Figure 2, and the results of the measurements are as follows:

<b>Task</b>	<b>Development cost (h)</b>	<b>Testing cost (h)</b>	<b>Automated Tests' cost(h)</b>	<b>Requirements' Coverage</b>
<b>P3T1</b>	0,25	11,83	10,65	100%
<b>P3T2</b>	31,69	79,65	48,36	94%
<b>P3T3</b>	18,75	45,02	36,23	100%
<b>P3T4</b>	19,88	104,38	58,65	83%
<b>P3T5</b>	7,69	12,56	10	100%
<b>P3T6</b>	15,56	20,02	14,83	100%
<b>P3T7</b>	6,75	8,25	4,50	70%
<b>P3T8</b>	6,56	16,52	14,33	100%
<b>P3T9</b>	8	1	1	66%
<b>P3T10</b>	7,13	10,04	7,67	100%
<b>P3T11</b>	11	9,67	7,25	80%
<b>P3T12</b>	2,88	1,96	0,75	100%
<b>P3T13</b>	5,88	9,96	4	100%
<b>P3T14</b>	16,38	14,46	4,50	80%
<b>P3T15</b>	18,75	22	7,56	80%
<b>P3T16</b>	1	2	2	100%

<b>P3T17</b>	64,83	121,58	85,70	95%
--------------	-------	--------	-------	-----

Table 5: Period 3 measurements.

And to contrast the result of the improved process to the baseline:

<b>Period</b>	<b>P1</b>	<b>P2</b>	<b>P3</b>
<b>Development cost (h)</b>	271,02	413,37	242,96
<b>Testing cost (h)</b>	460,22	777,50	490,90
<b>Initial requirements' coverage</b>	32%	58%	91%
<b>Initial test automation cost</b>	13,34%	28,37%	64,78%
<b>Final requirements' coverage</b>	48%	70%	91%
<b>Final test automation cost</b>	46,31%	41,55%	64,78%
<b>Number of Live bugs</b>	26	16	7
<b>Number of hotfixes</b>	18	10	1

Table 6: Baseline and improved process data gathering results.

Table 6 serves as a validation for the first modification, and as a confirmation for the correlation between requirements' coverage and effectiveness. The significantly low number of hotfixes and live bugs compared to **P1** and **P2** shows that the increased requirements' coverage is at effect, all within the acceptable testing cost range (testing cost = 2 \* development cost ( $\pm$  development cost/10)).

### Introducing Test Generation

The second modification, which the success of the proposed solution relies on, is to propose a structure for writing automated tests that would render generating them automatically possible, and then to implement a tool that helps writing the tests. Figure 2 represents this modification with a lighter colour for the last two activities of automating a test case, which means that they would take less time than they used to. This second part of the solution attacks the remaining third problem **Prob3**.

It's important to note that the solution relies on the combination of the two modifications. As the first modification was validated, the rest of this chapter will be dedicated to the analysis and description of the second modification.

## 6.2 Analysis

The previously mentioned omission from Figure 2 is related to user interface automated test. The focus will be on the Fitness tests solely for the following reasons:

- Only 20% of the requirements are about the user interface.



- Only 30% of the code base is traced to functionality for the user interface.
- 94% of the automated tests are written in Fitnesse, and the user interface tests provide little to no value due to the nature of our system.
- The user interface is a simplistic and minimalistic one, and testing it is better done manually because automating it would require too much work then the benefits are worth.

Now that it's established that the goal is to make Fitnesse tests more structured and possible to be generated, we can proceed by taking a close look at the existing tests and look for some patterns.

## **Test Types**

Excluding the user interface, there are five logical areas which constitute our system, and thus five types of Fitnesse tests:

1. File import tests: these are not interesting in the context of the problems at hand, because they are generic test pages, and require no or very little scripting. Adding a test for a process of such type, is as simple as
  - a. Adding a valid test file into the specified directory.
  - b. Then adding the process name to a list of processes to be ran.
  - c. Finally adding a check for the status of the process after running.

The three steps are highlighted in Figure 3, assuming that the process being added is named `IMPORT_FILE_ATM_SETTLEMENT_VISA`.

```
import
ee.lhv.acq.fitness.fixture
ee.lhv.acq.fitness.fixture.query
```

```
variable defined: date=12.03.2058
variable defined: testFolderForFiles=fitness
```

## Check error messages

- | Query: Get errors by journal period id |  | \$journalPeriodId   |
|--|--|---------------------|
| CODE                                   |  | DESCRIPTION JOJO_ID |

script	Journal Job Fixture
delete all files for date	12.03.2058

script	Rollback Test Data
--------	--------------------

2. File extract tests: these are tests for processes which parse imported files and store the parsed information into the database. In addition to the steps described in the previous type, this type of tests has an additional part, which is a query table that checks the parsing resulting from parsing to set of predefined values  
Figure 4 highlights this table

## Purpose

- To test the **positive** case of Extract ATM Settlements Visa job.
- All fields are mandatory, exceptions are:
  - If Entry\_type='V01' (number of transactions) then field "Ccy" (record currency) is empty
  - If Entry\_type='V01' (number of transactions) then field "DB\_CR" (credit/debit indicator) is empty

## Test steps

import
ee.lhv.acq.fitnessse.fixture
ee.lhv.acq.fitnessse.fixture.query

### Initialize variables and check journal job status

variable defined: date=12.03.2058

variable defined: testFolderForFiles=fitnessse

script	Journal Job Fixture				
prepare graph for types	EXTRACT_FILE_ATM_SETTLEMENT_VISA				
init all files from folder	fitnessse	with name	SuccessTestFile	for date	12.03.2058
\$journalPeriodId=	run journal jobs with date	12.03.2058			
check	get journal job status by period id	\$journalPeriodId	and type code	EXTRACT_FILE_ATM_SETTLEMENT_VISA	COMPLETED
\$fdAtmSettlementVisaJojoId=	get journal job id by period id	\$journalPeriodId	and type code	EXTRACT_FILE_ATM_SETTLEMENT_VISA	
delete all files for date	12.03.2058				

### Check atm visa records

Query: Get File By Name And Journal Job Id					
ATM_SETTLEMENT	\$fdAtmSettlementVisaJojoId				
CLEA_ROW_NO	DEBIT_CREDIT	AMOUNT	REP_DATE	TYPE	JOJO_ID
1	C	1899.72	2015-03-06 00:00:00.0	VISA	\$fdAtmSettlementVisaJojoId
2	D	108338.66	2015-03-06 00:00:00.0	VISA	\$fdAtmSettlementVisaJojoId
3	null	0.00	2015-03-06 00:00:00.0	VISA	\$fdAtmSettlementVisaJojoId
4	C	338.61	2015-03-06 00:00:00.0	VISA	\$fdAtmSettlementVisaJojoId

### Check error messages

- Shouldn't be any as it is a positive test case

Query: Get errors by journal period id		\$journalPeriodId
CODE	DESCRIPTION	

Figure 4. Extract file Fitnessse test example.

3. Calculation tests: this is most useful and critical type of tests, and it's for the processes that take the parsed data coming from external systems, or an aggregation of existing data from our system, perform complex calculations, and store the result into the database, and in some cases send it to external systems.

At first glance many of these tests seem long, complicated, and confusing, but they follow the same pattern, which will be described in the next section, for now it can be thought of roughly as an arrangement of: data preparation, running a process, and checking the result.

The confusion springs up from the inconsistent ways the tests are written, from a structural perspective, for example data checking is done sometimes with query tables, and on other times using script tables. The other way in which the tests are not consistent is the naming of the instructions (Fixtures), for example for checking a result, sometimes 'query' + sentence is used, sometimes 'get', other times 'check', or none of those.

Figure 5 is a section of the data preparation part of a calculation test, the highlighted are different steps for data preparation.

#### Initialize variables and check the parent job status

variable defined: date=12.03.2058

variable defined: testFilePathPos=fitnesse/calcmERCHANTfees/calcmERCHANTinterchangeFee/successMultipleMerchants

variable defined: testFilePathRep=fitnesse/calcmERCHANTfees/calcmERCHANTinterchangeFee/successMultipleMerchantsRep

script	Journal Job Fixture					
prepare graph for types	MERCHANT_TERMINALS_CHECK					
init file	fitnesse/calcmERCHANTfees/calcmERCHANTinterchangeFee/successMultipleMerchants	with type	IPM_NETS_MP	for date	12.03.2058	
init file	fitnesse/calcmERCHANTfees/calcmERCHANTinterchangeFee/successMultipleMerchantsRep	with type	IPM_NETS_MP_REPORT	for date	12.03.2058	
\$JournalPeriodId=	run journal jobs with date 12.03.2058					
check	get journal job status by period id	\$JournalPeriodId	and type code	MERCHANT_TERMINALS_CHECK	COMPLETED	
delete all files for date	12.03.2058					
\$usacId=	get usac id by member id code	15808				

#### Display merchant ID's

Merchant id from branch id	
usmeId?	branchId
\$usmeId1=	8280091
\$usmeId2=	8280133

#### Display extracted first presentment data

Query: Get ipm first presentment daily by journal period id		\$JournalPeriodId								
IPPE_ID	TRANSACTION_DTIME	TRIMMED_CARD_ACCEPTOR_ID_CODE	REFERENCE_NO	PROCESSING_CODE	CARD_TYPE	AMTREC_NET	CURREC	CHARGE_FEE	PROCESSING_FEE	
\$ipfpId1=	87654321 001	2730	0		801MC	D000000000001010	978	null	null	
\$ipfpId2=	87654321 001	2730	0		801MC	C000000000001010	978	null	null	
\$ipfpId3=	87654321 001	2730	0		801MC	C000000000002020	978	null	null	
\$ipfpId4=	87654321 001	2731	0		689MT	C000000000003030	978	null	null	
\$ipfpId5=	12345678 001	2732	0		801MC	D000000000001010	978	null	null	
\$ipfpId6=	12345678 001	2732	0		801MC	C000000000001010	978	null	null	
\$ipfpId7=	12345678 001	2732	0		801MC	C000000000002020	978	null	null	
\$ipfpId8=	12345678 001	2733	0		689MT	C000000000003030	978	null	null	

#### Display IC fee claims data

##### IC fee claims data for merchant 1

Query: Get merchant revenue and fee claims by joype id and usme id and type		\$JournalPeriodId	\$usmeId1	ACQ_MERCHANT_IC_FEE_LOCAL						
ID	AMTREC_NET	CURRENCY	CARD_TYPE	CLAIM_NAME	FEE_PERCENT	FEE_FIXED_AMOUNT				
\$ipfpId1	D000000000001010	EUR	801MC	ACQ_MERCHANT_IC_FEE_LOCAL	0.300000					
\$ipfpId2	C000000000001010	EUR	801MC							
\$ipfpId3	C000000000002020	EUR	801MC							
\$ipfpId4	C000000000003030	EUR	689MT	ACQ_MERCHANT_IC_FEE_LOCAL		0.012700				

##### IC fee claims data for merchant 2

Query: Get merchant revenue and fee claims by joype id and usme id and type		\$JournalPeriodId	\$usmeId2	ACQ_MERCHANT_IC_FEE_LOCAL						
ID	AMTREC_NET	CURRENCY	CARD_TYPE	CLAIM_NAME	FEE_PERCENT	FEE_FIXED_AMOUNT				
\$ipfpId5	D000000000001010	EUR	801MC	ACQ_MERCHANT_IC_FEE_LOCAL	0.300000					

Figure 5. Calculation Fitnessse, data preparation example.

Figure 6 is a section of the data check part of the same test, in this case the check is easily understandable, as the table used is consistently the query table.

#### Check claim entries for merchant 1

Query: Get claim entries by journal period id and job type code for merchant	\$journalPeriodId	CALC_CLAIM_MERCHANT_LOCAL_IC_FEE	\$usmeId1			
DEBIT_CREDIT	AMOUNT	CUR_CODE	USIS_ID	USME_ID	USAC_ID	
D	0.03	EUR	null	\$usmeId1	\$usacId	
C	0.10	EUR	null	\$usmeId1	\$usacId	

#### Check account entries for merchant 1

Query: Get acc entries by journal period id and job type code for merchant	\$journalPeriodId	CALC_CLAIM_MERCHANT_LOCAL_IC_FEE	\$usmeId1			
ACCOUNT_NO_D	ACCOUNT_NO_C	AMOUNT	CURRENCY_CODE	USIS_ID	USME_ID	USAC_ID
3038	1792	0.03	EUR	null	\$usmeId1	\$usacId
1792	3038	0.10	EUR	null	\$usmeId1	\$usacId

#### Check claim entries for merchant 2

Query: Get claim entries by journal period id and job type code for merchant	\$journalPeriodId	CALC_CLAIM_MERCHANT_LOCAL_IC_FEE	\$usmeId2			
DEBIT_CREDIT	AMOUNT	CUR_CODE	USIS_ID	USME_ID	USAC_ID	
D	0.30	EUR	null	\$usmeId2	\$usacId	
C	0.92	EUR	null	\$usmeId2	\$usacId	

#### Check account entries for merchant 2

Query: Get acc entries by journal period id and job type code for merchant	\$journalPeriodId	CALC_CLAIM_MERCHANT_LOCAL_IC_FEE	\$usmeId2			
ACCOUNT_NO_D	ACCOUNT_NO_C	AMOUNT	CURRENCY_CODE	USIS_ID	USME_ID	USAC_ID
3038	1792	0.30	EUR	null	\$usmeId2	\$usacId
1792	3038	0.92	EUR	null	\$usmeId2	\$usacId

#### Check first presentment data

- IC\_FEE column should be updated

Query: Get ipm first presentment daily by journal period id	\$journalPeriodId							
IPFP_ID	TRANSACTION_DTIME	TRIMMED_CARD_ACCEPTOR_ID_CODE	REFERENCE_NO	CARD_TYPE	CHARGE_FEE	PROCESSING_FEE	IC_FEE	AM
\$ipfpId1					null	null	-0.030300	null
\$ipfpId2					null	null	0.030300	null
\$ipfpId3					null	null	0.060600	null
\$ipfpId4					null	null	0.012700	null
\$ipfpId5					null	null	-0.303030	null
\$ipfpId6					null	null	0.303030	null
\$ipfpId7					null	null	0.606060	null
\$ipfpId8					null	null	0.012700	null

#### Check error messages

- Shouldn't be any as it is a positive test case

Query: Get errors by journal period id	\$journalPeriodId
CODE	DESCRIPTION

Figure 6. Calculation Fitnessse, result checking example.

4. Crosscheck tests: these are tests specific to processes that check the result of the calculation job, these are also the processes that are heavily tested and covered by automated tests, because they ensure that the calculations were done correctly, or indicate that they are not, so they can be fixed in time, before any legal liabilities arise.

There are no new techniques used in writing this type of tests, roughly the same pattern is respected; data preparation, running the checking process, and checking that the checking process behaves as expected.

5. General tests: these are tests that can't be classified under the previous four types, and are testing other logical areas of the system, and are similar to the previously described test types in structure and components. Examples, for tests which classify under this type are: transactions' execution, customer contracts' manipulation, sending emails, etc.

Figure 7 is an example of an invoice email sending process test.

## Purpose

- To test the **positive** case for **sending merchant terminal fee invoices**.

## Test steps

### Import required fixtures

import
ee.lhv.acq.fitness.fixture
ee.lhv.acq.fitness.fixture.query
ee.lhv.acq.fitness.fixture.decision

### Initialize variables and check journal job status

variable defined: date=02.03.2058

script	Journal Job Fixture				
prepare graph for types	JOURNAL_PERIOD_OPEN				
\$journalPeriodId=	run journal jobs with date	02.03.2058			
check	get journal job status by period id	\$journalPeriodId	and type code	JOURNAL_PERIOD_OPEN	COMPLETED

Merchant id from branch id	
usmeId?	branchId
\$usmeIdLegal=	8280091

script	Terminal Fee Fixture
add missing invoice email address to merchants	ali.belakehal@lhv.ee

script	Journal Job Fixture				
prepare graph for types	SEND_MERCHANT_TERMINAL_FEE_INVOICES				
run journal job by period id	\$journalPeriodId	and type code	CALC_CLAIM_MERCHANT_TERMINAL_FEE		
run journal job by period id	\$journalPeriodId	and type code	ACC_TRANSACTION_GENERATE_MERCHANT_TERMINAL_FEE		
run journal job by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES		
check	get journal job status by period id	\$journalPeriodId	and type code	ACC_TRANSACTION_GENERATE_MERCHANT_TERMINAL_FEE	COMPLETE
check	get journal job status by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES	COMPLETE
\$jobId=	get journal job id by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES	

### Check terminal fee invoice deliveries

Query: Get terminal fee invoice deliveries by journal period id for merchant	\$journalPeriodId	\$usmeIdLegal			
USME_ID	PERIOD_START_DTIME	PERIOD_END_DTIME	STATUS	STATUS_DTIME	EMAIL
\$usmeIdLegal	2058-02-01 00:00:00.0	2058-02-28 00:00:00.0	SENT		ali.belakehal@lhv.ee

### Check error messages

\* No error messages since this is a positive test case

Query: Get errors by journal period id	\$journalPeriodId
CODE	DESCRIPTION

Figure 7. Invoice sending process Fitnessse example.

To summarize the findings up to this point, from a syntax point of view:

All tests are an arrangement of three types of sections, not organized in any particular order is some test types, but seem to have a specific order of appearance in other types:

- Data preparation, which is through prepared files, or database manipulation instructions.
- Running processes, which are an already existing set of instructions used across the tests.
- Data verification, which are a set of instructions for checking the database state. Most of the time they are Query tables, but sometimes they can be under Script tables.

From what has been presented so far, it seems like introducing some rules to the vague pattern that revealed itself, could lead to a more formal expression of these tests, which in turn could make their automatic generation attainable.

Putting this information aside for a moment, it's time to introduce a useful concept that will help parse those tests and enforce structure.

## Grammar

Roughly expressing, a grammar is a formal notation, detailed enough to describe how a language is built, or how the alphabets of a language should be combined to construct syntactically sound instances of that language [2].

Without getting into the discussion of whether *Fitnessse* is a formal language<sup>10</sup>, and of what type<sup>11</sup>. Let there be the explicit assumption that it is a formal language with a context-sensitive grammar<sup>12</sup>, and that'll prove to be a **correct enough** assumption for the implementation purposes to take advantage of the grammar's properties.

### *Fitnessse Alphabet (Tokens)*

These are the elementary symbols which *Fitnessse* is built with:

1. Separator: the character '|'
2. Import keyword: the lower case literal 'import'
3. Script keyword: the lower case literal 'script'
4. Check keyword: the lower case literal 'check'
5. Query keyword: the capitalised literal 'Query'
6. Comment keyword: the lower case literal 'comment'
7. Word sequence: a whitespace separated sequence of lowercase literals; used for wiki instructions
8. Capitalized words' sequence: a whitespace separated sequence of capitalized literals; used for Fixture names
9. Capitalised word sequence: a capitalized word followed by a word sequence; used for Query instructions
10. Camel case literal: a lower camel case string
11. Capitalized camel case literal: an upper camel case string
12. Database entity name: an all upper case snake case literal
13. Variable name: the character '\$' concatenated with a lower camel case or snake case string.
14. Fixture class path: a '.' Separated sequence of lower case literals

Figure 8 highlights examples of occurrences of some tokens in the list.

---

<sup>10</sup> [https://en.wikipedia.org/wiki/Formal\\_language](https://en.wikipedia.org/wiki/Formal_language)

<sup>11</sup> [https://en.wikipedia.org/wiki/Chomsky\\_hierarchy](https://en.wikipedia.org/wiki/Chomsky_hierarchy)

<sup>12</sup> [https://en.wikipedia.org/wiki/Context-sensitive\\_grammar](https://en.wikipedia.org/wiki/Context-sensitive_grammar)

## Test steps

### Import required fixtures

import	2
ee.lhv.acq.fitness.fixture	
ee.lhv.acq.fitness.fixture.query	14
ee.lhv.acq.fitness.fixture.decision	

### Initialize variables and check journal job status

variable defined: date=02.03.2058

script	3	Journal Job Fixture	8
prepare graph for types		JOURNAL_PERIOD_OPEN	
\$journalPeriodId=	13	run journal jobs with date	02.03.2058
check	4	get journal job status by period id	\$journalPeriodId and type code JOURNAL_PERIOD_OPEN COMPLETED

Merchant id from branch id	
usmeId?	branchId
\$usmeIdLegal=	8280091

script	Terminal Fee Fixture
add missing invoice email address to merchants	ali.belakehal@lhv.ee

script	Journal Job Fixture			
prepare graph for types	7	SEND_MERCHANT_TERMINAL_FEE_INVOICES		
run journal job by period id	\$journalPeriodId	and type code	CALC_CLAIM_MERCHANT_TERMINAL_FEE	
run journal job by period id	\$journalPeriodId	and type code	ACC_TRANSACTION_GENERATE_MERCHANT_TERMINAL_FEE	
run journal job by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES	
check	get journal job status by period id	\$journalPeriodId	and type code	ACC_TRANSACTION_GENERATE_MERCHANT_TERMINAL_FEE
check	get journal job status by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES
\$jobId=	get journal job id by period id	\$journalPeriodId	and type code	SEND_MERCHANT_TERMINAL_FEE_INVOICES

### Check terminal fee invoice deliveries

Query: Get terminal fee invoice deliveries by journal period id for merchant		\$journalPeriodId	12	\$usmeIdLegal		
USME_ID		PERIOD_START_DTIME	PERIOD_END_DTIME	STATUS	STATUS_DTIME	EMAIL
\$usmeIdLegal		2058-02-01 00:00:00.0	2058-02-28 00:00:00.0	SENT		ali.bel

Figure 8. Highlighted Fitness alphabets.

## Formalising Fitness instructions

Now that the core Fitness constituents are known, it's time to identify where structure can be introduced. Two areas where this formalisation can be injected:

1. Organization: which is the order in which the wiki tables appear, or the three test parts (data preparation, process execution, and result verification), but that would be limiting if done too strictly, because many tests rely on these order being loosely controlled, for example, crosscheck processes' tests prepare data for multiple processes, and execute them in a specific order.
2. Instruction wording: this is where instructions can be improved from English sentences that follow no specific pattern, to English sentences that respect given rules for writing instructions. This would make them possible to parse, and later on generate to Java code.

As a sample of the instruction wording enhancement, let's look at an example from the English language, and see how it translates to the Fitness context. A very simplistic representation of writing sentences in English [2] is:

- A sentence is a noun followed by a verb, or a sentence followed by a conjunction followed by a sentence.
- A conjunction can be an 'and', an 'or', or a 'but'.
- A noun can be a 'bird', or a 'fish'.
- A verb can be 'fly', or 'swim'.



Formally, this is expressed as follows:

- **Sentence:**
  - Noun Verb
  - Sentence Conjunction Sentence
- **Conjunction:**
  - 'and'
  - 'or'
  - 'but'
- **Noun:**
  - 'bird'
  - 'fish'
- **Verb:**
  - 'fly'
  - 'swim'

To build a sentence within this framework, is to have it conform to the defined rules, so Figure 9 shows how the sentence 'birds fly and fish swim' maps to those rules.

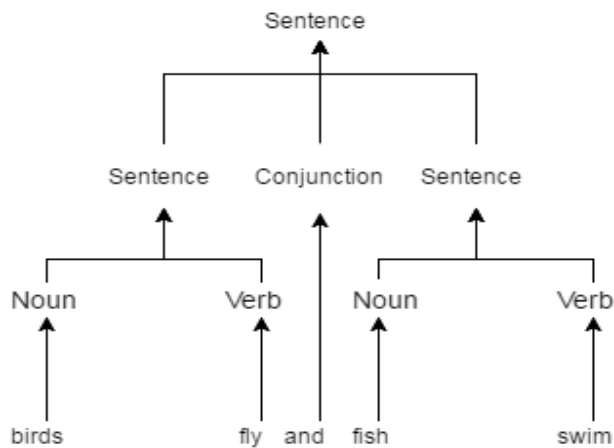


Figure 9. Sentence mapping to rules.

Following the same logic, a query instruction grammar can be stated as follows:

- A query instruction is a query keyword 'Query' followed by a colon ':' followed by 'Get' followed by a table name, followed by the conjunction 'by', and followed by a list of parameter names.
- A table name is a word sequence, which doesn't contain the word 'by' (in the formal notation this restriction is implied from the parent rule).
- A list of parameter names, is a parameter name, or a parameter name followed by the conjunction 'and', followed by a list of parameter names.
- A parameter name is a word sequence, which doesn't contain the word 'and' (in the formal notation this restriction is implied from the parent rule).

In a formal notation, this becomes:

- **Query instruction:**
  - 'Query: Get' + table name + 'by' + parameter names list
- **Table name:**

- Word sequence
- **Parameter names list:**
  - Parameter name
  - Parameter name + 'and' + parameter names list
- **Parameter name:**
  - Word sequence

Figure 10 shows the mapping of the query instruction ‘Query: Get merchant payment by transaction date and card type’.

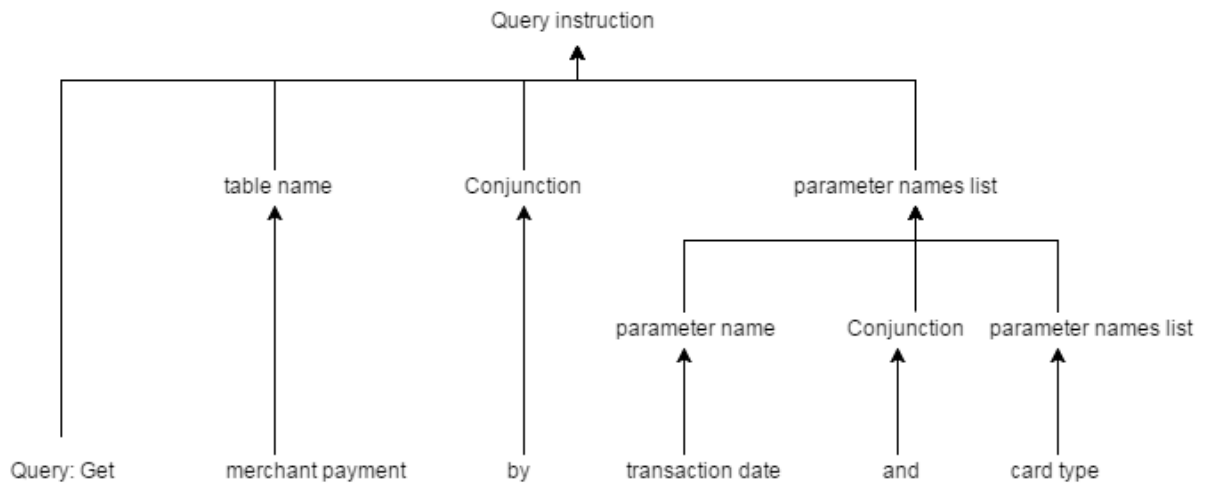


Figure 10. Query instruction mapping to rules.

This implies that a parser can be implemented along these rules. Subsequently, a logic that uses the parser’s result will have all the required information to generate the code for this query instruction: which table to query, and which lookup conditions to apply.

A parsing expression grammar<sup>13</sup> can be formulated for the remaining types of instructions, and table. It’s important to note that the full power of this technique comes from its extendibility, whenever a new pattern in writing Fitness tests is detected, it can be formalized as demonstrated, then implemented into the parser, and from there Java and SQL code can be generated based on that pattern.

The grammar that is used in the tool implementation, which is the realization of the proposed solution, will be presented in the next section.

### 6.3 Solution

In section 2 (Analysis) we reached the conclusion that tests are comprised of three logical parts: data preparation, processes’ execution, and result verification. An additional conclusion was that the process execution part is generic and is reusable from existing fixtures, so it can be removed from the context of this attempt of a solution.

The attention now move to those data related parts, and now they should be expressed in further detail, so that a concrete solution can be reached. The first of which (data preparation) will be solved along two dimensions; Fixture generation, and test data generation, and

<sup>13</sup> [https://en.wikipedia.org/wiki/Parsing\\_expression\\_grammar](https://en.wikipedia.org/wiki/Parsing_expression_grammar)

the second of which was mostly solved in the previous section, and will be entirely solved in the next.

## Fixture Generation

The full parsing grammar is present in Appendix 0 to serve as a reference, and this section will focus on the two most relevant issues.

1. **The first issue** is data preparation, which we can address by reintroducing the decision table from Section 3.1, and proposing it as an alternative to the use of test files as the main source of test data for reasons which will be uncovered in the Test Data Generation section. As referred to in the full grammar:

Batch insert fixture table

- Separator + Fixture class path + Capitalized camel case entity name + Separator + newline
- Separator + columns to be matched + 'get' + Capitalized camel case entity name + '?'
- values table

ee.lhv.acq.fitness.fixture.MerchantPaymentInsertFixture										
processingCode	cardAcceptorIdCode	amtrecDb	amtrecCr	countDebits	countCredits	merchantId	cardTypeId	referenceNo	currec	getMerchantPaymentId?
000000	87654321 001	D000000000001010	C000000000003030	1	2	8280091	401MA	2711	978	\$mpRow1=
000000	87654321 001	D000000000001010	C000000000003030	1	2	8280091	689MB	2712	978	\$mpRow2=
000000	87654321 001	D000000000001010	C000000000003030	1	2	8280091	A00AX	2713	978	\$mpRow3=
000000	87654321 001	D000000000001010	C000000000003030	1	2	8280091	767AX	2714	978	\$mpRow4=

Figure 11. Batch insert table.

After subjecting the table from Figure 11 to the batch insert fixture table grammar, the resulting information will be:

1. The fixture class name, and to path for where to generate it, if it doesn't exist, from the first line, in this example it is 'ee.lhv.acq.fitness.fixture.MerchantPaymentInsertFixture'.
2. The database table for which the insert fixture is to be created, from the last column of the last line, in this example it is 'MerchantPayment'.

The remaining lines in the table are the values to be inserted, and are used when the test is executed, that is, they are not used in generating the fixture, and can be discarded.

Now that the parsing part is over, the generation logic uses those information following the algorithm:

1. Convert the table name to an upper camel case.
2. Get all tables which are related to the table at hand by a non-null foreign key recursively.
3. For each related table, which doesn't have an existing fixture generated:
  1. Get the table metadata (columns' name, data type, size if applicable, and whether the column is mandatory foreign key).
  2. Generate the corresponding Java fields, with their getters and setters.
  3. Generate the SQL insert statement, and the insert method
  4. Amalgamate the generated code into a syntactically sound insert fixture class.
  5. Save the class name, path, and the corresponding database table into the existing fixtures table for future usage.

To eliminate any ambiguities, let's go through the algorithm following the example started in Figure 11. Assuming the following database structure:

Table 7: Demonstration database structure

Table	Columns	Foreign keys	Related Tables
MERCHANT_PAYMENT	[PAYMENT_ID] INT NOT NULL IDENTITY [PROCESSING_CODE] VARCAHR(16) NOT NULL [CARD_ACCEPTOR_ID_CODE] VARCAHR(13) NOT NULL [AMTREC_DB] VARCAHR(16) NOT NULL [AMTREC_CR] VARCAHR(16) NOT NULL [COUNT_DEBITS] INT [MERCHANT_ID] VARCAHR(15) NOT NULL [CARD_TYPE_ID] VARCAHR(5) NOT NULL [REFERENCE_NO] INT NULL [CURREC] NUMERIC(3) NOT NULL	MERCHANT_ID	MERCHANT
MERCHANT	[MERCHANT_ID] VARCAHR(15) NOT NULL [NAME] VARCAHR(50) NOT NULL [EMAIL] VARCAHR(50) NULL [PHONE] VARCAHR(50) NULL [USER_ID] INT NOT NULL [REPRESENTATIVE_ID] NULL	USER_ID REPRESENTATIVE_ID	USER REPRESENTATIVE
USER	[USER_ID] INT NOT NULL IDENTITY [NAME] VARCAHR(50) NOT NULL [REGISTRATION_CODE] VARCAHR(50) NOT NULL [TYPE] VARCAHR(50) NOT NULL	-	-

Given that, the steps would be:

1. The table name obtained from the parser as 'MerchantPayment' is converted to MERCHANT\_PAYMENT
2. The table MERCHANT\_PAYMENT refers to the table MERCHANT, which in turn refers to the tables USER and REPRESENTATIVE, but as the relation to REPRESENTATIVE is not mandatory, then only USER is kept, and USER table doesn't refer to any other tables, so the final list of tables for which fixtures are to be created is {MERCHANT\_PAYMENT, MERCHANT, USER}  
The reason for creating fixtures for the related tables as well will be explained and become clear when the generation is done.
3. For the purpose of demonstration, let's assume that the table MERCHANT already has a generated insert fixture class named 'MerchantInsertFixture', so the following steps will apply for MERCHANT\_PAYMENT, and USER.
  - a. First iteration :
    1. The table meta data which will be used to create the fields and their helper methods:

Column name	Data type	Size	Related Table
PROCESSING_CODE	VARCAHR	16	
CARD_ACCEPTOR_ID_CODE	VARCAHR	13	
AMTREC_DB	VARCAHR	16	
AMTREC_CR	VARCAHR	16	
COUNT_DEBITS	INT		
MERCHANT_ID	VARCAHR	15	MERCHANT
CARD_TYPE_ID	VARCAHR	5	
REFERENCE_NO	INT		
CURREC	NUMERIC	3	

2. For this step, all fields should be generated into code, but for the purpose of demonstration, only 3 will be presented:

Field	Corresponding Java code
PRO- CESSING_CODE	<pre> @Setter private static String processingCode; public String getProcessingCode () {     if(processingCode == null) {         processingCode = getRandomString();     }     return processingCode; } </pre>
COUNT_DEBITS	<pre> private static Integer countDebits; public Integer getCountDebit () {     if(countDebits == null) {         countDebits = getRandomInteger();     }     return countDebits; }  public void setCountDebit(String value) {     countDebit = Integer.parseInt(value); } </pre>
MERCHANT_ID	<pre> @Setter private static String merchantId; private static MerchantInsertFixture merchantIdInserter =     new MerchantInsertFixture(); public String getMerchantId () {     if(merchantId == null) {         merchantId = merchantIdInserter.getMerchantId();     }     return merchantId; } </pre>

Further explanation about the getters' role will be at the end of the demonstration.

3. Figure 12 shows the resulting SQL insert query for this table.
  4. Figure 13 shows the whole generated insert fixture class, some of the fields and their helper methods were omitted for conciseness.
  5. The table name and the generated Fixture data will be saved in a table that is meant to track the existing fixtures.
- b. Second iteration:
- The table USER will go through the same steps described in the first iteration.

```

public Long getMerchantPaymentId() {
    scriptHelper.runScript(String query = String.format("INSERT INTO [ACQ].[MERCHANT_PAYMENT]
([PROCESSING_CODE],[CARD_ACCEPTOR_ID_CODE],[AMTREC_DB],[AMTREC_CR],[COUNT_DEBITS], " +
"[MERCHANT_ID],[CARD_TYPE_ID],[REFERENCE_NO],[CURREC]) VALUES (%s, %s, %s, %s, %d, %s, %s, %d, %s)",
getProcessingCode(), getCardAcceptorIdCode(), getAmtrecDb(), getAmtrecCr(), getCountDebits(),
getMerchantId(), getCardTypeId(), getReferenceNo(), getCurrec());
    return scriptHelper.getLong("SELECT @@IDENTITY");
}

```

Figure 12. The resulting insert query method.

```

import lombok.Setter;
import ee.lhv.acq.unit.ScriptHelper;
import ee.lhv.acq.fitness.spring.BeanUtil;
import org.springframework.jdbc.core.JdbcTemplate;
import org.apache.commons.lang.StringUtils;

public class MerchantPaymentInsertFixture extends BaseInsertFixture {

    @Setter private static String processingCode;
    public String getProcessingCode () {
        if(processingCode == null) {
            processingCode = getRandomString();
        }
        return processingCode;
    }

    private static Integer countDebits;
    public Integer getCountDebit () {
        if(countDebits == null) {
            countDebits = getRandomInteger();
        }
        return countDebits;
    }

    public void setCountDebit(String value) {
        countDebit = Integer.parseInt(value);
    }

    @Setter private static String merchantId;
    private static MerchantInsertFixture merchantIdInserter =
        new MerchantInsertFixture();
    public String getMerchantId () {
        if(merchantId == null) {
            merchantId = merchantIdInserter.getMerchantId();
        }
        return merchantId;
    }

    // The remaining fields
    //...

    public Long getMerchantPaymentId() {
        scriptHelper.runScript(String query = String.format("INSERT INTO [ACQ].[MERCHANT_PAYMENT]
([PROCESSING_CODE],[CARD_ACCEPTOR_ID_CODE],[AMTREC_DB],[AMTREC_CR],[COUNT_DEBITS], " +
"[MERCHANT_ID],[CARD_TYPE_ID],[REFERENCE_NO],[CURREC]) VALUES (%s, %s, %s, %s, %d, %s, %s, %d, %s)",
getProcessingCode(), getCardAcceptorIdCode(), getAmtrecDb(), getAmtrecCr(), getCountDebits(),
getMerchantId(), getCardTypeId(), getReferenceNo(), getCurrec());
        return scriptHelper.getLong("SELECT @@IDENTITY");
    }
}

```

Figure 13. The full resulting insert fixture class.

Before continuing, some explanations are required to clarify remaining ambiguities. The way Fitnesses executes the general purpose table, which is the batch insert table in the context of this solution, is by executing the insert method for every line in the values section. The fields are initialized from the strings provided under the field's column in the wiki table,

that's the reason why setters are required. For strings the default setter would do, but for other types we have to define a setter that parses the string provided in the wiki, into a value of the corresponding field's type.

In the case where:

1. A column in the wiki doesn't have the value to initialize the corresponding field
2. The column is missing all together from the wiki page

The corresponding Java field will be null, and when the insert statement is executed, it will be rejected due to the NOT NULL constraints. An example illustrating these cases is shown in Figure 14.

ee.lhv.acq.fitness.fixture.MerchantPaymentInsertFixture								
processingCode	cardAcceptorIdCode	amtrecDb	amtrecCr	countDebits	merchantId	cardTypeId	referenceNo	getMerchantPaymentId?
000000	87654321 001	D0000000000001010	C0000000000003030	1	8280091	401MA	2711	\$mpRow1=
000000	87654321 001	D0000000000001010	C0000000000003030		8280091	689MB		\$mpRow2=
000000	87654321 001	D0000000000001010	C0000000000003030	1	8280091	A00AX	2713	\$mpRow3=
000000	87654321 001	D0000000000001010	C0000000000003030	1	8280091	767AX	2714	\$mpRow4=

Figure 14. Missing currec column, and some values.

The getters then fill these null values in case of normal fields with random values, and in case of foreign key fields they are filled by making the corresponding related table's insert fixture class return a valid foreign key value to refer to. The called insert fixture class will do the same in case its underlying table refers to other tables also. This recursive process is guaranteed to work because of step 2 of the algorithm, which ensures that any missing insert fixture classes for related tables are generated.

The fields are static to ensure that this process of recursive foreign key handling is done once only (the first time around), and not for every line that is missing a column.

This automatic filling and foreign key handling feature is important to cut down the time that is wasted on maintaining a valid state of the database, and the foreign keys relations. This way testers can focus on inserting valid data that is for the process being tested, and that only.

**The second issue** is result verification, for which the parsing grammar has already been explored. The final parsing grammar for the query table becomes:

Query fixture table

- query fixture table header + newline
- separator + columns to be matched

Query fixture table header

- separator + "Query: " + Query fixture class name + 's' + separator + separated arguments
- separator + "Query: " + Query fixture class name + separator + separated arguments

Query fixture class name

- Get + database table name + connected parameter names

Connected parameter names

- connector + lowercase entity name
- connected parameter names + connector + lowercase entity name

Connector

- "by" + white spaces
- "and" + white spaces
- "with" + white spaces
- "for" + white spaces
- "from" + white spaces

Display merchant payment data

Query: Get merchant payments by card type id and merchant id	401MA	8280091					
PAYMENT_ID	REFERENCE_NUMBER	PROCESSING_CODE	CARD_TYPE_ID	AMTREC_DB	AMTREC_CR	CURREC	MER
	2711	0	401MA	D0000000000001010	C000000000003030	978	8280
	2712	0	401MA	D0000000000010101	C0000000000030303	978	8280
	2713	0	401MA	D00000000000101010	C00000000000303030	978	8280
	2714	0	401MA	D000000000001010101	C000000000003030303	978	8280

Figure 15. Query table example.

Again, the first line is the only useful section from the table for the parsing and code generation purposes, the rest of the table is when the tests is ran.

After applying the improved grammar on the first line the same way we did in Formalising Fitness instructions, the information extracted are:

- The main queried table name; in the Figure 15 example, it would be 'merchant payment'
- The name of the fixture query class that is to be generated
- The query constraints; in this example 'card type id' and 'merchant id'

Adding that the generation algorithm is:

1. Convert the table name to the upper snake case format.
2. Convert the parameter names to upper snake case format.
3. Use them to generate the query.
4. Convert the fixture class name to an uppercase camel case format.
5. Generate the remaining of the class code

Figure 16 highlights these steps.

```

package ee.lhv.acq.fitness.fixture.query;

public class GetMerchantPaymentsByCardTypeIdAndMerchantId extends SqlQueryFixture {

    private final static String SQL = "SELECT MERCHANT_PAYMENT.* FROM [ACQ].[MERCHANT_PAYMENT]
    + "WHERE CARD_TYPE_ID = ? AND MERCHANT_ID = ?";

    public GetMerchantPaymentsByCardTypeIdAndMerchantId(Integer cardTypeId, String MerchantId) {
        super(SQL, cardTypeId, MerchantId);
    }
}

```

Figure 16. Generated query fixture class.

For now this simple version is enough as a proof of concept, but it's important to note that this can be extended to generate more complex queries, including multiple joins and complex conditions.

A projection of how it could be extended (which is also what's currently being implemented):

Query fixture class name



- Get + database table names + 'by' + connected parameter names

Database table names

- lowercase entity name
- database table names + 'and' + lowercase entity name

Connected parameter names

- lowercase entity name
- connected parameter names + connector + lowercase entity name

## Test Data Generation

The other part of data preparation is getting the test data right, adequately to the test case at hand. Up until **P3** this was done by preparing test files. For Import and extract tests' types files are necessary, but for those tests there aren't complicated test cases to prepare for, and this simple type of files is prepared by our partners. In order to create complicated scenarios for calculation tests' cases, those simple test files are then modified, Appendix 9.3 shows an example of a test file.

The advantage of files is that our system has checking processes which make sure that the files prepared for testing have valid data. However, the disadvantages are many, and outweigh the advantages by a large margin:

- They are machine readable, as shown in Appendix 9.3.
- They have checks encoded in them, so changing or adding values to them would render them invalid and thus unusable. In order to do so successfully, a tester has to have a working memory on the structure of the file, or refer to the file sender documentation in order to know what else to add, and where else to modify the files to maintain their integrity.
- They contain more data than necessary for a test case, and preparing that extra data is necessary because before reaching the calculation process to be tested, the files go through extraction and check processes which have to succeed, so that the calculation test can have valid data to work with.
- It's not good practice to put test data out of the test wiki page (into separate files), it makes the test less readable.

These factors makes working with files a laborious and extremely time consuming task. For this reason, in **P3** I set out to prove that setting up the database state, that is, by only inserting and manipulating data that's relevant to the process being tested, can be as effective as the usage of files, given that the tester is careful to maintain a valid database state (valid to the process being tested). That goal was accomplished on both fronts (cost, and effectiveness), in comparison between **P1** and **P3**, as demonstrated in Sections 5.2 and 6.1.

After introducing the batch insert feature, this data preparation alternative ought to be much simpler because of the feature of filling in complementary required data with random values when not provided, with the assumption that it's not provided because it's not needed.

The next section will be one extra facility, which combined with what has been discussed so far, aims to:

2. Make the data preparation an even faster process.
3. Eliminate the assumption of the 'tester being careful in maintaining a valid database state'.

4. Eliminate the assumption of the ‘data not being provided because it’s not needed’ to the most possible extent.

### ***Scrambled Data***

The idea here is to create a database that is a big enough sample (a year worth of data), from the live environment database, with the sensitive data censored. Given that the database is fairly normalized (normalized to a practical extent), the sample should remain meaningful, as the sensitive data is used for reporting and user interface purposes, and not needed for calculations.

This procedure of creating a scrambled data test database is a one time job using a simple script. Unfortunately, the script would reveal the internal database structure, so it’s not made public in the context of this thesis. Nevertheless, an example should illustrate the gist of this script.

Assuming that the database structure is as follows:

Table 8: Demonstration database structure

<b>Table</b>	<b>Columns</b>	<b>Foreign keys</b>	<b>Related Tables</b>
MERCHANT_PAYMENT	[PAYMENT_ID] INT NOT NULL IDENTITY [PROCESSING_CODE] VARCAHR(16) NOT NULL [CARD_ACCEPTOR_ID_CODE] VARCAHR(13) NOT NULL [CARD_ACCEPTOR_LOCATION] VARCAHR(50) NOT NULL [AMTREC_DB] VARCAHR(16) NOT NULL [AMTREC_CR] VARCAHR(16) NOT NULL [COUNT_DEBITS] INT [MERCHANT_ID] VARCAHR(15) NOT NULL [CARD_TYPE_ID] VARCAHR(5) NOT NULL [REFERENCE_NO] INT NULL [CURREC] NUMERIC(3) NOT NULL [MESSAGE] VARCHAR (255) NOT NULL [MESSAGE_HASH] VARCHAR (255) NOT NULL	MERCHANT_ID	MERCHANT
MERCHANT	[MERCHANT_ID] VARCAHR(15) NOT NULL [NAME] VARCAHR(50) NOT NULL [EMAIL] VARCAHR(50) NULL [PHONE] VARCAHR(50) NULL [USER_ID] INT NOT NULL [REPRESENTATIVE_ID] NULL	USER_ID REPRESENTATIVE_ID	USER REPRESENTATIVE
USER	[USER_ID] INT NOT NULL IDENTITY [NAME] VARCAHR(50) NOT NULL [REGISTRATION_CODE] VARCAHR(50) NOT NULL [TYPE] VARCAHR(50) NOT NULL	-	-

The algorithm to get the test database would be:

1. Copy a year worth of data from the backup live database into a newly created database.

2. Disable all foreign key constraints temporarily.
3. Identify the sensitive columns. In this example:

Table	Column	Reason
MERCHANT_PAYMENT	CARD_ACCEPTOR_ID_CODE	The real payment terminal id
	CARD_ACCEPTOR_LOCATION	The real address location of that payment terminal. It consists of two parts, the first is another merchant specific code, similar to merchant_id, and the second part is a 3 digit number indicating the number of the terminal, the two parts are separated by a space, example: 87654321 001.
	MERCHANT_ID	The merchant identification code from a private register
	MESSAGE	The original file row(s) from which this database row was extracted. The original message row has all the payment information.
	MESSAGE_HASH	The hash of that message
MERCHANT	NAME	The real merchant name
	EMAIL	The real merchant email
	PHONE	The real merchant phone
USER	NAME	The user name with which the merchant is registered
	REGISTRATION_CODE	Private registration code

4. Censor those sensitive columns

Table	Update
MERCHANT_PAYMENT	<p>CARD_ACCEPTOR_LOCATION, MESSAGE, and MESSAGE_HASH can be censored by generic values, because they are not used in any calculations, but CARD_ACCEPTOR_ID_CODE, and MERCHANT_ID should be carefully reformatted, as they are used in calculations, and as join conditions in queries.</p> <p>MERCHANT_ID: we can use the USER_ID as it's a database identity and isn't in any private register.</p> <p>CARD_ACCEPTOR_ID_CODE: we can replace the first secret code by the USER_ID.</p> <pre> UPDATE MP SET MP.MERCHANT_ID = convert(varchar(15), M.USER_ID), MP.CARD_ACCEPTOR_ID_CODE = concat(convert(varchar(15), M.USER_ID), ' ', RIGHT(MP.CARD_ACCEPTOR_ID_CODE, 3)), MP.CARD_ACCEPTOR_LOCATION = 'TERMINAL_LOCATION', MP.MESSAGE = 'MESSAGE', MP.MESSAGE_HASH = 'MESSAGE_HASH'  FROM [ACQ].[MERCHANT_PAYMENT] MP JOIN [ACQ].[MERCHANT] M ON M.MERCHANT_ID = MP.MERCHANT_ID </pre>
MERCHANT	<p>MERCHANT_ID should be replaced by USER_ID, all remaining column values can be replaced by generic values.</p> <pre> UPDATE [ACQ].[MERCHANT] SET MERCHANT_ID = USER_ID, NAME = concat('MERCHANT_', convert(varchar(15), M.USER_ID)), EMAIL = concat('EMAIL_', convert(varchar(15), M.USER_ID)), PHONE = concat('PHONE_', convert(varchar(15), M.USER_ID)) </pre>

USER	All columns with sensitive data can be replaced by generic values
------	---

##### 5. Enable all foreign key constraints.

With the scrambled database containing valid test data, the previous concerns - about the validity of the database state to a process, and the reliability on randomly generated values to fill the missing data- can be discarded, because the data source for the preparation contains valid data, and the filler values now can be taken from the test database, instead of being randomly generated.

Admittedly this last claim is not currently backed by implementation, but thanks the flexibility of our established grammar. It's not farfetched that it will be implemented by the time this thesis is being assessed.

To justify this claim, let's explore a simple grammar modification that would make it possible to not only generate fixtures (Java code), but wiki tables as well.

Batch insert fixture table

- Separator + Fixture class path + Capitalized camel case entity name + Separator + newline
- Separator + columns to be matched + 'get' + Capitalized camel case entity name + '?'
- values table

Removing the last sub-rule and giving the rule another name:

Batch insert wiki table and fixture

- Separator + Fixture class path + Capitalized camel case entity name + Separator + newline
- Separator + columns to be matched + 'get' + Capitalized camel case entity name + '?'

Means that along with the fixture class, the values for the table should be generated when not given. Figure 17 shows an example of a table without values as an input for the parsing and generation.

ee.lhv.acq.fitness.fixture.MerchantPaymentInsertFixture									
processingCode	cardAcceptorIdCode	amtrecDb	amtrecCr	countDebits	merchantId	cardTypeId	referenceNo	currec	getMerchantPaymentId?

Figure 17. Valueless-table example.

After the fixture class is generated the same way as described before, the column list parsed could be generated into a query that is shown in Figure 18, and then the query is executed and it results in a random sample from the scrambled data database. The result is then concatenated with the original table to result in a valid an executable batch insert table, shown in Figure 19.

```
SELECT TOP 15 CONCAT(' | ', [PROCESSING_CODE], ' | ', [CARD_ACCEPTOR_ID_CODE], ' | ', [AMTREC_DB],
' | ', [AMTREC_CR], ' | ', [COUNT_DEBITS], ' | ', [MERCHANT_ID], ' | ', [CARD_TYPE_ID],
' | ', [REFERENCE_NO], ' | ', [CURREC], ' | ')
FROM scrambled_db.[ACQ].[MERCHANT_PAYMENT] ORDER BY newid()
```

Figure 18. Table values generation query.

ee.lhv.acq.fitness.fixture.MerchantPaymentInsertFixture									
processingCode	cardAcceptorIdCode	amtrecDb	amtrecCr	countDebits	merchantId	cardTypeId	referenceNo	currec	getMerchantPaymentId?
0	8446119 001	D00000000000000000	C00000000000000590	0	8446119	689MD	785	978	\$merchantPaymentId1=
0	8456745 002	D00000000000000000	C00000000000042000	0	8456745	767MA	371	978	\$merchantPaymentId2=
0	8425487 200	D00000000000000000	C00000000000047898	0	8425487	767MC	2241	978	\$merchantPaymentId3=
0	8411834 013	D00000000000000000	C0000000000002440	0	8411834	401VC	92600	978	\$merchantPaymentId4=
0	8434953 010	D00000000000000000	C00000000000005000	0	8434953	401VD	356	978	\$merchantPaymentId5=
0	8458692 231	D00000000000000000	C0000000000008399	0	8458692	M02MC	65	978	\$merchantPaymentId6=
0	8436506 058	D00000000000000000	C0000000000000390	0	8436506	401VD	103	978	\$merchantPaymentId7=
0	8449480 011	D00000000000000000	C00000000000004340	0	8449480	767MC	681	978	\$merchantPaymentId8=
0	8447982 013	D00000000000000000	C0000000000002100	0	8447982	401MC	1684	978	\$merchantPaymentId9=
0	8446593 020	D00000000000000000	C00000000000024005	0	8446593	401MD	511	978	\$merchantPaymentId10=
0	8115540 001	D00000000000000000	C0000000000000583	0	8115540	V02VE	384	978	\$merchantPaymentId11=
0	8422546 120	D00000000000000000	C00000000000030115	0	8422546	767MA	112200	978	\$merchantPaymentId12=
0	8412510 001	D00000000000000000	C00000000000001000	0	8412510	767MD	31	978	\$merchantPaymentId13=
0	8351982 013	D00000000000000000	C0000000000003482	0	8351982	689MD	927	978	\$merchantPaymentId14=
0	8422202 031	D00000000000000000	C0000000000002420	0	8422202	401VE	831	978	\$merchantPaymentId15=

Figure 19. The generated wiki table.

## Files

Before wrapping up, some comments should be made about an attempt to generate files, which was successful to a limited extent only.

Before suggesting the adoption of the database setup model, sometime was spent trying to reverse engineer the files and find formulas to generate them from a set of parameters. The solution was to prepare a template of the most minimalistic version of a file, where it has only one row of data, with parts to be replaced by the given parameters. Then a file specific logic was implemented to make it extendible to contain multiple rows, depending on the input. To explain further, let's go through an example. The test file template:

```
16448000010000010000020000000000000006970400105025102xxxxxx00000015808012
280122001P000000001
```

```
1544A01001000041C00002000008000000000proces160102170753501cardacceptor
3720148004cux037200712402000380017D000000000000010000381017C00000000000000
22500384017C000000000000012500390017amtrecdebitamount0391017amtreccredia-
mount039201500D000000000000000039301500C0000000000000000394017C00000000000011
280395017D00000000000000000000396017C0000000000000000000400010countdebit040101
0countcredi0402010counttotal1014007mer-
chid10160036891017005cardt1018004refn10190060000000cuxcurmsgcount06015808
```

```
1644800001810001C00002000008000000000685686106491083404014800497820165001
B0300025102140630000000015808012280302001A037200712402000374002000375003
POS0378001O0380017D000000000000010000381017C000000000000022500384017C000
00000000000000000390017D0000000000000000000391017amtrecamoun-
tamoun039201800D0000000000000000000039301800C00000000000000000394017C0000000
0000144530395016D00000000000000000000396017C000000000000144530400010000000000
00401010countcount0402010counttotal1017005cardtcuxcurmsgcount06015808
```

```
1644800001810001C00002000008000000000685686106491083404014800497820165001
B0300025102140630000000015808012280302001A037200712402000374002000375003
POS0378001R0380017D000000000000010000381017C000000000000022500384017C000
00000000000000000390017amtrecamoun-
tamoun0391017C00000000000000000000039201800D0000000000000000000039301800C000000
0000000000000394017C000000000000144530395016D00000000000000000000396017C0000000
0000144530400010countcount0401010000000000000402010countto-
tal1017005cardtcuxcurmsgcount06015808
```

164480000100000100000200000000000000006950700105025102xxxxxx00000015808012  
28030101600000000000000000003060080000001600000016

The input applied to this template is a table where each row represents the parameters required to generate the minimum number of rows, which we can call a set of logically related rows. Figure 20 shows an example of input to the facility that uses the aforementioned template.

processingCode	cardAcceptorIdCode	amtrecDb	amtrecCr	countDebits	countCredits	merchantId	cardTypeId	referenceNo	currec	getMpRowId?
000000	87654321 001	D0000000000001010	C0000000000003030	1	2	8280091	401MA	2711	978	\$mpRow1<-[1]
000000	87654321 001	D0000000000001010	C0000000000003030	1	2	8280091	689MB	2712	978	\$mpRow2<-[2]
000000	87654321 001	D0000000000001010	C0000000000003030	1	2	8280091	A00AX	2713	978	\$mpRow3<-[3]
000000	87654321 001	D0000000000001010	C0000000000003030	1	2	8280091	767AX	2714	978	\$mpRow4<-[4]

Figure 20. Input for file generation.

The generated file is too long to include here, so it's put to Appendix 9.4 Generated File Example.

The problems with this approach and why it was discontinued are:

1. There are tens of file types, each with its own specific logic, and it's time consuming to keep implementing extendible file generation facilities, for an increasing number of file types.
2. Some files have to be generated together, because they have complementary file specific logics, and that raises the complexity and cost to an even more unacceptable level.
3. The whole process is counterproductive compared to the batch insert solution: the files are reverse engineered to be generated from a set of parameters, and then extracted so that those parameters are inserted to the database in a way that doesn't compromise its validity.

### Implementation note

All implementation discussed in this proposed solution chapter is available under [3]. The tool is a Java desktop application that relies on SQL Server, it is designed with minimal user interface interaction in mind so that the logic can be moved to a private library once well tested, approved, and refactored to a cleaner codebase.

## 6.4 Validation

Since there hasn't been enough time for testers to get acquainted with the proposed test writing rules, and the parsing and generation tool, there hasn't been enough feedback and data to make conclusions from based on testers' experience. Alternatively this section will go through the steps of the second modification. Measure the time that will take me as a developer to write a test in this manner, then compare the result to the time taken to write old tests of similar type, complexity, done by myself, or by another developer in the past.

Both of the following tests perform the following:

1. Prepare data for the process to run
  - a. Check that the data was prepared correctly (in case of file based test)
2. Execute the process
3. Verify the result

Figure 21 shows the test in the old file based model, with these sections highlighted.

variable defined: date=12.03.2058

variable defined: testFilePath=fitnesse/calciissuermastercardfee/success

script	Journal Job Fixture					
prepare graph for types	CALC_CLAIM_ISSUER_MASTERCARD_FEE					
init file	fitnesse/calciissuermastercardfee/success	with type	IPM_NETS_MC	for date	12.03.2058	
\$journalPeriodId=	run journal jobs with date	12.03.2058				
set journal job status	COMPLETED	by period id	\$journalPeriodId	and type code	IMPORT_FILE_IPM_NETS_MC	
run journal job by period id	\$journalPeriodId	and type code	EXTRACT_FILE_IPM_NETS_MC			
check	get journal job status by period id	\$journalPeriodId	and type code	CALC_CLAIM_ISSUER_MASTERCARD_FEE	COMPLETED	2
delete all files for date	12.03.2058					
\$usisId=	get usis id by issuer id code	M00		1		
\$usacId=	get usac id by member id code	15808				

#### Check data used in calculations

Query: Get reconciliations by journal period id							
CLEA_ROW_NO	SETTLEMENT_INDICATOR	MEMBER_RECONCILIATION_INDICATOR	CURREC	FEEAMTREC_NET	TX_DEST_INST_ID_CODE	IPRE_ID	
9	M	UNDEFINED	978	D000000000000957	15808	\$ipreId1=	
10	M	M00MA	978	D000000000001043	15808	\$ipreId2=	
11	M	M00MA	978	C000000000000033	15808	\$ipreId3=	
12	M	M00MA	840	D0000000000055055	15808	\$ipreId4=	
13	M	M00MA	840	C000000000000055	15808	\$ipreId5=	
14	M	M00MB	978	D000000000000101	15808	\$ipreId6=	
15	M	M00MB	978	C000000000000111	15808	\$ipreId7=	
16	M	M00MC	840	D000000000000202	15808	\$ipreId8=	
17	M	M00MC	840	C000000000000202	15808	\$ipreId9=	
18	M	M01MD	978	D000000000000303	15808	\$ipreId10=	
19	M	M01MR	978	C000000000000404	15808	\$ipreId11=	
20	M	M02MT	840	D000000000000505	15808	\$ipreId12=	

Query: Get issuer mastercard fee view by journal period id						
feeamtrec_net	currency	comma_placement	usac_id	usis_id	ipre_id	
D000000000000957	EUR	2	\$usacId	\$usisId	\$ipreId1	
D000000000001043	EUR	2	\$usacId	\$usisId	\$ipreId2	
C000000000000033	EUR	2	\$usacId	\$usisId	\$ipreId3	
D0000000000055055	USD	2	\$usacId	\$usisId	\$ipreId4	
C000000000000055	USD	2	\$usacId	\$usisId	\$ipreId5	
D000000000000101	EUR	2	\$usacId	\$usisId	\$ipreId6	
C000000000000111	EUR	2	\$usacId	\$usisId	\$ipreId7	
D000000000000202	USD	2	\$usacId	\$usisId	\$ipreId8	
C000000000000202	USD	2	\$usacId	\$usisId	\$ipreId9	
D000000000000303	EUR	2	\$usacId	\$usisId	\$ipreId10	
C000000000000404	EUR	2	\$usacId	\$usisId	\$ipreId11	
D000000000000505	USD	2	\$usacId	\$usisId	\$ipreId12	

#### Check created claim entries

Query: Get claim entries by journal period id and job type code						
DEBIT_CREDIT	AMOUNT	CUR_CODE	USIS_ID	USME_ID	USAC_ID	IPRE_ID
D	18.56	EUR	\$usisId	null	\$usacId	\$ipreId1
D	555.05	USD	\$usisId	null	\$usacId	\$ipreId4

Figure 21. File based test.

Figure 22 shows the new test written in the model of the proposed solution.



#### Prepare the job context

script	Journal Period Fixture				
\$journalPeriodId=	get journal period for date	2058.04.30			
\$mcExtractJojoId=	prepare graph for period	\$journalPeriodId	and job type	EXTRACT_FILE_IPM_NETS_MC	
\$visaExtractJojoId=	prepare graph for period	\$journalPeriodId	and job type	EXTRACT_FILE_IPM_NETS_VISA	
\$mcUsisId=	get usis id by issuer id code	M00			
\$visaUsisId=	get usis id by issuer id code	V00			
\$usacId=	get usac id by member id code	15808			

#### Prepare the job data

ee.lhv.acq.fitness.fixture.IpmChargebackFixture						
amtrecCb	currencyExponent	functionCode	currec	txDestInstIdCode	jojoId	getIpmChargebackId?
9521	84029782	452	978	15808	\$mcExtractJojoId	\$ipmChargebackId1=
45912	98629782	452	978	15808	\$mcExtractJojoId	\$ipmChargebackId2=
9425	84029782	452	978	15808	\$mcExtractJojoId	\$ipmChargebackId3=
39917	84029782	452	978	15808	\$mcExtractJojoId	\$ipmChargebackId4=
34091	84029782	450	978	15808	\$mcExtractJojoId	\$ipmChargebackId5=
45912	98629782	451	978	15808	\$mcExtractJojoId	\$ipmChargebackId6=
32500	8402	450	840	15808	\$mcExtractJojoId	\$ipmChargebackId7=
19912	84029782	255	978	15808	\$visaExtractJojoId	\$ipmChargebackId8=
92782	84029782	255	978	15808	\$visaExtractJojoId	\$ipmChargebackId9=
49997	84028402	451	840	15808	\$visaExtractJojoId	\$ipmChargebackId10=
13566	9782	450	978	15808	\$visaExtractJojoId	\$ipmChargebackId11=
21173	98529782	452	985	15808	\$visaExtractJojoId	\$ipmChargebackId12=
90457	84029782	450	978	15808	\$visaExtractJojoId	\$ipmChargebackId13=
29800	82628402	452	840	15808	\$visaExtractJojoId	\$ipmChargebackId14=
10000	84028402	451	840	15808	\$visaExtractJojoId	\$ipmChargebackId15=

#### Execute chargeback job

script	Journal Period Fixture				
\$execJojoId=	run job by period	\$journalPeriodId	and type	CALC_CLAIM_ISSUER_CHARGEBACK	

#### Check created claim entries

Query: Get claim entries by jojo id \$execJojoId						
DEBIT_CREDIT	AMOUNT	CUR_CODE	USIS_ID	USME_ID	USAC_ID	IPCH_ID
D	94.25	EUR	\$mcUsisId	null	\$usacId	\$ipmChargebackId3
D	459.12	EUR	\$mcUsisId	null	\$usacId	\$ipmChargebackId6
D	325.00	USD	\$mcUsisId	null	\$usacId	\$ipmChargebackId7
D	499.97	USD	\$visaUsisId	null	\$usacId	\$ipmChargebackId10
D	135.66	EUR	\$visaUsisId	null	\$usacId	\$ipmChargebackId11
D	904.57	EUR	\$visaUsisId	null	\$usacId	\$ipmChargebackId13
D	100.00	USD	\$visaUsisId	null	\$usacId	\$ipmChargebackId15

Figure 22. Proposed solution model.

The comparison between these two tests is ideal, because if the first test would be rewritten in the second test's model, or vice versa, they would look almost identical.

Structure-wise, the second test is much more organized and reflects how the real process operates, whereas the first one is less structured and requires extra checks to make sure that the data was inserted correctly, which is testing the data preparation processes in addition to the main process being tested, and that's not a good testing practice.

Timewise, the first test logged five and a half hours. Half an hour was spent on reading the process requirements, or test cases, and thinking of the test values to be used, three hours on preparing the test file, and two hours writing the fixture classes and methods, and debugging or fixing syntax error and such. Those four test writing stages measurements are estimated for the old test, as there's no traceable record of what was done, but the cost of writing the whole test is precise with respect to the method described in Section 4.1.

In contrast, the second test took me about one hour to finish, half an hour was to read the requirements and think of how the test case should look like. Then ten minutes or so was



spent on generating and verifying the insert and query fixtures, then ten minutes getting the test data from the prepared database and placing it in the test page, finally few minutes on fixing my own syntax errors here and there.

After conducting two more experiments of such nature it seems that the benefits theorized in the proposed solution are taking effect. The new model for writing tests proved to be at least three times more cost efficient as the old model. Admittedly, some of that difference is attributed to experience gained from analysing, and writing many tests, but most of it comes from the methodical elimination or remodelling of activities that are archaic and less efficient.

Table 9 shows the three comparisons.

	Old test 1	New test 1	Old test 2	New test 2	Old test 3	New test 3
<b>Cost (h)</b>	5.5	1	43	11.66	13.72	3.5

Table 9: Test generation validation

## **7 Conclusion**

### **7.1 Summary**

The thesis started by a detailed analysis of the testing process data, during which two key periods were measured for a variety of factors, and considered as a baseline for future comparisons. After the data analysis the problems were asserted to be an inefficient testing process that caused time to be wasted in manual testing and retesting, and also caused the requirements' coverage by the regression tests to be deficient. The comparison between the measurements of the two periods determined that the two factors of cost and effectiveness are highly correlated with the testing process structure.

Improving those two factors meant improving the testing process's structure, by moving the test automation as the main testing process, rather than an activity that can be done whenever time is present. A third period was then designed within the framework of this thesis in order to validate this structure improvement by comparing the result to the other two periods. Additionally, this period also validated afore made assertions, and helped quantify the effectiveness of test automation within our project.

The second part of the solution was to introduce automation into the main aspects of the test automation process. The solution then was the combination of these two ideas, of which the first was practically applicable only if the second one was accomplished.

At this point the whole focus of the solution became reliant on somehow automating certain aspects of writing Fitnesse tests. The concept of the Parsing Expression Grammar was found to be an elegant way to add some formal rules into writing Fitnesse instructions so that they can be parsed and generated automatically. The parsing grammar also proved to be flexible and showed promise of how this solution can be extendible for future possibilities of generating both Java code and wiki tables with the addition or adjustment of the grammar rules.

For the time being the two main usages for the implemented grammar and generation logic are the data preparation, and data verification parts of the tests. This was then combined with the idea of getting a sample of data from the live environment database and censoring the sensitive data, so that all aspects of data preparation for the tests would be handled, with very few unchecked assumptions.

The proposed solution of the new test automation model was applied and compared against tests written in the old model of laboriously preparing test files, and the result was positive in favour of the new model.

### **7.2 Goals**

With this, the five goals stated at the beginning of this thesis can be considered achieved. As of now, the upcoming work, is to incrementally use this model throughout future development, and slowly get rid of the old model. Once there's a big enough sample of data to make decisions based on, this model, and implemented tool will be presented to other teams within the company.

The ultimate goal is to integrate the logic developed in the test parsing and generation tool into Fitnesse directly – as it's open source – and keep adding features and new patterns there when appropriate.

## 8 References

- [1] W. E. Wong, J. R. Horgan, S. London ja H. Agrawal, „A study of effective regression testing in practice,“ %1 *Proceedings The Eighth International Symposium on Software Reliability Engineering*, Albuquerque, 1997.
- [2] B. Stroustrup, *Programming Principles and Practice Using C++*, Addison-Wesley Professional, 2008.
- [3] A. Belakehal, „Fixture-gen,“ LHV, 05 04 2017. [Vörgumaterjal]. Available: <https://bitbucket.org/alihk47/fixture-gen>.

## 9 Appendix

### 9.1 Task Names Mapping

Thesis Task Code	Real Task Code
P3T1	ACQ-1283
P3T2	ACQ-1289
P3T3	ACQ-1305
P3T4	ACQ-1308
P3T5	ACQ-1311
P3T6	ACQ-1315
P3T7	ACQ-1324
P3T8	ACQ-1325
P3T9	ACQ-1320
P3T10	ACQ-1326
P3T11	ACQ-1334
P3T12	ACQ-1341
P3T13	ACQ-1362
P3T14	ACQ-1364
P3T15	ACQ-1416
P3T16	ACQ-1412
P3T17	ACQ-1230
P2T1	ACQ-1132
P2T2	ACQ-645
P2T3	ACQ-1083
P2T4	ACQ-1084
P2T5	ACQ-1085
P2T6	ACQ-1086

<b>P2T7</b>	<b>ACQ-1092</b>
<b>P2T8</b>	<b>ACQ-1093</b>
<b>P2T9</b>	<b>ACQ-1142</b>
<b>P2T10</b>	<b>ACQ-377</b>
<b>P2T11</b>	<b>ACQ-1112</b>
<b>P2T12</b>	<b>ACQ-1105</b>
<b>P2T13</b>	<b>ACQ-1106</b>
<b>P2T14</b>	<b>ACQ-1109</b>
<b>P2T15</b>	<b>ACQ-1070</b>
<b>P2T16</b>	<b>ACQ-1173.2</b>
<b>P2T17</b>	<b>ACQ-1237</b>
<b>P2T18</b>	<b>ACQ-1173</b>
<b>P2T19</b>	<b>ACQ-1237.2</b>
<b>P1T1</b>	<b>ACQ-750</b>
<b>P1T2</b>	<b>ACQ-766</b>
<b>P1T3</b>	<b>ACQ-764</b>
<b>P1T4</b>	<b>ACQ-742</b>
<b>P1T5</b>	<b>ACQ-1004</b>
<b>P1T6</b>	<b>ACQ-1013</b>
<b>P1T7</b>	<b>ACQ-1016</b>
<b>P1T8</b>	<b>ACQ-1024</b>
<b>P1T9</b>	<b>ACQ-1062</b>
<b>P1T10</b>	<b>ACQ-1061</b>
<b>P1T11</b>	<b>ACQ-1042</b>
<b>P1T12</b>	<b>ACQ-1036</b>
<b>P1T13</b>	<b>ACQ-1019</b>

<b>P1T14</b>	<b>ACQ-1009</b>
<b>P1T15</b>	<b>ACQ-1043</b>
<b>P1T16</b>	<b>ACQ-112</b>
<b>P1T17</b>	<b>ACQ-913</b>
<b>P1T18</b>	<b>ACQ-914</b>
<b>P1T19</b>	<b>ACQ-915</b>
<b>P1T20</b>	<b>ACQ-911</b>
<b>P1T21</b>	<b>ACQ-907</b>
<b>P1T22</b>	<b>ACQ-979.1</b>
<b>P1T23</b>	<b>ACQ-979.2</b>

The .1 and .2 notation means that during that Fitness update task, two different sets of requirements related to multiple tasks were addressed, and the attributes for those tasks were measured separately.

## 9.2 Full Parsing Grammar

### Assumptions

- '+' means concatenation
- Whitespaces are explicitly specified (not considered characters)
- Digits are considered characters
- A newline will be explicitly specified, and is not considered a whitespace
- For Import fixture table, Batch insert fixture table, Script fixture table, and Query fixture table, the sub-rules are in successive conjunction of each other (first sub-rule, **and then** the second sub-rule and so on), unlike the remaining of the rules, which are in disjunction (first sub-rule **or** second rule, and so on).

### The parsing grammar

#### Requirements

- title 'Requirements'
- Description

#### Purpose

- title 'Purpose'
- Description

#### Description

- newline
- line of test
- confluence link
- Description + newline + Description

#### Fixture table

- import fixture table
- batch insert fixture table (general)
- script fixture table
- comment fixture table
- query fixture table

#### Import fixture table

- import fixture table header + newline
- import fixture table body

#### Batch insert fixture table

- Separator + Fixture class path + Capitalized camel case entity name + Separator + newline
- Separator + columns to be matched + 'get' + Capitalized camel case entity name + '?'
- values table

#### Script fixture table

- script fixture table header + newline
- script fixture table body

#### Comment fixture table

- separator + "comment" + separator + line of text + separator + newline

#### Query fixture table

- query fixture table header + newline
- separator + columns to be matched
- query fixture table body

#### Import fixture table header

- separator + "import" + separator

#### Script fixture table header

- separator + "script" + separator + fixture class name + separator

#### Query fixture table header

- separator + "Query: " + Query fixture class name + separator + separated arguments

#### Separator

- "|"
- white spaces + "|"
- "|" + white spaces
- white spaces + "|" + white spaces

#### Fixture class name

- capitalized entity name

#### Query fixture class name

- Get + database table name + connected parameter names

#### Database Table name

- lowercase entity name

#### Connected parameter names

- connector + lowercase entity name
- connected parameter names + connector + lowercase entity name

#### Connector

- "by" + white spaces
- "and" + white spaces
- "with" + white spaces
- "for" + white spaces
- "from" + white spaces

#### Separated arguments

- variable + separator
- separated arguments + variable + separator

#### Import fixture table body

- separator + full class name + separator + newline
- import fixture table body + separator + full class name + separator + newline

#### Script fixture table body

- lowercase entity name
- lowercase entity name + script arguments

#### Script arguments

- connector + entity name + separator + variable + separator
- connector + entity name + separator + constant + separator
- script arguments + connector + entity name + separator + variable + separator
- script arguments + connector + entity name + separator + constant + separator

#### Columns to be matched

- database column name + separator
- columns to be matched + database column name + separator

#### Query fixture table body

- values table

#### Values table

- separator + values to match + newline
- query fixture table body + separator + values to match + newline

#### Values to match

- variable + separator
- constant + separator
- whitespace + separator
- values to match + variable + separator
- values to match + constant + separator
- values to match + whitespace + separator



#### Capitalized entity name

- capitalized word + whitespace
- capitalized entity name + lowercase word + whitespace

#### Lowercase entity name

- lowercase word + whitespace
- lowercase entity name + lowercase word + whitespace

#### Camel case entity name

- Lowercase word
- Camel case entity name + Capitalized word

#### Capitalized camel case entity name

- Capitalized word
- Capitalized camel case entity name + Capitalized word

#### Lowercase word

- Lowercase character sequence

#### Capitalized word

- uppercase character + lowercase character sequence

#### Lowercase character sequence

- lowercase character
- lowercase character sequence + lowercase character

#### Uppercase character sequence

- uppercase character
- uppercase character sequence + uppercase character

#### Fixture Class path

- lowercase word + '.'
- lowercase word + '.' + Fixture Class path

#### Variable

- \$ + variable name

#### Variable name

- lowercase word
- variable name + Capitalized word

#### Constant

- lowercase character sequence
- uppercase character sequence
- constant + lowercase character sequence
- constant + uppercase character sequence

#### White spaces

- ' '
- '\t'
- white spaces + ' '

- white spaces + '\t'

### 9.3 IPM File Content Example

16448000010000010000020000000000000000697040010502510204045800000015808012  
280122001P00000001  
1544A00001000041C000020000080000000000000050187654321 001  
37201480049782037200712402000380017D00000000000010000381017C000000000000  
22500384017C00000000000012500390017D0000000000001010391017C000000000000  
01010039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C000000000000000040001000000000004010  
100000000002040201000000000021014007828009110160036891017005401MA101800  
4271010190060000009789780000000206015808  
1544A00001000041C000020000080000000000000050187654321 001  
37201480049782037200712402000380017D00000000000010000381017C000000000000  
22500384017C00000000000012500390017D00000000000002020391017C000000000000  
01515039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C000000000000000040001000000000004010  
100000000002040201000000000021014007828009110160036891017005401MB101800  
4271110190060000009789780000000306015808  
1544A00001000041C000020000080000000000000050187654321 001  
37201480049782037200712402000380017D00000000000010000381017C000000000000  
22500384017C00000000000012500390017D00000000000003030391017C000000000000  
02020039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C000000000000000040001000000000004010  
100000000002040201000000000021014007828009110160036891017005767VC1018004  
271210190060000009789780000000406015808  
1544A00001000041C000020000080000000000000050187654321 001  
37201480049782037200712402000380017D00000000000010000381017C000000000000  
22500384017C00000000000012500390017D00000000000004040391017C000000000000  
02525039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C000000000000000040001000000000004010  
100000000002040201000000000021014007828009110160036891017005767VE1018004  
271310190060000009789780000000506015808  
1544A00001000041C000020000080000000000000050187654321 002  
37201480049782037200712402000380017D00000000000003390381017C000000000000  
09940384017C00000000000006550390017D00000000000009090391017C000000000000  
05050039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C00000000000006550400010000000000104010  
100000000002040201000000000031014007828009110160036891017005767AX101800  
4271810190060000009789780000000606015808  
1544A00001000041C000020000080000000000000050187654321 002  
37201480049782037200712402000380017D00000000000003390381017C000000000000  
09940384017C00000000000006550390017D000000000000013130391017C000000000000  
07070039201500D000000000000039301500C0000000000000394017C00000000000000  
000395017D00000000000000000396017C00000000000006550400010000000000104010  
100000000002040201000000000031014007828009110160036891017005793MT101800  
4271910190060000009789780000000706015808  
1644800001810001C00002000008000000000685686106491083404014800497820165001  
B030002510214063000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D00000000000000000381017C000000000000144530384017C000

00000000144530390017D0000000000000000391017C0000000000001010039201800D  
00000000000000000039301800C0000000000000000394017C000000000000144530395016  
D000000000000000000396017C0000000000001445304000100000000000040101000000000  
000402010000000000001017005401MA9789780000000806015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C00000000000000000384017D000  
00000000011050390017D00000000000001010391017C0000000000000000039201800D  
00000000000000000039301800C0000000000000000394017D00000000000011050395016  
D000000000000000000396017D00000000000011050400010000000000004010100000000  
000040201000000000001017005401MA9789780000000906015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D00000000000000000381017C000000000000144530384017C000  
00000000144530390017D00000000000000000391017C0000000000001515039201800D  
00000000000000000039301800C0000000000000000394017C000000000000144530395016  
D000000000000000000396017C000000000000144530400010000000000004010100000000  
000402010000000000001017005401MB9789780000001006015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C00000000000000000384017D000  
00000000011050390017D00000000000002020391017C0000000000000000039201800D  
00000000000000000039301800C0000000000000000394017D00000000000011050395016  
D000000000000000000396017D00000000000011050400010000000000004010100000000  
000402010000000000001017005401MB9789780000001106015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D00000000000000000381017C000000000000144530384017C000  
00000000144530390017D00000000000000000391017C0000000000002020039201800D  
00000000000000000039301800C0000000000000000394017C000000000000144530395016  
D000000000000000000396017C000000000000144530400010000000000004010100000000  
000402010000000000001017005767VC9789780000001206015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C00000000000000000384017D000  
00000000011050390017D00000000000003030391017C0000000000000000039201800D  
00000000000000000039301800C0000000000000000394017D00000000000011050395016  
D000000000000000000396017D00000000000011050400010000000000004010100000000  
000402010000000000001017005767VC9789780000001306015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D00000000000000000381017C000000000000144530384017C000  
00000000144530390017D00000000000000000391017C0000000000002525039201800D  
00000000000000000039301800C0000000000000000394017C000000000000144530395016  
D000000000000000000396017C000000000000144530400010000000000004010100000000  
000402010000000000001017005767VE9789780000001406015808  
1644800001810001C0000200000800000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C00000000000000000384017D000  
00000000011050390017D00000000000004040391017C0000000000000000039201800D

0000000000000000000039301800C0000000000000000394017D000000000000011050395016  
D0000000000000000000396017D000000000000011050400010000000000004010100000000  
0000402010000000000001017005767VE97897800000001506015808  
1644800001810001C00002000008000000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D0000000000000000000381017C0000000000000144530384017C000  
000000000144530390017D0000000000000000000391017C00000000000005050039201800D  
0000000000000000000039301800C0000000000000000000394017C0000000000000144530395016  
D0000000000000000000396017C00000000000001445304000100000000000040101000000000  
0004020100000000000001017005767AX97897800000001606015808  
1644800001810001C00002000008000000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C0000000000000000000384017D000  
00000000011050390017D000000000000009090391017C000000000000000000039201800D  
0000000000000000000039301800C0000000000000000000394017D000000000000011050395016  
D0000000000000000000396017D0000000000000110504000100000000000040101000000000  
0000402010000000000001017005767AX97897800000001706015808  
1644800001810001C00002000008000000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001O0380017D0000000000000000000381017C0000000000000144530384017C000  
000000000144530390017D0000000000000000000391017C00000000000007070039201800D  
0000000000000000000039301800C0000000000000000000394017C0000000000000144530395016  
D0000000000000000000396017C00000000000001445304000100000000000040101000000000  
0004020100000000000001017005793MT97897800000001806015808  
1644800001810001C00002000008000000000685686106491083404014800497820165001  
B0300025102140630000000015808012280302001A037200712402000374002000375003  
POS0378001R0380017D000000000000011050381017C0000000000000000000384017D000  
00000000011050390017D000000000000013130391017C000000000000000000039201800D  
0000000000000000000039301800C0000000000000000000394017D000000000000011050395016  
D0000000000000000000396017D0000000000000110504000100000000000040101000000000  
0000402010000000000001017005793MT97897800000001906015808  
1644800001000001000002000000000000000695070010502510204045800000015808012  
280301016000000000000000000003060080000002000000020

#### 9.4 Generated File Example

1644800001000001000002000000000000000697040010502510222052900000015808012  
280122001P00000001

1544A01001000041C0000200000800000000000000016010217075350187654321 001  
37201480049782037200712402000380017D000000000000010000381017C0000000000000  
22500384017C000000000000012500390017D000000000000010100391017C00000000000  
03030039201500D0000000000000039301500C00000000000000394017C000000000000011  
280395017D0000000000000000000396017C00000000000000000400010000000000104010  
100000000002040201000000000031014007828009110160036891017005401MA101800  
4271110190060000000978978000000020601580880c528fd-d3d0-49e0-8ede-  
49de4bc693ce

1544A01001000041C0000200000800000000000000016010217075350187654321 001  
37201480049782037200712402000380017D000000000000010000381017C0000000000000  
22500384017C000000000000012500390017D0000000000000101010391017C00000000000  
30303039201500D0000000000000039301500C00000000000000394017C000000000000011





- 1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
- 1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis **Test Automation Case Study**, supervised by Dietmar Alfred Paul Kurt Pfahl

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **13.05.2017**