CourJette: a Java implementation of Cucumber blended with QuickCheck

Author:  
Hugo Legrand  
H00256134

Supervisor:  
Dr. Manuel Maarek

Second Reader:  
Dr. Stefano Padilla

A Dissertation submitted in fulfilment of the requirements  
for the degree of MSc. Information Technology (Software Systems)  
in the  
School of Mathematical and Computer Sciences  

August 2017
Declaration of Authorship

I, Hugo Legrand, confirm that this work titled 'CourJette: a Java implementation of Cucumber blended with QuickCheck' and submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date:
“Quality means doing it right even when no one is looking.”

— Henry Ford, founder of the Ford Motor Company
Abstract

One thing challenging in Software Engineering is getting the right idea of the product: this means understanding the client’s requirements, making sure the latter understands the marketing people, who in turn have to understand engineers and vice versa.

With the help of Behaviour-driven development (BDD), communication and apprehension between engineers and marketing people improved. Now, with the aid of simple quasi-natural languages, both can write simple specifications understandable by each other, and easily converted into a skeleton that would become a fully-featured product, in accordance with the specifications.

However, this kind of tools does not offer at the moment any way of generating Property-based testing (PBT) tests, which are effective in Software Engineering, as they offer approaches to combine multiple test cases into one.

This paper aims to provide a solution that will allow Software Engineers, marketing people and even average clients to write not only clear specifications but also powerful Unit tests. In order to do that, I will instantiate this approach with Cucumber, QuickCheck and Java, and demonstrate its usability on a test program.
I would like to thank my supervisor Dr. Manuel MAAREK who has been of a great help in the process of finding a subject, finding a nice name for the final software, during the research, the implementation and the writing of this dissertation.

My family and friends have been a great help and support during this research, I would like to thank them.

I would like to thank Daniel WAGHORN for his evaluation of my project. His efficiency has been remarkable and helped me a lot in finishing this project.

I also thank all other anonymous students and staff from the School of Mathematical and Computer Sciences and the School of Languages & Intercultural Studies of Heriot-Watt University for answering a questionnaire that served me as a base for the project.

I would like to thank the organisers of the LinUX-AD workshops at Heriot-Watt University for their great idea of gathering students in workshops to realize a software, using Agile methods. It inspired me a lot for this project.

Moreover, I would like to thank Heriot-Watt University for letting me do this research project, which has taught me a lot, in various ways.

Finally, I would like to thank you, who are reading this dissertation. Please excuse me if you find any mistake in it, I did my best to proofread this document even if English is not my native language.
Contents

Declaration of Authorship i
Abstract iii
Acknowledgements iv
Contents v
List of Figures vii
List of Tables ix

1 Introduction 1
1.1 Background ............................................. 2
1.1.1 Software Engineering ..................................... 2
1.1.2 The role of requirements ................................... 3
1.1.3 Testing in Software Engineering ......................... 3
1.1.4 Agile development ....................................... 3
1.1.5 Test-driven development .................................. 4
1.2 Motivation .................................................. 5
1.3 Objectives ................................................... 5
1.3.1 Assess the benefits of Behaviour-driven development ...... 5
1.3.2 Integrate QuickCheck into Cucumber seamlessly ........... 5
1.3.3 Generate QuickCheck tests ................................ 5

2 Literature Review 6
2.1 Communication between Domain experts and Software Engineers .... 7
2.1.1 Process Modeling .......................................... 7
2.1.2 Controlled natural language ................................ 10
2.2 Test-driven development ..................................... 11
2.3 Behaviour-driven development ................................ 13
2.3.1 Cucumber and Gherkin ..................................... 13
2.4 Property-based testing ....................................... 16

3 Methodology and Requirements 19
3.1 Requirements ................................................. 19
3.2 Methodology .................................................. 22
3.2.1 Implementation of the solution ............................ 22
3.2.2 Evaluation ................................................ 22
## Contents

### 4 Design

4.1 Tools and technologies ........................................... 23
4.2 Code analysis of the existing software .......................... 24
  4.2.1 Analysis of the gherkin-java project ......................... 24
  4.2.2 Analysis of the cucumber-jvm project ....................... 27
4.3 First design of CourJette ........................................... 30

### 5 Prototyping, redesign and implementation

5.1 First modifications .................................................. 32
5.2 Enhancements .......................................................... 36
  5.2.1 Adding more patterns ........................................... 36
  5.2.2 Generate QuickCheck Generators .............................. 37

### 6 Evaluation

6.1 Output of CourJette ................................................ 45
6.2 External evaluation of CourJette ................................ 47
6.3 Strengths and weaknesses of CourJette .......................... 49
6.4 Future work ........................................................... 49
6.5 Conclusion ............................................................ 50

### References

52

### List of terms and acronyms

55
List of Figures

2.1 Modeling languages history ........................................... 7
2.2 Simple UML diagram .................................................. 8
2.3 Code generated based on previous UML diagram ................ 9
2.4 Simple BPMN diagram ................................................ 10
2.5 Example of the ACE language ....................................... 11
2.6 The test pyramid ....................................................... 12
2.7 The test pyramid - Unbalanced ..................................... 12
2.8 How Cucumber works ................................................ 14
2.9 Gherkin syntax example for a simple Feature ................. 14
2.10 Step definition generated for the Given Step of the “banana” example .......................... 14
2.11 User Story example for the Feature written in Figure 2.9 .... 15
2.12 The test iceberg ....................................................... 15
2.13 Example of placeholders in Gherkin .............................. 16
2.14 A simple QuickCheck test, using the @InRange annotation .. 17
2.15 Output of the test in Figure 2.14 .................................. 18
2.16 Template to create a QuickCheck Generator .................... 18
2.17 Using a complex QuickCheck Generator .......................... 18
2.18 Using a simple QuickCheck Generator ............................ 18

3.1 Example of the keyword “any” that could be used ................ 19

4.1 Cucumber Regular Expression extracted from the original Gherkin .... 24
4.2 Pickle class diagram .................................................. 25
4.3 Output of the Main method of the gherkin-java project ...... 26
4.4 Output of the GenerateTokens method of the gherkin-java project .... 27
4.5 Output of the Main method of the cucumber-jvm project ....... 28
4.6 my-example.feature Gherkin file used to test the software .... 29
4.7 Part of the argumentTypes method of the SnippetGenerator class .... 29
4.8 ArgumentPattern UML class diagram | Values of ArgumentPattern for the When I have 1 scenario Step ........................................ 30
4.9 Static array returned by argumentPatterns() function ........ 30
4.10 Options given on the questionnaire to find the more usable generalization of BDD .............................................. 31

5.1 Modified DEFAULT_ARGUMENT_PATTERNS array ............ 32
5.2 getSnippet() function generating the Java snippets .......... 33
5.3 Default template used for each Step ................................ 33
5.4 Documentation of the template() method of the Snippet interface ... 34
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>First attempt at <strong>QuickCheckSnippet</strong>’s template</td>
<td>34</td>
</tr>
<tr>
<td>5.6</td>
<td>First attempt at modifying the <code>getSnippet()</code> function</td>
<td>35</td>
</tr>
<tr>
<td>5.7</td>
<td>Output of the first attempt for CourJette</td>
<td>35</td>
</tr>
<tr>
<td>5.8</td>
<td>Gherkin example on adding more patterns</td>
<td>36</td>
</tr>
<tr>
<td>5.9</td>
<td>Including two more patterns to be recognized</td>
<td>37</td>
</tr>
<tr>
<td>5.10</td>
<td>Second attempt at <strong>QuickCheckSnippet</strong>’s template</td>
<td>37</td>
</tr>
<tr>
<td>5.11</td>
<td>List of core Generators provided by <code>junit-quickcheck Source: junit-quickcheck documentation and Oracle Java documentation</code></td>
<td>38</td>
</tr>
<tr>
<td>5.12</td>
<td><code>QUICKCHECK_GENERATOR_PATTERNS</code> array</td>
<td>39</td>
</tr>
<tr>
<td>5.13</td>
<td>Searching for QuickCheck patterns</td>
<td>40</td>
</tr>
<tr>
<td>5.14</td>
<td>Constructing the succession of parameters for the final snippet</td>
<td>41</td>
</tr>
<tr>
<td>5.15</td>
<td>Third attempt at <strong>QuickCheckSnippet</strong>’s template</td>
<td>42</td>
</tr>
<tr>
<td>5.16</td>
<td>Second attempt at modifying the <code>getSnippet()</code> function</td>
<td>42</td>
</tr>
<tr>
<td>5.17</td>
<td><code>isQuickCheckStep()</code> function determining the use of the QuickCheck template</td>
<td>43</td>
</tr>
<tr>
<td>5.18</td>
<td><code>argumentTypes()</code> function gathering the future arguments of the final snippet</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Complex Gherkin example to evaluate CourJette</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Snippets generated by CourJette</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Libraries to import for QuickCheck to run</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>Risk Assessment Form of the project</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Requirements .................................................. 21
3.2 Legend for Table 3.1 .......................................... 21
Chapter 1

Introduction

Developing a project nowadays cannot be done without testing at least the final product. It has been more and more important since the last two decades and the apparition of the Agile approach (Scrum, 1996), which has brought developers, managers and customers closer in the process of implementing of a solution. Agile methods are more pragmatic than traditional methods, involve as much as possible the applicant (i.e. the client) and allow a great reactivity to his requests. They are based on an iterative, incremental and adaptive cycle of development.

Test-driven development (TDD) is a software development technique that advocates writing unit tests before writing the source code of a software, and is particularly used with Agile methods. In 2003, a new software process has appeared in response to TDD. Behaviour-driven development (BDD) is a process used between Agile teams and customers, to replace TDD. BDD is more of a business approach, thanks to its simplicity, and allows to gather all the tests from the specifications given by the customer. Simple explanations to the customer are sufficient to let him specify his requirements, using natural language.

However, although BDD allows more interaction between executives and developers, it is still a challenge for developers to test the product, and it gets harder if they do not have an overview of the project wide enough. Some tools allow developers to simplify testing inputs, such as QuickCheck.

In this dissertation, I will explain the main differences between the two types of software process I have mentioned (BDD and TDD), what are their strengths along with their weaknesses and how it could be enhanced. The next chapter will discuss existing work regarding these subjects, it is the Literature Review. Next, the Methodology and Requirements will explain how far I will go in this project within the allocated time, and
discuss the materials needed in order to achieve the objectives. After that, the Design chapter will explain the first thoughts on the project and how the CourJette tool was first composed. Next, the Prototyping, redesign and implementation chapter will analyse the characteristics of the first prototype and clarify the final design of the product. It will also describe the multiple attempts of carrying out this project and the final solution adopted. Finally, an Evaluation is conducted in order to assess the project, show its strengths and weaknesses, propose future work and conclude this dissertation.

1.1 Background

This section aims to explain a bit more the technical terms cited in the above introduction. I will discuss Software Engineering, the role of requirements, the tests, Agile development and TDD. I will go into more details of these topics in the Literature Review chapter.

1.1.1 Software Engineering

Software engineering is an industrial engineering science that studies the working methods and best practices of engineers who develop software. In particular, software engineering focuses on systematic procedures to ensure that large software meets customer expectations, is reliable, has a low maintenance cost and performs well while meeting deadlines and construction costs.

Multiple development methods and practices exist to perform the various software development works: cascading, iterative, sprinting or parallel. Cascade is the classical method of engineering. It consists in successively performing, in cascade, functional analysis, then design, programming and testing. Iterative is another method that aims to perform the analysis, programming, testing and validation work first on a restricted set of software functions, and then a new iteration will be used to repeat these operations on a more refined set of functions, and so on, in a spiral cycle. Agile refers to various development processes in breach of the conventional engineering processes. These processes focus on constantly changing specifications and software source code, close collaboration and strong end-user involvement, and a spiral development cycle with many short iterations. Scrum, Extreme programming (XP) and Rational Unified Process are Agile methods.
1.1.2 The role of requirements

In engineering, particularly in public and private tendering procedures, requirements are the expression of a documented need for what a particular product or service should be or should be doing. They are most often used in a formal sense in software engineering. In the conventional approach to engineering, requirements are considered as prerequisites for the design and development stages of a product.

Requirements are essential in building a good software because it forces the client to formalize his thoughts and wishes concerning the product, and allow the developers to build the exact software matching the requirements. Hence, requirements are often the cause of misunderstandings between the two parties.

1.1.3 Testing in Software Engineering

A test designates a procedure for partial verification of a system. Its main objective is to identify a maximum number of problem behaviours of the software in order to increase the quality. Nevertheless, the test may also aim to provide information about this quality in order to allow decision making.

The purpose of verification or validation tests is to verify that this system reacts as intended by its developers (specifications) or conforms to customer requirements, respectively.

A test is like a scientific experiment. It examines a hypothesis expressed in terms of three elements: the input data, the object to be tested and the expected observations. This examination is carried out under controlled conditions in order to draw conclusions.

1.1.4 Agile development

Agile methods must respect four fundamental values declined into twelve principles from which flow a basis of practices, either common or complementary.

The four fundamental values are:

- Individuals and interactions rather than processes and tools
- Operational features rather than exhaustive documentation
- Working with the client rather than contracting relationships
• Acceptance of change rather than compliance with plans

The twelve general principles are:

• Satisfy the customer first
• Welcoming requests for change
• Deliver operational versions of the application as often as possible
• Ensure ongoing cooperation between the client and the project team
• Building projects around motivated individuals
• Focus on face-to-face conversation
• Measure project progress in terms of application functionality
• Advancing the project at a sustainable and sustainable pace
• Continuous attention to technical excellence and design
• Make simple
• Empowering teams
• Regularly adjust its behaviour and processes to be more efficient

1.1.5 Test-driven development

Using TDD allows to make it possible to specify the specifications of the code, and therefore its behaviour according to the situations to which it will be exposed. This facilitates the production of a valid code in all circumstances. This gives a more accurate and reliable code. By writing the tests first, one uses the program before it even exists. This ensures that the product code can be tested individually. It is therefore imperative to have a precise vision of how one will use the program before to even consider its implementation. This often avoids design errors due to precipitation in the implementation before defining the objectives.

A more detailed explanation of TDD is written on the Literature Review chapter, section Test-driven development.
1.2 Motivation

Comprehension between expert practitioners (developers) and non-expert practitioners (clients) is still a problem. Requirements have to be clearly defined in order for a project to be successful. The comprehension and good implementation of these requirements is sometimes difficult because of their ambiguity.

Cucumber helps reducing this gap between expert and non-expert practitioners but still do not include other testing methods. Using Unit test to test a software is useful, and QuickCheck propose an attractive solution for simplifying and empowering these tests. Hence, combining Cucumber and QuickCheck makes the new software powerful and reduces the possible errors in projects that shall use it.

1.3 Objectives

1.3.1 Assess the benefits of Behaviour-driven development

BDD is still quite young hence it is not very known. Still, it is interesting to wonder whether this software process is effective during the development of a product. Therefore, I will try to assess the benefits of one BDD tool: Cucumber.

1.3.2 Integrate QuickCheck into Cucumber seamlessly

The main objective of the project would be to extend the actual possibilities of BDD by integrating a simple method, that would allow it to create test cases easily on top of the structure already defined by the BDD tool Cucumber. This would be done by extending the Gherkin syntax already used by Cucumber.

1.3.3 Generate QuickCheck tests

An objective is to allow the user to generate QuickCheck tests with the same Gherkin file, without generating the whole structure with Cucumber. The idea is to use Gherkin to generate Property-based testing (PBT) tests, without using Cucumber.
Chapter 2

Literature Review

Testing in software development is mandatory in order to propose a viable product (Beck, 2003). Business people often used to be reluctant to do testing, blaming the time it usually takes, thus the resources put into it, for a result that they did not find noteworthy (Beck, 2003).

In the beginning of the century, this behaviour has changed and business people along with engineers agree on the efficiency of testing by using Test-driven development (TDD) (see section 2.2) (Aniche and Gerosa, 2015). However, executives do still have a hard time understanding this branch of software development, and therefore cannot participate with their teams in the production of relevant testing code (Nelson et al., 2015).

Lately, a new software process arose: Behaviour-driven development (BDD) (see section 2.3). It is stated by Rose et al. (2015) that it suppresses the barrier between executives and software developers by letting one define its requirements using a Controlled natural language (CNL). Software developers can therefore implement code around these requirements, keeping the product free from misconceptions.

However, while BDD integrates Unit tests, it is still lacking of a better method to efficiently check properties of a software (see section 2.4). Hence it is a significant idea to try to reunite BDD and Property-based testing (PBT) in one tool, yet increasing the possibilities of executives of a software project and simplifying the work of software engineers at the same time.
Chapter 2. Literature Review

2.1 Communication between Domain experts and Software Engineers

Practitioners and Software Engineers often do not think the same way. Hence it is difficult for them to understand each other. These divergences become decisive when developing a software project. This section will go through some attempts to reduce the issues created by these Human aspects of Software Engineering.

2.1.1 Process Modeling

Modeling languages helped teams to work together on software projects since the early 1990s, as stated by Figure 2.1.

For 20 years now, Unified Modeling Language (UML) has evolved and earned its place in every modern companies. Since 2008, another method of modeling has emerged: Business Process Model and Notation (BPMN) (Birkmeier et al., 2010). Few years before, the UML Activity Diagram was deemed as being too technically oriented (White, 2004). However, it is still significantly more effective in data handling and adequacy than BPMN (Birkmeier et al., 2010).
UML is used to specify, view, modify and construct the documents necessary for the good development of an object-oriented software (Ageriou et al., 2005). UML provides a modeling standard to represent the software architecture. Some of the different elements that can be represented are, according to Object Management Group (OMG) (2011):

- Activity of an object/software
- Actors
- Processes
- Database Schema
- Software Components

Thanks to the UML modeling tools, it is also possible to automatically generate all or part of the code of a software application, for example in Java language, from the various documents produced (Harrison et al., 2000). For instance, the following UML class diagram (see Figure 2.2) is quite simple and can be understood overall by practitioners.

![Figure 2.2: Simple UML diagram](Source: IdioTechie)

The following code (see Figure 2.3) is part of what is automatically generated from the previous UML example (Figure 2.2). It translates the UML diagram into effective code (here in Java), usable by Software Engineers as a base for developing the
Chapter 2. Literature Review

product, and maintain consistency between a model (Figure 2.2) and its implementation (Figure 2.3) (Harrison et al., 2000).

```java
public class Car {
    private String model;
    public void printPrice() {
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}

public class hatchback extends Car {
    private String model;
    public void printPrice() {
        System.out.println("Hatchback Price");
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
```

Figure 2.3: Code generated based on previous UML diagram

Source: IdioTechie.com

The main goal of BPMN is to provide a notation that is easily understood by all users of the company (Birkmeier et al., 2010). This includes domain experts, business analysts who create the initial draft processes, and the developers responsible for implementing the technology that will execute the Processes. Ultimately, the users of the company that will implement these processes will also be able to understand this notation (Recker, 2008).

Below is a simple BPMN diagram that expresses the multiple steps in the process of satisfying the hunger of a user (see Figure 2.4). Even though a BPMN diagram is similar to an UML Activity Diagram (Birkmeier et al., 2010), it is simpler to understand thanks to the familiar and easy basic graphical notation that makes it easier to learn than UML (Wahl and Sindre, 2006).

However, the use of BPMN within other domains is limited (e.g. the object-oriented domain), but it can be used to model general processes also outside the business domain (Wahl and Sindre, 2006).

Therefore, UML and BPMN both aim at reducing, or even breaking, the barrier
between one or several departments in a modern company. This is appropriate to know that there are already multiple attempts to solve this issue, and that the related works will help to enhance this communication, which is essential for a maximum productivity.

### 2.1.2 Controlled natural language

Sometimes called processable, simplified, technical, structured or basic natural languages, CNL refer to a wide variety of languages emerged from 1930s until today (Kuhn, 2013). CNL can be very different one from another, some are ambiguous, other formal, sometimes they look like natural language, sometimes they don’t. A simple definition may be:

> “A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties.”

— (Kuhn, 2013, p. 123)

CNL have been suggested in order to surmount the problem that common formal languages are often hard to understand for people unfamiliar with formal notations (Kuhn, 2009).

Traditionally, CNL have been grouped into two categories: human-oriented CNL and machine-oriented CNL (Schwitter, 2010). Human-oriented CNL aim to improve readability and comprehensibility of technical documentation while machine-oriented CNL aim to improve the translatability of technical documents (Schwitter, 2010).

Since this project aims to enhance the translatability of a technical document, I will illustrate here a machine-oriented CNL: Attemto Controlled English (ACE). It is a CNL based on a subset of English and is used for automated reasoning (Schwitter,
ACE is defined by a small set of construction rules (syntax) and another set of interpretation rules that allows the language to suppress any ambiguity inherited from the “full English” (Fuchs et al., 2008). The vocabulary of ACE comprises the following items (see Figure 2.5).

- Predefined function words (e.g. determiners, conjunctions)
- Predefined phrases (e.g. “it is false that ...”, “it is possible that ...”)
- Content words (e.g. nouns, verbs, adjectives, adverbs)

A simple example of an ACE text with a query may be:

1. Every company that buys at least three machines gets a discount.
2. Six Swiss companies each buy one machine.
3. A German company buys four machines.
4. Who gets a discount?

Figure 2.5: Example of the ACE language
Source: (Schwitter, 2010, p. 1115)

In section 2.3 we will talk about Gherkin, another CNL which is highly customizable, easy-to-use, and of course looks like a natural language.

2.2 Test-driven development

TDD is not a testing technique, it is a software design technique (Nelson et al., 2015). It is used a lot in Agile development, and is composed of three cycles:

- **Red phase** - Write tests until one fails
- **Green phase** - Write production code until tests pass
- (optional) **Blue phase** - Refactor the code to clean up, improve design

To make it simple, the philosophy of TDD is “Clean code that works” (Beck, 2003), because of the maintainability, the teamwork, and the easy while writing it. In order to do that, one must write code only if an automated test failed: it means that the project starts with the requirements, then tests cases are written and only after that the development of the main features may begin. Moreover, TDD tries to eliminate duplication of code. For this approach to work, each programmer must write its own
tests (for efficiency), and every small change in the code must be rapidly taken into account by the development environment.

Lately, methods have emerged to test more efficiently using TDD. For example, one must follow the test pyramid principles (see Figure 2.6): only implement the layers you need, and test them at the lowest level possible.

![Figure 2.6: The test pyramid](source: Morris, 2016, p. 198)

In order to achieve the maximum efficiency, one must continuously review the testing effectiveness (Morris, 2016). On low-level testing, Unit test allows the user to check for intended results. Combined with Syntax checking and Static analysis, nowadays fully provided by IDE, the programmer's productivity has climbed and bad habits and common errors remain rare. On mid-level testing, the developer has to prove that the individual components of the software work when integrated together. On high-level testing, the programmer has to prove that the multiple elements of the infrastructure work properly together.

But to be useful, the tests must respect the pyramid shown on Figure 2.6, which is sometimes complicated for some developers who tend to write a lot of high-level tests and omit the low-level tests. This situation is depicted in Figure 2.7.

![Figure 2.7: The test pyramid - Unbalanced](source: Morris, 2016, p. 199)

However, one can ask if using TDD to design a software really enhance the overall quality. According to Aniche and Gerosa (2015), most studies on TDD focus
only on external quality (e.g., bug proneness) and never try to answer the question of class design. After conducting a qualitative study with professional developers, they found out that the programmers did produce a high quality code using TDD, thanks to the repetitive question “is your test easy to be tested or not?” that the developers ask themselves a lot when using this technique.

2.3 Behaviour-driven development

According to Rose et al. (2015), when Software Engineers write TDD acceptance tests, it is usually from the outside-in, by trying to fix a failing behaviour from the customer’s point of view, while BDD acceptance test are readable and understandable by everyone on the team, using examples. It means that before starting to develop anything, the product should already be acceptable. In fact, large parts of code are regularly being thrown away because of some misunderstanding of specifications in the first place. BDD tries to solve this problem (Rose et al., 2015).

BDD works very well with Agile development. To make sure everything behaves as it should be, meetings with the customer allow to get ideas in order and write efficient acceptance tests. Also, in a lot of cases it allows the project to deliver great results quickly and efficiently (Pham et al., 2013). As stated by Diepenbeck et al. (2014), BDD extends TDD by using ‘natural language written user stories’.

2.3.1 Cucumber and Gherkin

The “natural language written user stories” in Cucumber are also called Scenarios. Each Scenario is made of Sentences, which in turn are made of Steps. The file that holds the Scenario is a Feature file. A Feature is composed of one or multiple Scenarios. This is summarized on Figure 2.8.

An example of the syntax used by Cucumber is shown on Figure 2.9. This syntax is called Gherkin, “a Business Readable, Domain Specific Language that lets you describe software’s behaviour without detailing how that behaviour is implemented”

When a Feature file is created and tested with Cucumber, Step definitions are called. They consist of a keyword, a regular expression, and a test code (see Figure 2.10). The user has to implement them in order for Cucumber to validate the tests. Usually, their test codes are composed of JUnit assertions. Furthermore, Gherkin syntax has been

\[1\]https://github.com/cucumber/cucumber/wiki/Gherkin, accessed: 20/02/2017
translated into more than 40 different languages, in order for teams to really concentrate on features, and not trying to translate. Gherkin also aims to help stimulate working teams by being specific and thinking of concrete examples while writing a Scenario. The following code in Figure 2.10 is actually generated automatically by Cucumber.

```java
@Given("the price of a \[\(\text{"banana"}\)\] is \(\text{40c}\)")
public void thePriceOfALsbC(String arg0, int arg1) throws Throwable {
    // Write code here that turns the phrase above
    // into concrete actions
    throw new PendingException();
}
```

According to Colombo et al. (2014), by establishing some convention rules, it is possible to allow preconditions, actions and postconditions on respectively each of the Given-When-Then element of the typical Gherkin Scenario. It is in fact similar to the User
Stories already widely used in Software development (Landhäußer and Genaid, 2012), since it is written in the same way (see Figure 2.11).

1. As a cashier, I want to get the price of a certain number of bananas
2. * The price of a banana is 40c.
3. * Bananas are sold by unit

Figure 2.11: User Story example for the Feature written in Figure 2.9

The executives are deciding if a behaviour should be in the Feature files or not, and the only parameter here is if they are interested in it. Therefore, it leads to the Figure 2.12 below: the test iceberg. Everything above the “water” is what is business readable. Below the “water” are tests only readable by software people. Obviously, some tests in each category (i.e. end-to-end test; tests that verify integrated components of subsystems; tests that verify components in isolation) are readable by the two parties, but End-to-end tests will be much more readable for executives than Unit test. It is important to make a parallel with the test pyramid (Figure 2.6) seen in section 2.2. The “Technical” part of the iceberg is actually referring to the “low-level tests” of the test pyramid, while “Business-readable” tests are the “high-level tests”. The part in between can be understood as the “medium-level tests”.

![Business-readable](image)

Figure 2.12: The test iceberg
Source: (Rose et al., 2015, p. 250)

An interesting feature in the Gherkin language would be the use of placeholders. In the following Figure 2.13, some words are between angle brackets and serve as a placeholder. In fact, the user may provide an example table (see line 7-9) to allow Cucumber to test these entries while running the corresponding Step definitions.
Scenario Outline: Eating

Given there are <start> cucumbers
When I eat <eat> cucumbers
Then I should have <left> cucumbers

Examples:

<table>
<thead>
<tr>
<th>start</th>
<th>eat</th>
<th>left</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 2.13: Example of placeholders in Gherkin
Source: Behat.org

2.4 Property-based testing

Tests in software engineering aims to validate semantic properties of a program’s behaviour (Fink and Bishop, 1997). A simple situation depicting the need of tests is shown below.

“
Me to co-worker: “We need a function that will add two numbers together, would you mind implementing it?”

(a short time later)
Co-worker: “I just finished implementing the ‘add’ function”
Me: “Great, have you written unit tests for it?”
Co-worker: “You want tests as well? Ok.”

(a short time later)
Co-worker: “I just wrote a test. Look! ‘Given 1 + 2, I expect output is 3’.”
Co-worker: “So can we call it done now?”
Me: “Well that’s only one test. How do you know that it doesn’t fail for other inputs?”
Co-worker: “Ok, let me do another one.”

(a short time later)
Co-worker: “I just wrote another awesome test. ‘Given 2 + 2, I expect output is 4’!”
Me: “Yes, but you’re still only testing for special cases. How do you know that it doesn’t fail for other inputs you haven’t thought of?”
Co-worker: “You want even more tests?”

— scottw (slightly modified)

https://fsharpforfunandprofit.com/posts/property-based-testing/, accessed: 08/03/2017
Following this scenario, we can wonder “How many test are enough?”. This question is answered in part by PBT. As we discussed before in Test-driven development, one must validate her program by running tests at the lowest level possible first. In order to validate higher-level properties, properties must be formalized. PBT answers this issue by requiring one or more properties to run the testing process.

Moreover, PBT encourages the programmer to think harder about the code. One implementation of PBT is QuickCheck (Claessen and Hughes, 2011). Originally developed for programs written in Haskell, QuickCheck is now available for multiple languages, such as Java in our case\(^3\), named junit-quickcheck.

PBT is a generative testing method. The developer has to specify one or multiple properties of the method he wants to test. The PBT framework, here QuickCheck, will then generate random values, respecting the rules set by the developer, and checking for each one if the property is still correct. QuickCheck will in fact bombard the method with a great number of different inputs to find whether a weakness exists. When an input causes an error, QuickCheck will specify it to the developer and will attempt to shrink the set of values until it finds the smallest set that disprove the property.

---

```java
@Property(trials=5)
public void inInRange(@InRange(minInt = 0, maxInt = 100) int num) {
    System.out.println("InRange: " + num);
    assertTrue(num>0);
}
```

---

Figure 2.14: A simple QuickCheck test, using the @InRange annotation

A simple example of a QuickCheck property is shown on Figure 2.14. The @Property annotation notifies QuickCheck of the methods to check. This annotation is followed by a parameter, trials, that allows the user to tell QuickCheck how much random inputs it should use (by default, the number of inputs generated is a hundred). Then, the @InRange annotation allows the user to define a range for QuickCheck to choose inputs from. Finally, the assertTrue method lets the user define the assertion that will validate this test case. The output printed by this method is shown on Figure 2.15.

Another interesting feature of QuickCheck is the ability to choose among lots of Generators. A Generator is a class that will generate random values of a certain type. For example, the IntegerGenerator class will generate integers, while ArrayListGenerator will help generate ArrayLists. Every core types of Java are supported, along with some others (junit-quickcheck documentation). Of course, it is also possible to create our own

\(^3\)https://github.com/pholser/junit-quickcheck, accessed: 27/02/2017
Figure 2.15: Output of the test in Figure 2.14

Generators, using the following basic template (see Figure 2.16). Using Generators in `junit-quickcheck` may be quite verbose sometimes, especially when trying to generate complex values such as generic types. For example, considering a list of integers, the implementation of the `junit-quickcheck` is shown on Figure 2.17. While using a simple type such as an integer is more simple (Figure 2.18).

```java
import java.awt.Dimension;

public class Dimensions extends Generator<Dimension> {
    public Dimensions() {
        super(Dimension.class);
    }

    @Override public Dimension generate(
        SourceOfRandomness r,
        GenerationStatus status) {
        // ...
    }
}
```

Figure 2.16: Template to create a QuickCheck Generator

Source: junit-quickcheck documentation

```java
@Property
public void list_of_int_example(
    @From(ArrayListGenerator.class) ArrayList<Integer> list)
    // ...
}
```

Figure 2.17: Using a complex QuickCheck Generator

```java
@Property
public void int_example(
    @From(IntegerGenerator.class) Integer integer)
    // ...
}
```

Figure 2.18: Using a simple QuickCheck Generator
Chapter 3

Methodology and Requirements

3.1 Requirements

Regarding the multiples objectives of the project (see chapter Introduction, section Objectives), it is possible to define some requirements for the project. For example, regarding the blending of QuickCheck into Cucumber, I will certainly have to fork the existing Java project of Cucumber. By modifying its behaviour, I will be able to add my own keywords to the Gherkin language (see FR-L-1 in Table 3.1). This will allow me to interpret what the user wants to do and generate QuickCheck tests. This may be done in various ways, as shown in Figure 3.1.

The aim of this project is to tighten the gap between software engineers and business people, by enhancing their mutual comprehension.

The main objective for this project is to integrate QuickCheck into Cucumber seamlessly.

```
Feature: Checkout
  Scenario: Checkout any Fruit
    Given the price of any Fruit is 40c
    When I checkout 1 Fruit
    Then the total price should be 40c
```

Figure 3.1: Example of the keyword “any” that could be used

In the above Figure 3.1, let a Fruit class, thanks to the any keyword a binding with QuickCheck would allow the user to generate a Property-based testing (PBT) tests

1 https://github.com/cucumber/cucumber-jvm, accessed: 25/02/2017
using his own `Fruit` class. This is only one example of what is possible with the extended keywords.

Next, regarding the generation of PBT tests using Gherkin language (see FR-G-3 in Table 3.1), I will have to divert the original function of the Cucumber tool to allow only the generation of the PBT tests without generating or altering existing Java code.

In order to assess the QuickCheck possibilities, I will have to do some research before beginning the implementation of the solution (see FR-Q-4 in Table 3.1) in order to assess what is possible and what is not with `junit-quickcheck`.

Regarding the IDE support, I will try to add some support to the solution I will be implementing. This is to facilitate the use of the tool inside an IDE, for example JetBrains’ IntelliJ IDEA\(^2\). As stated in the Table 3.1, this requirement has a low priority since it is not directly related to the core of the project.

Finally, while it is stated in the Table 3.1 that the NFR-P-6 requirement is related to the objective 1.3.1, I will of course follow the feedback I will get from the multiple discussions with my supervisor, the participants of the workshop, and other actors of the project.

Below, in Table 3.1, all the requirements for the project are listed. Some are more important than other, and all relate to an objective explained in Chapter 1. It uses the best practice nowadays, following the RFC 2119 (Bradner, 1997). Requirements using ‘shall’ will be mandatory, while the ones using ‘should’ are secondary. Requirements with ID beginning by FR are Functional Requirements, while the ones beginning by NFR are Non-Functional Requirements. Then, a letter specifying the category of the requirement is added to the ID, followed by a unique integer. This notation for the requirements will allow me to quickly make a reference to any requirement listed below, and still be able to understand which one it is. This is summarized in Table 3.2.

\(^2\)https://www.jetbrains.com/idea/, accessed: 3/04/2017
### Table 3.1: Requirements

<table>
<thead>
<tr>
<th>Req. ID</th>
<th>Title</th>
<th>Description</th>
<th>Priority</th>
<th>Category</th>
<th>Rel. Obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR-L-1</td>
<td>Extend <strong>Gherkin</strong> language</td>
<td>The <strong>Gherkin</strong> language shall be extended. New keywords shall be added.</td>
<td>High</td>
<td>Language <strong>(Gherkin)</strong></td>
<td>1.3.2</td>
</tr>
<tr>
<td>FR-L-2</td>
<td>Generate <strong>QuickCheck</strong> annotations</td>
<td>Using the new keywords, CourJette shall recognize patterns and generate the corresponding annotations.</td>
<td>High</td>
<td>Language <strong>(Gherkin)</strong></td>
<td>1.3.2</td>
</tr>
<tr>
<td>FR-G-3</td>
<td>Generate PBT tests using <strong>Gherkin</strong> language</td>
<td>The <strong>Cucumber</strong> tool should be able to generate only PBT tests from a <strong>Scenario</strong> written in <strong>Gherkin</strong>.</td>
<td>Low</td>
<td>Generator <strong>(Cucumber)</strong></td>
<td>1.3.3</td>
</tr>
<tr>
<td>FR-Q-4</td>
<td>Assess the <strong>QuickCheck</strong> possibilities</td>
<td>The final solution should include the main <strong>QuickCheck</strong> properties.</td>
<td>Medium</td>
<td>QuickCheck</td>
<td>1.3.2</td>
</tr>
<tr>
<td>NFR-I-5</td>
<td>Update the IDE support</td>
<td>The IDE should include the new keywords in the syntax coloration, and should provide a simple access to the final solution.</td>
<td>Low</td>
<td>IDE</td>
<td>1.3.2</td>
</tr>
<tr>
<td>NFR-P-6</td>
<td>Follow feedbacks</td>
<td>The final solution shall include the outcome of the discussions of the benefits of Behaviour-driven development (BDD) with neophytes.</td>
<td>High</td>
<td>Project</td>
<td>1.3.1</td>
</tr>
</tbody>
</table>

### Table 3.2: Legend for Table 3.1

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR</td>
<td>Functional Requirement</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirement</td>
</tr>
<tr>
<td>L</td>
<td>Language Category</td>
</tr>
<tr>
<td>G</td>
<td>Generator Category</td>
</tr>
<tr>
<td>Q</td>
<td>QuickCheck Category</td>
</tr>
<tr>
<td>I</td>
<td>IDE Category</td>
</tr>
<tr>
<td>P</td>
<td>Project Category</td>
</tr>
</tbody>
</table>
3.2 Methodology

3.2.1 Implementation of the solution

In response to the FR-L-1 requirement (see Table 3.1), I will use the feedbacks gathered during my first experiment. I will try to use the BDD tool Cucumber to clearly define the scope of the project. I will develop the solution in Java, by extending the current work of cucumber-jvm\(^3\).

3.2.2 Evaluation

The evaluation will be done in several steps. It is an important part of the project where I will be checking whether the requirements have been fulfilled and the overall solution is good enough to answer existing problems. First, I will try to gather some students of the School of Languages & Intercultural Studies (LINCS) and the School of Mathematical and Computer Sciences schools (Heriot-Watt University, Edinburgh), who already took part in a User Experience (UX) project together. In order to answer the NFR-P-6 requirement (see Table 3.1), I will discuss with them what they thought of the multiple workshops that composed the UX project through a questionnaire. I will propose them some simple scenarios using BDD and let them use it and mark it (usability, comprehension, ...). Then, I will gather their feedbacks and try to extract the maximum of useful information. These informations will help me to guide the integration of PBT into BDD.

Secondly, I will contact a BDD expert in order to assess my solution, and compare it against existing BDD solutions. I will invite him to try my solution, observe him using it, and discuss with him the original BDD tool Cucumber and its features. Again, this will answer the NFR-P-6 requirement (see Table 3.1). This evaluation will be done close to the end of the project, so the expert would have a nearly finished solution to compare to existing ones.

Having non-expert practitioners feedback and expert feedback will help me assess the two facets of BDD, which is oriented to be used by non-expert (business people, clients, people with no knowledge of IT) as well as by expert people (developers).

\(^3\text{https://github.com/cucumber/cucumber-jvm, accessed: 25/02/2017}\)
Chapter 4

Design

This project being an extension of an already existing open-source product, I tried to put efforts into implementing the solution without perturbing its original functioning. This design phase has been important in the development of the software because it helped give a direction to the project. Thanks to this, I have been able to make decisions more quickly when facing choices.

4.1 Tools and technologies

This project being an extension of an already existing one (cucumber-jvm\textsuperscript{1}), I had to use the same language already used by the cucumber team: Java. Moreover, since Cucumber is using Gherkin to generate Step definitions, I learned and used Gherkin a lot in order to develop and test the final solution. Then, I had to use regular expressions to detect keywords in the Gherkin files.

In order to simplify the development process, I chose to use an IDE: JetBrains’ IntelliJ IDEA\textsuperscript{2}, because I am very familiar with it and it is a great IDE, also satisfying thanks to its look but also its user-friendliness. I chose to use GitHub\textsuperscript{3} and BitBucket\textsuperscript{4}, using the Git Version Control System, in order to keep a detailed history of the project software and the project dissertation. That allowed me not to lose any piece of work during this project.

\textsuperscript{1}https://github.com/cucumber/cucumber-jvm, accessed: 25/02/2017
\textsuperscript{2}https://www.jetbrains.com/idea, accessed: 3/04/2017
\textsuperscript{3}https://github.com, accessed: 13/07/2017
\textsuperscript{4}https://bitbucket.org, accessed: 13/07/2017
4.2 Code analysis of the existing software

A big challenge of enhancing other people’s projects is to adapt to the existing code. Professional projects such as cucumber-jvm\(^5\) and gherkin-java\(^6\) are difficult to read and understand. Moreover, these projects are tremendous by the number of files and lines of code. This is why, in order to find the correct design for my software, I had to understand the design of the already existing software.

My project resting on parsing the Steps of the Gherkin file and adding new keywords to be matched, I searched for the part in the code where the Steps were parsed. I already knew they were parsed somewhere because of one feature of Cucumber: when processing a Step containing an integer or a string between quotation marks, Cucumber replaces it by the corresponding regular expression (see Figure 4.1).

```java
Feature: Checkout
  Scenario: Checkout a banana
  Given the price of a "banana" is 40c
  When I checkout 1 "banana"
  Then the total price should be 40c

@Given("the price of a \d+ c\$")
public void thePriceOfAIsC(String arg0, int arg1) throws Throwable {
  // Write code here that turns the phrase above
  // into concrete actions
  throw new PendingException();
}
```

![Figure 4.1: Cucumber Regular Expression extracted from the original Gherkin](https://github.com/cucumber/cucumber-jvm, accessed: 25/02/2017)

4.2.1 Analysis of the gherkin-java project

The gherkin-java project may be run through two different classes: Main and Generate Tokens. In order to find a suitable design for the software, I used the IDE (see Tools and technologies) and its Debug mode to follow the path of execution, and search for what is relevant for my project.

I found out during my research that this Main class eventually prints out the whole Gherkin document as a JSON object. Moreover, it makes use of multiple classes that helped me understand how Gherkin worked. For instance, a Pickle object is what defines a Scenario. It contains the Steps, but without the keywords Given-When-Then.

\(^5\)https://github.com/cucumber/cucumber-jvm, accessed: 25/02/2017
\(^6\)https://github.com/cucumber/gherkin-java, accessed: 13/07/2017
The **Pickle** class and its primary dependencies are shown in the **Unified Modeling Language (UML)** class diagram in Figure 4.2.

![Figure 4.2: Pickle class diagram](image)

By looking at the UML class diagram above, it is quite clear how is structured a **Scenario** in the **Gherkin** language. Also, the **GherkinDocument** object will store the **Feature** part of the **Gherkin** document, along with comments. Therefore, this object will contain the **Steps**, as well as the **Pickle** objects. The **GherkinDocument** object is supposed to be larger than the **Pickle** object, since the latter represents a **Scenario**, while the **GherkinDocument** represents a **Feature** (see **Cucumber** and **Gherkin** in the Literature Review).

The output of the **Main** class is shown on Figure 4.3, for a sample **Gherkin** file. Please note that some repeating parts have been omitted in order to make this output clearer and shorter. The important things to point out here are the number of information extracted from a single small **Gherkin** file. The language in which it is written is recovered, and every keyword (**Given-When-Then**) is extracted along with the following text of the **Step** and its location in the **Gherkin** document.
Scenario: My example is good
Given an example of a scenario
When I have 1 scenario
Then I am happy

```json
{
    "type": "gherkin-document",
    "uri": "testdata/good/my-example.feature",
    "document": {
        "feature": {
            "tags": [],
            "language": "en",
            "keyword": "Feature",
            "name": "my example",
            "children": [{
                "tags": [],
                "keyword": "Scenario",
                "name": "My example is good",
                "steps": [...] {
                    "keyword": "When",
                    "text": "I have 1 scenario",
                    "type": "Step",
                    "location": {
                        "line": 5,
                        "column": 5
                    }
                }, [...] },
                "type": "Feature",
                "location": {
                    "line": 5,
                    "column": 5
                }
            }, [...] ]
        },
        "type": "GherkinDocument"
    }
}
```

Figure 4.3: Output of the Main method of the gherkin-java project
Finally, by running the debug mode through this, I concluded there were no parsing of the Steps made here, but only a parsing to match the Given-When-Then keywords, and construct the corresponding Pickle and GherkinDocument objects. In fact, in the Step I have 1 scenario (see Figure 4.3 at line 15-21), the number 1 is not treated as an integer but as a string, therefore not yet recognize as a modifiable parameter by gherkin-java. This is why I continued searching in the second runnable class of gherkin-java: GenerateTokens.

The GenerateTokens class prints out the parsed document with tokens and lines (see Figure 4.4).

```
<table>
<thead>
<tr>
<th>Feature: Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background: a simple background</td>
</tr>
<tr>
<td>Given the minimalism inside a background</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario: minimalistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given the minimalism</td>
</tr>
</tbody>
</table>
```

Figure 4.4: Output of the GenerateTokens method of the gherkin-java project

It is quite clear that this runnable class does not help much in the search for parsed Step. However, combined with the Main class, it helps to understand the purpose of the gherkin-java project, which is in fact all the tools required for a software to “understand” the Gherkin language. It gives the means to parse any Gherkin document and other extra information (location of the keywords, languages used, ...), but it will not parse anything within the text of the Steps: that is the role of the cucumber-jvm project.

### 4.2.2 Analysis of the cucumber-jvm project

The cucumber-jvm project, which in the following shall be renamed courjette-jvm when the first modifications appear, may be run through only one class. The Main class of this project prints out the output of the original Cucumber tool (see Figure 4.5).
Chapter 4. Design

Feature: Checkout
Scenario: Checkout a banana
   Given the price of a "banana" is 40c
   When I checkout 1 "banana"
   Then the total price should be 40c

1 Scenarios (1 undefined)
3 Steps (3 undefined)
0m0,.000s

You can implement missing steps with the snippets below:

```java
@Given("the price of a \[\d+\] is \[\d+\]c")
public void the_price_of_a_is_c(String arg1, int arg2)
    throws Throwable {
    // Write code here that turns the phrase above
    // into concrete actions
    throw new PendingException();
}

@When("I checkout \[\d+\] \[\d+\]c")
public void i_checkout(int arg1, String arg2)
    throws Throwable {
    // Write code here that turns the phrase above
    // into concrete actions
    throw new PendingException();
}

@Then("the total price should be \[\d+\]c")
public void the_total_price_should_be_c(int arg1)
    throws Throwable {
    // Write code here that turns the phrase above
    // into concrete actions
    throw new PendingException();
}
```

Figure 4.5: Output of the Main method of the cucumber-jvm project

Before making the project run, I had to resolve some configuration problems. In fact, the cucumber-jvm project containing multiple modules, it can be run using different classpath. Since I am more interested in the Java project, I had to define the classpath to cucumber-java. Next, I had to check the dependencies versions, and used the newer ones from the Maven repositories. Finally, in order to make it run, I had to configure the cucumber-jvm-deps and gherkin dependencies to compile within the cucumber-java project. After that, the project was runnable when specifying a argument: this argument is in fact the path of the Gherkin file to run with Cucumber. I chose it to be the following file, that I shall use later in this dissertation (see Figure 4.6).

When finally being able to run in debug mode this Main class, I followed carefully the execution path. This cucumber-jvm project was a lot more difficult than the
Chapter 4. Design

Feature: my example

Scenario: My example is good
  Given an example of a scenario
  When I have 1 scenario
  Then I am happy

Scenario: My new example will work
  Given any new scenario
  When I have 1 scenario
  Then my example works

Scenario: This is a complex example with lists
  Given any list of list of integers
  When I have 1 integer positive
  Then any other integers and one "banana" are positive

java/src/test/resources/cucumber/runtime/java/test/my-example.feature

Figure 4.6: my-example.feature Gherkin file used to test the software

gherkin-java project, by its size, but also by its inherent complexity. It took time and multiple attempts to find the correct part of the code that was searching for patterns in the Steps. The part of the code in charge of this feature is actually the SnippetGenerator class (see Figure 4.7). On the following Figure, the argumentTypes method calls for multiple patterns and adds the type of each the pattern matched in the resulted list.

```java
private List<Class<?>> argumentTypes(Step step) {
  Matcher[] matchers = new Matcher[argumentPatterns().length];
  for (int i = 0; i < argumentPatterns().length; i++) {
    matchers[i] = argumentPatterns()[i].pattern().matcher(name);
  }
  [...]
  while (true) {
    int matchedLength = 1;
    for (int i = 0; i < matchers.length; i++) {
      Matcher m = matchers[i].region(pos, name.length());
      if (m.lookingAt()) {
        Class<?> typeForSignature = argumentPatterns()[i].type();
        argTypes.add(typeForSignature);
        matchedLength = m.group().length();
        break;
      }
    }
  }
  [...]
```

Figure 4.7: Part of the argumentTypes method of the SnippetGenerator class

When looking what is the argumentPatterns()[i].type() (Figure 4.7, line 14) of the When I have 1 scenario Step (Figure 4.6, line 5), one can see a call to the ArgumentPattern class, that is filled with the following data (see Figure 4.8).
It is safe to assume that the parsing of Steps is made here, when Cucumber tries to construct the snippets of code, therefore towards the end of the program. This is on this basis that I shall explain my design for CourJette in the next section.

### 4.3 First design of CourJette

The previous section stated:

"""
My project resting on parsing the Steps of the Gherkin file and adding new keywords to be matched, I searched for the part in the code where the Steps were parsed.
""

- Section 4.2: Code analysis of the existing software

Therefore, the main idea would be to match the keyword “any” and enhance the snippet provided by the original Cucumber with the Property-based testing (PBT) predicate that suits. In order for CourJette to match the new keyword “any” that would trigger a QuickCheck context, I would have to add a new pattern to the result of the `argumentPatterns()` function (see Figure 4.7, lines 3-6). In fact, this function just returns a static array declared in the `SnippetGenerator` class (see Figure 4.9).

```
private static final ArgumentPattern[] DEFAULT_ARGUMENT_PATTERNS
    = new ArgumentPattern[]{
        new ArgumentPattern(Pattern.compile("(\d+)"), Int.class),
        new ArgumentPattern(Pattern.compile("(\d+)"), Integer.TYPE)
    };
```

**Figure 4.9:** Static array returned by `argumentPatterns()` function

The elements of the `DEFAULT_ARGUMENT_PATTERNS` array are `ArgumentPattern` objects (see Figure 4.8). They consist of a `Pattern`, which is in fact a Regular Expression, and a `Type`, that is the Java type of the pattern. Here, the first element of the array is a
pattern that will match any string that is inside quotes. The second element will match any integer. Therefore, I will add the “any” keyword in this list of patterns. This will be discussed in the next chapter Prototyping, redesign and implementation.

In order to propose an effective design, I asked some people at Heriot-Watt University to fill in a questionnaire. One of the question was to determine the best approach for triggering the QuickCheck context in Cucumber. I proposed two kind of solutions to the participants, and they elected option 1 as the best one (see Figure 4.10).

```
Feature: Checkout
Scenario: Checkout any Fruit
  Given the price of any Fruit is 40c
  When I checkout 1 Fruit
  Then the total price should be 40c
```

Option 1 – 66.7%

```
Feature: Checkout
Scenario: Checkout $Fruit$
  Given the price of $Fruit$ is 40c
  When I checkout 1 Fruit
  Then the total price should be 40c
```

Option 2 – 33.3%

Figure 4.10: Options given on the questionnaire to find the more usable generalization of Behaviour-driven development (BDD)

This has comforted me in my initial choice, since it is the simplest way of inducing a generalization into a Controlled natural language (CNL) by respecting its main property: using natural language.
Chapter 5

Prototyping, redesign and implementation

Following the big first steps of code analysis, I began to prototype my software and adjusted its main design. The nature of this project being to enhance an existing software made this prototyping and redesign part blend with the implementation part. In fact, since I was not constructing this project from scratch, but only modifying key parts of the code, I was not totally free to choose the design, but I had to follow the existing structure and adapt my modifications to it.

5.1 First modifications

In regard of the recognition of the “any” keyword, I chose to modify the DEFAULT_ARGUMENT_PATTERNS array (see Figure 4.9). Therefore, the new DEFAULT_ARGUMENT_PATTERNS array looks like in the below Figure 5.1, modification being on line 5.

```java
private static final ArgumentPattern[] DEFAULT_ARGUMENT_PATTERNS
    = new ArgumentPattern[]{
    new ArgumentPattern(Pattern.compile("\|\n\|[-|"\n]Š|"\n\|\n\|d+\n\|\n\|any\n\|\n\|class\n\|\n\|class\n\|\n\|class\n\|\n\|class
```
with the ArgumentPattern object corresponding to the “any” keyword matched the **When** I have 1 scenario Step (Figure 4.6, line 5).

Hence, the next step would be to alter the snippet generated by the original Cucumber tool and generate a Property-based testing (PBT) predicate. I shall start with a simple @Property annotation, used by QuickCheck to locate the methods to run.

The method used by Cucumber to generate a snippet is shown on Figure 5.2. This method, in the SnippetGenerator class, calls for a template. This template (see Figure 5.3) contains arguments (numbers between curly brackets) which are defined by the getSnippet method. In fact, the MessageFormat.format() function allows the template to be filled with the arguments passed to this later function.

```java
public String getSnippet(Step step,
FunctionNameGenerator functionNameGenerator) {
    return MessageFormat.format(
            snippet.template(),
            I18n.codeKeywordFor(step.getKeyword()),
            snippet.escapePattern(patternFor(step.getName())),
            functionName(step.getName(), functionNameGenerator),
            snippet.arguments(argumentTypes(step)),
            REGEXP_HINT,
            step.getRows() == null ? "": snippet.tableHint()
    );
}
```

**Figure 5.2:** getSnippet() function generating the Java snippets

```java
@Override
public String template() {
    return "@{0}("{1}"|"
    + "public void {2}({3}) throws Throwable '{|n" + 
    "| }|n" + 
    "|} throw new PendingException();|n" + 
    "|}''|n";
}
```

**Figure 5.3:** Default template used for each Step

The documentation of the Snippet interface explains the multiples arguments called in the template (see Figure 5.4). This has helped me understand the functioning of the MessageFormat.format() method and how the template was filled to create a meaningful snippet.

In order to include the @Property QuickCheck annotation, I decided to create a new QuickCheckSnippet class to define my own template. The first attempt was to insert the QuickCheck snippet before the original Cucumber snippet. The template() method of the QuickCheckSnippet class was very short (see Figure 5.5)
java.lang.String template()

Returns:
a java.text.MessageFormat template used to generate a snippet.
The template can access the following variables:

{0} : Step Keyword
{1} : Value of escapePattern(String)
{2} : Function name
{3} : Value of arguments(java.util.List)
{4} : Regexp hint comment
{5} : value of tableHint() if the step has a table

Figure 5.4: Documentation of the template() method of the Snippet interface

@Override
public String template() {
    return "@{0}({1})\n";
}

Figure 5.5: First attempt at QuickCheckSnippet’s template

This first attempt resulted in a change in the getSnippet() method (see Figure 5.6). I chose to insert the @Property(trials = 5) annotation before the original Cucumber snippet each time the keyword “any” would be used. Therefore, a quickCheckSnippet variable is used to fill in the template whenever the output code is Java and a quickCheck variable is used whenever the “any” keyword is matched on a Step. If it is, this Step’s snippet shall have the @Property(trials = 5) annotation.

Hence, the output of CourJette for this first attempt is shown on Figure 5.7. This output was made using the my-example.feature Gherkin file (see Figure 4.6). The explanations of the @Property annotation and all other information on QuickCheck are explained in the Literature Review, section 2.4.
```java
public String getSnippet(Step step,
    FunctionNameGenerator functionNameGenerator) {
    String qcSnippet = (quickCheckSnippet == null ? "":
        MessageFormat.format(
            quickCheckSnippet.template(),
            "Property",
            "trials = 5"
        )
    );

    String cucumberSnippet = MessageFormat.format(
        snippet.template(),
        IiSn.codeKeywordFor(step.getKeyword()),
        snippet.escapePattern(patternFor(step.getName())),
        functionName(step.getName(), functionNameGenerator),
        snippet.arguments(argumentTypes(step)),
        REGEXP_HINT,
        step.getRows() == null ? "": snippet.tableHint()
    );

    return (!quickCheck ? "": qcSnippet) + cucumberSnippet;
}
```

Figure 5.6: First attempt at modifying the getSnippet() function

```
2 Scenarios (2 undefined)
6 Steps (6 undefined)
0m0,000s

You can implement missing steps with the snippets below:

@Given("an example of a scenario")
public void an_example_of_a_scenario() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

[...]

@Property(trials = 5)
@Given("any new scenario")
public void new_scenario(String arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

[...]
```

Figure 5.7: Output of the first attempt for CourJette
5.2 Enhancements

The first prototype explained in the previous section works for every Gherkin file, whether they contain the keyword “any” or not. However, the integration of QuickCheck is not yet high enough since there is no variation in the snippet between two different Steps containing the keyword “any”. Therefore, I enhanced the integration of QuickCheck.

5.2.1 Adding more patterns

An interesting functionality would be to refer to the same objects when using the keyword “any”. For example, in the following Gherkin file (see Figure 5.8), “any user” and “the user” should refer to the same thing, when “any list of users” and “the list of users” should refer to another thing.

```
1 Given any user
2 Given any list of users
3 When I add the user to the list of users
4 Then the user should be in the list of users
```

Figure 5.8: Gherkin example on adding more patterns

However, when trying to implement this feature, I felt this was not really adding much value to the software, since using the “any” keyword would allow QuickCheck to generate as much values as the user wants, therefore not binding two object together would allow a larger set of values to be tested.

Still, I managed to modify the code to include two more patterns to be matched by CourJette (see Figure 5.9, lines 6-9). I added two lines to the DEFAULT_ARGUMENT_PATTERNS array supposed to match for Steps containing “any [...] from [...]” or “any [...] to [...]. The original idea would have been to get the object that follows the “any” keyword in order to instantiate the corresponding QuickCheck Generator in the annotation, and do the same for the object that follows “from” or “to”.

This modification induced a change in the template used for QuickCheck. The previous template was the one shown on Figure 5.5. The new template is the one shown below on Figure 5.10. This template is much more elaborated than the first one. For example, it includes the original Cucumber’s template and adds the @Property annotation to it. Moreover, it inserts the Generators in the method signature: @From(Class).class) {4} should generate an input such as @From(IntegerGenerator.class) Integer arg1 for an integer.
5.2.2 Generate QuickCheck Generators

The previous section led me to think of how to design the integration of Generators into CourJette. First, I thought of two solutions:

1. Give a snippet to the user that will contain the basis for a custom Generator (to be implemented by the user);

2. CourJette will search for predefined keywords following the any keyword (“Given any list of user”; “Given any list of integer”) and insert core Generators provided by junit-quickcheck into the snippet.

The snippet given to the user for solution 1 would be the default snippet for customizing a Generator (see Figure 2.16, in the Literature Review). For solution 2, CourJette would generate snippets with signatures like: public void test(@From(Integer Generator.class) Integer integer).

In any case, the Generator, for a non-basic object (such as a “User” class) must be implemented by the user. However, for some predefined keywords such as the ones in Figure 5.11, there could be a feature that would search them, after the “any” keyword, and use core Generators provided by junit-quickcheck.
Therefore, the final solution will combine solution 1 and solution 2. In order for CourJette to match the previous keywords in Figure 5.11, I based my work on already existing methods of the original cucumber-jvm project. In fact, I created a new static array, named QUICKCHECK_GENERATOR_PATTERNS. This array, shown on Figure 5.12, has the same structure as the DEFAULT_ARGUMENT_PATTERNS array, but will match generic Java types in the Gherkin file.

This array is quite long for two reasons: first, listing this code in IMpX forces me to break a lot of lines; second, the patterns compiled in this array are the strength of CourJette. With this modification, all the basic types of Java are now recognized by CourJette, and in any form. In fact, the Regular Expression allows the words to be case-insensitive ((?i)) and to be in their singular or plural form ((s)?). Moreover, for some types, it allows the use of truncated forms or full forms (int(eger)\?, enum(eration)\?, array(s)\?list(s)\?array(\?)list(s)\?, ...). It makes CourJette a lot more flexible, which is the purpose of using a Controlled natural language (CNL) for generating code.

At last, you will distinguish the use of Integer.class in this array (line 8) compared to the use of Integer.TYPE in the original DEFAULT_ARGUMENT_PATTERNS (Figure 4.9, line 4). This modification was made to access the Integer class, and not the basic type int. This simplifies the next step which is constructing the corresponding Generator.

The previous patterns are searched within each Step of the provided Gherkin file thanks to the following method (see Figure 5.13). This quickCheckGenerator

```java
byte
short
int
long
float
double
boolean
char
Big(Decimal|Integer)
Date
enum
String
ArrayList
LinkedList
HashSet
LinkedHashSet
HashMap
LinkedHashMap
```
private static final ArgumentPattern[] QUICKCHECK_GENERATOR_PATTERNS = new ArgumentPattern[] {
    new ArgumentPattern(Pattern.compile("(\?s)byte(s)\?"), Byte.class),
    new ArgumentPattern(Pattern.compile("(\?s)short(s)\?"), Short.class),
    new ArgumentPattern(Pattern.compile("(\?s)integer(s)\?"), Integer.class),
    new ArgumentPattern(Pattern.compile("(\?s)long(s)\?"), Long.class),
    new ArgumentPattern(Pattern.compile("(\?s)float(s)\?"), Float.class),
    new ArgumentPattern(Pattern.compile("(\?s)double(s)\?"), Double.class),
    new ArgumentPattern(Pattern.compile("(\?s)boolean(s)\?"), Boolean.class),
    new ArgumentPattern(Pattern.compile("(\?s)char(acter)s\?"), Character.class),
    new ArgumentPattern(Pattern.compile("(\?s)b(ig)decimal(s)\?"), BigDecimal.class),
    new ArgumentPattern(Pattern.compile("(\?s)b(ig)integer(s)\?"), BigInteger.class),
    new ArgumentPattern(Pattern.compile("(\?s)d(ate)s\?"), Date.class),
    new ArgumentPattern(Pattern.compile("(\?s)e(num)eration(s)\?"), Enum.class),
    new ArgumentPattern(Pattern.compile("(\?s)s(tring)s\?"), String.class),
    new ArgumentPattern(Pattern.compile("(\?s)a(rray)s\?list(s)\?|l(is)t(s)\?list(s)\?"), ArrayList.class),
    new ArgumentPattern(Pattern.compile("(\?s)l(inked)list(s)\?"), LinkedList.class),
    new ArgumentPattern(Pattern.compile("(\?s)h(ash)set(s)\?"), HashSet.class),
    new ArgumentPattern(Pattern.compile("(\?s)l(inked)hash(set)s\?"), LinkedHashSet.class),
    new ArgumentPattern(Pattern.compile("(\?s)h(ash)set(s)\?map(s)\?"), HashMap.class),
    new ArgumentPattern(Pattern.compile("(\?s)l(inked)hash(map)s\?"), LinkedHashSet.class)
};

Figure 5.12: QUICKCHECK_GENERATOR_PATTERNS array

Arguments() method is directly inspired from the argumentTypes() method of the same SnippetGenerator class (see Figure 4.7). This method changes, of course, the static array of patterns from DEFAULT_ARGUMENT_PATTERNS to the newly created QUICKCHECK_GENERATOR_PATTERNS, respectively returned by the argumentPatterns() and quickCheckGeneratorPatterns() functions. Next, this method will search for these patterns in the currently analysed substep. Here, a substep is the part of a Step following the keyword “any”. When matching a pattern, it will add the type of the pattern matched to the returned list.
private List<Class<?>?> quickCheckGeneratorArguments(String substep) {
  List<Class<?>> argTypes = new ArrayList<Class<?>?>();
  Matcher[] matchers =
    new Matcher[quickCheckGeneratorPatterns().length];
  for (int i = 0; i < quickCheckGeneratorPatterns().length; i++) {
    matchers[i] = quickCheckGeneratorPatterns()[i].pattern().matcher(substep);
  }
  int pos = 0;
  while (true) {
    int matchedLength = 1;
    for (int i = 0; i < matchers.length; i++) {
      Matcher m = matchers[i].region(pos, substep.length());
      if (m.lookingAt()) {
        Class<?> typeForSignature =
          quickCheckGeneratorPatterns()[i].type();
        argTypes.add(typeForSignature);
        matchedLength = m.group().length();
        break;
      }
    }
    pos += matchedLength;
    if (pos == substep.length()) {
      break;
    }
  }
  return argTypes;
}

Figure 5.13: Searching for QuickCheck patterns

The previous `quickCheckGeneratorArguments()` method allows a list of types being returned. In order for these elements to be treated, I overrode the `arguments()` function of the `QuickCheckSnippet` class (see Figure 5.14), inherited from the `Snippet` class. This function used to concatenate the types recovered from the original `argumentTypes()` function and format the output to be a succession of parameters (e.g. “String arg1, int arg2”). Now, the new `arguments()` function recovers the types returned by the `quickCheckGeneratorArguments()` function and builds the corresponding succession of parameters and includes the `QuickCheck Generators` (e.g. for an integer, “@From(IntegerGenerator.class) Integer arg1”).

The intriguing part of this function is actually the one between line 16 and line 24. This part of the function allows the returned string to contain parameters of generic type. For instance, should CourJette detect a list of integers in the Gherkin file, it would be translated into an `ArrayList<Integer>` in Java. Using QuickCheck’s Generators, it would be written: `@From(ArrayListGenerator.class) ArrayList<@From(Integer...`
Chapter 5. Prototyping, redesign and implementation

Generator.class) Integer> arg1. The arguments() function allows CourJette to format correctly any number of generic type parameters. Generic types are recognized using the private method isGenericType() (Figure 5.14, lines 32-43).

```java
@override
public String arguments(List<Class<?>> argumentTypes) {
    StringBuilder sb = new StringBuilder();
    int prevArgType = 0;
    int n = 1;
    for (Class<?> argType : argumentTypes) {
        if (n > 1 & & prevArgType == 0) {
            sb.append(“, ”);
        }
        sb.append("@From(")
            .append(getArgType(argType))
            .append("Generator.class")
            .append(" @")
            .append(getArgType(argType));
        if (isGenericType(argType)) {
            prevArgType += 1;
            sb.append("<”);
            continue;
        }
        while (prevArgType > 0) {
            prevArgType -= 1;
            sb.append(“>”);
        }
        sb.append(" “)
            .append("arg")
            .append(n++);
    }
    return sb.toString();
}

private boolean isGenericType(Class<?> argType) {
    Boolean result = Boolean.FALSE;
    if (getArgType(argType).equals("ArrayList") ||
        getArgType(argType).equals("LinkedList") ||
        getArgType(argType).equals("HashSet") ||
        getArgType(argType).equals("LinkedHashSet") ||
        getArgType(argType).equals("HashMap") ||
        getArgType(argType).equals("LinkedHashMap") ) {
        result = Boolean.TRUE;
    }
    return result;
}
```

**Figure 5.14:** Constructing the succession of parameters for the final snippet

Therefore, a little change was made to the template used for the snippet. The new version shown below (see Figure 5.15) is the definitive one. Compared to the previous version (Figure 5.10), this version does not include the “@From({3}.class) {4}” part in the signature. Instead, only a simple argument ({3}) will get the result of the above arguments() function.
The previous `quickCheckGeneratorArguments()` method is called inside the already existing `argumentTypes()` method, which is called by the `getSnippet()` method. I also had to make changes to this method for the project to work. The new version is shown below on Figure 5.16, while the previous version is on Figure 5.6. This new version introduces a new method `isQuickCheckStep()` that determines if a `Step` should trigger the `QuickCheck` template or not (i.e. by recognizing the keyword “any”). If it is triggered, the template used is the one above (Figure 5.15), else it is the default template that is used (Figure 5.3). The string of arguments of the final snippet generated is still recovered by the `argumentTypes()` function.

```java
public String getSnippet(Step step, FunctionNameGenerator functionNameGenerator) {
    Snippet currentSnippet = isQuickCheckStep(step) ? quickCheckSnippet : snippet;
    return MessageFormat.format(
        currentSnippet.template(),
        I18n.codeKeywordFor(step.getKeyword()),
        currentSnippet.escapePattern(patternFor(step.getName())),
        functionName(step.getName(), functionNameGenerator),
        currentSnippet.arguments(argumentTypes(step)),
        REGEXP_HINT,
        step.getRows() == null ? "": currentSnippet.tableHint()
    );
}
```

Figure 5.15: Third attempt at QuickCheckSnippet’s template

Figure 5.16: Second attempt at modifying the `getSnippet()` function

The following `isQuickCheckStep()` function (see Figure 5.17) is actually quite simple since it is only a copy of the `argumentTypes()` function (Figure 4.7): it browses the `Step` in order to find the “any” keyword and thus determine the correct snippet to apply. In fact, if CourJette only checked for the “any” keyword when called at “currentSnippet.arguments(argumentTypes(step))” (Figure 5.16, line 8), it would be too late to order a change of snippet, which is done earlier in the same function (“currentSnippet.template()”, Figure 5.16, line 4). Therefore, the following `isQuickCheckStep()` takes care of enabling the `QuickCheck` template, while the new `argumentTypes()` function takes care of the string of arguments to construct.
private boolean isQuickCheckStep(Step step) {
    Boolean result = Boolean.FALSE;
    String name = step.getName();
    Matcher[] matchers = new Matcher[argumentPatterns().length];
    for (int i = 0; i < argumentPatterns().length; i++) {
        matchers[i] = argumentPatterns()[i].pattern().matcher(name);
    }
    int pos = 0;
    while (true) {
        int matchedLength = 1;
        for (Matcher matcher : matchers) {
            Matcher m = matcher.region(pos, name.length());
            if (m.lookingAt()) {
                // If we are in a QuickCheck situation
                if (m.group().equals("any")) {
                    result = Boolean.TRUE;
                }
                matchedLength = m.group().length();
                break;
            }
        }
        pos += matchedLength;
        if (pos == name.length()) {
            break;
        }
    }
    return result;
}

Figure 5.17: isQuickCheckStep() function determining the use of the QuickCheck template

Hence, the new argumentTypes() function is shown below (see Figure 5.18). When recognizing a QuickCheck situation (e.g. matching the “any” keyword), it will add all the values returned by the quickCheckGeneratorArguments() function to the result list. If not in a QuickCheck situation, it will do what it already did in the original Cucumber project: add the type of the pattern matched (i.e. either an integer or a string, as shown by Figure 4.9) to the result list.

This was the last of many modifications made to the code of the original Cucumber project (cucumber-jvm). The final results will be discussed in the next chapter Evaluation.
private List<Class<?>> argumentTypes(Step step) {
    String name = step.getName();
    List<Class<?>> argTypes = new ArrayList<Class<?>>();
    Matcher[] matchers = new Matcher[argumentPatterns().length];
    for (int i = 0; i < argumentPatterns().length; i++) {
        matchers[i] = argumentPatterns()[i].pattern().matcher(name);
    }
    int pos = 0;
    while (true) {
        int matchedLength = 1;
        for (int i = 0; i < matchers.length; i++) {
            Matcher m = matchers[i].region(pos, name.length());
            if (m.lookingAt()) {
                // If we are in a QuickCheck situation,
                // use the QuickCheck arguments function
                if (m.group().equals("any")) {
                    argTypes.addAll(quickCheckGeneratorArguments(
                        name.substring(m.end())));
                    break;
                    // If not, add the argument to the list
                } else {
                    Class<?> typeForSignature =
                        argumentPatterns()[i].type();
                    argTypes.add(typeForSignature);
                }
            }
        }
        pos += matchedLength;
        if (pos == name.length()) {
            break;
        }
    }
    if (step.getDocString() != null) {
        argTypes.add(String.class);
    }
    if (step.getRows() != null) {
        argTypes.add(DataTable.class);
    }
    return argTypes;
}

Figure 5.18: argumentTypes() function gathering the future arguments of the final snippet
Chapter 6

Evaluation

After implementing the features of CourJette (see previous chapter Prototyping, redesign and implementation), I evaluated the work done. First of all, in this chapter I will demonstrate the final result of CourJette, based on one situation. Next, I will use the help of Daniel Waghorn who evaluated my program. Moreover, thanks to fellow IT and Business students who answered a questionnaire, I was able to compare some of the answers to the final solution. Finally, I will explain the strengths and weaknesses of my program, and try to foresee some possible enhancements.

6.1 Output of CourJette

Based on an already existing project, my work on this program has only been to modify the behaviour of the original Cucumber tool to allow the snippets proposed by the tool to include Property-based testing (PBT) statements. Cucumber already proposed snippets at the end of its execution. In CourJette, the snippets are now adapted to some QuickCheck situations, and triggered by the keyword “any”.

In order to evaluate my project, I used the following Gherkin Scenario (see Figure 6.1). This Scenario doesn’t necessarily refer to any real-world situation, but it doesn’t need to do so in order to test the software.

```
Scenario: This is a complex example with lists
  Given any list of list of integers
  When I have 1 integer positive
  Then any other integers and one "banana" are positive
```

Figure 6.1: Complex Gherkin example to evaluate CourJette
Considering the previous Scenario, the following snippets are generated (see Figure 6.2). As expected, these snippets contain the necessary QuickCheck statements for them to be run through the QuickCheck tool. In the end, it transforms a complex Gherkin statement (“Given any list of list of integers”) into a QuickCheck method signature, respecting the syntax of both Java and QuickCheck. Also, it still supports the original Cucumber features, as shown by the second Step (“When I have 1 integer positive”, Figure 6.1, line 3), by generating exactly the same snippet (Figure 6.2, lines 11-15) as the original Cucumber tool.

```java
@Given("any list of list of integers")
@Property
public void list_of_list_of_integers(
  @From(ArrayListGenerator.class) ArrayList<
    @From(ArrayListGenerator.class) ArrayList<
      @From(IntegerGenerator.class) Integer arg1>) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("I have (|d+| integer positive")
public void i_have_integer_positive(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("any other integers and one |"([-|n]*")| are positive")
@Property
public void other_integers_and_one_are_positive(
  @From(IntegerGenerator.class) Integer arg1,
  @From(StringGenerator.class) String arg2) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

Moreover, even if it seems normal, one can note the fact that these snippets actually work. Indeed, when testing the software, I tried to run the output of CourJette, with the help of an IDE, and there was no clash between Cucumber and QuickCheck. Of course, in order for the snippets to be run through QuickCheck, there has to be the junit-quickcheck library installed. The following lines must be added to the concerned Java file (see Figure 6.3). The important line here is the `@RunWith(JUnitQuickcheck.class)` annotation (line 13). It specifies to JUnit the class to use to run the tests. Without it, QuickCheck will never test the snippets.
Chapter 6. Evaluation

47

```java
import com.pholser.junit.quickcheck.From;
import com.pholser.junit.quickcheck.Property;
// Import necessary Generators
import com.pholser.junit.quickcheck.generator.java.lang.IntegerGenerator;
[...]
import com.pholser.junit.quickcheck.runner.JUnitQuickcheck;
import cucumber.api.PendingException;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import org.junit.runner.RunWith;

@RunWith(JUnitQuickcheck.class)
public class Stepdefs {
    // Paste snippets here
}
```

Figure 6.3: Libraries to import for QuickCheck to run

6.2 External evaluation of CourJette

Before starting the implementation of CourJette, I managed to survey some fellow students at Heriot-Watt University who participated to a workshop, named LinUX-AD (for User eXperience and Agile Development), that gathered IT students and students from the School of Languages & Intercultural Studies (LINCS) in small groups to develop a software. This workshop illustrated quite well the need of a communication between business people (played by LINCS students) and developers (IT students). There were approximately 20 students participating in these workshops. I managed to survey 6 of them (3 IT students, 3 LINCS students).

Among the questions asked to these students, some justified the need of such a software. When asked “What do you think worked well during the LinUX-AD workshops?”, only 17% of them thought that the clarity of requirements was effective. In the same spirit, when asked “During the workshops, did you have some difficulties to understand/explain the requirements?”, 83% of them replied it happened at least once. Another important figure is that 66% of the respondents never heard of Behaviour-driven development (BDD) before. Only one respondent has already used it. However, when showed a simple example of a Gherkin Feature, 66% of the respondents find it understandable. Half of the respondents say they would use BDD in a project similar to the LinUX-AD they participated in because “it seems like a good way to make developers and clients communicate better”.

Therefore, it is now a certainty that this project is responding to a real problem between developers and business people. The implementation I made aims to answer some of this problems.
In order for this MSc project to be meaningful, an evaluation made by a third party is mandatory. Cucumber being a very specific tool, the number of users is not very high. Therefore, thanks to Dr. Manuel Maarek, I managed to find an user of Cucumber, who already knows the software and what it does. This knowledge is crucial in evaluating CourJette, because of its nature being an enhancement of the existing software. A fellow student from Heriot-Watt University, Daniel Waghorn, who has agreed to use his expertise to evaluate the program, helped me evaluate CourJette.

In order for him to test the software, I gave him the link to the Git repository of courjette-jvm. To make it simpler, I added to the repository a working JAR archive that includes all the necessary libraries. This allowed him to successfully run my project to experiment with. This archive is available at the following path: courjette-jvm/classes/artifacts/cucumber_java_jar/cucumber-java.jar.

According to Daniel, CourJette “is definitely a worthwhile extension to the tooling typically available in Cucumber / BDD” testing. It allows implementations to be easily tested more comprehensively than would be typically possible with Gherkin multi-line argument sets and tables.

Moreover, the main feature that Daniel found useful is how through the use of generics it would be possible in CourJette to interchange the factory classes (e.g. IntegerGenerator.class) with more complex classes such as for example Object-relational mapping (ORM) classes to represent an entity such as a “User”.

Furthermore, he stated that the nesting seemed to work effectively, which would allow one to create a list of “User” classes extending the example above, using the simple Gherkin statement Given any list of User.

Finally, Daniel thinks this tool would bring ease to testing behavioural specifications over a comprehensive set of test data, saving a great deal of time and effort. By offloading much of the seed data generation, it maintains readability in the specifications (something which multi-line and table arguments in the Gherkin language can sometimes sacrifice). He feels that he would use this tool in a Java based system which features a test suite utilising BDD and Cucumber, since CourJette “is definitely something that is interesting”, and now that he has seen it in practice, “exposes quite a significant limitation in the Cucumber / Gherkin ecosystem”.

\[1\] https://github.com/Archigog/courjette-jvm
6.3 Strengths and weaknesses of CourJette

CourJette has fulfilled multiple requirements of the project (see Table 3.1 in Methodology and Requirements). First of all, the Gherkin language has been extended: the keyword “any” has been added to the keywords recognized by Cucumber when parsing Steps (Req. ID FR-L-1). Next, QuickCheck annotations have been generated following the recognition of patterns in the Steps (Req. ID FR-L-2).

In order for the final solution to contain the main QuickCheck properties, the QuickCheck solution has been assessed. It makes the features of CourJette a lot more meaningful by trying to fill in a gap in the original testing solution (Req. ID FR-Q-4).

I also put a lot of efforts in following feedbacks given by neophytes of BDD, gathered at the beginning of the project (Req. ID NFR-P-6).

CourJette has all the features of Cucumber and only adds some others. Therefore it can replace the original Cucumber for users who want to generate PBT tests along with their Java Step definitions.

Unfortunately, I have not been able to generate only PBT tests using the Gherkin language (Req. ID FR-G-3). The IDE support has not been implemented (Req. ID NFR-I-5).

One other weakness of CourJette has been discovered during the testing period: some Java types have not been properly implemented, such as HashMap and LinkedHashMap. In fact, these objects have a special syntax when instantiated, they are in the form of new Map<Key, Value>, which is different from other generic types such as an ArrayList(new ArrayList<Value>).

6.4 Future work

I noticed some improvements that could be made to CourJette. First of all, for CourJette to be better, it would be a great value to integrate more QuickCheck features. At this moment, CourJette only recognize one trigger keyword (“any”) and only use the Generators aspect of QuickCheck. Other features such as shrinking, which is one of the most compelling features of a proper QuickCheck, could make the blending of QuickCheck into Cucumber a lot more convincing.

Moreover, I would have liked to implement the IDE support for CourJette, as suggested by the requirement NFR-I-5 (see Table 3.1 in Methodology and Requirements).
This support would allow users to directly generate the snippets of CourJette without copy-pasting the output of the Command-Line Interface (CLI) version of CourJette. This IDE support could be made to IntelliJ IDEA, that I used to develop this software, and that already supports the original Cucumber.

Another improvement, linked to the requirement FR-6-3, would have been to generate only PBT tests with the Gherkin file. In fact, QuickCheck is a difficult tool, the syntax is sometimes complicated, hence the generation of the method signatures via CourJette could be an interesting feature.

On a different aspect, I think CourJette or Cucumber may be used at Heriot-Watt University for the future Linux-AD workshops, or other significant project involving Agile methods. It could help the developers (i.e. students) to follow the requirements of a project and simplify the development process.

### 6.5 Conclusion

This report details the development of a software for generating stubs of Java methods, following specific requirements given by the user, and tries to integrate a successful method of testing when it can be. This aims to help developers and business-people who sometimes don’t understand each other. Now, developers who use this software are not only following precisely the requirements that they have been given (Cucumber), but also benefit of a tool to test their implementations (QuickCheck). All of this with a single click.

Evaluation clearly indicates that the developed application does meet the aims of the project, even though it could still be enhanced (see previous section Future work). I have understood the concept of BDD and PBT, while developing a useful software for conscientious developers. I also faced and overcame a number of challenges such as analysing an already existing software, that uses some features of a language that I didn’t master. This project has broadened my knowledge, understanding and programming skills in this subject.
MACS Risk Assessment Form (Project)

**Student:** Hugo LEGRAND (H00258134)

**Project Title:** CourJelle: a Java implementation of Cucumber blended with QuickCheck

**Supervisor:** Manuel MAAREK

### Risks:

<table>
<thead>
<tr>
<th>Risk</th>
<th>Present (give details) (risk if present)</th>
<th>Control Measures and/or Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Office environment-includes purely software projects</td>
<td>✓</td>
<td>Nothing</td>
</tr>
<tr>
<td>Unusual peripherals</td>
<td>e.g., Robot, VR helmet, haptic device, etc.</td>
<td></td>
</tr>
<tr>
<td>Unusual Output</td>
<td>e.g., Laser, loud noises, flashing lights, etc.</td>
<td></td>
</tr>
<tr>
<td>Other risks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4:** Risk Assessment Form of the project
References


Lecture Notes in Computer Science, pages 104–124. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-85658-0_3.


List of terms and acronyms

ACE

Attempted Controlled English. 10, 11, Glossary: Attempted Controlled English

Agile

The Agile approach is a Project Management paradigm. It allows a better conception of a project by using iterative cycles and other specificities. iv, 1–3, 11, 13, 49

Attempted Controlled English

Attempted Controlled English is a Controlled natural language (CNL). ACE can serve as knowledge representation, specification, and query language, and is intended for professionals who want to use formal notations and formal methods, but may not be familiar with them. 10, 53

BDD

Behaviour-driven development. iii, 1, 5, 6, 13, 21, 30, 46–49, 54, Glossary: Behaviour-driven development

Behaviour-driven development

Behaviour-driven development is a software development methodology in which an application is specified and designed by describing how its behaviour should appear to an outside observer. iii, 1, 6, 21, 30, 46, 53

BPMN

Business Process Model and Notation. 7, 9, 10, Glossary: Business Process Model and Notation

Business Process Model and Notation

Business Process Model and Notation is a business process model and a notation to describe value chains and business activities of an organization in the form of a standardized graphic representation. 7, 53
CLI

Command-Line Interface. 49, Glossary: Command-Line Interface

CNL

Controlled natural language. 6, 10, 11, 30, 37, 53, Glossary: Controlled natural language

Command-Line Interface

A command line interface is a human-machine interface in which the communication between the user and the computer is performed in text mode. 49, 54

Controlled natural language

Controlled natural languages are subsets of natural languages that are obtained by restricting the grammar and vocabulary in order to reduce or eliminate ambiguity and complexity. 6, 30, 37, 53, 54

Cucumber

Cucumber is a BDD tool. iii, vii, 5, 13–15, 19–23, 26, 27, 29, 30, 32, 33, 35, 42, 44, 45, 47–49

End-to-end tests

An end-to-end test is a test that checks a software as a whole. It follows Unit tests. 15

Gherkin

Gherkin is the syntax used by Cucumber to write BDD features. 5, 11, 13–16, 19–24, 26–29, 33, 35, 37, 39, 44–49

Git

Git is a Version Control System (VCS). It allows to track changes of computer files. 22, 47

IDE

“An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development”\(^2\). 12, 20–23, 45, 48, 49, 56

Java

“Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible”3. iii, 8, 17, 19, 20, 22, 27, 29, 32, 33, 37, 39, 45, 47–49

JSON

JavaScript Object Notation (JSON) is a textual data format derived from the notation of JavaScript language objects. It allows to represent structured information as XML allows for example. 23

JUnit

JUnit is an implementation of Unit testing on Java application. “JUnit is a simple framework to write repeatable tests”4. 13, 45

Maven

Apache Maven is a tool for managing and automating the production of Java software projects in general and Java EE in particular. The aim is similar to the make system under Unix: producing software from its sources, optimizing the tasks carried out for this purpose and guaranteeing the correct production order. 27

Object-relational mapping

Object-relational mapping is a computer programming technique that creates the illusion of an object-oriented database from a relational database by defining correspondences between this database and objects of the language used. 47, 55

ORM

Object-relational mapping. 47, Glossary: Object-relational mapping

PBT

Property-based testing. iii, 5, 6, 17, 19–21, 29, 32, 44, 48, 49, 56, Glossary: Property-based test

Property-based test

A method of testing that goes beyond simple unit tests by checking a property instead of a value. iii, 5, 6, 19, 29, 32, 44, 55


4http://junit.org/junit4/, accessed: 22/02/2017
QuickCheck

QuickCheck is a PBT tool. iii, 1, 5, 17–21, 29, 30, 32, 33, 35, 39, 41, 42, 44–46, 48, 49

Regular Expression

A regular expression is a string of characters that describes, in a precise syntax, a set of possible strings. 23, 29, 37

Static analysis

Static analysis allows the user to get informations about the behaviour of a software while executing it, but without executing it. Most of the time, it is done by an IDE. 12

Syntax checking

Syntax checking allows the user to detect syntax errors on her code at editing time, and can even propose better solutions for outdated methods. 12

TDD

Test-driven development. 1, 2, 4, 6, 11–13, Glossary: Test-driven development

Test-driven development

Test-driven development is a software development technique that advocates writing unit tests before writing the source code of a software. 1, 6, 56

UML

Unified Modeling Language. 7–9, 24, 29, Glossary: Unified Modeling Language

Unified Modeling Language

The Unified Modeling Language is a graphical modeling language based on pictograms designed to provide a standardized method for visualizing the design of a system. 7, 24, 56

Unit test

“Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation”5. iii, 6, 12, 15, 54, 55

5http://searchsoftwarequality.techtarget.com/definition/unit-testing, accessed: 22/02/2017
User Experience

User experience is a concept that emerged in the 2000s in an attempt to qualify the result (benefit) and the feeling of the user (experience) during a manipulation of a functional object or a human-machine interface (via a user interface). 57

UX

User Experience. *Glossary: User Experience*

VCS

Version Control System. 54, *Glossary: Version Control System*

Version Control System

A version control system is a software that allows you to store a set of files while keeping the chronology of all the changes that have been made on it. It allows to find the different versions of a batch of related files. 54, 57