An Automated Assistant for Reducing Duplication in Living Documentation

A dissertation submitted to the University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences.

2015

SIA WAI SUAN

SCHOOL OF COMPUTER SCIENCE

Table of Contents

List of Figures 6

List of Tables 9

List of Abbreviations 10

Abstract 11

Declaration 12

Intellectual Property Statement 13

Acknowledgements 14

Chapter 1 : Introduction 15

1.1 Overview 15

1.2 Aim 17

1.2.1 Objectives 17

1.3 Dissertation Outline 17

Chapter 2 : Background 18

2.1 Overview 18

2.2 Requirement Engineering (RE) 18

2.3 Specification by Example (SbE) 19

2.4 Acceptance Test-Driven Development (ATDD) 24

2.4.1 ATDD Exemplar: FitNesse 26

2.5 Behaviour-Driven Development (BDD) 28

2.5.1 BDD Exemplar: Cucumber 30

2.6 Conclusion 35

Chapter 3 : Duplication Detection and Analysis 36

3.1 Overview 36

3.2 Why BDD & Cucumber? 36

3.3 What is Duplication in BDD? 42

3.3.1 Good vs. Bad Duplication 42

3.4 Importance of Reducing Duplication in BDD 45
3.5 Near-duplicates 46
3.6 Rules for Detecting Duplication 47
3.6.1 Rule 1 48
3.6.2 Rule 2 49
3.6.3 Rule 3 49
3.6.4 Rule 4 50
3.6.5 Decision Tree 50
3.7 Duplicate Detection Algorithm 51
3.7.1 Exact Duplicate Detection 51
3.7.2 Near-Duplicate Detection 52
3.7.3 Model of Duplication 56
3.7.4 Process Flow 57
3.7.5 Limitations 61
3.8 Refactoring 61
3.9 Methodology 64
3.9.1 Information Gathering 64
3.9.2 Development 65
3.9.3 Deliverables 65
3.9.4 Evaluation 66
3.10 Conclusion 66
Chapter 4 : Realizing the SEED tool 67
4.1 Overview 67
4.2 Software Design 67
4.2.1 Eclipse Plug-in Architecture 67
4.2.2 SEED Architecture 68
4.2.2 Domain Model Diagram 71

4.2.3 Class Diagram......72
4.3 Version Control System......73
4.4 Functionality......74
4.4.1 Duplicate Cucumber Feature Titles Detection......74
4.4.2 Duplicate Cucumber Scenario Titles Detection......76
4.4.3 Duplicate Cucumber Scenario Steps Detection......77
4.4.4 Duplicate Cucumber Examples Table Rows Detection......79
4.4.5 Duplicate Cucumber Steps Detection......79
4.4.6 Duplicate Cucumber Pre-condition Steps Detection......81
4.4.7 Duplicate Cucumber Scenarios Detection......83
4.5 Testing......83
4.5.1 Unit Testing with Xtext......84
4.6 Conclusion......86
Chapter 5 : Evaluation......87
5.1 Overview......87
5.2 Hypothesis & Prediction......87
5.3 Approach......88
5.4 Experiment 1......90
5.4.1 Duplication......91
5.4.2 Refactoring......93
5.5 Experiment 2......94
5.5.1 Duplication......94
5.5.2 Refactoring......96
5.6 Experiment 3......97
5.6.1 Duplication......98
5.6.2 Refactoring......99
5.7 Results & Discussion......99

5.8 Conclusion 103
Chapter 6 : Conclusion 104
6.1 Future Work 105
Bibliography 106
Appendix A: Gherkin Grammar (Gherkin.xtext) 109

Word Count: 22825

List of Figures

Figure 2.1: User story structure. 19
Figure 2.2: Key process patterns of SbE. 20
Figure 2.3: Using examples to drive a conversation between customer, developer, and tester. 21
Figure 2.4: Refined examples. 22
Figure 2.5: Refined specification. 22
Figure 2.6: An overview of the automation architecture. 23
Figure 2.7: An example of an acceptance test. 25
Figure 2.8: The process of ATDD and its relationship with TDD. 26
Figure 2.9: FitNesse decision table. 27
Figure 2.10: Decision table in markup format. 27
Figure 2.11: Fixture code (in Java). 28
Figure 2.12: Executed acceptance test. 28
Figure 2.13: The language of BDD. 29
Figure 2.14: Full example of a BDD Story. 30
Figure 2.15: Describing a software feature (in Gherkin) using Cucumber. 31
Figure 2.16: An example of Cucumber's Background. 31
Figure 2.17: An example of Cucumber's Scenario Outline. 32
Figure 2.18: An example of potential Scenario Outline candidates. 32
Figure 2.19: An example of Cucumber's Doc String. 33
Figure 2.20: An example of Cucumber's Data Table. 33
Figure 2.21: An example of using descriptions in Cucumber. 33
Figure 2.22: An example of a step definition (written in Ruby). 34
Figure 2.23: Executed Cucumber feature file. 34
Figure 3.1: An example of a repeated event condition in scenarios. 43
Figure 3.2: An example of duplicated scenario descriptions. 43
Figure 3.3: An example of duplicated feature titles. 44
Figure 3.4: An example of a repeated pre-condition in scenarios. 44
Figure 3.5: Side-by-side comparison of bad and good Cucumber features. 45
Figure 3.6: An example of (semantically) equivalent steps. 47
Figure 3.7: A high level outlook of the goal of the project. 48
Figure 3.8: Decision Tree Diagram. 51
Figure 3.9: A working example of the Dice Coefficient algorithm. 53
Figure 3.10: Duplication Model in SEED. 57
Figure 3.11: Flow of Duplication Detection. 59

Figure 3.12: Abstract syntax tree produced by the parser after parsing a Cucumber feature.60
Figure 3.13: An example of quick fixes on Eclipse. 62
Figure 3.14: SEED's quick fixes. 62
Figure 3.15: Example of renaming duplications. 63
Figure 3.16: Example of moving pre-condition steps to the background. 63
Figure 3.17: Example of combining scenarios into a scenario outline. 64
Figure 3.18: Flow of refactoring. 64
Figure 4.1: Overview of Eclipse plug-in architecture. 68
Figure 4.2: A simple plug-in that adds a new item to the menu. 68
Figure 4.3: Architecture of the SEED plugin. 69
Figure 4.4: Detailed outlook of SEED's architecture. 71
Figure 4.5: Domain Model of SEED. 72
Figure 4.6: Class Diagram. 73
Figure 4.7: Pseudocode for detecting duplicate Cucumber feature titles. 75
Figure 4.8: Two different feature files with the same title. 75
Figure 4.9: Two different feature files with similar titles. 75
Figure 4.10: Pseudocode for detecting duplicate Cucumber Scenario titles. 76
Figure 4.11: Two scenarios with the same title/description. 77
Figure 4.12: Two scenarios with equivalent title/descriptions. 77
Figure 4.13: Pseudocode for detecting duplicate list of Cucumber steps. 78
Figure 4.14: Two different scenarios having similar list of steps. 78
Figure 4.15: Pseudocode for detecting duplicate Examples table rows. 79
Figure 4.16: Repeated rows in Examples table. 79
Figure 4.17: Pseudocode for detecting duplicate Cucumber steps. 80
Figure 4.18: Exact and equivalent steps detected. 80
Figure 4.19: Pseudocode for detecting steps that already exist the Background section. 81
Figure 4.20: Exact and equivalent steps already exist in the background section. 81
Figure 4.21: Pseudocode for detecting pre-condition steps repeated in every scenario. 82
Figure 4.22: Pre-condition steps repeated in every scenario of the feature. 82
Figure 4.23: Pseudocode for detecting similar scenarios. 83
Figure 4.24: Scenarios that differ in their input/output values can be combined into a scenario outline (through refactoring/quick fix). 83
Figure 4.25: Unit test code for detecting duplicate scenario titles. 85
Figure 4.26: JUnit test cases for SEED. 85
Figure 4.27: Expected vs. actual results from a failed unit test. 86
Figure 5.1: Classification of evaluation results. 88
Figure 5.2: History of Git commits. 89

Figure 5.3: Coverage of Duplication Detection in Experiment 1. 92
Figure 5.4: snippet_search.feature 95
Figure 5.5: Coverage of Duplication Detection in Experiment 2. 96
Figure 5.6: Coverage of Duplication Detection in Experiment 3. 98
Figure 5.7: Results for Experiment 1. 101
Figure 5.8: Results for Experiment 2. 101
Figure 5.9: Results for Experiment 3. 102
Figure 5.10: SEED deployed on Eclipse Marketplace. 103

List of Tables

Table 3.1: ATDD Tools. 39
Table 3.2: BDD Tools. 40
Table 3.3: FitNesse Support Tools. 40
Table 3.4: Cucumber Support Tools. 41
Table 3.5: Dice coefficient for syntactically and semantically equivalent statements. 54
Table 3.6: String similarity-matching algorithms ran against syntactically and semantically equivalent statements. 56
Table 5.1: Total amount of feature files and commits in the Cucumber project. 90
Table 5.2: Count of duplications & errors detected within the Cucumber features. 91
Table 5.3: Count of duplications detected. 91
Table 5.4: Count of refactorings suggested/done. 94
Table 5.5: Total amount of feature files and commits in the Gitlab project. 94
Table 5.6: Count of duplications & errors detected within the Cucumber features. 94
Table 5.7: Count of duplications detected. 95
Table 5.8: Count of refactorings suggested/done. 97
Table 5.9: Total amount of feature files and commits in the RadiantCMS project. 97
Table 5.10: Count of duplications & errors detected within the Cucumber features. 98
Table 5.11: Count of duplications detected. 98
Table 5.12: Count of refactorings suggested/done. 99

List of Abbreviations

TDD	Test-Driven Development
ATDD	Acceptance Test-Driven Development
BDD	Behaviour-Driven Development
SbE	Specification by Example
RE	Requirement Engineering

Abstract

Acceptance Test-Driven development and Behaviour-Driven development are two software development methodologies that help software development teams to write better requirements specifications and to allow customers to convey their needs easily. Automation tools have emerged to automate this process and allow the specifications to be executed as acceptance tests for the software. However, these tools have drawbacks that lead to difficulty in maintaining the specifications. Users of the tools are prone to mistakes such as repeating test scenarios since the writing of the tests is not automated.

Duplication is an issue that stems from writing the tests manually. Test suites can grow to a large scale as a software development project progresses. It is easy to create duplication in large tests unknowingly. However, removing/searching for them is not as easy. By allowing duplicates to occur and stay, maintenance and readability issues would arise. This affects both the development team and the customers.

This project aimed to reduce duplication in BDD specifications by experimenting with the Cucumber BDD automation tool. On top of that, the project also delivered a plugin tool to not only detect exact- and near-duplication in these specifications but also provide helpful refactoring diagnostics for them. Evaluation results showed that the plugin tool was able to not only cover the duplicates detected by human experts but also duplicates that went undetected.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual Property Statement

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library's regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University's Guidance for the Presentation of Dissertations.

Acknowledgements

I would like to thank my supervisor, Dr. Suzanne M. Embury, for her guidance and help throughout this project.

I would also like to express my gratitude to my family for their continuous support throughout the duration of my studies.

Chapter 1 : Introduction

1.1 Overview

In every software development project, there is a requirements gathering process. The main agenda of this phase is to determine, analyse, and document the system requirements through a series of discussions with the stakeholders/customer [1]. The output of this phase is a set of documents called the *requirements specification*. The development team would then refer to these specifications for building the system.

The requirements documents, however, need to be interpreted by a human to gain an understanding of the software's expected behaviour. This leaves room for misinterpretation and misunderstanding. As a result, rework may have to be done later when the software does not meet the customer's needs.

Specification by example (SbE) is an approach to this problem. It was devised by Martin Fowler in 2002 [2]. In SbE, the requirements of the system are represented as *examples*. The examples depict the software's functionality in the form of specific scenarios. It encourages frequent collaboration and communication between the customers, developers, and testers of the system such that each participant of the discussion has a shared understanding on what needs to be done. This helps to reduce ambiguity when it comes to interpreting the requirements specification.

To make validating the system through the examples less tedious and much quicker, automated testing tools have been introduced. The tools make the specification executable without modifying its original content. The execution of an example reports success if the software under construction implements the behaviour described in the example, and reports failure if it does not. The results from execution could not only trigger further discussions with the customer(s) on clarifying the requirements but also help keep the system in line with the specification – something that passive/typical documentation tends to fail at. By making the specification a single source of truth on the requirements of the system and automating the validation of the example(s) on the software under construction, the requirements specification becomes a living documentation [3].

Acceptance Test-Driven Development (ATDD) and Behaviour-Driven Development (BDD) are two software development methodologies that were introduced based on SbE/living documentation. They are extensions to SbE [4]. ATDD promotes the

practice of creating necessary acceptance tests/executable examples before coding [5]. The key concept here is not to create the tests but to create the necessary tests such that the development team would only need to write the required code around these tests. BDD promotes an easy-to-understand and common language when writing customer examples of the required software behaviour [6]. A non-technical person can not only understand these examples but also write them if necessary.

This project aimed to identify a challenging aspect of writing Cucumber tests/features and to create a software tool that could identify problems and propose remedies for them. The project began by surveying existing literature and tool base to identify the problems that users are currently seeing in their Cucumber test suites.

Initial survey of the various existing ATDD/BDD automated testing tools revealed that they share a common issue in that users of these tools have to put effort into writing consistent and maintainable tests. Writing the tests with these tools is, unfortunately, not automated and is highly dependent on the user of the tool. Thus, this process is prone to human mistakes. An example of this is unknowingly creating duplication in an acceptance test case. With large scale test suites, it gets increasingly difficult to manually identify duplication in the tests. Duplication is a problem because the specification then becomes difficult to maintain and read.

We selected BDD and the Cucumber tool for this project and for researching the duplication problem. The project included establishing a definition for duplication in BDD specifications and identifying viable refactoring solutions for this form of detected duplication. A plugin tool was then developed to incorporate the results of the research.

The tool is founded on the assumption that it detects the same duplications and fixes them in the same/similar way as expert writers of Cucumber tests do. The evaluation of the tool was carried out to prove whether the behaviour of the tool corresponded to the actions of Cucumber experts. The evaluation results showed that the plugin tool was able to capture the duplicates and refactor them accordingly albeit with further evaluation required on duplicates that were not detected by Cucumber experts.

1.2 Aim

The aim of the project is to determine whether a software tool is able to detect the same or similar duplication in BDD/Cucumber test cases as a human expert does, as well as providing the same or similar suggestions for refactoring the duplication.

1.2.1 Objectives

1. Compile a list of existing ATDD/BDD tools and identify their aims and drawbacks.
2. Invent a set of rules that denotes whether duplication exists in a BDD/Cucumber test.
3. Develop a plugin tool with the intent of highlighting duplication within Cucumber specifications and providing refactoring suggestions for them.
4. Evaluate the quality of the tool and how well it solves the problem.

1.3 Dissertation Outline

The report is structured as follows:-

- Chapter 2: Background – This chapter discusses in detail the concept of requirement engineering, SbE, ATDD, and BDD and how they differ from one another.
- Chapter 3: Duplication Detection and Analysis – This chapter gives an in-depth review of the duplication problem as well as motivating our focus on BDD and Cucumber in the project. The methods used for detecting and refactoring duplications are also discussed in this chapter.
- Chapter 4: Realizing the SEED tool – This chapter examines the overall architecture of the plugin tool and its implementation details.
- Chapter 5: Evaluation – This chapter describes how we have evaluated the plugin tool and provides the final results of the project.
- Chapter 6: Conclusion – This chapter summarizes the work done and lessons learned in the project as well as describes potential future work.

Chapter 2 : Background

2.1 Overview

This chapter discusses the background research done in the project. It starts by discussing the requirement engineering process from an agile perspective. Then, the discussion will branch out into SbE, ATDD, and BDD, where it is shown that they share the same goals but still have their own distinct features.

2.2 Requirement Engineering (RE)

Requirement engineering [7] [8] [9] is an essential part of software engineering. It serves as the foundation for building the right software product. The purpose of requirement engineering is to gather the needs of the software from the customer/stakeholders, analyse them, and document them as necessary. A document containing the software requirements is known as a *requirements specification*.

In an agile software development process, requirements are gathered from the customer iteratively [10]. A general overview of the software functionality is first established at the start of the project by the customer. It does not have to cover every single aspect of the software as it is meant to serve as a starting point of the project. This information is then documented in the form of *user stories*. As shown in Figure 2.1, a user story is a short description of a piece of the software's functionality. It can be written on a piece of paper such as a card or a post-it note. The user stories are then prioritized (by the customer) according to their importance before moving forward with implementing them. The development team then selects a subset of user stories to work on for the upcoming iteration. At the beginning of each iteration, detailed requirements and clarifications are gathered from the customer on the user stories that have to be implemented before the end of the iteration. An iteration runs for approximately 1 - 4 weeks. [1]

---

[1] *Iteration* [Online]. Available at: http://guide.agilealliance.org/guide/iteration.html

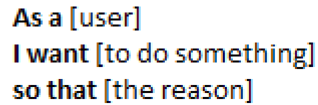

Figure 2.1: User story structure.

Throughout the development timeline, new user stories might be added and existing ones might be altered. Part of the agenda of agile is to promote customer collaboration within software development projects. [2] Therefore, the customer is constantly involved in the development process. The process includes clarifying requirements, writing acceptance tests, and getting feedback/validation on the implemented software features. The benefit from this is that the need for major re-work after the software has been delivered to the end users is reduced [11].

However, due to the simplistic nature of user stories, they do not contain enough information to formally represent the software requirements. There is still ambiguity and uncertainty surrounding it. A user story mainly triggers further communication and discussions between the customer and the software development team. These discussions take place in order for the developers to get a better understanding of how the software needs to function.

It should be obvious by now that the clarity of the software requirements is dependent on the communication with the customer. The developers ask the questions whilst the customer does the answering. It is essential for the requirements to be as accurate as possible in order to reduce the risk of building the wrong software i.e. something that the customer does not want. However, customers can sometimes have trouble conveying their business objectives/what they want clearly [11]. Of course, getting frequent feedback from the customer throughout the iteration helps keep the development on the right track. Still, this can be improved upon.

2.3 Specification by Example (SbE)

Specification by Example helps software development teams to build the right software from the start of the project/iteration [12]. An illustration of the SbE process

---

[2] *The Agile Manifesto* [Online]. Available at: http://www.agilealliance.org/the-alliance/the-agile-manifesto/

is shown in Figure 2.2 (taken from G. Adzic [3]). Building it right from the start reduces the need for re-work during and after an iteration. Re-work leads to unnecessary delays in the development time. SbE does this by reducing the communication gap between the customer and development team when it comes to understanding software requirements. Questions and clarifications can still come up during development but there is less back and forth for feedback between the customer and the development team. This leads to shorter iterations and reduced (overall) development time.

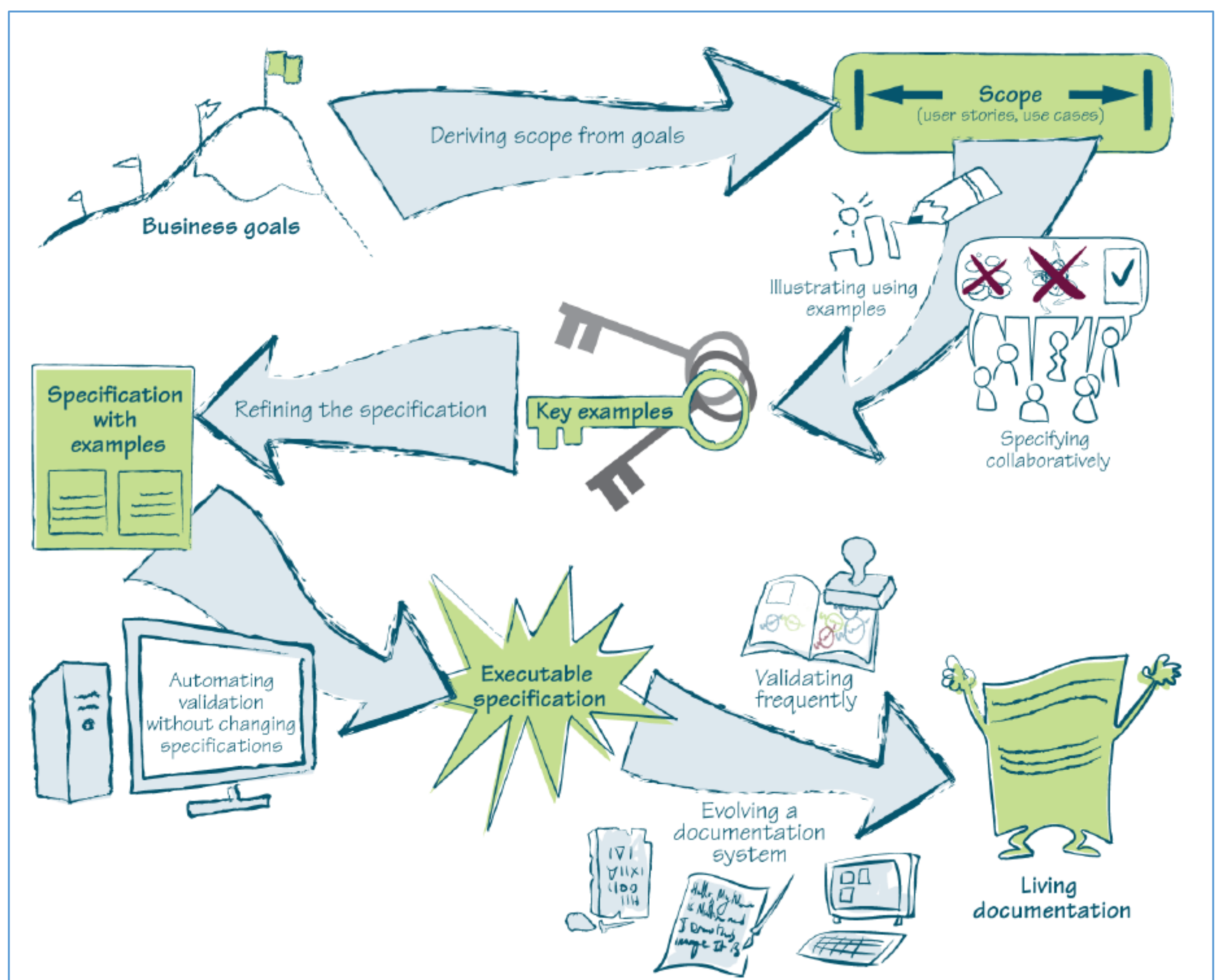

Figure 2.2: Key process patterns of SbE.

In SbE, as we have said, examples are used when describing software requirements [3]. An example in SbE is a concrete and unambiguous description of the behaviour of a software feature. Figure 2.3 (taken from G. Adzic [3]) gives an insight into what an example in SbE looks like. *Illustrating using examples* is a way of preventing misinterpretation/misunderstanding of the software requirements amongst the

different participants of the software development project (e.g. customer, testers, developers, and business analysts). As mentioned previously, misinterpretation leads to re-work. During the discussions before an iteration begins, developers and testers are able to utilize examples as a way of flushing out inconsistencies and edge cases of the software feature under development. It is also a way of gaining a shared understanding of what needs to be done. In addition, the examples are used to drive the entire development process, using (as we shall see) ATDD processes.

BARBARA: The next thing on the list is free delivery. We have arranged deal with Manning to offer free delivery on their books. The basic example is this: If a user purchases a Manning book, say *Specification by Example*, the shopping cart will offer free delivery. Any questions?

David, a developer, spots a potential functional gap. He asks: Is this free delivery to anywhere? What if a customer lives on an island off South America? That free delivery will cost us much more than we earn from the books.

BARBARA: No, this isn't worldwide, just domestic.

Tessa, a tester, asks for another example. She says: The first thing I'd check when this comes for testing is that we don't offer free delivery for all books. Can we add one more case to show that the free delivery is offered only for Manning books?

Figure 2.3: Using examples to drive a conversation between customer, developer, and tester.

Building the right software is a team effort and requires co-operation from both the customer and the development team. Customers are not software designers [3]. Therefore, it is unfair to expect them to cover the entire scope of the software (e.g. user stories and use cases) whilst the development team handles the implementation side of things. The result of this is an end product with functional gaps and a lack of customer satisfaction. Instead, the development team should collaborate with the customer in establishing the scope of the project. The customer is in charge of conveying their business goals. The team would then come up with efficient/feasible solutions that help achieve these goals. This is what it means by *deriving scope from goals*.

As for *specifying collaboratively*, people from various domain-specific backgrounds and knowledge work together in getting the software requirements right. Developers use their experience in technology to locate functional gaps within the requirements. Testers are able to specify the potential issues of certain software features. Inputs and

opinions from different expertise help make the requirements more specific and accurate. This leads to a more refined set of software requirements.

After the set of examples have been listed out by the customer, the next step for the team would be to clean them up. The examples described by the customer tend to be from a user perspective. This means that they contain user interface details such as clicking buttons and links [3]. These extra details show how the software works when the examples are meant to identify what it needs to do. Keeping them may obscure the key information of an example and therefore, needs to be removed. As the examples are used to guide development and testing, it is important that they remain unambiguous and precise. Figure 2.4 (taken from G. Adzic [3]) shows what refined examples look like. They can be further organized into what is shown in Figure 2.5 (taken from G. Adzic [3]). The requirements specification is then represented by these refined examples. This falls under the *refining the specification* stage of the SbE process.

Key Examples: Free delivery

- VIP customer with five books in the cart gets free delivery.
- VIP customer with four books in the cart doesn't get free delivery.
- Regular customer with five books in the cart doesn't get free delivery.
- VIP customer with a five washing machines in the cart doesn't get free delivery.
- VIP customer with five books and a washing machine in the cart doesn't get free delivery.

Figure 2.4: Refined examples.

Free delivery

- Free delivery is offered to VIP customers once they purchase a certain number of books. Free delivery is not offered to regular customers or VIP customers buying anything other than books.
- Given that the minimum number of books to get free delivery is five, then we expect the following:

Examples

Customer type	Cart contents	Delivery
VIP	5books	Free,Standard
VIP	4books	Standard
Regular	10books	Standard
VIP	5washingmachines	Standard
VIP	5books,1washingmachine	Standard

Figure 2.5: Refined specification.

As mentioned, the examples are frequently used to validate the software. This means that the team would refer to them as a way of making sure that they are not getting off track from the software requirements during development. However, doing this manually is tedious, slow, and prone to human error. SbE introduces the concept of *automating validation without changing specifications* as a solution to this problem. The examples themselves are used as acceptance tests for the software [3] and automation makes them *executable*. There are various existing tools that help with the automation process. Figure 2.6 illustrates an overview of how the tool relates to the examples/specification and the software. The tools work by sending inputs from the executed examples to the production code and comparing the outputs (returned by the production code) against the expected outputs in the examples [3]. An executed example reports success if the software under construction implements the behaviour described in the example, and reports failure if it does not. The support code is part of the tool and is there to connect the examples to the production code (i.e. the software). It is important to note that the developers will have to write this support code. These tools can be divided into two classifications – Acceptance Test-Driven development (ATDD) and Behaviour-Driven Development (BDD). They will be further explored in the following sections. It is important that the examples are not altered in any way through the process of automation [3]. They should remain understandable and accessible to the customer and the development team. This is so that the purpose of creating these examples in the first place is not defeated.

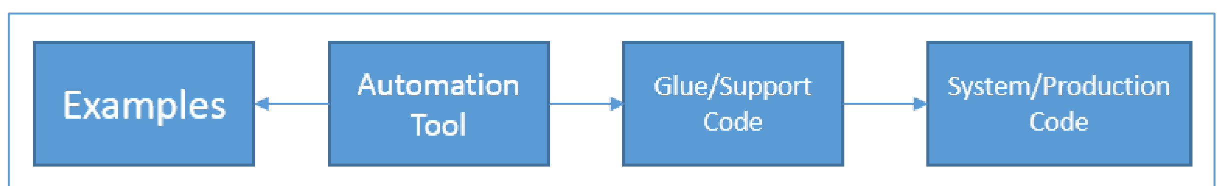

Figure 2.6: An overview of the automation architecture.

Now that the specification is executable, the team can proceed with *validating the software frequently*. Aside from making sure that development is on the right track of meeting the software requirements, the benefit of this is also to keep the specification in-line with the production/software code. The team is able to see if things get broken due to the changes made to both the document and the code. Apart from missing functionality/behaviour, a failing example could also indicate that something is broken in the software under development. This allows the customer and the team to gauge

the progress of the project. The goal here is to ensure that all of the examples pass by the end of the project timeline.

In the end, the specification becomes a *living document*. "Living documentation is a reliable and authoritative source of information on system functionality that anyone can access." (p. 24 [3]). It acts as a single source of truth during the development process. It gets updated when there are changes. Developers refer to it as guidance during development. Testers use the examples to aid in testing. Customers use it to measure whether the software is complete. This is what is meant by *evolving a documentation system* in SbE.

2.4 Acceptance Test-Driven Development (ATDD)

ATDD is a software development technique that helps the customer and development team know when a software requirement has been completed. This is done by capturing the software requirements in the form of *acceptance tests*. The acceptance tests are discussed and written before doing any coding. [3] An acceptance test works the same way as an executable example whereby it has a set of input and expected output values [13]. The expected output values denote whether the test passes or fail. Figure 2.7 (taken from G. Adzic [3]) shows an example of an acceptance test where the input and output values are arranged in a tabular format. An acceptance test is created from a user story. A user story can have many acceptance tests. Each test represents a specific scenario in the story. A user story is complete only when all of its acceptance tests pass. [4] Since the user stories represent the software requirements, the project can only be considered complete when all of its user stories have been completed.

---

[3] *Acceptance Test-Driven Development* [Online]. Available at: http://www.netobjectives.com/acceptance-test-driven-development

[4] *Acceptance Tests* [Online]. Available at: http://www.extremeprogramming.org/rules/functionaltests.html

The prize pool is divided among the winners using the following distribution for winning combinations (number of correct hits out of six chosen numbers). Example below is for $2M payout pool.

Prize Distribution for Payout Pool	2,000,000	
Winning Combination	Pool Percentage?	Prize Pool?
6	68	1,360,000
5	10	200,000
4	10	200,000
3	12	240,000

Figure 2.7: An example of an acceptance test.

Before going further, it is important that Test-Driven development (TDD) be briefly discussed. As the TDD process is contained within the ATDD process, it is therefore, a test-driven technique. TDD is a software development methodology that encompasses three steps. First, the developer writes a failing automated unit test for a particular software feature. A unit test is a low-level test case focusing on the functionality of a single unit of the system (typically, taken to be a class in object-oriented programming) i.e. ensuring that the production source code works as intended [14]. Second, the developer writes as little code as possible to make the test pass. Third, the code is refactored to remove code smells. This means removing duplicate code and hardcoded data amongst other things [15]. These three steps are then repeated for another unit of the system/software feature. The main benefit of this is to keep the number of coding errors low. If they do show up and cause a test to fail, it is easy to locate them since minimal code was changed since the tests last passed, and the error is likely to be found in these changes. TDD aims to ensure that the software's technical quality is maintained [13] such as well-written code and minimal software bugs.

ATDD uses TDD to build code in order to implement a single complete feature by involving multiple units/classes. It works at the level of a single software feature (instead of a unit as TDD does), as described by a set of acceptance tests. An acceptance test passes when the functionality it describes is implemented. The unit tests pass when the functionality that they collectively describe is implemented. According to Figure 2.8 (taken from L. Koskela [13]), it can be depicted that an acceptance test passes when all of its unit tests passes. Once an acceptance test passes, the ATDD process is then repeated for the next set of software features/functionality.

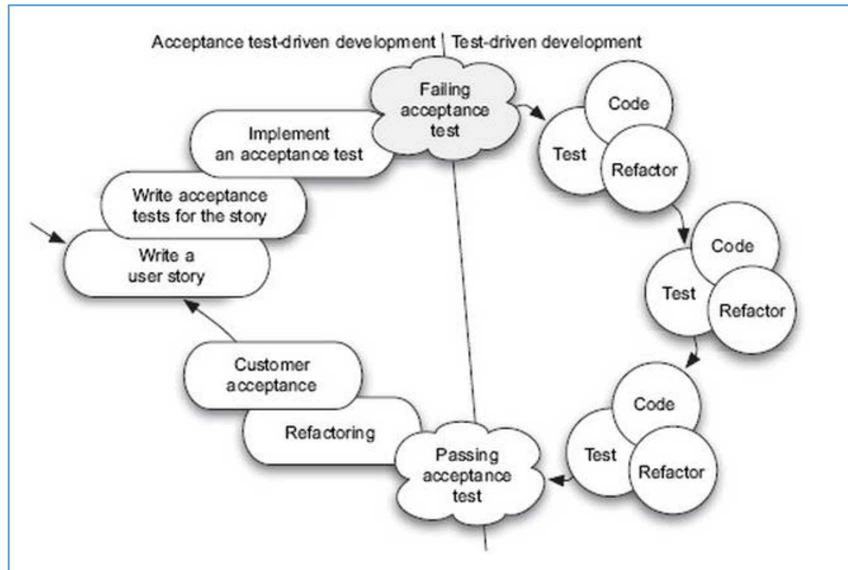

Figure 2.8: The process of ATDD and its relationship with TDD.

Like SbE, the acceptance tests can be automated/made executable through the support of automation tools. Automation allows the team to get quick feedback on whether a test passes or fails. It also makes regression testing of the software easier. This is to ensure that the changes made are error-free. The FitNesse[5] tool will be used to help illustrate how an ATDD automation framework works and what it looks like.

2.4.1 ATDD Exemplar: FitNesse

FitNesse is an automation testing tool that supports the writing of acceptance tests through a wiki.

In FitNesse, an acceptance test is expressed in a tabular format. An acceptance test table, in FitNesse, is referred to as a *decision table*[6]. Figure 2.9 shows an example of a "division operation" acceptance test written in the mentioned format. Each row in the table represents a scenario of the acceptance test and is read from left to right. Within each scenario, there are input and expected output values that will be used during the execution of the test. In this case, the "numerator" and "denominator" columns represent the inputs whereas the "result?" column represents the expected outputs (this is signified by the question mark suffix in the column name). For the first

---

[5] *FitNesse* [Online]. Available at: http://fitnesse.org/
[6] *Decision Table* [Online]. Available at: http://fitnesse.org/FitNesse.UserGuide.WritingAcceptanceTests.SliM.DecisionTable

row in the table, the scenario reads as follows: "If given a numerator of 10 and a denominator of 2, the result should be a 5".

Division		
numerator	denominator	result?
10	2	5.0
4	2	2.0
9	3	4.0

Figure 2.9: FitNesse decision table.

In FitNesse, creation of the decision tables is done by using the *markup* language. An example of this is shown in Figure 2.10. The vertical bars are required as delimiters for the table cells. [7]

```
||Division                               |
|numerator|denominator|result?|
|10        |2           |5.0    |
|4         |2           |2.0    |
|9         |3           |4.0    |
```

Figure 2.10: Decision table in markup format.

Before a FitNesse acceptance test can be executed, the glue code has to be written (by the developers) in order to pass values from the decision table to the production code (i.e. the software under development). In FitNesse, glue code is referred to as *fixture code*[8]. An example of some fixture code is shown in Figure 2.11. When an acceptance test is executed, FitNesse will attempt to map the decision table header to a fixture class name. Then, it maps the column headers to their respective fixture method names. The column headers representing the input values are mapped with *setter* method names i.e. has a "set" in front of the header name. The setter methods are in charge of setting the input values before processing them for results. Also, following code conventions, each header name is replaced with camel-casing whereby spaces are moved from the name, and the first letter in every word of a header name is

[7] *An Example FitNesse Test* [Online]. Available at: http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample
[8] *Fixture Code* [Online]. Available at: http://fitnesse.org/FitNesse.FullReferenceGuide.UserGuide.WritingAcceptanceTests.FixtureCode

capitalized. A column header ending with a question mark will be interpreted as an expected output value coming from its respective fixture method name (is not a setter method).

Table header.

```java
public class Division {
    private double num;
    private double denom;

    public void setNumerator(double numerator) {
        num = numerator;
    }

    public void setDenominator(double denominator) {
        denom = denominator;
    }

    public double result() {
        return Calculator.divide(num, denom);
    }
}
```

numerator column (input).

denominator column (input).

result? column (output).

Production code.

Figure 2.11: Fixture code (in Java).

After the fixture code has been written, the acceptance test is ready to be executed using FitNesse. Figure 2.12 shows the result of the test after it has been executed. The green cells indicate that the values returned from the production code match the expected values whereas the red cell indicates otherwise. To ease developers in debugging failed tests, red cells are accompanied by the actual values returned. In the figure shown, the cell was expecting "4.0" but it received "3.0" instead.

Division		
numerator	denominator	result?
10	2	5.0
4	2	2.0
9	3	[3.0] expected [4.0]

Figure 2.12: Executed acceptance test.

2.5 Behaviour-Driven Development (BDD)

BDD is a software development methodology created by Dan North [5] that "uses natural language to capture customer examples in a domain-specific language" [16]. The idea for BDD stemmed from the original creator's frustrations with TDD. These frustrations include figuring out what to test, how much testing was involved, and where to begin testing [5] [17]. Therefore, the BDD framework was developed to address these questions. In BDD, customer examples are known as *scenarios* and tests

are known as *behaviours* of the software[9] . The scenarios represent examples of the software behaviour when given different situations.

BDD introduced a common language for writing examples, software requirements, and (acceptance) tests. This language is the core of BDD. Its purpose is to reduce miscommunication between the customer and development team since everyone within the discussion is using the same terminology. It also removes ambiguity when it came to describing the software requirements or the behaviour of a software feature. This ensures that everyone participating in the discussion has a collective understanding on the customer's needs of the software. In order for that to happen and as shown in Figure 2.13, the language had to not only be natural enough such that the customer can easily understand it but also structured in a way that they can be automated by automation tools [5]. Figure 2.14 (taken from D. North [18]) shows an example of the BDD's *given-when-then* expressions being used in one of the scenarios in a user story. A scenario represents an example of the behaviour of the software feature. In order to fully specify the story, there can be more than one scenario in a story.

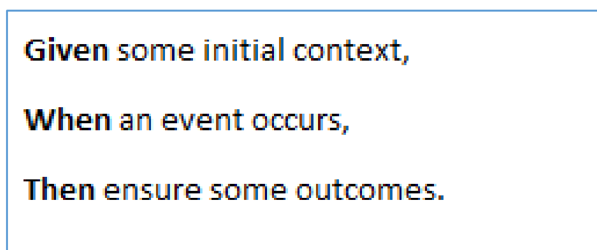

Figure 2.13: The language of BDD.

[9] *Bdd* [Online]. Available at: http://guide.agilealliance.org/guide/bdd.html

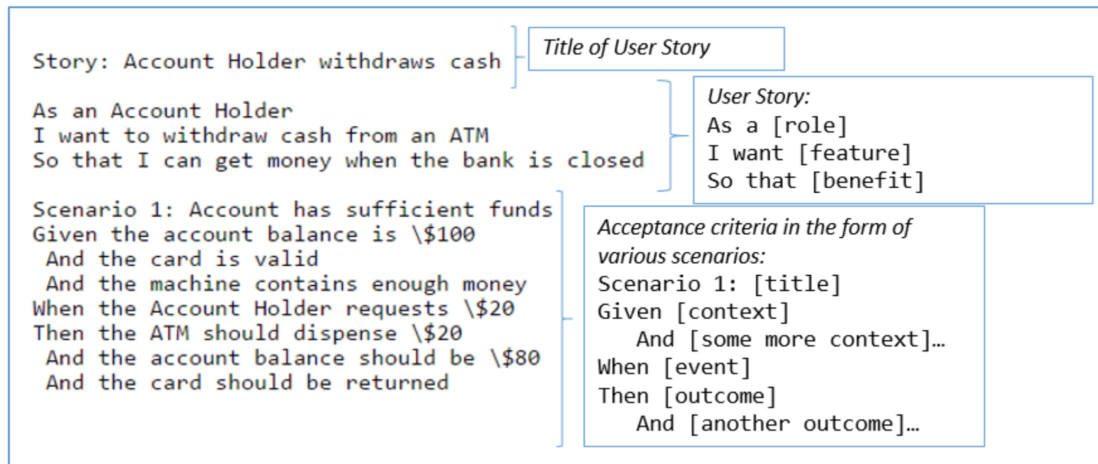

Figure 2.14: Full example of a BDD Story.

Like SbE and ATDD, BDD expressions/examples can be automated. The automation tools that support BDD are not only able to make the customer examples executable but also recognize BDD's language, such as the given-when-then notation. This means that the tool is able to extract the scenario fragments from the user story/specification and parse them as input arguments for the underlying test code. The Cucumber[10] tool will be used to help illustrate how a BDD automation framework works and what it looks like.

2.5.1 BDD Exemplar: Cucumber

Cucumber was developed by Aslak Hellesøy to provide support for customer examples to be written in plain English whilst still being executable. Cucumber refers to BDD's language as *Gherkin*[11].

As shown in Figure 2.15, a user story/software behaviour is referred to as *Feature*[12]. A Cucumber feature is stored in a plain text file with a ".feature" extension so that Cucumber can find and execute it. In Cucumber, each scenario within the feature consists of a list of *steps*[13], also known as *Givens*, *Whens*, and *Thens*,

---

[10] *Cucumber* [Online]. Available at: https://cukes.info/
[11] *Gherkin* [Online]. Available at: https://cucumber.io/docs/reference#gherkin
[12] *Feature* [Online]. Available at: https://cucumber.io/docs/reference#feature
[13] *Steps* [Online]. Available at: https://cucumber.io/docs/reference#steps

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers
    Given I have entered 20 into the calculator
    And I have entered 5 into the calculator
    When I press add
    Then the result should be 25 on the screen
```

Figure 2.15: Describing a software feature (in Gherkin) using Cucumber.

In Cucumber and Gherkin, a feature is allowed to have a *Background*[14] section. Figure 2.16 shows an example of this. It is optional in a feature and is placed before the scenarios and below the feature title. This section collects preconditions (*given* steps) that are required/repeated in all of the scenarios under it into a single location. In turn, this makes each scenario shorter and easier to maintain. However, it may make the feature as a whole less readable.

```
Feature: Search

  Background:
    Given I am logged in as a user

  Scenario: I should see user comments that matches my query
    When I search for "waisuan"
    And I click on the "Search For Comments" button
    Then I should see all of the comments that I have made in the past
    And also comments that have mention of my name

  Scenario: I should see thread titles that matches my query
    When I search for "cucumber"
    And I click on the "Search For Threads" button
    Then I should see all of the thread posts that has the word "cucumber" in it
```

Figure 2.16: An example of Cucumber's Background.

Cucumber introduces additional entities on top of Gherkin. Cucumber allows a *Scenario Outline*[15] section. An example of a scenario outline is shown in Figure 2.17. A scenario outline can be described as a scenario with placeholders in its steps. Placeholders (i.e. variables in a scenario) are contained within the "< >" delimiters. Scenario outline exists to avoid the need for repetitive scenarios which only differ in their input/output values such as is shown in Figure 2.18. Therefore, an *Examples* section is introduced below a scenario outline. This section is essentially a table giving sets of placeholder values. It acts as a way of resolving the repetition problem by

[14] *Background* [Online]. Available at: https://cucumber.io/docs/reference#background
[15] *Scenario Outline* [Online]. Available at: https://cucumber.io/docs/reference#scenario-outline

combining the repetitive scenarios into a single scenario (outline). Their differing input/output values are put into a table instead of separate scenarios.

```
Scenario Outline: feeding a suckler cow
  Given the cow weighs <weight> kg
  When we calculate the feeding requirements
  Then the energy should be <energy> MJ
  And the protein should be <protein> kg

  Examples:
    | weight | energy | protein |
    |    450 |  26500 |     215 |
    |    500 |  29500 |     245 |
    |    575 |  31500 |     255 |
    |    600 |  37000 |     305 |
```

Figure 2.17: An example of Cucumber's Scenario Outline.

```
Scenario: feeding a small suckler cow
  Given the cow weighs 450 kg
  When we calculate the feeding requirements
  Then the energy should be 26500 MJ
  And the protein should be 215 kg

Scenario: feeding a medium suckler cow
  Given the cow weighs 500 kg
  When we calculate the feeding requirements
  Then the energy should be 29500 MJ
  And the protein should be 245 kg
```

Figure 2.18: An example of potential Scenario Outline candidates.

For cases where a step requires more than a single line to convey its intention, the user can rely on *Doc Strings*[16] (Figure 2.19) and *Data Tables*[17] (Figure 2.20). It is important to note that doc strings are written within delimiters consisting of three double-quote marks whereas each cell in a data table is delimited by the "|" character.

---

[16] *Doc Strings* [Online]. Available at: https://cucumber.io/docs/reference#doc-strings
[17] *Data Tables* [Online]. Available at: https://cucumber.io/docs/reference#data-tables

```
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
  """
```

Figure 2.19: An example of Cucumber's Doc String.

```
Given the following users exist:
  | name   | email              | twitter         |
  | Aslak  | aslak@cucumber.io  | @aslak_hellesoy |
  | Julien | julien@cucumber.io | @jbpros         |
  | Matt   | matt@cucumber.io   | @mattwynne      |
```

Figure 2.20: An example of Cucumber's Data Table.

In Cucumber, on the lines following a *feature*, *background*, *scenario*, *scenario outline*, or *examples*, the user is allowed to write any amount of plain text as long as the text does not start with any of Cucumber's keywords (e.g. *Given, When, Then*, *And*, *But*). This piece of text is known as a *Description*[18]. It can be used to provide further details on important aspects of the feature or even for user comments. An example of this is shown in Figure 2.21.

```
Feature: Refund item

  Sales assistants should be able to refund customers' purchases.
  This is required by the law, and is also essential in order to
  keep customers happy.

  Rules:
  - Customer must present proof of purchase
  - Purchase must be less than 30 days ago
```

Figure 2.21: An example of using descriptions in Cucumber.

Once the features have been written, the next step would be to write some code to implement the features. As mentioned previously, the automation process is not complete until the examples have been linked to the production code. This is handled by the glue code which needs to be written by the developers. The same process applies to Cucumber. In Cucumber, glue code is known as *Step Definitions*[19]. A step

[18] *Descriptions* [Online]. Available at: https://cucumber.io/docs/reference#descriptions
[19] *Step Definitions* [Online]. Available at: https://cucumber.io/docs/reference#step-definitions

definition helps translate the Gherkin syntax from the customer examples into the same actions at the code level. Figure 2.22 shows the step definitions that are able to support the feature file from Figure 2.15. Cucumber looks for matching step definitions (written in Ruby) when executing a scenario. Therefore, a step definition is accompanied by a *pattern* statement. Once a match is found, the *code* associated with the step definition is then executed. The code in this case refers to the production code (the application/software itself) which the developers will also have to write. It also indicates whether or not the scenario passes. A passing scenario indicates that the behaviour (of the software under construction) the scenario describes has been implemented/is working correctly. However, a failing scenario indicates otherwise.

Pattern.

```
Given /I have entered (\d+) into the calculator/ do |n|
  @calc.push n.to_i
end

When /I press (\w+)/ do |op|
  @result = @calc.send op
end

Then /the result should be (.*) on the screen/ do |result|
  @result == result.to_f
end
```

Code.

Matches *Given* keywords.

Matches *When* keywords.

Matches *Then* keywords.

Figure 2.22: An example of a step definition (written in Ruby).

After the step definitions and production code have been created, the only thing left to do is to execute the feature file. Figure 2.23 illustrates what is output when the feature is executed using the command line (after Cucumber has been installed). The results are informative enough such that the developers should not have much trouble debugging a failed scenario.

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers                    # features\addition.feature:7
    Given I have entered 20 into the calculator # features/step_definitions/calculator_steps.rb:14
    And I have entered 5 into the calculator    # features/step_definitions/calculator_steps.rb:14
    When I press add                            # features/step_definitions/calculator_steps.rb:18
    Then the result should be 25 on the screen  # features/step_definitions/calculator_steps.rb:22

1 scenario (1 passed)
4 steps (4 passed)
0m0.031s
```

Figure 2.23: Executed Cucumber feature file.

2.6 Conclusion

This chapter has provided the underlying foundation for the project. The concepts and purpose of Software Requirements Engineering, SbE, ATDD, and BDD have been presented. Additionally, FitNesse and Cucumber were introduced to show how ATDD and BDD are used in practice. In the following chapter, an analysis of the project's problem domain will be discussed.

Chapter 3 : Duplication Detection and Analysis

3.1 Overview

Now that the definitions of ATDD and BDD have been established, the discussion can shift towards the research question introduced in Chapter 1 i.e. *can a software tool detect the same duplications in BDD specifications as human experts do and provide refactoring suggestions that human experts think are worth applying?*

The chapter will begin by discussing the reasons for using BDD and Cucumber in this project. The discussion will then proceed with an in-depth review of the project's problem domain i.e. the duplication problem. The algorithms used for detecting and refactoring duplications in BDD specifications are also explained in this chapter. Finally, the methodologies used in this project are briefly discussed.

3.2 Why BDD & Cucumber?

There are various existing ATDD and BDD tools. For the purpose of this project, Cucumber was selected out of the many to help answer the project's research question. However, this was not decided at random. During the initial stages of the project, a survey was done on the existing ATDD and BDD tools. The survey included understanding the purpose of the tool, what makes it unique, and potential issues observed from the author's hands-on use of the tool. The results of this survey have been tabulated into Tables 3.1, 3.2, 3.3, and 3.4. Gathering this data helped narrow down the choices of ATDD/BDD tools for the project.

In Tables 3.1 and 3.2, the "automation language" column refers to the primary programming language used for the tool's glue code and the "test syntax" column refers to the primary structure of a test written with the tool. The "aim(s)/purpose" column was necessary to help understand what makes a particular ATDD/BDD tool different from the others and whether it had an advantageous feature over them. As for the "problem(s) observed" column, the observations were gathered through hands-on experience with the tools. One of the purposes of gathering this information was to gain an insight as to how customer-friendly the tools are. This is an important factor because (in theory, at least) the customers will be using the ATDD/BDD tools as much as the development team will. Therefore, it is best to not turn customers off from them for the reasons mentioned in Section 2.3.

As for Tables 3.3 and 3.4, the findings were limited to the support tools (i.e. extensions) of FitNesse and Cucumber. In order to reduce the scope of ATDD/BDD tools, the project had to shift its focus towards the most popular ATDD and BDD tools i.e., FitNesse and Cucumber, respectively. Gathering the data for these tables was necessary in order to gain further understanding on the limitations of FitNesse and Cucumber as well as why these support tools were needed in the first place. In addition, there was insufficient time to gather information on every existing support tool.

From the findings shown in the tables, most of the ATDD/BDD tools share a similar trait whereby the process of writing the tests is done manually and the test's consistency/structure is not maintained by the tool.

BDD was chosen as the focus for this project due to its nature of being more customer-focused than ATDD [19]. ATDD tends to be more developer-focused. It is geared towards capturing software requirements in the form of acceptance tests which in turn, help drive the development process. BDD cares more about the expected behaviour of the software system rather than testing its implementation details. BDD is also more widely adopted in the software industry [20]. Therefore, it is more beneficial to focus efforts on BDD for the project.

The Cucumber tool was selected for the project because not only is it popular amongst the BDD community but also due to its major feature which is that it could support plain text BDD specifications [21]. This was important since it eases the customer into joining the collaboration process. Since (in theory) Cucumber is optimized for customers (instead of testing/technical staff), it is essential that the tests/features created with the tool remain readable at all times [22].

ATDD Tools					
Tool name	**URL**	**Automation language**	**Test syntax**	**Aim(s)/purpose**	**Problems observed**
FitNesse	http://fitnesse.org/	Java	Decision table	To automate acceptance testing through a wiki.	Difficulty in editing tables (through FitNesse editor). For example, removing a column would require the user to remove each row within that column one-by-one.
Robot Framework	http://robotframework.org/	Python	Keyword-driven	To automate ATDD.	Readability is affected by a mixture of comments and test cases within a specification. Since both use plain text, it becomes difficult to distinguish one from the other.
Concordion	http://concordion.org/	Java	HTML	To allow customer examples be expressed in a natural	Writing of tests is done with basic HTML. The customer might not be open to that unless the writing is

				language (through HTML).	entirely done by the development team.

Table 3.1: ATDD Tools.

BDD Tools					
Tool name	**URL**	**Automation language**	**Test syntax**	**Aim(s)/purpose**	**Problems observed**
Cucumber	https://cukes.info/	Ruby	Gherkin	To automate BDD with plain text support.	Similar issue with FitNesse whereby making changes to a scenario that share similar steps as another scenario would require extra effort. Since this is done manually by the user, it is prone to careless mistakes (especially if the user needs to make the changes to many scenarios).
JBehave	http://jbehave.org/	Java	Gherkin	First automation tool created to support BDD.	Shares a similar issue with Cucumber.
Behat	http://docs.behat.org/en/v2.5/	PHP	Gherkin	BDD framework for PHP.	Shares a similar issue with Cucumber.

RSpec	http://rspec.info/	Ruby	Mixed	Inspired by JBehave and to provide Ruby support for BDD.	The automation code is written alongside the customer examples. This could potentially turn off customers from using the tool due to a lack of non-technical interface.
EasyB	http://easyb.org/	Java	Groovy	BDD framework for Java.	Shares a similar issue with RSpec.

Table 3.2: BDD Tools.

FitNesse Support Tools		
Tool name	**URL**	**Aim(s)/purpose**
GivWenZen	https://github.com/weswilliams/GivWenZen	Allow FitNesse to recognize BDD's given-when-then notation in its tests.
Cukable	http://www.automation-excellence.com/software/cukable	Allows running of Cucumber features in Fitnesse.

Table 3.3: FitNesse Support Tools.

Cucumber Support Tools		
Tool name	**URL**	**Aim(s)/purpose**
Cucover	https://github.com/mattwynne/cucover	Skips a scenario/feature if the code has not been changed since the last execution.
Cucumber-Eclipse	https://github.com/cucumber/cucumber-eclipse	An Eclipse IDE plugin for editing and running Cucumber features.
Guard:Cucumber	https://github.com/guard/guard-cucumber	Allows Cucumber features to be automatically executed when changes are detected.
Relish	https://relishapp.com/	Allows Cucumber features to be viewed from a web browser.

Table 3.4: Cucumber Support Tools.

3.3 What is Duplication in BDD?

Like any complex computational artefact, Cucumber features can contain “smells” [23]. In programming languages, these smells are referred to as code smells. A code smell is an indication that something *might* be wrong in the code [24]. It usually stems from a flaw in the code design. A code smell does not necessarily prevent the software from running i.e. is not a bug. However, if left unattended, the smell may increase the risk of bugs in the future. For developers/programmers, code smells are signs for when refactoring is needed.

There are good and bad code smells. *Duplicated code* is a bad code smell. An example of this is having the same fragment/block of code repeated in a method. This increases the size/length of the method; making it difficult to follow/read. An option for refactoring this problem would be to migrate the fragment into its own method. Now, the fragment has a name and the method has gotten shorter. The *single responsibility principle* is a good code smell. The principle states that every class is responsible for a specific part of the software functionality and that its methods are catered towards that responsibility. This keeps each class robust towards changes. Changes made to one class would require minimal (at best, none) changes in other classes.

Duplication is one of many bad code smells and is defined as the action of duplicating/repeating something. It is not necessarily a bug. For example, with duplication, Cucumber features can still be correctly expressed, in that they describe the desired behaviour. Most of the time, duplication is done unintentionally e.g. copy-and-pasting. However, duplication can be interpreted differently depending on the given context. To answer this question, a fair amount of familiarity with BDD/Cucumber had to be achieved. Therefore, this section will be dedicated to establishing what it means to have duplication in BDD. The Cucumber tool will be used as the source of examples.

3.3.1 Good vs. Bad Duplication

Like code smells, there are good and bad kinds of duplication in BDD. Good duplication refers to acceptable/necessary duplication in BDD specifications. For example, the *when* step in Cucumber represents an event/action triggered by the user and might be repeated in multiple scenarios. This is acceptable only if each of the scenarios has different pre-condition (*given*) and/or outcome (*then*) steps. As shown

in Figure 3.1, an event (*when*) with different preconditions (*given*) might lead to different outcomes (*then*). The two scenarios shown in the figure are part of the "addition" event but one scenario has two operands in its pre-condition whereas another scenario has three operands.

```
Feature: Addition

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen

  Scenario: Add three numbers
    Given I have entered 2 into the calculator
    And I have entered 3 into the calculator
    And I have entered 4 into the calculator
    When I press add
    Then the result should be 9 on the screen
```

Figure 3.1: An example of a repeated event condition in scenarios.

Bad duplication refers to duplication that does not contribute to the story that a BDD specification is trying to convey i.e. it is unnecessary. For example, in the Cucumber *feature* and shown in Figure 3.2, two or more *scenarios* have the exact same descriptions/titles. The title of a scenario is important because it represents a summary of the behaviour of the software i.e. how it should behave in different situations. It confuses the reader if there are two or more of the same scenario titles in the feature since the reader has to distinguish between the scenarios by reading their steps. This defeats the purpose of having a title in the first place. It is worse if the scenarios have the same list of *steps*. That shows a clear duplication and would only prove to be a waste of space.

```
Feature: Addition

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen

  Scenario: Add two numbers
    Given I have entered 2 into the calculator
    And I have entered 3 into the calculator
    When I press add
    Then the result should be 5 on the screen
```

Figure 3.2: An example of duplicated scenario descriptions.

Figure 3.3 shows another example of bad duplication. In the figure, both features have different scenarios/behaviours but they are part of the same software functionality. There is confusion when it comes to adding new "addition" scenarios to the feature i.e. "Which feature do we add the new scenario to and how do we decide this?" Each feature describes the behaviour of the software and has a set of scenarios that illustrate examples for that behaviour. Therefore, any scenario that is related to the feature should be grouped together in the same location. This allows the features to be cohesive and distinguishable from one another.

```
Feature: Addition

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```

```
Feature: Addition

  Scenario: Add no numbers
    Given I did not enter anything into the calculator
    When I press add
    Then the result should be 0 on the screen
```

Figure 3.3: An example of duplicated feature titles.

Another example of bad duplication comes with the *given* step. Each scenario should have a precondition to define its context for the example at hand. The *given* step is responsible for this task. At times, the same precondition is needed to set things up for the scenario. Figure 3.4 illustrates an example of when the precondition is used in more than one scenario. The consequence of repeating the preconditions is that if they are required to be changed, the changes would have to be made in all of the scenarios that have the same precondition step.

```
Feature: Search

  Scenario: I should see user comments that matches my query
    Given I am logged in as a user
    When I search for "waisuan"
    And I click on the "Search For Comments" button
    Then I should see all of the comments that I have made in the past
    And also comments that have mention of my name

  Scenario: I should see thread titles that matches my query
    Given I am logged in as a user
    When I search for "cucumber"
    And I click on the "Search For Threads" button
    Then I should see all of the thread posts that has the word "cucumber" in it
```

Figure 3.4: An example of a repeated pre-condition in scenarios.

This project will be focusing on reducing *bad* duplication in BDD/Cucumber specifications.

3.4 Importance of Reducing Duplication in BDD

It is important that duplication in BDD be avoided (or at least, kept to a minimum). As mentioned in Section 2.5, one of the main reasons for using BDD is to allow customers to get involved in writing and checking the specifications. So, it is vital for the specifications to remain readable and consistent at all times. If they are difficult to read/follow, the customer will possibly avoid using them. This leads to difficulty in getting clarifications for unambiguous software requirements from the customer. Additionally, the specifications' quality is affected greatly when duplication is introduced. This makes it difficult to trust their authenticity since there is a worry that duplication exists somewhere in them and that it may affect their outcome.

BDD specifications should be readable. This means that reading a BDD specification should not be complicated. The reader should be able to understand what each Cucumber feature is about through the feature title. The examples/scenarios should be descriptive enough such that the user does not have to refer to the steps in order to decipher their meaning. The steps are meant to provide a walkthrough towards achieving the goal that the scenario has specified. However, it is easy to write irrelevant steps that the user is forced to read around. Introducing duplication to the specifications will only make reading them difficult. This also makes the scenario longer than necessary. A good rule of thumb is to write steps pertaining to *what* needs to be done instead of *how* it is done [25]. Figure 3.5 (taken from D. Kennedy [26]) shows an example of what bad and good Cucumber features can look like. Both features represent the same example and behaviour. However, the feature on the right has removed irrelevant steps from its example and kept steps that are important for expressing the example. It is difficult to grasp the important points in the feature on the left due to the need to rummage through the less important steps. In addition, the feature on the right has added a narrative to make the story clearer.

```
Feature: Todo item management

Scenario: Adding a todo item
    Given: I have a todo list named "Mondays list"
    When I go to the home page
    And I fill in "username" with "dave"
    And I fill in "password" with "secret"
    And I press "Log In"
    And I go to the todo page
    And I click on link "Mondays list"
    And I fill in "todo" with "Grab some milk"
    And I press "Add todo"
    Then I should see "Todo item added successfully"
```

```
Feature: Todo item management

I want to track items I need to do in a list.
That way I will never forget them.
I want to add, edit, delete and mark todo items as finished.

Background:
    Given I am logged in as a normal user

Scenario: Adding a todo item
    Given: I have a todo list named "Mondays list"
    When I add a todo item "Grab some milk"
    Then it should be added to the todo list
```

Figure 3.5: Side-by-side comparison of bad and good Cucumber features.

Another reason for avoiding duplication is to help keep test suites small and efficient to execute, so that developers can maintain them. Therefore, BDD specifications should not only be readable but easily maintainable. If duplication is introduced into the specifications, the creator of the specifications is *expected* to fix it. It is harder than it sounds. For example, a repeated scenario has been inserted (possibly, copied-and-pasted) into a Cucumber feature unknowingly. This is possible if the feature already has a large number of scenarios in it and the user overlooked the scenario that already exists. Later on, the creator has to manually look through the feature for the duplication and remove it. Likewise, the same can be said when/if one of the duplicates needs to be updated/changed. This gets increasingly tedious and risky if there is more than one repeated scenario in the feature and there are many features to look through. Since the removal of the duplicates is done manually, it is easy to overlook a few. Another example is if the creator wanted to remove repeated pre-conditions from the scenarios. The creator might carelessly remove more than he/she was aiming for in the first place. Features that passed initially may fail after the changes were made or features that would have failed because of the deleted precondition may now pass. Ultimately, the creator would have to proceed with debugging the failing features and trace back his/her changes to locate the source of the problem. Again, the situation is worse if the changes were made towards many features.

If left untreated, duplication can build up and a significant amount of effort and time would have to be put into refactoring the specifications later on. The issues depicted here could be avoided if the creator of the specifications was aware that duplicates were being created in the early stages/as the features are being written.

3.5 Near-duplicates

So far, the discussion has been directed towards exact duplicates. However, there is a need to take into account near-duplicates as well. Near-duplicates detection is defined as identifying entities that might differ slightly from one another but are close enough to be considered as duplicates. For example, within the context of Cucumber, *given two textually distinct steps in a Cucumber feature, do they have the same meaning?* This is also known as *semantic equivalence*. Another instance of near-duplication could be when two Cucumber steps differ from one another only in the placement of punctuation marks (e.g. *full stop*). Figure 3.6 shows two steps that are not exact matches but are pointing to the same meaning and can be considered as duplicates of

one another. One of the objectives of this project is to flag for near-duplicates in BDD specifications.

```
...
Given there are eggs in my basket
...
Given my basket has eggs
...
```

Figure 3.6: An example of (semantically) equivalent steps.

Again, this comes back to improving the readability of specifications. Similar features/scenarios/steps would only serve to confuse the reader. Using BDD should not create further ambiguities. Having near-duplicates in the specifications also brings up further questions regarding which of the similarities should be kept and which should be refactored.

3.6 Rules for Detecting Duplication

There is the issue of how one can judge whether two or more BDD entities are truly bad duplicates of one another. In addition, good duplications should be allowed. If a software application were to detect duplications in the specifications, would human experts (ones who are well-versed with BDD) detect the same duplications? It is important for the application not to flag duplicates that are necessary to the specifications. This would only serve to confuse the application user. The point to make here is to ease the use of BDD for users and guide them into writing the right BDD specifications from the very beginning. Therefore, it is essential to establish what constitutes bad duplication and what does not in this project.

A set of rules (that has not been proposed before) was established for detecting whether duplications exist in BDD specifications/Cucumber features. These rules act as functional requirements and are adopted by the tool implemented for the project (also, to help answer the project's research question). The tool is in the form of an Eclipse IDE (integrated development environment) plugin named *SEED*[20].

The rules for detecting duplications does not describe how duplications are detected but what situations are considered as occurrences of duplication. The process of

---

[20] *SEED* [Online]. Available at: https://github.com/waisuan/SEED

developing the rules was based on the author's extensive use of the Cucumber tool. The rules had to fit within the aim of the project. Therefore, the process (of developing the rules) began with an overview of what the project was trying to achieve and how SEED fit into all of it. The overview is shown in Figure 3.7. The main entities involved in the project were the Cucumber specifications, SEED, and the user who *will* be using SEED. The aim of SEED is to detect bad duplications in the specifications. The user uses SEED in order to reduce/avoid duplications in the specifications that they, otherwise, would have to search for manually. Therefore, the result of using SEED is having the specifications *marked* with helpful diagnostics on places that contain duplicates (which we will see later on). In order for this to happen, SEED had to be implemented with information (i.e. rules) and methods (i.e. algorithms) for detecting the duplications.

It is worth noting that, due to insufficient implementation time, the rules only apply to duplications in *individual* Cucumber features. It would have been *better* to detect duplications across two or more Cucumber features. However, Cucumber feature titles are still compared and checked for duplications.

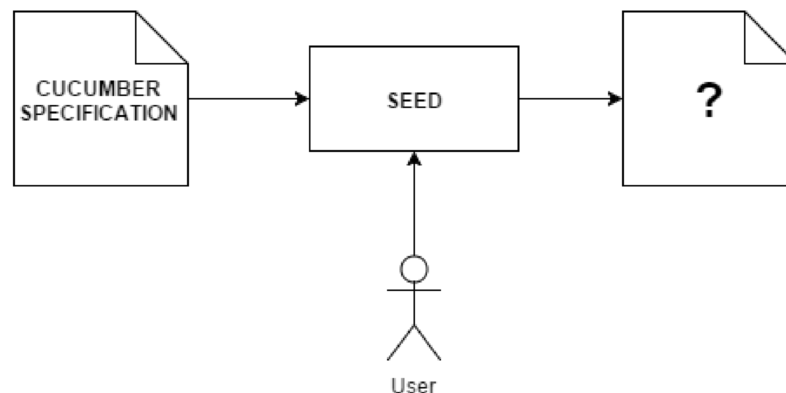

Figure 3.7: A high level outlook of the goal of the project.

3.6.1 Rule 1

The first rule is directed towards detecting duplications in *syntactically/textually equivalent* matches of Cucumber entities (e.g. feature, scenario, and steps). This means that the duplicates can either be exact or equivalent matches. The rule is as follows:-

- Two or more Cucumber features are duplicates of one another if their titles are syntactically equivalent.

- Two or more Cucumber scenarios/scenario outlines are duplicates of one another if their titles and/or list of steps are syntactically equivalent.
- Two or more Cucumber steps in a scenario/scenario outline/background are duplicates of one another if they are syntactically equivalent.
- Two or more rows in a scenario outline's examples table are syntactically equivalent.
- Duplication is found if a step(s) in a Cucumber scenario/scenario outline is syntactically equivalent to a step(s) in the background section.

Exact matching is not enough to locate the duplicates since the matches need to be character-by-character identical. For example, if the Cucumber entities consisted of extra trailing spaces, the match would be broken. Therefore, the rule takes that into account by including equivalent matches. Equivalence matching is *similar* to comparing parse/syntax trees where wasteful information (e.g. extra spaces) is not included in the matching process.

3.6.2 Rule 2

The second rule deals with near-duplications (briefly discussed in Section 3.5) of Cucumber entities. When syntax matches fail to produce any results, it is worth looking at semantic matches/equivalences. However, it is difficult (for SEED) to confirm whether the semantic/similar matches should be considered as duplications. There may be cases where further evaluation from the user of SEED on the matches found is required. The rule is as follows:-

- Two or more Cucumber features are duplicates of one another if their titles have the same meaning.
- Two or more Cucumber scenarios/scenario outlines are duplicates of one another if their titles have the same meaning.
- Two or more Cucumber steps in a scenario/scenario outline/background are duplicates of one another if they have the same meaning.
- Duplication is found if a step(s) in a Cucumber scenario/scenario outline has the same meaning as a step(s) in the background section.

3.6.3 Rule 3

The third rule focuses on the pre-condition step(s) (i.e. *given* steps) of the Cucumber scenarios in a feature. As mentioned in Section 2.5.1, Cucumber provides a

background section for pre-condition steps that are required by all of the scenarios in the feature. The section helps reduce the number of steps in a scenario, making it smaller and easier to read. The user of Cucumber might not *always* be aware of this. Therefore, if a syntactically or semantically equivalent pre-condition step(s) is repeated in *all* of the scenarios in the feature, they are considered as duplicates. However, if there are only two scenarios in the feature, it *might* not be necessary to move the pre-conditions into the background.

3.6.4 Rule 4

The fourth rule deals with Cucumber scenarios that are different from one another *only* by their input or output values as shown in Section 2.5.1. They are not wrong but if there are many of these scenarios, the feature can get quite large. In Cucumber, the scenario outline section is able to solve this issue by combining the scenarios into a single section whilst using the examples table to hold the different input/output values. Therefore, if two or more scenarios differ only by their input/output values, they are considered as duplicates. However, if the scenario outline makes it harder to read (e.g. through the examples table) than the individual scenarios, combining them *might* not be the best option.

3.6.5 Decision Tree

As a visual aid of the rules (of detecting duplication) discussed previously, a decision tree diagram (as shown in Figure 3.8) was constructed. The decision tree encompasses the decision-making process that comes with looking for duplications in a Cucumber feature. The flowchart-like structure starts with a Cucumber feature and progresses according to *yes* or *no* answers. The goal is to reach either one of the following results: *No duplicates*, *Duplicates found*, and *Invalid*. Additionally, the decision tree made it easier to track the development progress of SEED since each block represents a behaviour/action that SEED should have. Therefore, by following the flow of the tree, the development of SEED could be done incrementally until completion.

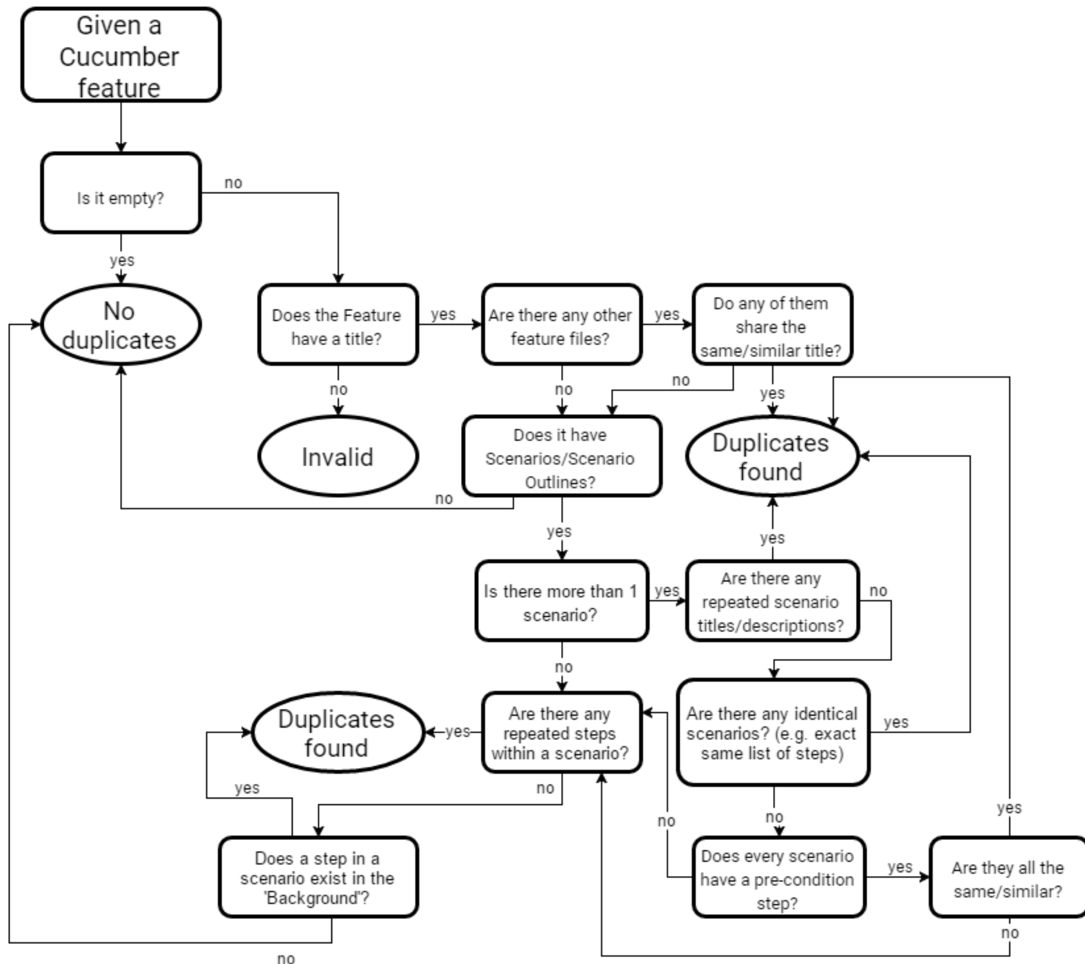

Figure 3.8: Decision Tree Diagram.

3.7 Duplicate Detection Algorithm

The algorithms (implemented in SEED) for detecting duplications is divided into two categories; exact- and near-duplication. Although both algorithms differ in logic and steps, their intentions are the same; searching for duplications in BDD specifications/Cucumber features.

3.7.1 Exact Duplicate Detection

The following equation is used to indicate when two Gherkin expressions are *equal*.

$$Gherkin_1 = Gherkin_2$$

In Cucumber features, plain text follows the Cucumber keywords. The plain text is first extracted from a Cucumber entity (e.g. scenario, step, and feature) before being used as input(s) for the detection algorithm. The algorithm can be broken down into the following important points:-

- The pieces of text are compared character-by-character.

- In order to avoid getting false negatives (e.g. two pieces of text are actually duplicates but the algorithm failed to detect it), each character of the text is converted into *lower* case before comparison. The algorithm also removes extra (trailing) spaces within the text before comparison. As long as two pieces of text have the exact same characters, they are considered as duplicates.
- To avoid unnecessary checks, candidates (i.e. two pieces of text) that differ in length are excluded from duplicate detection.

3.7.2 Near-Duplicate Detection

The following equation is used to indicate when two Gherkin expressions are *equivalent*.

$$Gherkin_1 \equiv Gherkin_2$$

As mentioned in Section 3.5, near-duplicate detection not only covers minor differences in syntax but also semantic equivalences. The algorithm chosen for detecting semantic equivalences is based on the *Dice coefficient* algorithm.

Dice coefficient is a "term based similarity measure" [27]. The calculation for the similarity measure is based on the formula shown in Equation 3.1 where *C* is the number of common terms (in the form of *bigrams*) found in both pieces of text, *L1* is the number of terms in the first piece of text, and *L2* is the number of terms in the second piece of text. A bigram is "a pair of consecutive written units such as letters, syllables, or words"[21]. For the algorithm, words are used as bigrams. In summary, the measure is defined as *twice the number of common terms divided by the total number of terms in both pieces of text*. The result of this algorithm is a value ranging between 0 and 1 whereby 0 indicates (complete) dissimilarity between the pieces of text and 1 indicates that the pieces of text are identical.

$$DiceCoefficient = \frac{2 * C}{L1 + L2}$$

Equation 3.1: Dice coefficient formula.

[21] *Bigram* [Online]. *Oxford Dictionaries.* Available at: http://www.oxforddictionaries.com/definition/american_english/bigram

The algorithm can be broken down into the following steps:-

1. Let the first piece of text be *S* and the second piece of text be *T*
2. Split *S* and *T* into two arrays of bigrams.
3. Calculate the length of each array.
4. Combine both arrays into a single array of bigrams.
5. Calculate the length of the single array.
6. Calculate the total number of *common* terms/bigrams found in *S* and *T*.
7. Finally, calculate Dice coefficient based on the formula shown in Equation 3.1.

Figure 3.9 illustrates an example of the Dice coefficient algorithm applied to two pieces of text. The intersection process collects the bigrams that exist in both pieces of text. The formula is only applied after they have been collected. The vertical bars in the formula represents the magnitude of an array. In conclusion, the similarity between "MALAYSIA" and "MALDIVES" is 30% (the coefficient value is expressed in percentage for easy viewing and is rounded to the nearest whole number) i.e. they are not similar.

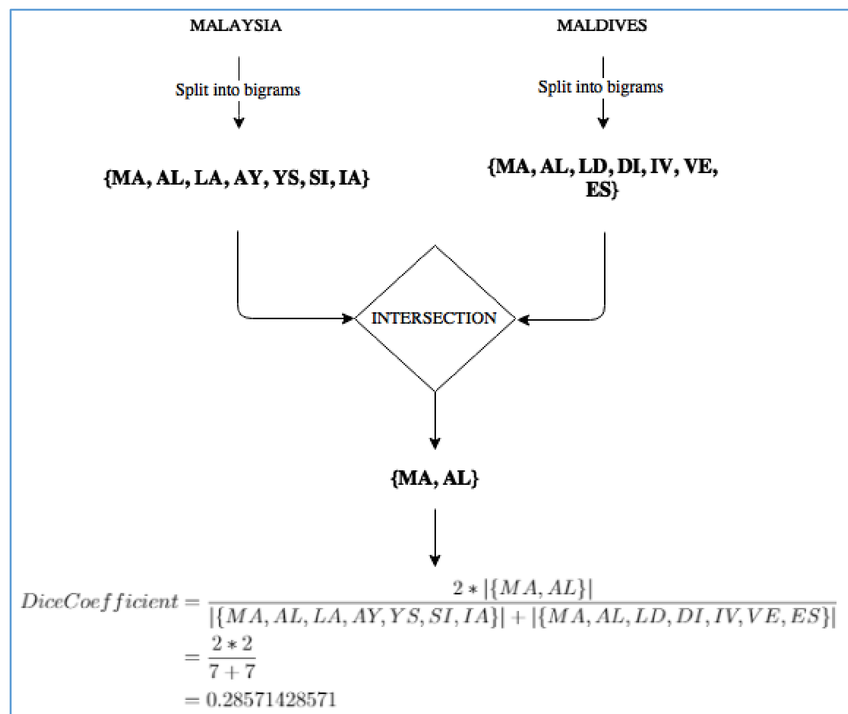

Figure 3.9: A working example of the Dice Coefficient algorithm.

It is possible that the algorithm can produce a misleading result i.e. sometimes state that two strings are similar when a typical native speaker would disagree. At the time of writing, there is ongoing research on improving the strength of finding near-

duplicates amongst strings. Therefore, for this MSc project, it is necessary to choose a threshold/Dice coefficient for recognizing equivalent statements in BDD specifications. This threshold must be carefully selected to reduce the number of false matches. In other words, two pieces of text are regarded as similar if their Dice coefficients are above or equal to the threshold. To establish this threshold, the algorithm had to be run against a set of situations where near-duplicates should be found and could exist in Cucumber features. The results are shown in Table 3.5.

Context	Example	Similarity Measure
Steps differ in only their input/output values.	"there are 12 cucumbers" "there are 20 cucumbers"	93%
	"the cow weighs 450 kg" "the cow weighs 500 kg"	92%
Statements differ by placement of punctuation mark(s).	"the weather is good today" "the weather is good today**.**"	97%
	"however**,** this is still incorrect" "however this is still incorrect"	98%
Statements that have different arrangement of words but have the same meaning (e.g. active/passive verb forms) (i.e. semantic equivalency).	"the professor teaches the students" "the students are taught by the professor"	73%
	"James washes the dishes" "the dishes are washed by James"	85%
	"I repaired the car" "the car was repaired by me"	85%
	"she is making dinner tonight" "dinner is going to be made by her tonight"	74%
	"I have to complete the project before the deadline" "the project will have to be completed by me before the deadline"	90%

Table 3.5: Dice coefficient for syntactically and semantically equivalent statements.

Table 3.5 highlights three situations where near-duplicates could occur in Gherkin expressions. Multiple examples were used to ensure that the similarity measure threshold is consistent against a given situation. The challenge resided in identifying whether two pieces of text are semantically equivalent. Therefore, more tests were run for that particular situation. From the results shown in the table, a safe/reasonable threshold for identifying near-duplicates in Cucumber features (with the Dice coefficient algorithm) is **73%**. The percentage is taken from the lowest (possible) similarity measure generated within the given situations. In conclusion, if statements within a Cucumber feature are checked for near-duplication and the similarity measure is above or equal to 73%, they are considered as near-duplicates/equivalent/similar. It is important to note that there is still a need for the user of SEED to decide whether the statements are *truly* similar/dissimilar (to avoid false positives/negatives).

There are other algorithms similar to Dice coefficient that focus on measuring similarities between pieces of text (i.e. strings). Due to the nature of the project requirements for detecting near-duplicates in Cucumber features (as shown in Table 3.5), the Dice coefficient algorithm was deemed to be the most suitable for the project. However, this was only decided after a series of experiments (as shown in Table 3.6) using various other algorithms to determine their strength in identifying equivalency within the context of BDD. These algorithms were chosen based on their popularity in the field of approximate string matching. As shown in Table 3.6, the Dice coefficient algorithm proved to be the most consistent in identifying similarity between the chosen statements.

Example	Levenshtein Distance	Longest Common Substring	Jaro-Winkler Distance	Dice Coefficient
"the cow weighs 450 kg" "the cow weighs 500 kg"	90%	29%	**98**%	92%
"however**,** this is still incorrect" "however this is still incorrect"	97%	25%	**99**%	98%

“the professor teaches the students” “the students are taught by the professor”	40%	68%	**78**%	73%
“James washes the dishes” “the dishes are washed by James”	37%	67%	66%	**85**%
“I repaired the car” “the car was repaired by me”	35%	62%	0%	**85**%
“she is making dinner tonight” “dinner is going to be made by her tonight”	46%	**76**%	0%	74%
“I have to complete the project before the deadline” “the project will have to be completed by me before the deadline”	54%	68%	77%	**90**%

Table 3.6: String similarity-matching algorithms ran against syntactically and semantically equivalent statements.

3.7.3 Model of Duplication

Figure 3.10 shows a representation of the duplication model in SEED. The model was created based on the rules in Section 3.6. Each component in the model represents a specific (class of) duplication in Cucumber. Duplicates detected by SEED are grouped together under the components of the model. The model is described as follows:-

- *Step in Scenario/Outline/Background.* This duplication is part of Rule 1 and 2 whereby two or more Cucumber steps are either equivalent or have the same meaning.
- *List of steps.* This duplication is part of Rule 1 where two or more Cucumber scenarios/scenario outlines have the same list of steps.

- *Feature title.* This duplication is part of Rule 1 and 2 whereby two or more Cucumber feature titles are equivalent or have the same meaning.
- *Scenario.* This duplication is part of Rule 4 whereby two or more Cucumber scenarios differ only by their input/output values and can be combined into a scenario outline.
- *Step already exists in background.* This duplication is part of Rule 1 and 2 whereby steps in the background section and steps in the Cucumber scenarios/scenario outlines are equivalent or have the same meaning.
- *Scenario/Scenario outline title.* This duplication is part of Rule 1 and 2 whereby two or more Cucumber scenario/scenario outline titles are equivalent or have the same meaning.
- *Examples table row.* This duplication is part of Rule 3 whereby two or more rows in an Examples table are equivalent.

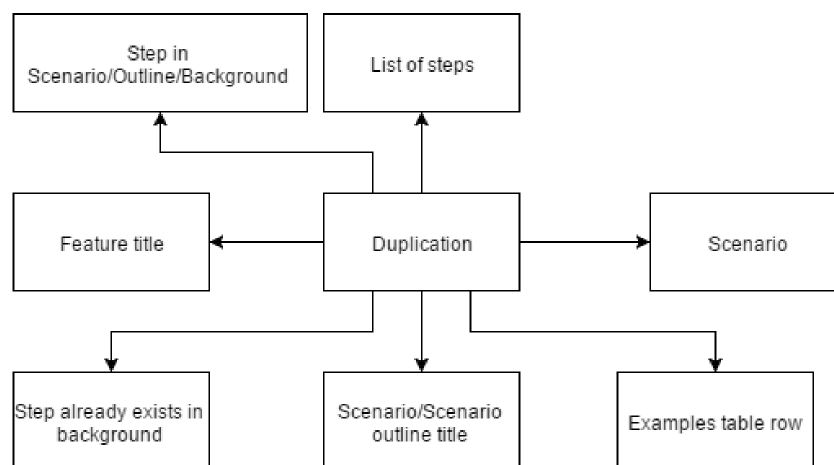

Figure 3.10: Duplication Model in SEED.

3.7.4 Process Flow

Figure 3.11 shows where the algorithms (discussed earlier) fit in the overall process of detecting duplication in Cucumber features. Note that the entire process takes place within the Eclipse IDE (where SEED resides).

SEED has a parser for parsing Cucumber features. The parser uses the Gherkin grammar to convert the strings (from the Cucumber feature) into an *abstract syntax tree* (AST) during runtime. An AST is a representation of the source code (which in this case is the Gherkin grammar) as a tree whereby each node in the tree represents a

construct occurring in the source code (e.g. step keywords and step descriptions). [22] Figure 3.12 shows an example of the tree after the parser is run against a Cucumber feature. The leaf nodes represent the strings of the Cucumber entities. The parsing of a Cucumber feature (by the parser) is triggered (automatically) when the feature file is opened on the Eclipse IDE editor.

After the Cucumber feature(s) has been parsed, SEED is able to call upon the implemented methods (encompassing the algorithms) to detect exact- and near-duplication (from the AST). The leaf nodes of the AST are the core entities in determining whether duplication has been found or not. In SEED, every rule defined in Section 3.6 has its own (implementation) method and each method takes in the appropriate Cucumber entity (as an input parameter) that is related to the rule. For the example shown in this process flow, duplicate Cucumber scenario titles were found. The method for checking whether duplicate scenario titles exist accepts the Scenario node/entity as its input parameter. The method compares the scenario titles by their *string* (the leaf nodes of the AST) using the exact- and near-duplicates detection algorithms.

At the implementation level, *each* of the detected duplicates is contained in a *wrapper* object. The wrapper object was specifically designed and implemented for this project. The wrapper consists of the following details: Cucumber entity *type* (e.g. Scenario), the *string* of the entity, and the *message* to appear on the Eclipse IDE editor for the duplicate text found. The contents of the wrapper are passed (through a method call) to the internal components of the Eclipse IDE and Eclipse *underlines* the duplicates in its editor (similarly to how Eclipse typically marks issues on its editor). Apart from duplication detection and creating the wrapper, this operation is done *entirely* by Eclipse. The underlined/marked duplicate is also shown in the process flow figure (at the end of the process).

---

[22] *Abstract Syntax Tree* [Online]. Available at: http://c2.com/cgi/wiki?AbstractSyntaxTree

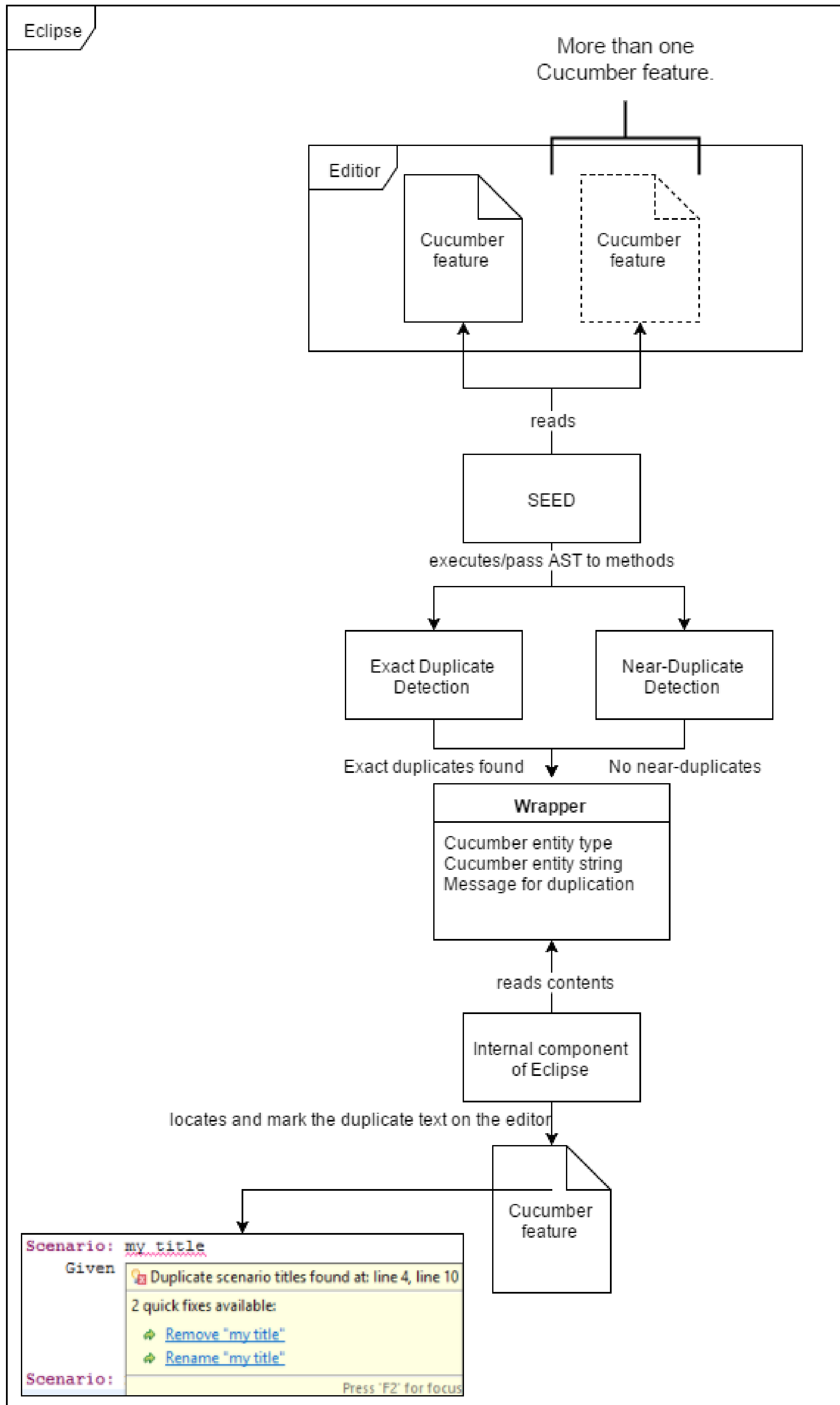

Figure 3.11: Flow of Duplication Detection.

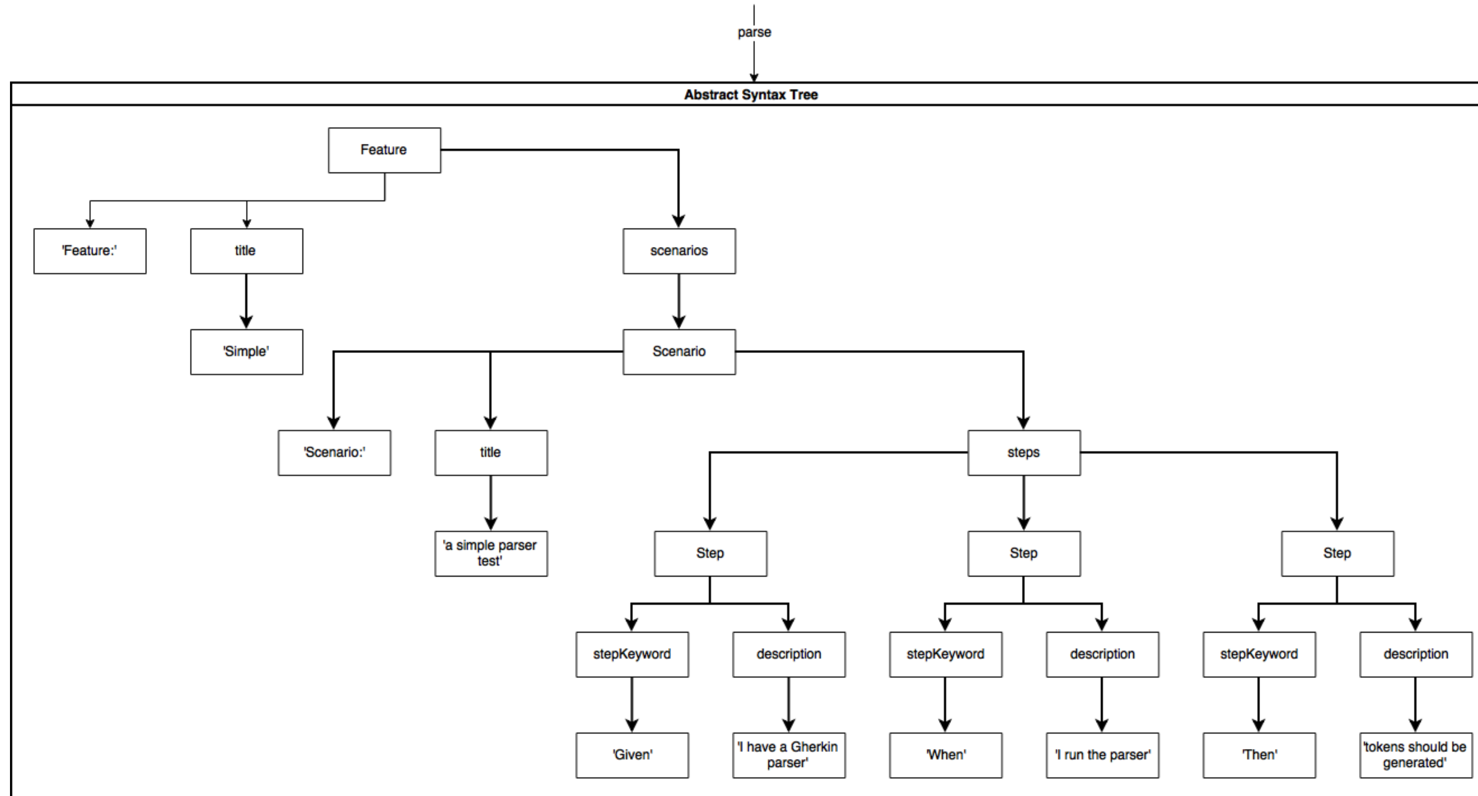

Figure 3.12: Abstract syntax tree produced by the parser after parsing a Cucumber feature.

3.7.5 Limitations

As mentioned, there is a risk of getting misleading results when searching for near-duplication/similar matches. The Dice coefficient threshold established in the previous section does not guarantee that there will be no false matches. Additionally, string similarity matching algorithms are more suited for checking syntactic equivalence than semantic equivalence. It is difficult to judge whether two pieces of text have the same meaning (through a software tool) since the text can have different arrangements of words but still contain their meaning.

An idea for a more reliable alternative would be comparing *parse trees*. A parse tree represents the syntax of a string in the form of an ordered tree and is generated from the parser. Duplication is detected if two or more strings have the *same* parse trees. To bypass the different arrangements of strings, one could just compare the leaf nodes of the trees. If the trees have the same list of leaf nodes, then they are considered as *equivalent* and are duplicates.

3.8 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure." (p. 9 [24]). The refactoring process typically refers to the software code. However, this process can also be adopted in BDD specifications. It generally means making descriptions of computational behaviour (like code and BDD specifications) cleaner and clearer without changing its underlying functionality (e.g. step definitions/glue code).

After providing helpful diagnostics to the Cucumber user regarding potential duplications in the specifications, the next step for the user/creator of the specifications would be to remove the duplications i.e. refactor. The refactoring operation can be automated. The *Eclipse*[23] IDE does this by listing down all valid refactoring steps for a particular issue (as shown in Figure 3.13) and allows the user of the IDE to choose which to apply. SEED also acts similarly (as shown in Figure 3.14). Automation removes the need for making the refactoring changes manually, leaving it open to

[23] *Eclipse* [Online]. Available at: https://eclipse.org/home/index.php

careless mistakes as described previously. However, in order for it to be useful, the refactoring options (through automation) should reflect the same options that human experts generally think of e.g. *would a Cucumber user typically migrate repeated pre-conditions into the Background such that it becomes a valid refactoring option?*

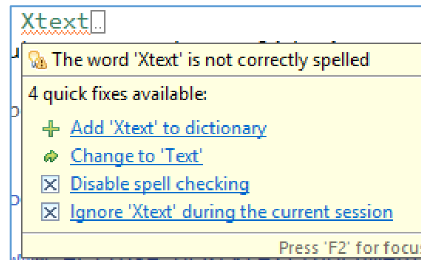

Figure 3.13: An example of quick fixes on Eclipse.

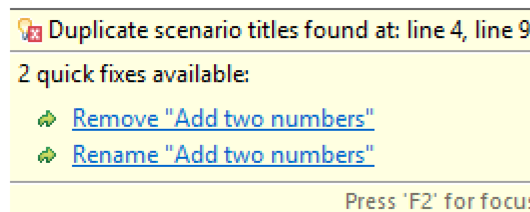

Figure 3.14: SEED's quick fixes.

Figure 3.18 shows the overall process flow of refactoring Cucumber features with SEED. SEED uses the information from the duplications detected to carry out the refactoring operations. SEED's refactoring operations/methods are responsible for *locating* the duplications on the Eclipse editor and *manipulating* the string of the duplications. The *updated* strings are passed (by SEED) to Eclipse's internal components for the *actual* removal of the duplications on the editor (by *replacing* the duplications with the updated/new strings). There are multiple options that the user (of SEED) is able to choose for removing duplications. The following is a list of the available refactoring options that SEED provides:-

- `Rename`. This option is available when exact duplicates (e.g. feature titles, scenario titles, and step descriptions) are found (Rule 1 & 2). SEED appends the duplicate's line number (in the Eclipse editor) onto the to-be-refactored text (as shown in Figure 3.15). It would have been better to put up a dialogue for the user (of SEED) to enter the new name since appending the line number does not necessarily improve the quality of the Cucumber feature. Unfortunately, there was insufficient time to implement this functionality.

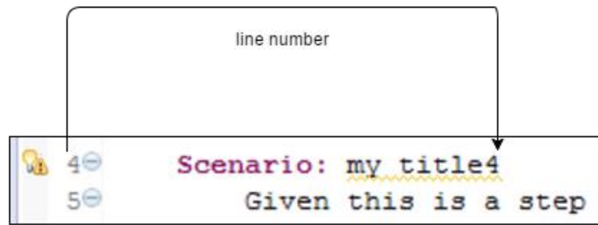

Figure 3.15: Example of renaming duplications.

- `Remove`. This option is available when exact/equivalent duplicates (e.g. feature titles, scenario titles, and step descriptions) are found (Rule 1 & 2). SEED will remove the duplicate text entirely.
- `Migrate to Background`. This option is available when repeated pre-conditions are found in the Cucumber feature (Rule 3). SEED moves these steps into the background portion (will be created if it does not exist) of the feature. An example of this is shown in Figure 3.16.

```
Feature: Shopping Cart

    Background:
        Given I am a normal user
       Given I added an item to my cart

    Scenario: 1 item in my shopping cart
        Given I added an item to my cart
        And my cart was empty previously
        Then I have 1 item in my shopping cart

    Scenario: 2 items in my shopping cart
        Given I added an item to my cart
        And my cart had 1 item previously
        Then I have 2 items in my shopping cart
```

Figure 3.16: Example of moving pre-condition steps to the background.

- `Combine`. This option is available when similar scenarios are found in the feature i.e. scenarios with steps that differ in only input/output values (Rule 4). SEED offers to help combine these scenarios into a scenario outline. Figure 3.17 shows an example of this. The question marks serve as placeholders for the examples table's column header names. The naming is left to the users of SEED.

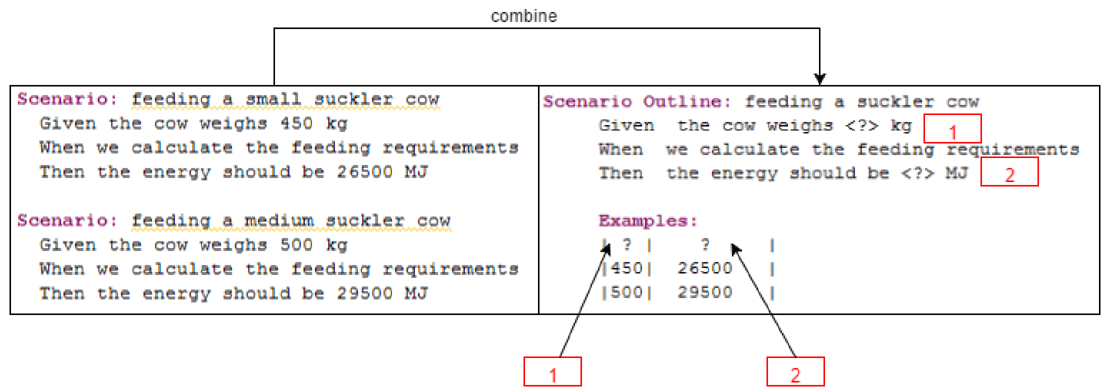

Figure 3.17: Example of combining scenarios into a scenario outline.

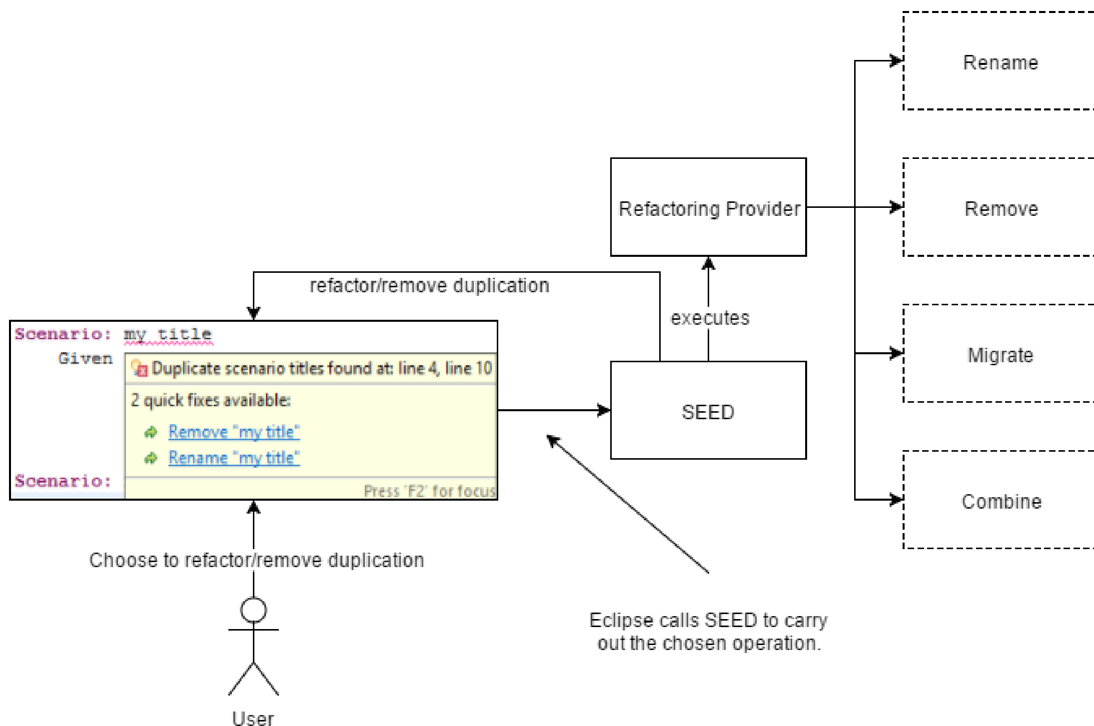

Figure 3.18: Flow of refactoring.

3.9 Methodology

The following sub-sections outline the steps taken in the project.

3.9.1 Information Gathering

The majority of the effort and time during the initial phase of the project was dedicated towards gathering information on the existing ATDD/BDD tools and our current understanding of the ATDD/BDD process. The aim was to gain a strong understanding of the project requirements. A problem domain (i.e. duplications in BDD) was then selected to focus on following the conclusion of the background research.

The tasks undertaken are as follows:-

1. Reading theses and scientific papers – The literature covered the study of ATDD and BDD. This helped highlight the advantages and drawbacks of both concepts.
2. Surveying online resources, specifically blogs and forums focused on ATDD and BDD tools – These contained discussions of the problems that users faced whilst using the tools.
3. Investigating existing support tools – The support tools are extensions/plug-ins to the parent tools (e.g. FitNesse/Cucumber/Robot). They were created either to ease the use of the parent tools or solve an identified issue with the parent tool. (For example, a plug-in that gives Python support to FitNesse). The project aimed to tackle an unsolved problem(s) surrounding the tools. Therefore, it was necessary to exclude existing work and solved problems from the project.
4. Hands-on use of FitNesse and Cucumber – In order to gain a perspective on the issues surrounding these tools, both the tools were experimented with.

3.9.2 Development

A plugin tool was developed for the project. The design of the tool took place after the project context had been fully understood. The development process followed a broadly agile approach of organising the tasks into small iterations. An iteration lasted for two weeks. Each iteration consisted of the following phases: requirements gathering, design & implementation, and testing.

1. Requirements gathering: Requirements of the tool were gathered through discussions with the supervisor.
2. Design: Feature designs of the tool were discussed with supervisor before coding.
3. Testing & Implementation: This phase followed the process of TFD (Test-First Driven Development) where unit tests were written before any coding took place. The tests helped measure the progress of development.

3.9.3 Deliverables

The final set of deliverables are divided into the following artefacts:-

1. A survey of ATDD/BDD tools (and their respective supporting tools) with the results compiled into a tabular format for easy viewing. The table describes the key features of the tools and the issues that have been observed for each tool.
2. A set of rules for detecting duplications within BDD specifications.
3. A plugin tool that detects duplications in Cucumber features.
4. A plugin tool that proposes refactoring suggestions to the detected duplications.
5. A dataset containing the results from the evaluation of the tool.

3.9.4 Evaluation

The evaluation process occurred towards the end of the project. The process sought to answer the research question introduced in Chapter 1. The project's final results were the outcome of this process. For the approach, Cucumber features were retrieved from open source projects that make use of the Cucumber tool. These projects were located within GitHub.

3.10 Conclusion

This chapter has provided a detailed outlook on the problem tackled by the project and the implications of having (bad) duplication in BDD specifications. The methods used to tackle the duplication problem were also shown. At the time of writing, Cucumber/BDD tools have yet to address the problem outlined in this chapter. The following chapter gives an overview of the architecture design and implementation details of the tool implemented for the project.

Chapter 4 : Realizing the SEED tool

4.1 Overview

This chapter will give an overview of SEED's architectural design as well as its implementation details.

4.2 Software Design

This section discusses the overall design and architecture of SEED.

4.2.1 Eclipse Plug-in Architecture

As stated in Section 3.6, SEED is an Eclipse IDE plug-in application. Before discussing SEED's components, it is necessary to show the overall architecture of an Eclipse IDE plug-in and how it fits into the Eclipse IDE.

An IDE is a software development workspace that integrates multiple development utilities into a single location such that developers can maximize their productivity by not worrying about configuration/setup tasks. The Eclipse IDE architecture is made up of layers of plug-ins. Each plug-in represents a specific task and it is these plug-ins that bring end user functionality to Eclipse. It is important to note that these plug-ins are dependent on the IDE and don't work by themselves. The IDE/platform itself manages the complexity of integrating these plug-ins together to form a running application. Eclipse comes with a few default plug-ins that represent the *basic* functionality of the IDE. However, the IDE is able to integrate with *additional* plug-ins for adding (more) functionality to the platform.

Figure 4.1 shows a high level overview of the Eclipse plug-in architecture. The *workbench UI* component represents the interfaces in Eclipse (e.g. menus and editors) and allows new UI/interface components to be added to it. The *extension point* serves as an extension slot for the Eclipse component that is able to be extended for new functionality (e.g. Workbench UI). The *plug-in interface* comprises of the services of the plug-in's behaviour/functionality. Eclipse connects the interface to the extension point in order for the new functionality (i.e. the plug-in) to work.

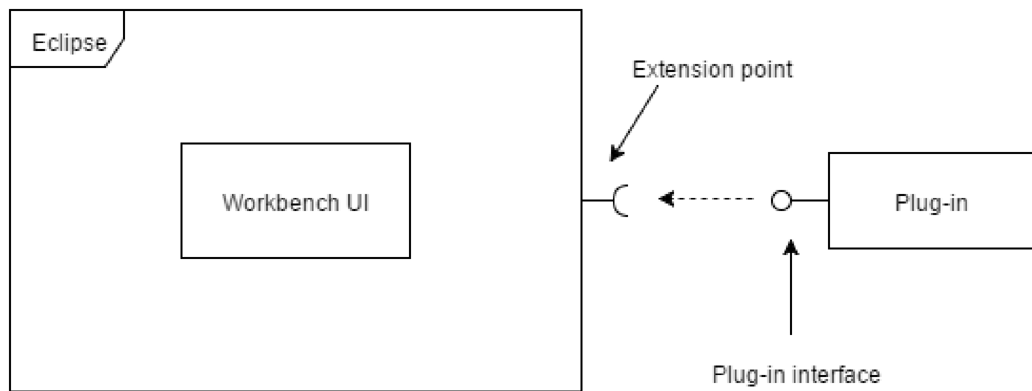

Figure 4.1: Overview of Eclipse plug-in architecture.

The minimal requirement for an Eclipse plug-in to work is having a *manifest* file (in the plug-in). The file provides important details about the plug-in such as the plug-in name, ID, version number, and the implementation code that makes the plug-in work. An example of a simple Eclipse plug-in that adds a new item to the Eclipse menu is shown in Figure 4.2.

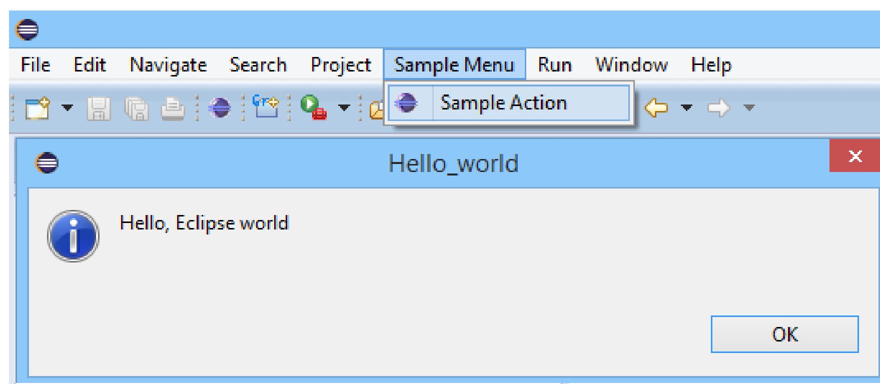

Figure 4.2: A simple plug-in that adds a new item to the menu.

4.2.2 SEED Architecture

Figure 4.3 shows an (abstract) overview of SEED's internal components. SEED is made up of two main components: *parser* and *engine*. At the moment, SEED is geared towards Cucumber features. However, it is possible to extend the plugin's functionality towards other BDD-based tests in future work.

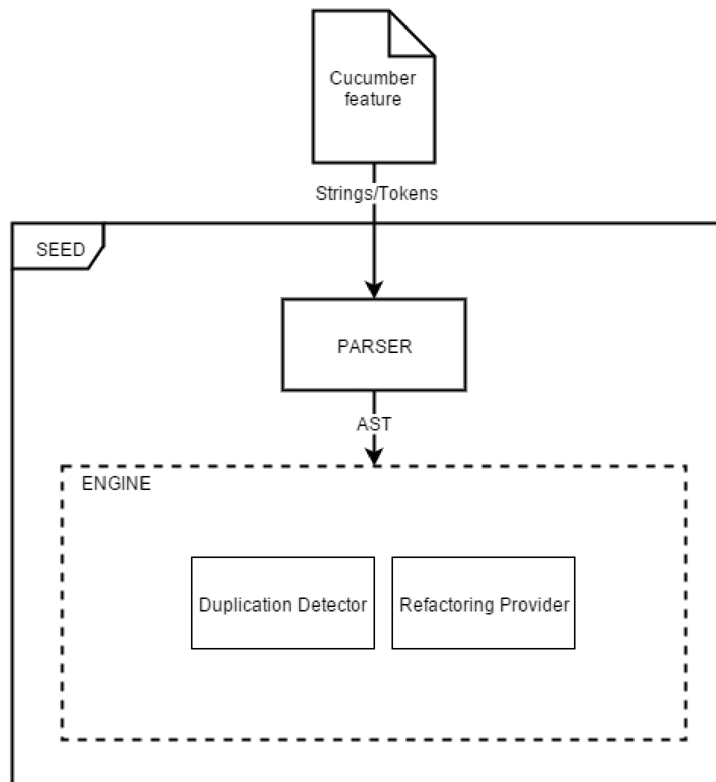

Figure 4.3: Architecture of the SEED plugin.

The purpose and functionality of the parser was examined in Section 3.7.4. *ANTLR*[24] was used to generate the *parser* component. ANTLR is a "parser generator for reading, processing, executing, or translating structured text" [25]. In order for ANTLR to generate the parser that is able to recognize the Gherkin language, it needs a *grammar*. A grammar is defined as a set of rules for constructing valid strings/text according to a language's syntax. For this project, a grammar needed to be built around the Gherkin language. With the help of the Cucumber development team[26] and external work[27] on integrating Cucumber into the Eclipse IDE, we managed to construct a working Gherkin grammar. The full Gherkin grammar can be found in Appendix A.

The *engine* component contains the sub-components for detecting and refactoring duplications in Cucumber features. The methods for detecting duplication are

---

[24] *ANTLR* [Online]. Available at: http://www.antlr.org/
[25] *ANTLR* [Online]. Available at: http://www.antlr.org/
[26] *BNF* [Online]. Available at: https://github.com/cucumber/gherkin/wiki/BNF
[27] *Cucumber Xtext* [Online]. Available at: https://github.com/rlogiacco/Natural/blob/master/org.agileware.natural.cucumber/src/org/agileware/natural/cucumber/Cucumber.xtext

contained in the *duplication detector* sub-component. The rules and algorithms discussed earlier are also implemented in this sub-component. They are used to process the AST (generated from the parser component) for duplications. The *refactoring provider* sub-component consists of the refactoring methods discussed in Section 3.8 and is in charge of removing duplications from Cucumber features.

Figure 4.4 illustrates a deeper look at SEED's architecture and *where* SEED's components are actually called/executed *during* runtime (i.e. when SEED is running in Eclipse). As seen in the figure, *Xtext* plays a pivotal role in the runtime of SEED. Xtext[28] is a framework that assists in the development and integration of domain specific languages (e.g. Gherkin) on the Eclipse IDE. It simplified the job of creating SEED by automating some of the work and removing the need of creating the plug-in from scratch. Xtext provides the necessary dependencies (libraries, files, and source folders) and requirements (manifest) for an Eclipse plug-in to work. All we had to do in order for SEED to fully function was to implement the duplication detection (and refactoring) code for Xtext to call/execute and the grammar for the parser. Xtext can be seen as the *foundation* of SEED's architecture and is responsible for facilitating the communication between Eclipse and SEED's components. When a Cucumber feature is opened (by the user) on Eclipse's editor, Eclipse *alerts* Xtext to start the duplication detection process. Xtext then passes this "message" on to SEED. During runtime/duplication detection, Xtext facilitates the execution of the parser (i.e. we do not have to *explicitly* call the parser in the implementation code) and passing of the AST (from the parser) to the duplication detection methods (as input parameters). SEED passes (any) detected duplications (in the form of wrappers) and refactoring outcomes (updated/new strings) to Xtext before Xtext hands them over to Eclipse. Eclipse then proceeds to update the Cucumber feature(s) accordingly i.e. underline duplicates and/or replace duplicates with new strings.

---

[28] *Xtext* [Online]. Available at: https://eclipse.org/Xtext/index.html

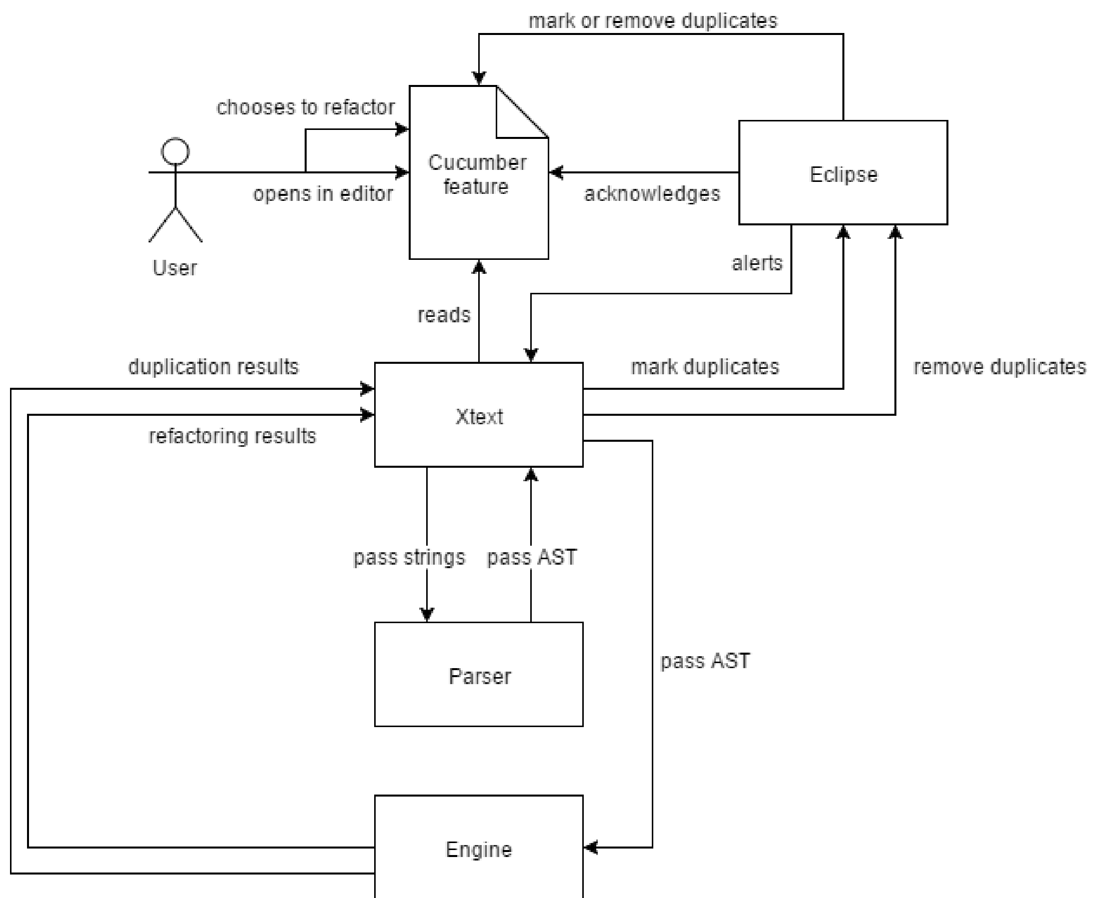

Figure 4.4: Detailed outlook of SEED's architecture.

4.2.2 Domain Model Diagram

Following the architecture of the system shown in the previous section, a conceptual model was constructed. In software engineering, this is known as a domain model. It represents a high level abstraction of real world entities and their relationships that cover the problem domain [28]. The model can go on to be translated into code and be used to solve problems related to the problem domain. In other words, the domain model diagram describes a link between the problem domain (in the real world) and the code. The *purpose* of creating the diagram was to illustrate how the (key) components within the SEED architecture interact with one another. The domain model diagram for SEED is shown in Figure 4.5 and was constructed/derived from the implementation code. Most of the models in the diagram have already been mentioned in the previous sections. The *user* model corresponds to the person who is using SEED and is writing the Cucumber features.

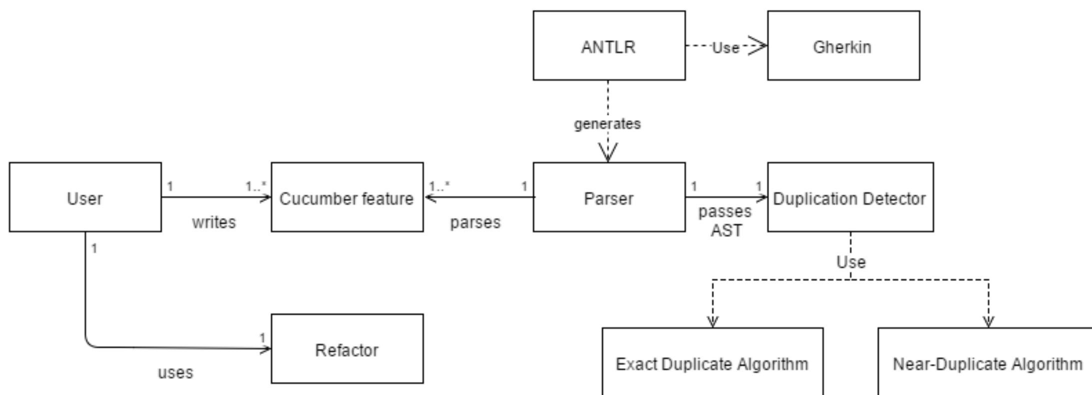

Figure 4.5: Domain Model of SEED.

4.2.3 Class Diagram

The classes in the domain model was then translated into programming code for a more detailed outlook on the software structure. These detailed models are illustrated as *classes* in a class diagram (as shown in Figure 4.6) which was also reverse-engineered from the implementation code. The arrows indicate *dependency* whereby the entity at the back of an arrow is dependent on the entity that the arrow is pointing to. Looking at the diagram, each class has its own responsibilities and minimal dependencies to one another. The *GherkinValidator* class was *generated* from Xtext. It is called when SEED wants to detect duplications in Cucumber features (after parsing). *GherkinValidator* then triggers the necessary classes into carrying out specific tasks. It acts as a gateway to the rest of the classes and is therefore, dependent on those classes. The classes are as follows:-

- *CucumberDuplicateChecker*. This class contains methods responsible for detecting exact- and near- duplicates.
- *CucumberBackgrounder*. This class contains methods responsible for identifying repeated pre-condition steps in the Cucumber feature.
- *CucumberFileHandler*. This class contains methods responsible for comparing all Cucumber features within the Eclipse project workspace for duplications.
- *GherkinQuickfixProvider*. This is the class of the refactoring provider component. It is dependent on the *GherkinValidator* class because each refactoring option/method is unique to the type of duplication found.

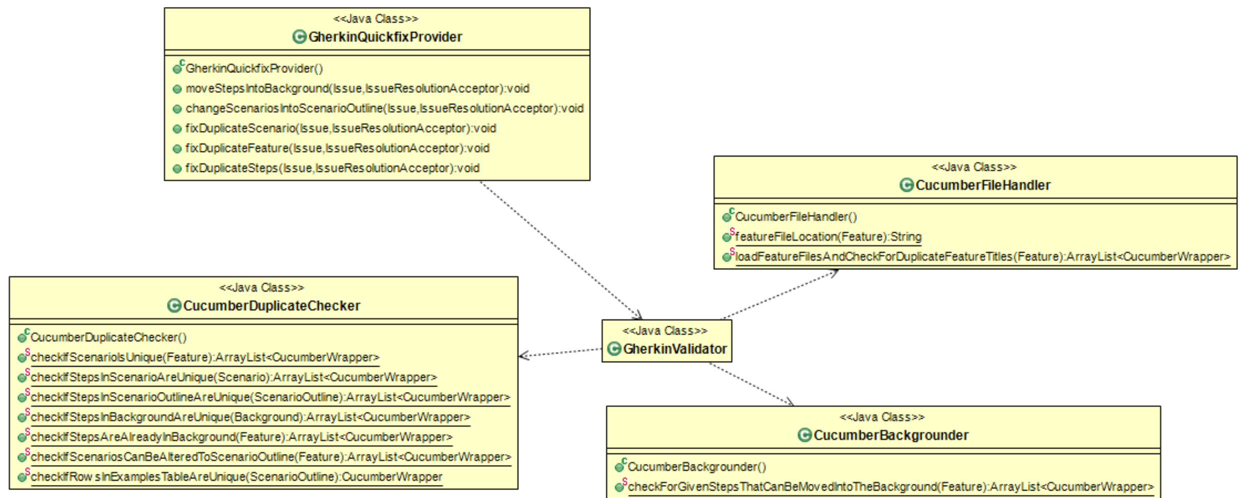

Figure 4.6: Class Diagram.

4.3 Version Control System

"*A version control system is a repository of files, often the files for the source code of computer programs, with monitored access. Every change made to the source is tracked, along with who made the change, why they made it, and references to problems fixed, or enhancements introduced, by the change.*" [29]

It is important for any software development project to adopt a backup strategy for its source code. This is to avoid the risk of losing any of the project's important files. Also, any unwanted changes made to the code can be reverted to its previous version.

For this project, *Git*[29] was the tool used to handle the version control. Backup copies of the source files are stored in a *remote* repository. The working/original copy is kept in the *local* environment/repository for development. Git offers several useful commands for manipulating these repositories but the following had the most uses in the project:-

- `Commit`. Save any changes to the source files into the local repository.
- `Push`. Send these changes to the remote repository for saving.
- `Pull`. Retrieve the latest copy of the files from the remote repository and into the local repository.

[29] *Git* [Online]. Available at: https://git-scm.com/

GitHub and *GitLab* were used in this project as well. They are web-based versions of Git i.e. they provide access to remote repositories through a web browser and various other protocols. GitLab (a repository server provided by the University of Manchester) was used to store the source code whereas a public GitHub account served as the download/update site for users to install the SEED plugin. Eclipse requires a URL/site for downloading and installing plugins into the IDE. They can be accessed through the following links:-

- GitLab: https://gitlab.cs.man.ac.uk/waisuan.sia/seed
- GitHub: https://github.com/waisuan/SEED

4.4 Functionality

The following sub-sections illustrates examples and the pseudocode of SEED's implemented functionality. Each functionality conforms to the rules defined in Section 3.6.

4.4.1 Duplicate Cucumber Feature Titles Detection

The pseudocode for detecting Cucumber feature titles is shown in Figure 4.7. The process of detecting duplications is slight different from detecting them within individual Cucumber features. SEED has to firstly read all of the files in the currently active project workspace on Eclipse. SEED then iterates through the files and a file is detected for duplication if it is a Cucumber feature file and it is *not* the same file as the currently opened/active Cucumber feature file (on the Eclipse IDE editor). After a file has been selected for detection, SEED's parser converts the file into an abstract syntax tree (AST). We will refer to this file as *other feature* to distinguish itself from the active feature file. The *titles* are then extracted from the active Cucumber feature and the other feature. Both titles are subjected to the removal of extra spacing and changed into lower cases so that they do *not* escape from (possibly) being detected as duplications. As seen in the pseudocode, exact matches (==) are first checked. If no matches were found, the titles are then checked for near-duplications (*diceCoefficient*).

The input of this algorithm/pseudocode is the (active) Cucumber feature node/object from the feature's AST. The output, however, is a *list* of duplications that were detected. Each duplication (i.e. Cucumber feature object) is stored in a wrapper object which was briefly discussed in Section 3.7.4. Figure 4.8 and 4.9 shows examples of duplicate Cucumber feature titles detected by SEED.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

let aF = all of the files in the current Eclipse project workspace

FOR EACH file IN aF DO
    IF file has a ".feature" extension AND it is not the same file as f THEN
        let otherF = parse(file)

        featureTitle = removeExtraSpacingAndChangeToLowerCase(f.title)
        otherFeatureTitle = removeExtraSpacingAndChangeToLowerCase(otherF.title)

        IF featureTitle == otherFeatureTitle THEN
            IF l is empty THEN
               let w = wrapper(f)
               add w to l
            END IF

            let w = wrapper(otherF)
            add w to l
        ELSEIF diceCoefficient(featureTitle, otherFeatureTitle) >= 73% THEN
            IF l is empty THEN
               let w = wrapper(f)
               add w to l
            END IF

            let w = wrapper(otherF)
            add w to l
        END IF

    END IF
END FOR
```

Figure 4.7: Pseudocode for detecting duplicate Cucumber feature titles.

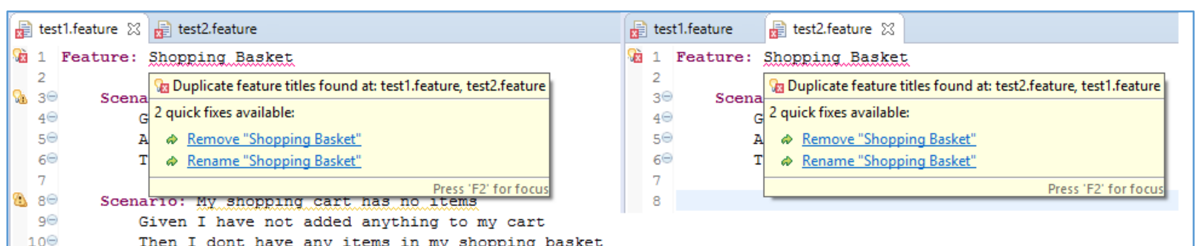

Figure 4.8: Two different feature files with the same title.

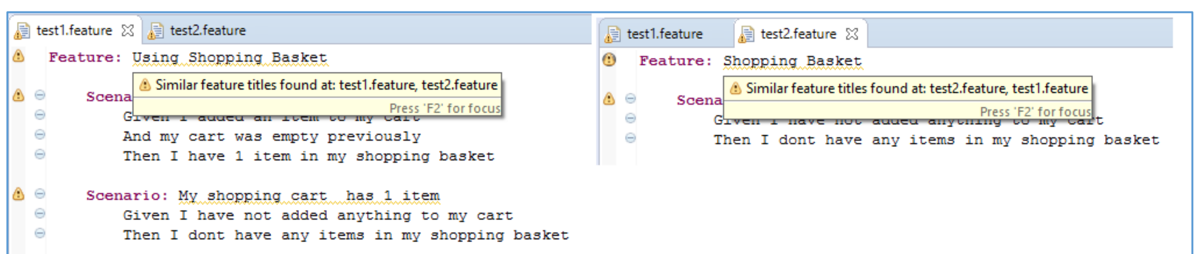

Figure 4.9: Two different feature files with similar titles.

4.4.2 Duplicate Cucumber Scenario Titles Detection

The pseudocode for detecting duplicate Cucumber Scenario titles (in a feature) is shown in Figure 4.10. The logic/process applies to Scenario Outline titles as well. The steps are quite similar to detecting duplicate feature titles in terms of detecting exact- and near-duplications. As seen in the pseudocode, the implementation consists of *nested* loops. SEED iterates through *all* of the scenarios in the feature for the detection process. For each scenario in the loop, SEED then iterates through *all* of the scenarios (again) excluding the current scenario (that is being compared against) and compare their titles. Figure 4.11 and 4.12 illustrates examples of duplicate scenario titles detected by SEED.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

FOR EACH scenario IN f.getAllScenarios DO
    scenarioTitle = removeExtraSpacingAndChangeToLowerCase(scenario.title)

    FOR EACH otherScenario IN f.getAllScenarios DO
        IF scenario != otherScenario THEN
            otherScenarioTitle = removeExtraSpacingAndChangeToLowerCase(otherScenario.title)

            IF scenarioTitle == otherScenarioTitle THEN
                IF l is empty THEN
                   let w = wrapper(scenario)
                   add w to l
                END IF

                let w = wrapper(otherScenario)
                add w to l
            ELSEIF diceCoefficient(scenarioTitle, otherScenarioTitle) >= 73% THEN
                IF l is empty THEN
                   let w = wrapper(scenario)
                   add w to l
                END IF

                let w = wrapper(otherScenario)
                add w to l
            END IF

        END IF

    END FOR

END FOR
```

Figure 4.10: Pseudocode for detecting duplicate Cucumber Scenario titles.

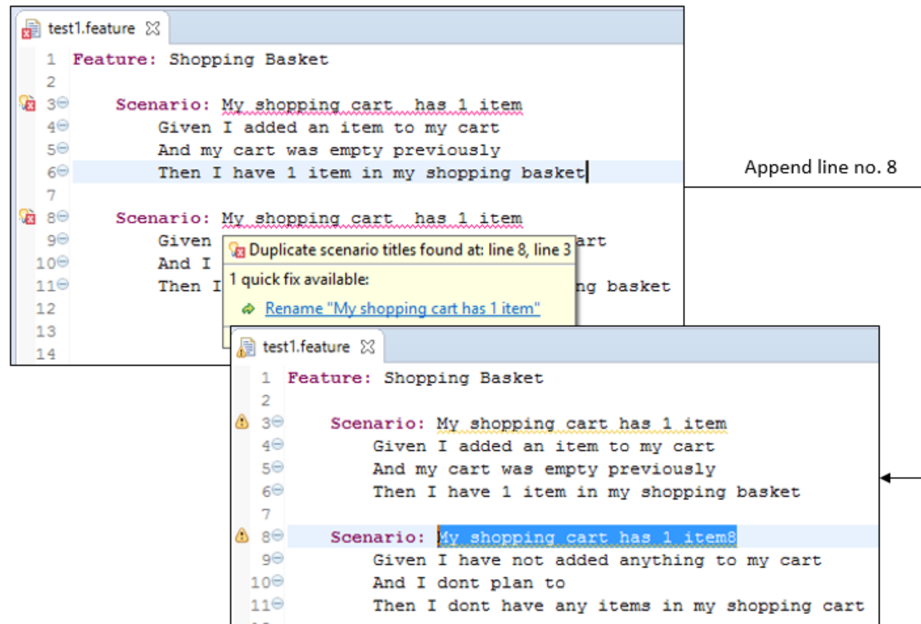

Figure 4.11: Two scenarios with the same title/description.

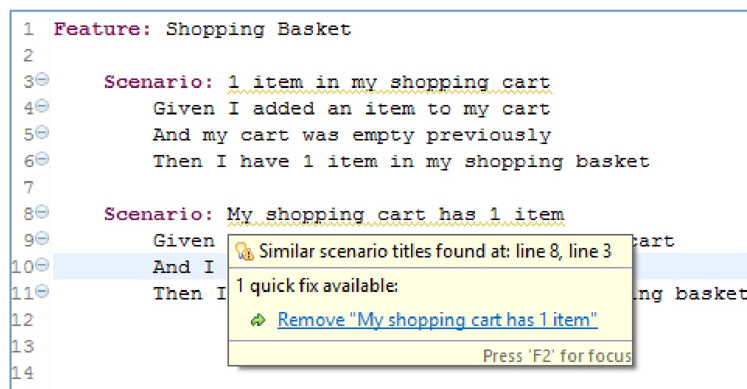

Figure 4.12: Two scenarios with equivalent title/descriptions.

4.4.3 Duplicate Cucumber Scenario Steps Detection

The pseudocode for detecting duplicate list of Cucumber steps between scenarios is shown in Figure 4.13. The pseudocode is slightly different from the earlier pseudocodes. There are a total of *three* (nested) loops. The first two loops serve the same purpose of iterating through all of the scenarios. The third loop compares both scenarios *step-by-step*. The comparison is done between the step's descriptions (strings). Again, the descriptions are first checked for exact matches before checking for near-duplications. However, the duplications (i.e. Cucumber scenario objects) are not added to the *list of wrappers* right away. Since we are dealing with a *list* of steps, each step has to be compared before determining whether both scenarios have the same list of steps. The *numberOfDuplicationsDetected* counter is responsible for keeping

track of the number of duplicate steps that has been detected so far. After each step has been compared, the counter is compared against the total number of steps that the scenarios have. If they are equal, it means that there are duplicates. If not, it means that there are different step(s) in each of the scenario. Figure 4.14 shows an example of duplicate list of steps detected by SEED.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

FOR EACH scenario IN f.getAllScenarios DO
    listOfSteps = scenario.getAllSteps

    FOR EACH otherScenario IN f.getAllScenarios DO
        IF scenario != otherScenario THEN
            listOfOtherSteps = otherScenario.getAllSteps

            IF sizeOf(listOfSteps) == sizeOf(listOfOtherSteps) THEN
                let numberOfDuplicationsDetected = 0

                FOR EACH step IN listOfSteps AND otherStep IN listOfOtherSteps DO
                    IF step.keyword == otherStep.keyword THEN
                        stepDescription = removeExtraSpacingAndChangeToLowerCase(step.description)
                        otherStepDescription = removeExtraSpacingAndChangeToLowerCase(otherStep.description)

                        IF stepDescription == otherStepDescription THEN
                            numberOfDuplicationsDetected = numberOfDuplicationsDetected + 1
                        ELSEIF diceCoefficient(stepDescription, otherStepDescription) >= 73% THEN
                            numberOfDuplicationsDetected = numberOfDuplicationsDetected + 1
                        END IF

                    END IF

                END FOR

                IF numberOfDuplicationsDetected == sizeOf(listOfSteps) THEN
                    IF l is empty THEN
                        let w = wrapper(scenario)
                        add w to l
                    END IF

                    let w = wrapper(otherScenario)
                    add w to l
                END IF

            END IF

        END IF

    END FOR

END FOR
```

Figure 4.13: Pseudocode for detecting duplicate list of Cucumber steps.

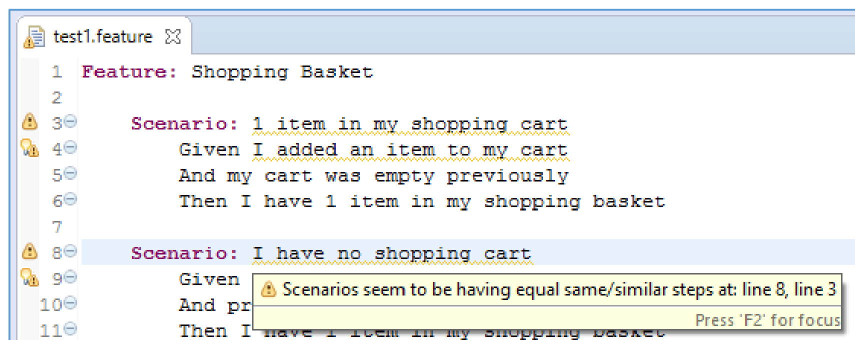

Figure 4.14: Two different scenarios having similar list of steps.

4.4.4 Duplicate Cucumber Examples Table Rows Detection

The pseudocode for detecting duplicate rows in Cucumber Examples tables is shown in Figure 4.15. The input is a Cucumber Scenario outline object. For this implementation, a *collection* object/data structure is used to track the duplicates in the table. The collection does not contain duplicate elements. Before adding a row into the collection, if the row already exists in the collection, then duplication is found. Also, each row is compared in its entirety i.e. not cell by cell. This is done by (first) converting the entire row into a *string*. Figure 4.16 shows an example of duplicate table rows detected by SEED.

```
input: Scenario Outline, so
output: A single wrapper object containing the duplications found, w

let set = a data structure that contains no duplicate elements

FOR EACH row IN so.getAllExamplesTableRows DO
    let rowString = convertRowIntoString(row)

    IF set already contains rowString THEN
        w = wrapper(so)

        break loop;
    END IF

    add rowString to set

END FOR
```

Figure 4.15: Pseudocode for detecting duplicate Examples table rows.

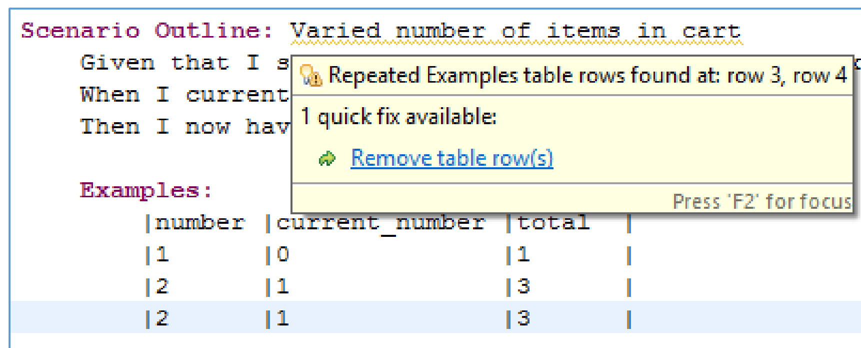

Figure 4.16: Repeated rows in Examples table.

4.4.5 Duplicate Cucumber Steps Detection

The pseudocode for detecting duplicate Cucumber steps within a Scenario/Scenario Outline/Background is shown in Figure 4.17. The implementation code is very similar

to detecting duplicate Scenario titles (Section 4.4.2). Instead of comparing titles, the step descriptions are compared for duplications. Figure 4.18 shows an example of duplicate steps detected by SEED.

```
input: List of Cucumber steps belonging to Scenario/Scenario Outline/Background, lcs
output: List of wrappers containing the duplications found, l

FOR EACH step IN lcs DO
    stepDescription = removeExtraSpacingAndChangeToLowerCase(step.description)

    FOR EACH otherStep IN lcs DO
        IF step != otherStep THEN
            otherStepDescription = removeExtraSpacingAndChangeToLowerCase(otherStep.description)

            IF step == otherStep THEN
                IF l is empty THEN
                   let w = wrapper(step)
                   add w to l
                END IF

                let w = wrapper(otherStep)
                add w to l
            ELSEIF diceCoefficient(stepDescription, otherStepDescription) >= 73% THEN
                IF l is empty THEN
                   let w = wrapper(step)
                   add w to l
                END IF

                let w = wrapper(otherStep)
                add w to l
            END IF

        END IF

    END FOR

END FOR
```

Figure 4.17: Pseudocode for detecting duplicate Cucumber steps.

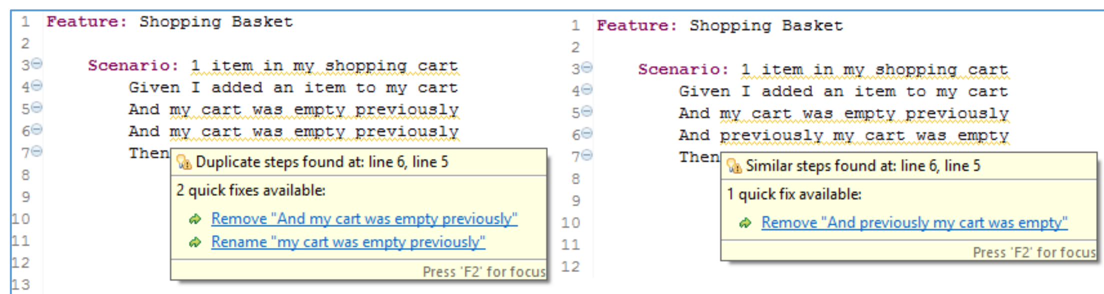

Figure 4.18: Exact and equivalent steps detected.

Figure 4.19 shows the pseudocode for detecting steps (in scenarios/scenario outlines) that already exist in the Cucumber background section. The implementation code works by comparing each step in the background against each step in every scenario/scenario outline. Figure 4.20 shows an example of steps that are already in the background section.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

let background = f.getBackground

FOR EACH backgroundStep IN background.getAllSteps DO
    backgroundStepDescription = removeExtraSpacingAndChangeToLowerCase(backgroundStep.description)

    FOR EACH scenario IN f.getAllScenarios DO
        FOR EACH step in scenario.getAllSteps DO
            stepDescription = removeExtraSpacingAndChangeToLowerCase(step.description)

            IF backgroundStepDescription == stepDescription THEN
                let w = wrapper(step)
                add w to l
            ELSEIF diceCoefficient(backgroundStepDescription, stepDescription) >= 73% THEN
                let w = wrapper(step)
                add w to l
            END IF

        END FOR

    END FOR

END FOR
```

Figure 4.19: Pseudocode for detecting steps that already exist the Background section.

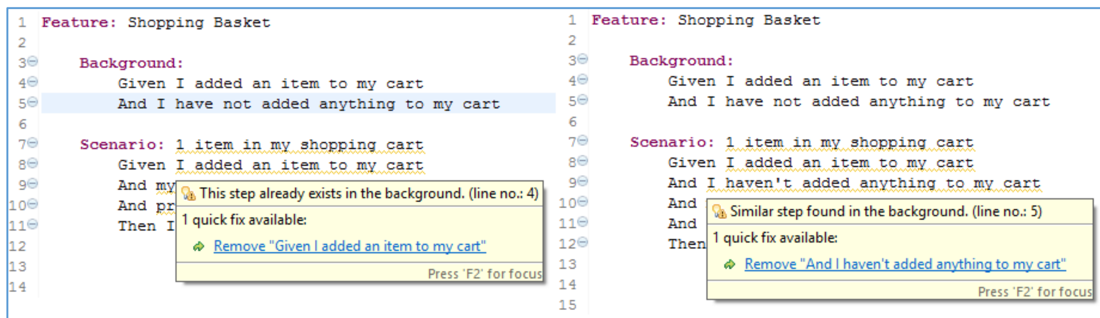

Figure 4.20: Exact and equivalent steps already exist in the background section.

4.4.6 Duplicate Cucumber Pre-condition Steps Detection

The pseudocode for detecting pre-condition steps that exist in *every* Cucumber scenario (or scenario outline) in the feature is shown in Figure 4.21. The *getAllPreconditionSteps* is responsible for gathering the *given-and* steps (there may be more than one) from a scenario. As for comparing the precondition steps, the method is similar to Section 4.4.3 where the comparison is done between lists of steps. A *COUNTER* is used to keep track of the number of duplicates. If the number matches the total number of scenarios in the feature, then there are duplicates. Figure 4.22 shows the refactoring provider of SEED suggesting to put the repeated pre-conditions into the background section.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

let firstScenario = getFirstScenarioFromCucumberFeature(f)

let preconditionStepsInFirstScenario = getAllPreconditionSteps(firstScenario)

let restOfTheScenarios = getRestOfScenariosFromCucumberFeature(f)

let COUNTER = 1

FOR EACH scenario IN theRestOfTheScenarios DO
    preconditionStepsInOtherScenario = getAllPreconditionSteps(scenario)

    IF preconditionStepsInFirstScenario == preconditionStepsInOtherScenario THEN
        COUNTER = COUNTER + 1
    ELSEIF diceCoefficient(preconditionStepsInFirstScenario, preconditionStepsInOtherScenario) >= 73% THEN
        COUNTER = COUNTER + 1
    END IF

END FOR

IF COUNTER == sizeOf(f.getAllScenarios) THEN
    let w = wrapper(preconditionStepsInFirstScenario)
    add w to l

    FOR EACH scenario IN theRestOfTheScenarios DO
        preconditionStepsInOtherScenario = getAllPreconditionSteps(scenario)

        let w = wrapper(preconditionStepsInOtherScenario)
        add w to l
    END FOR
END IF
```

Figure 4.21: Pseudocode for detecting pre-condition steps repeated in every scenario.

```
test1.feature
1  Feature: Shopping Basket
2
3     Background:
4      Given blank
5
6     Scenario: 1 item in my shopping cart
7         Given I added an item to my cart
8         And my cart was empty previously
9         Then I have 1 item in my shopping basket
10
11    Scenario: My shopping cart has 1 item
12        Given I added an item to my cart
13        Given I have not added anything to my cart
14        And I dont plan to
15        Then I dont have any items in my shopping cart
16
17    Scenario Outline: Varied number of items in cart
18        Given I added an item to my cart
19        Given
20        When I
21        Then I
22
23        Exampl
24            |number |current_number |t
25            |1      |0              |1
26            |2      |1              |3
27            |2      |1              |3
28
29
```

This step has been repeated in every scenario. Do you want to move it into the Background?
1 quick fix available:
Put step(s) into background.

Refactor

```
*test1.feature
1  Feature: Shopping Basket
2
3     Background:
4         Given blank
5         Given I added an item to my cart
6
7     Scenario: 1 item in my shopping cart
8         And my cart was empty previously
9         Then I have 1 item in my shopping basket
10
11    Scenario: My shopping cart has 1 item
12        Given I have not added anything to my cart
13        And I dont plan to
14        Then I dont have any items in my shopping cart
15
16    Scenario Outline: Varied number of items in cart
17        Given that I select <number> of items from the stock
18        When I currently have <current_number> in my cart
19        Then I now have a total of <total> items in my cart
20
21        Examples:
22            |number |current_number |total  |
23            |1      |0              |1      |
24            |2      |1              |3      |
25            |2      |1              |3      |
26
```

Figure 4.22: Pre-condition steps repeated in every scenario of the feature.

4.4.7 Duplicate Cucumber Scenarios Detection

Figure 4.23 shows the pseudocode for detecting whether two or more scenarios in a feature can be combined into a single scenario outline. This is done by comparing the scenarios' titles and steps. If they are different by a *single* value/word, then they can be combined. For example, "the cow weighs **450** kg" and "the cow weighs **500** kg" differ in only their weight. Figure 4.24 depicts scenarios that have different input/output values and SEED offers the option of refactoring them. The "?" placeholder in the generated scenario outline allows the user (of SEED) to rename the column headers according to his/her preference.

```
input: Active Cucumber feature on Eclipse IDE editor, f
output: List of wrappers containing the duplications found, l

let allScenarios = f.getAllScenarios

filteredScenarios = compareScenarioTitlesForDifferences(allScenarios)

possibleScenarioOutlineCandidates = compareScenarioStepsForDifferences(filteredScenarios)

FOR EACH scenario IN possibleScenarioOutlineCandidates DO
    let w = wrapper(scenario)
    add w to l
END FOR
```

Figure 4.23: Pseudocode for detecting similar scenarios.

```
Scenario: feeding a small suckler cow
  Given the cow weighs 450 kg
  When we calculate the feeding requirements
  Then the energy should be 26500 MJ
  And the protein should be 215 kg

Scenario: feeding a medium suckler cow
  Given th
  When we
  Then the
  And the
```

```
Similar scenario titles found at: line 35, line 29
1 quick fix available:
Convert scenario(s) into a Scenario Outline
```

```
Scenario Outline: feeding a suckler cow
    Given  the cow weighs <?> kg
    When  we calculate the feeding requirements
    Then  the energy should be <?> MJ
    And  the protein should be <?> kg
    Examples:
        | ? | ? | ? |
        |450|26500|215|
        |500|29500|245|
```

Figure 4.24: Scenarios that differ in their input/output values can be combined into a scenario outline (through refactoring/quick fix).

4.5 Testing

Testing is an important aspect of any software engineering project. The purpose of creating tests is to ensure that the software application works as expected and fulfils

the needs of the customer/user. It is no different for this project and the development of SEED.

4.5.1 Unit Testing with Xtext

"*At a high-level, unit testing refers to the practice of testing certain functions and areas – or units – of our code. This gives us the ability to verify that our functions work as expected.*" [30]

Unit tests help ensure that the methods in SEED work as expected. Each failing test indicates a possible error in the implementation. This helps a lot when changes are made to source code or when new methods are introduced. By having a suite of unit tests, the functionality of SEED can be quickly and continually verified. When a test fails after the changes have been made, the new piece of code can be quickly debugged/fixed. SEED's unit tests are ran frequently to ensure that the functionality is not broken at any time during development.

Xtext and JUnit were used to facilitate the testing process. JUnit[30] is a unit testing framework designed for testing Java code. Xtext simplified the procedure of testing SEED. Due to the nature of SEED's functionality i.e. it relies on *parsed* Cucumber feature files in order to detect duplications, the parsing of the Cucumber features had to be simulated before SEED's functions could be tested. Xtext provides the tools/libraries necessary for carrying out these tests. These tools can be seen on the test code (for detecting duplicate scenario titles) shown in Figure 4.25. The test begins by initialising a *string* consisting of a Cucumber feature with two scenarios. The *parser* object is provided and executed by Xtext. It is used to convert the string into a Java object (i.e. *model*). The *assertNoErrors* method is also provided by Xtext and its function is similar to JUnit's *assertion* methods. Internally, the method would go on to call SEED's duplication detection methods. In this case, the method is expecting *no* errors (i.e. no duplications) from the execution of the methods. The execution of the test is done by JUnit (called by Xtext) i.e. provides the pass or fail results for the tests.

---

[30] *JUnit* [Online]. Available at: http://junit.org/

```
@Test
def void testDuplicateScenario() {
    val model = parser.parse(
        "Feature: MyFeature \n\n
            Scenario: MyScenario \n\n Given this is a step \n\n
            Scenario: MyScenario \n\n Given this is also a step"
    );
    validationTestHelper.assertNoErrors(model);
}
```

Done by Xtext

Cucumber feature object

Done by Xtext

Figure 4.25: Unit test code for detecting duplicate scenario titles.

Unfortunately, Xtext does not provide a method for *expecting* errors in the tests. Therefore, the expectation of running SEED's tests is to see *failures*. In this case, a failed test is *equivalent* to passing the test. That is, duplication should be detected and the appropriate warning/error messages should be displayed (as would be shown on the Eclipse IDE editor). Figure 4.26 shows SEED's suite of tests cases.

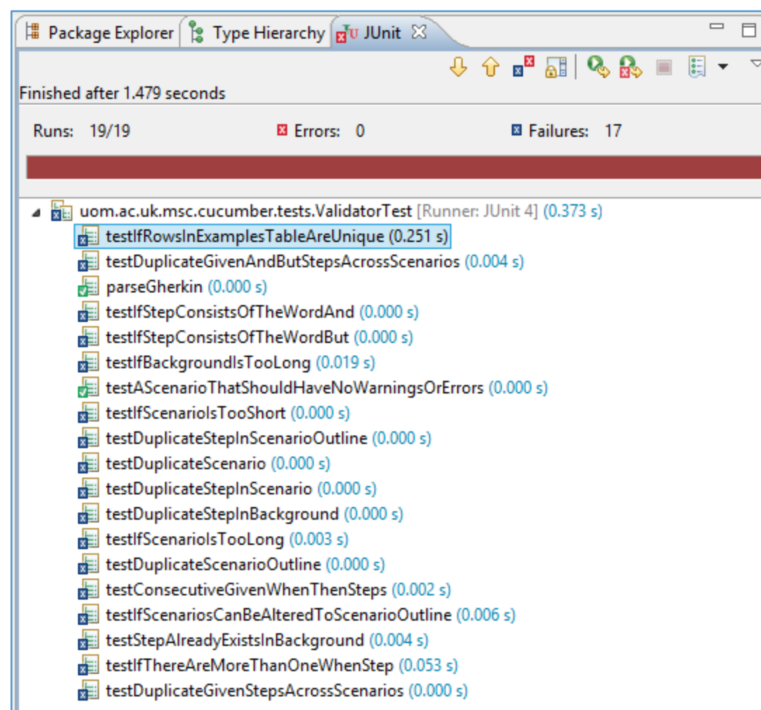

Figure 4.26: JUnit test cases for SEED.

Figure 4.27 shows the failure message containing the expected and actual output results after running the test (from Figure 4.25). By looking at the failure message, SEED's functionality works as intended since duplicate scenario titles should be

detected. The test is stating that it was expecting no errors from the Cucumber feature but it received an error for the respective duplication detected.

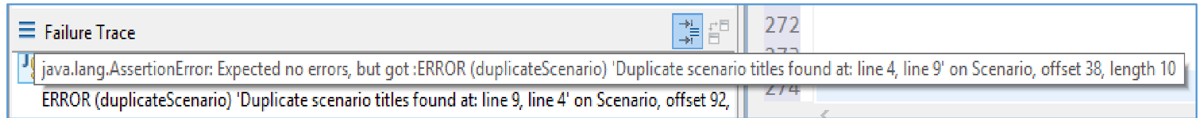

Figure 4.27: Expected vs. actual results from a failed unit test.

4.6 Conclusion

The chapter showed an overview of SEED's architectural design. Also, SEED's implementation details were shown in the form of pseudocode and example diagrams. The next chapter will be examining the evaluation process of the project as well as answer the project's research question.

Chapter 5 : Evaluation

5.1 Overview

The final phase of the project consists of evaluating the rules proposed in Section 3.6 for detecting and removing duplications in Cucumber features. This chapter begins by laying out the purpose of evaluation and the expected evaluation results. It will then go on to examine the approach taken for the evaluation process and the series of experiments undertaken in the process. The results generated from the evaluation process are compared with our predicted results and producing an answer to the research question proposed in the project.

5.2 Hypothesis & Prediction

The research question can be broken down into the following:-

i. *Can a software tool detect the same duplications in Cucumber features as human experts do?*
ii. *Can a software tool provide refactoring operations in Cucumber features that human experts think are worth applying?*

The results in the evaluation process can be divided into several classes. This classification is illustrated in Figure 5.1 and their definitions are as follows:-

- *False Positives*. Results in this class indicate that a Cucumber feature has duplications when in fact it does not. Additionally, SEED proposes refactoring changes that worsens the quality of the Cucumber feature.
- *True Positives*. Results in this class indicate that a Cucumber feature has duplications and it truly does. Additionally, SEED proposes refactoring changes that genuinely improves the quality of the Cucumber feature. The *intersection* in the figure refers to the duplication/refactoring proposals detected by both SEED and the human experts i.e. confirmed as true by the human experts.
- *False Negatives*. Results in this class indicate that a Cucumber feature does not have duplications when in fact it does.

If the tool does well, it would spot the same sets of duplications and make the same refactorings as human experts do. We will declare SEED a success if it detects some (though probably not all) of the duplications found by the human experts (of

Cucumber) and proposes refactoring changes that match the actions of the human experts. We also aspire to a small number of *false positives* from SEED and *high* proportion of confirmed *true positives*.

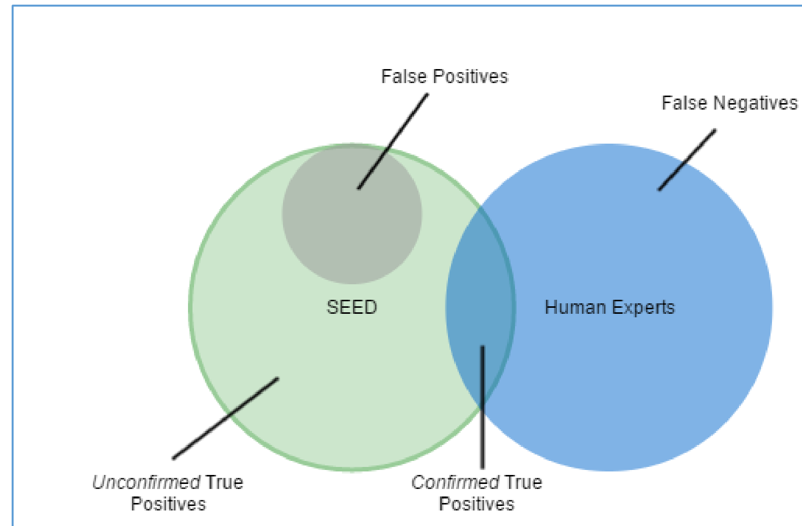

Figure 5.1: Classification of evaluation results.

5.3 Approach

The approach taken for evaluating the effectiveness and usefulness of SEED was to run SEED against a set of Cucumber features from external/third-party software development projects. The projects were chosen for their use of the Cucumber tool. They are also open-source and hosted on GitHub. Fortunately, the Cucumber team has compiled a list of projects that specifically used Cucumber on Cucumber's own GitHub page. This list can be found in the following link:-

https://github.com/cucumber/cucumber/wiki/Projects-Using-Cucumber

The aim of running these tests against SEED is to determine if it is able to detect the same issues as developers of the project did with the tests and refactor them in similar ways as the developers. This method of comparison utilizes the commit/change log that GitHub provides for each file in the project repository. The log encompasses the history of changes that have been made to each file. Each commit in the changelog represents one snapshot of the whole project. An example of a GitHub changelog (taken from the Cucumber project: https://github.com/cucumber/cucumber-ruby) is shown in Figure 5.2. From the log, specific answers can be deduced to questions such as "*what was the original content of the Cucumber feature?*" and "*what changes did the developer(s) make to the feature and were the changes related to removing*

duplications from the feature?" These answers help determine whether SEED is doing the same or similar things as the developers.

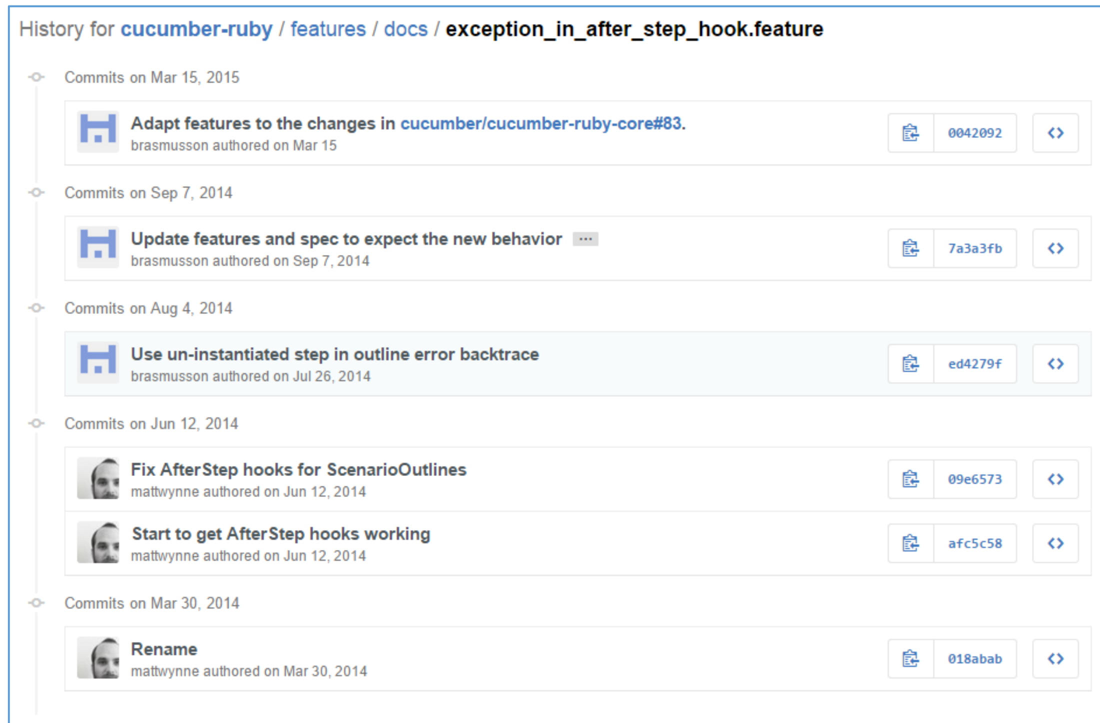

Figure 5.2: History of Git commits.

The reason for using GitHub projects in the evaluation process is due to the need for subjective views of human experts on the Cucumber features and for us to compare SEED's actions to. It would prove challenging to gather together lots of Cucumber developers into a single location and carry out the evaluation process in their presence. Therefore, as a proxy for the opinions of human experts, the GitHub changelog was used. The changelog denotes past actions of the developers/human experts. An assumption of this evaluation approach is that all changes made by the human experts in the changelog are assumed to be necessary and *right*. SEED's behaviour would then be matched against these changes for verification.

The steps undertaken in the evaluation approach were as follows:-

1. Select a project (that has Cucumber features).
2. Select a Cucumber feature from the project.
3. If the feature does not have any changes in its changelog, go back to step 2.
4. Select the first/initial version of the feature file from the changelog.
5. Run the feature file against SEED (in the Eclipse IDE).

6. Identify the warnings/errors generated by SEED.
7. Examine the versions/snapshots after the first version of the feature file. The examination includes noting down the differences between versions i.e. what changes were made by the developers.
8. Deduce whether the duplication issues identified and changes made to each version correspond to SEED's behaviour.
   - Do the refactoring suggestions match with what the developers have done?
     - If so, are they sensible?
9. Repeat steps 2 – 8 until every cumber feature in the project has been examined/evaluated.
10. Go back to step 1 *until* three projects have been examined.

After every Cucumber feature has been tested against SEED, the results are gathered and analysed (as shown later on). The *experiments* (on the projects) are conducted in order to gather the results. The *discussion* section will be assessing the success criteria of the evaluation process.

5.4 Experiment 1

The Cucumber features used in this experiment were taken from the official *Cucumber* project stored in the following repository:-

https://github.com/cucumber/cucumber-ruby

Running SEED against the Cucumber features that were created by the Cucumber team themselves is a good indication of how useful SEED is. The expectation here is that the Cucumber team is following their own recommended practices for writing Cucumber features and that they are also careful about what to put on them. SEED will then try to detect any (bad) duplications that the team might have missed out on. Table 5.1 shows the total number of Cucumber features that were evaluated against and analysed as part of the experiment.

Total No. of Cucumber features	Total No. of commits/changes
69	375

Table 5.1: Total amount of feature files and commits in the Cucumber project.

Since this was the first experiment conducted, the prediction made here was that *most* of the exact/equal duplications would be detected by SEED. There might be some similarity matches/near-duplicates detected as well. Also, every refactoring proposal by SEED would match with the human expert's commit actions.

The experiment began by identifying and splitting different types of errors/warnings that were discovered by both SEED and the Cucumber developers. The findings were then calculated and tabulated into Table 5.2. *Duplication* refers to the total number of duplications found (by SEED) within the Cucumber features and their respective versions in the changelog. Two or more of the same duplicates are calculated as *one* duplication. *Errors* refer to issues detected when parsing the Cucumber features e.g. missing scenario titles.

Type	Count
Duplication	55
Errors	7

Table 5.2: Count of duplications & errors detected within the Cucumber features.

5.4.1 Duplication

The commits in the changelog help determine whether the human experts have detected duplications in a Cucumber feature. A commit related to duplications is understood to mean that the human experts have detected duplications in the feature and has made an attempt to fix them *if* the changes made (in the commit) relates to removing duplications.

Duplication	Count of Duplicates
Detected by SEED and Human Expert	6
Should have been detected by Human Expert	22
Should not have been detected by SEED	27

Table 5.3: Count of duplications detected.

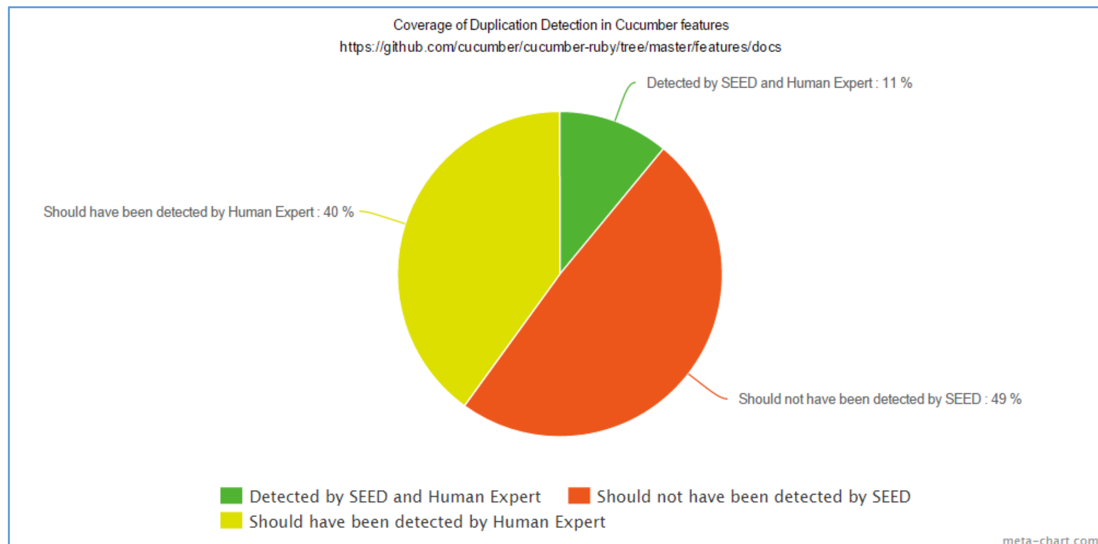

Figure 5.3: Coverage of Duplication Detection in Experiment 1.

Figure 5.3 shows an in-depth analysis of the total number of duplications found in the Cucumber features. The results for the Cucumber project were *positive* in terms of duplications coverage. SEED managed to detect every single duplication that the human experts (i.e. developers) found (i.e. *confirmed true positives*) and also those that were not found by the human experts. According to Table 5.3, 6 out of the 55 duplicates (10%) were detected by both SEED and the human experts. The human experts only realized the duplications after the Cucumber feature has been created and therefore, required a separate/new commit in order to rectify this error. The duplications found were from exact matches. There were a total of 6 exact match duplications found by SEED.

40% of the duplications found by SEED were not detected by the human experts (i.e. *unconfirmed true positives*). However, they should have been. This part of the coverage was determined by whether the duplications corresponded to the rules stated in Section 3.6 and the severity of the duplications. If the duplications were caused by exact matches between two/more pieces of text in the Cucumber feature, then it should have been resolved by the human experts. The near-duplicates were analysed before they were tabulated in order to determine if they were considered as true or false positives i.e. “were they semantically or syntactically equivalent?” The analysis was done by referring to the discussion in Section 3.7.2. Additionally, if the near-duplicates degraded the readability of the test (the similarities were *too* close to one another such

that it was difficult to distinguish between them), they were put into this category. The duplications consisted of 7 exact- and 15 near-duplicates.

The rest of the duplications (49%) detected by SEED were considered as *false positives*. They were not detected by the human experts and (possibly) should *not* have been detected by SEED as well. The duplications stemmed from the similarity matching portion of the Cucumber features. There were a total of 27 near-duplicates found. Apart from *one* false near-duplication, the rest of the similarities were correct in terms of being near-duplicates. For example, "Exception *before* the test case is run" and "Exception *after* the test case is run". Nevertheless, they did not fit into the rule of being near-duplicates within the context of BDD/Cucumber as discussed in Section 3.7.2 i.e. semantically equivalent.

5.4.2 Refactoring

In order to determine if the refactoring changes made by the human experts towards the Cucumber features were the same as SEED's suggestions, the duplications detected by both SEED and the human experts had to be further examined. The findings are shown in Table 5.4.

5 out of the 6 duplicates found by both SEED and the human experts had the same refactoring suggestions by both SEED and the human experts. The suggestions stemmed from the removal and renaming of duplicate text. The process began by first identifying what sort of refactoring suggestions/options SEED provided for the detected duplication. From there, the actual change made by the human experts was observed and compared against SEED's behaviour. If they matched, then it can be deduced that SEED was behaving correctly.

There was only one occurrence whereby the refactoring done by the human experts did not match with SEED's suggestions. The Cucumber feature had a pre-condition step repeated in every scenario. The feature did not have a background section. SEED detected this and suggested to move the pre-condition into the background section. However, the human experts opted to remove the pre-conditions entirely. Both are viable options in handling the duplication. It is possible that the human experts deemed the pre-condition unnecessary for the scenarios.

Refactoring	Count of Refactorings
Suggested by SEED and Human Expert	5
Suggested by Human Expert only	1
Suggested by SEED only	1

Table 5.4: Count of refactorings suggested/done.

5.5 Experiment 2

The Cucumber features used in this experiment were taken from the official *Gitlab* project stored in the following repository:-

https://github.com/gitlabhq/gitlabhq

Table 5.5 shows the total amount of Cucumber features that was evaluated against and analysed as part of the experiment.

Total No. of Cucumber features	Total No. of commits/changes
70	408

Table 5.5: Total amount of feature files and commits in the Gitlab project.

The project used in this experiment had significantly more cucumber features and commits than in Experiment 1.

Learning from Experiment 1, we know that SEED is able to capture exact duplications. Therefore, the success criteria remains the same. However, it is expected that there would be more near-duplicates (similarity matches) than exact duplicates detected. Refactoring proposals are still expected to match with the human experts.

Table 5.6 shows the total number of duplications found in this experiment.

Type	Count
Duplication	347
Errors	0

Table 5.6: Count of duplications & errors detected within the Cucumber features.

5.5.1 Duplication

There were more duplications found in this experiment than in Experiment 1. The average size of the Cucumber features were large i.e. had many lines. That may have contributed to the number of duplications discovered (more opportunities for finding

similarities in the feature). The duplications detected in this experiment had a significantly high number of near-duplicates compared to exact duplicates. Most of the Cucumber scenarios were long and focused too much on *how* things should be done instead of *what* should be done which is considered to be a bad practice as mentioned in Section 3.4. Figure 5.4 shows one of the Cucumber features in the project that consisted of this bad practice. As one can see, the similar steps make it harder to read and comprehend the scenario. Table 5.7 and Figure 5.5 show the coverage of duplications detected in Experiment 2.

```
@dashboard
Feature: Snippet Search
  Background:
    Given I sign in as a user
    And I have public "Personal snippet one" snippet
    And I have private "Personal snippet private" snippet
    And I have a public many lined snippet

  Scenario: I should see my public and private snippets
    When I search for "snippet" in snippet titles
    Then I should see "Personal snippet one" in results
    And I should see "Personal snippet private" in results

  Scenario: I should see three surrounding lines on either side of a matching snippet line
    When I search for "line seven" in snippet contents
    Then I should see "line four" in results
    And I should see "line seven" in results
    And I should see "line ten" in results
    And I should not see "line three" in results
    And I should not see "line eleven" in results
```

Figure 5.4: snippet_search.feature

Duplication	Count of Duplicates
Detected by SEED and Human Expert	21
Should have been detected by Human Expert	196
Should not have been detected by SEED	130

Table 5.7: Count of duplications detected.

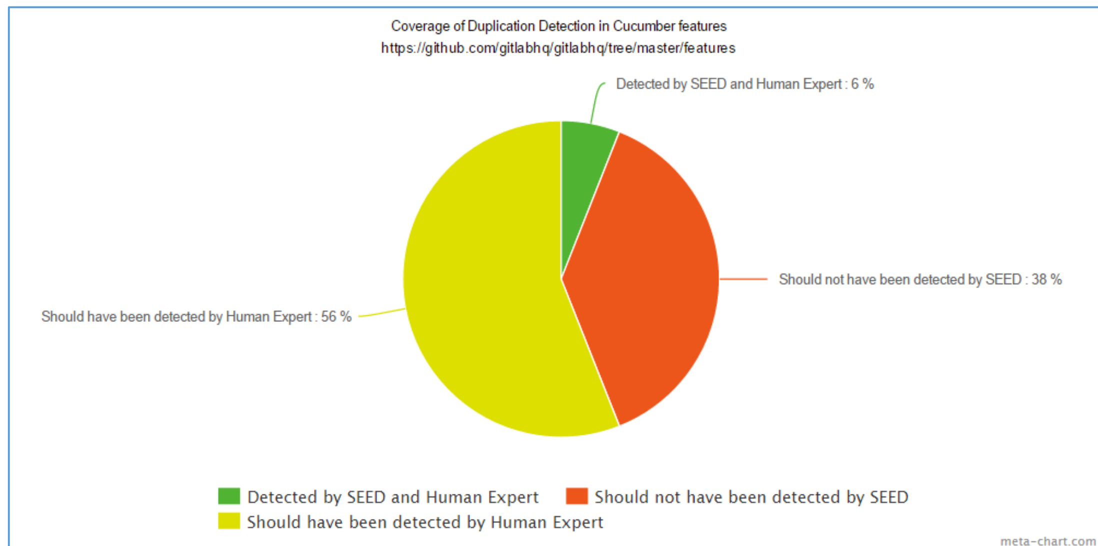

Figure 5.5: Coverage of Duplication Detection in Experiment 2.

The results were *positive* as none of the duplicates went undetected by SEED i.e. duplicates that were detected by the human experts were also detected by SEED. Out of 347 duplications, 21 of them (6%) were detected by both SEED and the human experts. Among the 21, there were 5 exact- and 16 near-duplicates.

56% of the duplications were considered as *true positives* but were not detected/confirmed by the human experts. The process for determining this was elaborated in Experiment 1. In this part of the coverage, there were 50- exact and 146 near-duplicates.

38% of the duplications were considered *false positives*. They were correct in terms of being similar to one another. However, there is a need for further verification (from the human experts) in order to confidently consider them as truly duplicates i.e. "do the human experts consider the findings as duplications?" The duplications consisted of only near-duplicates. There were a total of 130 near-duplicates in this part of the coverage.

5.5.2 Refactoring

As shown in Table 5.8, SEED managed to propose nearly all of the same refactoring options as the human experts' behaviour in the changelog. 20 out of the 21 duplicates found by both SEED and the human experts had the exact same refactoring behaviour. The refactoring proposed dealt with the removal of the duplicates.

There was only one occurrence whereby the refactoring proposed by SEED did not match with the human experts' actions. It had to do with the Cucumber feature having the same pre-condition step in all of its scenarios. The feature had a background section with three steps in the section. SEED proposed for a migration of the pre-condition step into the background section. However, the human experts removed the pre-condition instead.

Refactoring	Count of Refactorings
Suggested by SEED and Human Expert	20
Suggested by Human Expert only	1
Suggested by SEED only	1

Table 5.8: Count of refactorings suggested/done.

5.6 Experiment 3

The Cucumber features used in this experiment were taken from the official *RadiantCMS* project stored in the following repository:-

https://github.com/radiant/radiant

Table 5.9 shows the total amount of Cucumber features that was evaluated against and analysed as part of the experiment.

Total No. of Cucumber features	Total No. of commits/changes
9	60

Table 5.9: Total amount of feature files and commits in the RadiantCMS project.

The amount of data analysed in this experiment was less than in Experiment 1 and Experiment 2.

Learning from Experiment 1 and 2, the success criteria remains the same. Again, it is expected that there would be more near-duplicates (similarity matches) than exact duplicates detected. Although, the refactoring proposal for repeated pre-condition steps is expected to be different between SEED and the human experts.

Table 5.10 shows the total number of duplications found in this experiment.

Type	Count
Duplication	26

Errors	0

Table 5.10: Count of duplications & errors detected within the Cucumber features.

5.6.1 Duplication

Experiment 3 was the only experiment to have the majority of the duplications be detected by both SEED and the human experts. Table 5.11 and Figure 5.6 show the coverage of duplications detected in Experiment 3.

Duplication	Count of Duplicates
Detected by SEED and Human Expert	10
Should have been detected by Human Expert	8
Should not have been detected by SEED	8

Table 5.11: Count of duplications detected.

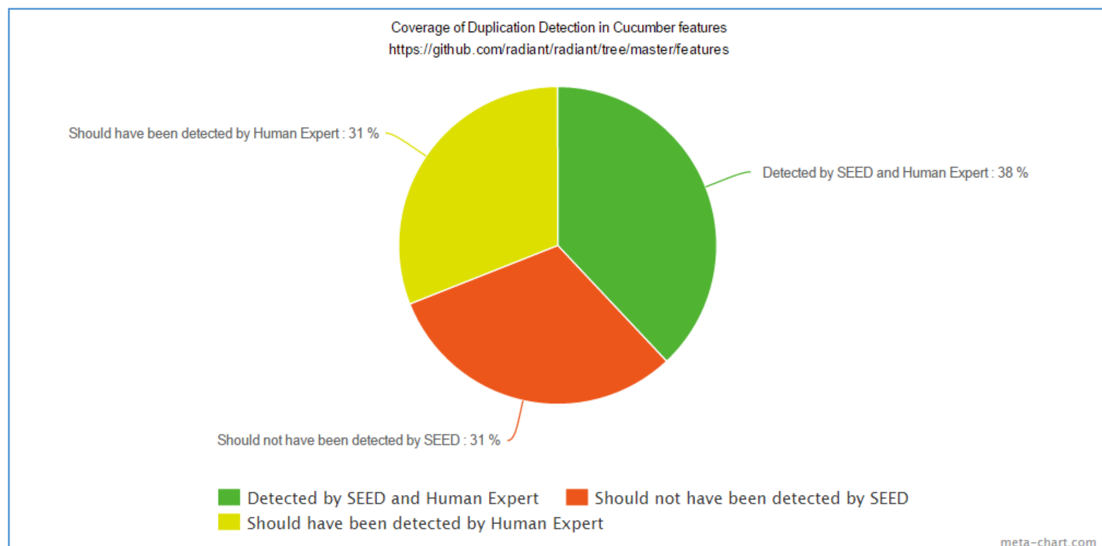

Figure 5.6: Coverage of Duplication Detection in Experiment 3.

There were a total of 10 out of 26 of the duplicates (38%) detected by both SEED and the human experts. Again, no duplications went undetected by SEED. Of those 10, 2 were exact duplicates and 8 were near-duplicates.

The total number of duplicates that fell under the category of *unconfirmed true positives* and *false positives* were equal. The true positives consisted of 6 exact- and 2 near-duplicates. The false positives, however, consisted of only 8 near-duplicates.

5.6.2 Refactoring

As shown in Table 5.12, there were no discrepancies for the refactoring proposals between Experiment 3 and the other experiments. There was only one occurrence whereby the refactoring proposed by SEED did not match with the human experts' actions. It had to do with repeated pre-conditions in the feature. The feature did not have a background section. The repeated pre-conditions were *renamed* by the human experts instead of being moved into the background section as suggested by SEED.

Refactoring	Count of Refactorings
Suggested by SEED and Human Expert	9
Suggested by Human Expert only	1
Suggested by SEED only	1

Table 5.12: Count of refactorings suggested/done.

5.7 Results & Discussion

The overall results of the experiments undertaken in the evaluation process proved to be *positive*. By experimenting with different sized external project repositories, the conclusions drawn became more trustworthy. Additionally, experimenting with more than one repository helped to gauge the consistency of SEED's behaviour and results.

SEED managed to detect every single duplication that were found by the human experts i.e. no duplicates that were detected by the human experts but not by SEED. Among the duplications detected, SEED also proposed the same refactoring options as the actual actions of the human experts albeit with minor differences. However, alongside the *true positives*, there were *false positives* detected in each of the experiments.

The occurrence of near-duplicates was high in all three experiments compared to the number of exact duplicates found. The reason for this was that the similarity matching algorithm used (Section 3.7.2) not only looked for semantic equivalence but also syntactically equivalent pieces of text. Therefore, the near-duplicates detected consisted of both equivalence types. The assessment of near-duplicates proved to be challenging as the similarities could fall under either *(unconfirmed) true* or *false* positives. Close examination of the similarities had to be carried out before a decision was made and the data was tabulated. Even though some of the similarity matches

were categorized as false positives, it did not mean that they were incorrect (an example was shown in Section 5.4.1). It meant that the false positives were not considered as duplications within the BDD/Cucumber context but could still possibly assist the user/human expert in improving their Cucumber features. Near-duplicates that had a high syntax equivalence match (as shown in Section 5.5.1) were placed under the (unconfirmed) true positive category since they worked against the readability of the Cucumber feature.

Exact duplicates were managed well by SEED. Those that were not caught by the human experts were categorized as *(unconfirmed) true positives*. However, there might be an undisclosed reason as to why some exact duplicates went undetected such as choosing to remove the duplicates at a later date or that the human experts did *not* consider the duplications as problematic to the Cucumber features. Therefore, further evaluation from on-site human experts may be required in order to verify the matches found.

Refactoring proposals made by SEED were consistent throughout the experiments. The refactoring mainly dealt with removal and renaming of the duplicates. They proved to be viable options for the human experts to take when it came to fixing duplications in the tests. However, the experiments shared a similarity of proposing different refactoring actions for repeated pre-condition Cucumber steps than SEED. This was the only refactoring suggestion that differed from the human experts. The reason for this could possibly be that the human experts did not want to create a Cucumber *background* section for only *one* pre-condition step.

All three experiments showed that there was a correlation between SEED and the human experts in terms of what was considered as duplication in a Cucumber feature.

Figure 5.7 shows the overall results gathered for Experiment 1. It can be seen that the results of SEED encapsulates the results of the human experts; results being the duplications discovered. The number of false positives were significantly high in this experiment compared to the rest of the detected duplications.

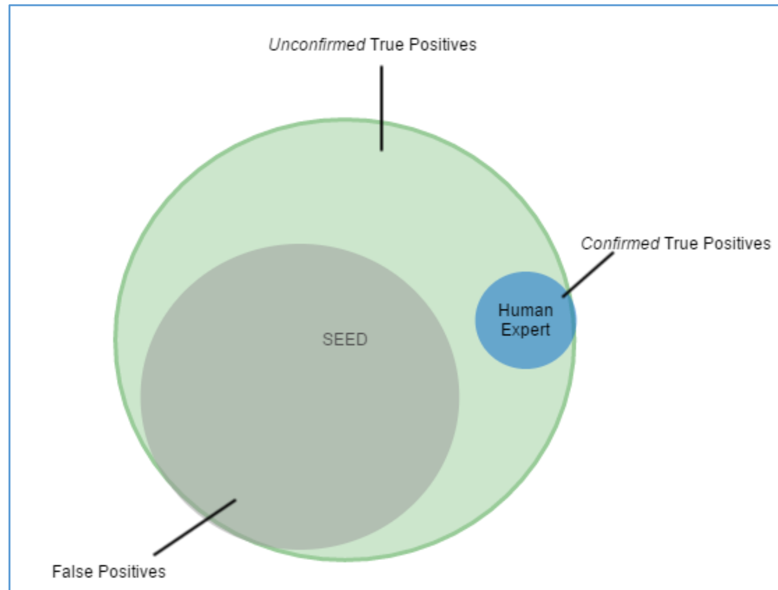

Figure 5.7: Results for Experiment 1.

Figure 5.8 shows the overall results gathered for Experiment 2. The number of (unconfirmed) true positives that went undetected by the human experts is higher than the number of false positives in this experiment.

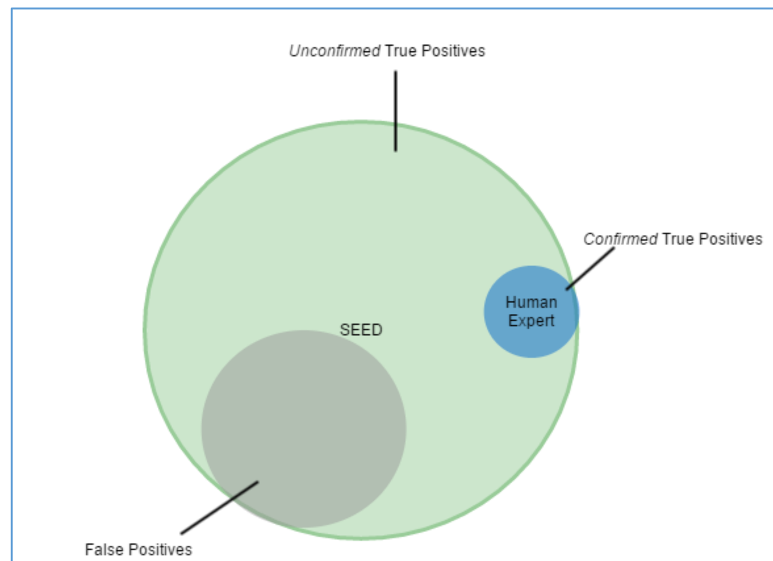

Figure 5.8: Results for Experiment 2.

Figure 5.9 shows the overall results gathered for Experiment 3. The number of (confirmed) true positives that were detected by both SEED and the human experts is higher than both the (unconfirmed) true positives that went undetected by the human experts and the false positives that were detected by SEED. Additionally, the number

of unconfirmed true positives is equal to the number of false positives in this experiment.

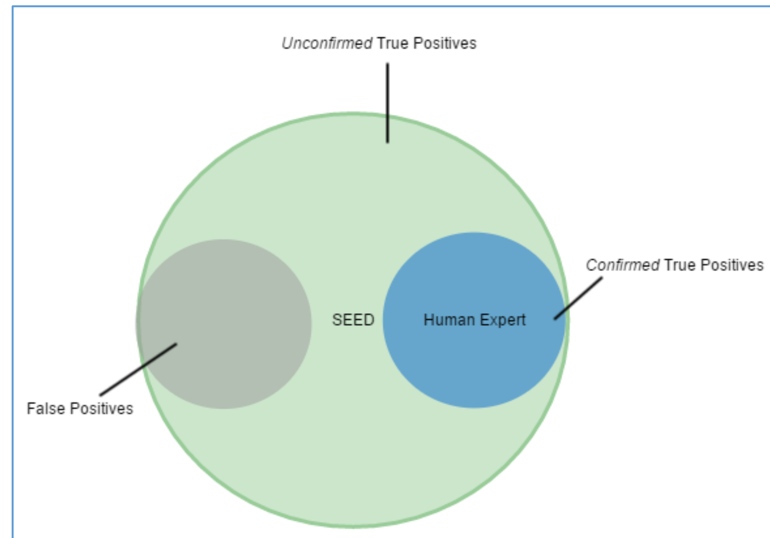

Figure 5.9: Results for Experiment 3.

Overall, it can be deduced that SEED is able to capture the same duplicates as human experts and refactor them in the same manner as human experts would. However, further evaluation (by the human experts) might be needed for verifying the near-duplicates detected by SEED and the ones that went undetected by the human experts.

Additionally, SEED has been deployed onto the Eclipse Marketplace. A snapshot of this can be seen on Figure 5. 10. SEED can be found on the following link: https://marketplace.eclipse.org/content/seed. The marketplace serves as a repository for users of Eclipse to download Eclipse plug-ins. From the snapshot shown, SEED has been downloaded/installed a total of five times since it was put on the marketplace.

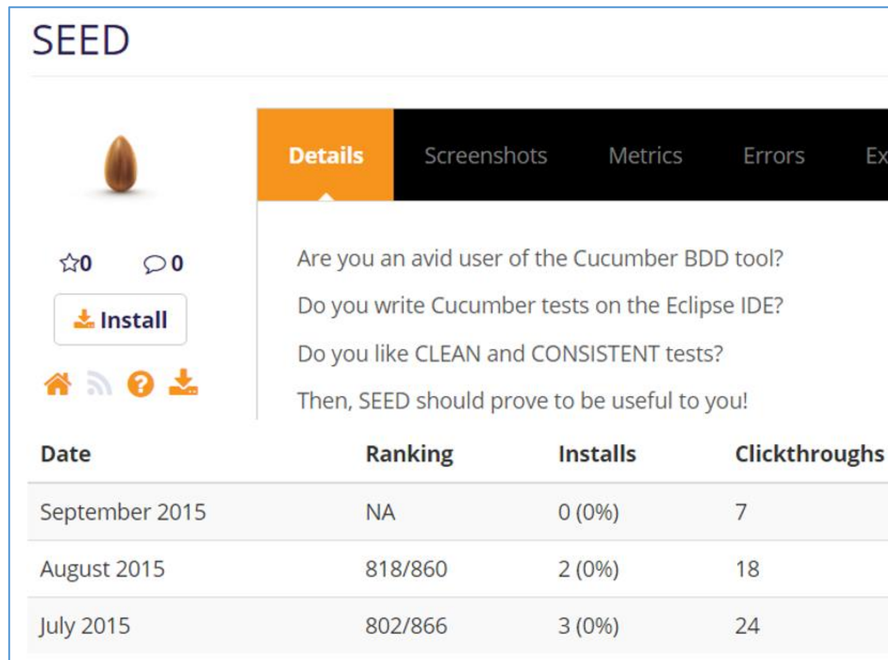

Figure 5.10: SEED deployed on Eclipse Marketplace.

5.8 Conclusion

The chapter has explained in detail of the evaluation approach undertaken in the project. The approach spanned over a total of three experiments. The raw data were then collected and analysed accordingly. The results then lead to answering the research question proposed in the project.

Chapter 6 : Conclusion

The project began by establishing its aim of determining whether a software tool could simulate the actions of a human expert in terms of detecting and refactoring duplications in BDD specifications. As a result, the tool would help reduce duplications in BDD specifications.

A thorough background research was conducted in order to gain a strong understanding of the duplication problem. Through the research, the author learned about the following concepts:-

- The limitations of gathering requirements in a software development project.
- The purpose of Specification-by-Example.
- The definitions of Acceptance Test-Driven Development and Behaviour-Driven Development.
- The functionality of the FitNesse tool.
- The functionality of the Cucumber tool.

The author also learned that having duplications in BDD specifications make the specifications difficult to maintain and read. However, removing these duplications is not a simple task. Therefore, it was necessary to experience and experiment with plausible sources of duplications in BDD specifications in order to define a set of rules that are able to tell (whoever is reading these rules) that duplications exist in the specifications. This process was done by the author's extensive use of the Cucumber tool. Then, the learning process extended towards algorithms that could be used to detect exact- and near-duplications in the specifications.

The implementation portion of the project focused on developing an Eclipse plugin to detect and refactor duplicates residing in Cucumber features. The algorithms and rules were implemented into the plugin as well. The plugin was created to help answer the aim/research question of the project.

The evaluation of the plugin and duplication rules proved to be successful according to the results gathered from the process. As part of the evaluation process, the plugin was compared against human experts' actions in terms of GitHub commits done to Cucumber features. After experimenting with three different GitHub repositories, the evidence showed that the plugin was able to detect duplications that were also detected

by the human experts and refactor them in the same way as the human experts did. However, apart from the positive results, there were unconfirmed findings as well. These unconfirmed findings consisted of duplications that were found by SEED but not by the human experts. These duplications could be further broken down into duplications that *should* have been detected by the human experts i.e. helpful and duplications that possibly *should not* have been detected by SEED i.e. not helpful. These duplications were determined by the author's analysis and observations of the findings. Therefore, further evaluation by the human experts may (still) be required to verify the extra duplicates reported by SEED.

6.1 Future Work

Although SEED succeeded in answering the project's research question, there is still room for improvement. Currently, SEED only accommodates for Cucumber features. Since Cucumber is not the only BDD tool out there, it would be beneficial to port SEED's functionality to span across other BDD-based tools. Therefore, SEED can help reduce duplications in specifications that were not only created with Cucumber. The rules created in this project for detecting duplications could be re-used and re-applied for the other tools as well.

Detecting near-duplicates in specifications could also be improved and made to be more reliable. As stated previously, alternative solutions could be looked at such as comparing *parse trees*. The goal is to reduce the number of misleading results (*false positives*) gathered when searching for near-duplications.

Work could also be done towards detecting duplications across multiple Cucumber features aside from only their titles. This way, SEED is able to detect duplicate scenarios that exist in two or more features. In addition, new rules will have to be introduced and defined for this newfound functionality.

SEED currently renames duplications by appending the duplicates' line numbers to their text as part of the refactoring operation. However, as stated previously, this does not improve the quality of the specifications i.e. it does not make it any more readable than it previously was. A better solution would be to create a dialogue for the users (of SEED) to type a new name of their choosing.

Bibliography

[1] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques., Springer, 2010.

[2] M. Fowler, "SpecificationByExample," 18 March 2004. [Online]. Available: http://martinfowler.com/bliki/SpecificationByExample.html . [Accessed February 2015].

[3] G. Adzic, Specification by example: How successful teams deliver the right software, Manning Publications, 2011.

[4] J. Gregory, "ATDD vs. BDD vs. Specification by Example vs …," 31 August 2010. [Online]. Available: http://janetgregory.blogspot.co.uk/2010/08/atdd-vs-bdd-vs-specification-by-example.html. [Accessed February 2015].

[5] D. North, "Introducing BDD," 20 September 2006. [Online]. Available: http://dannorth.net/introducing-bdd/. [Accessed February 2015].

[6] E. Hendrickson, "Acceptance Test Driven Development (ATDD): an Overview," 8 December 2008. [Online]. Available: http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/. [Accessed February 2015].

[7] T. u. Rehman, M. N. A. Khan and N. Riaz, "Analysis of Requirement Engineering Processes, Tools/Techniques and Methodologies," *Modern Education and Computer Science Press,* vol. 5, no. 3, p. 40 – 48, 2013.

[8] D. Pandey, U. Suman and A. K. Ramani, "An Effective Requirement Engineering Process Model for Software Development and Requirements Management," *IEEE International Conference on Advances in Recent Technologies in Communication and Computing,* pp. 287 - 291, 2010.

[9] H. v. Vliet, "Requirements Documentation and Management," in *Software Engineering: Principles and Practice [2nd edition]*, Wiley, 1999, p. 241 – 242.

[10] L. Cao and B. Ramesh, "Agile Requirements Engineering Practices: An Empirical Study," *IEEE Computer Society,* vol. 25, no. 1, p. 60 – 67, 2008.

[11] B. Ramesh, L. Cao and R. Baskerville, "Agile Requirements engineering practices and challenges: an empirical study," *Information Systems Journal,* vol. 25, no. 1, p. 449 – 480, 2010.

[12] M. Gartner, "Specification by Example – The Big Win," 5 February 2011. [Online]. Available: http://www.shino.de/2011/02/05/specification-by-example-the-big-win/. [Accessed April 2015].

[13] L. Koskela, "Practical TDD and Acceptance TDD for Java Developers," in *Acceptance TDD explained*, Manning Publications, 2007, p. 323 – 363.

[14] M. Fowler, "UnitTest," 5 May 2014. [Online]. Available: http://martinfowler.com/bliki/UnitTest.html. [Accessed August 2015].

[15] A. Vlachou, "Test-Driven Development for Aspect-Oriented Programming," University of Manchester, 2014.

[16] J. Gregory and L. Crispin, "Getting Examples," in *More Agile Testing: Learning Journeys for the Whole Team*, Addison-Wesley, 2014, p. 145 – 162.

[17] C. Solís and X. Wang, "A Study of the Characteristics of Behaviour Driven Development," *In Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011),* p. 383 – 387, 2011.

[18] D. North, "What's in a story?," 11 February 2007. [Online]. Available: http://dannorth.net/whats-in-a-story/. [Accessed April 2015].

[19] G. Esquivel, "Differences Between TDD, ATDD and BDD," 28 July 2014. [Online]. Available: http://gaboesquivel.com/blog/2014/differences-between-tdd-atdd-and-bdd/. [Accessed August 2015].

[20] A. Hellesøy, "Behaviour Driven Development (BDD) in the Finance Sector," 22 June 2015. [Online]. Available: http://www.excelian.com/blog/behaviour-driven-development-bdd-in-the-finance-sector/. [Accessed August 2015].

[21] J. Stenberg, "Behaviour Driven Development Tool Cucumber Questioned," 1 October 2013. [Online]. Available: http://www.infoq.com/news/2013/10/bdd-tool-cucumber-questioned. [Accessed August 2015].

[22] R. Lawrence, "Cucumber - Behavior Driven Development for Ruby," *Methods & Tools,* vol. 19, no. 4, pp. 51 - 56, 2011.

[23] C. Parsons, "Make Cucumber features more readable with this one weird trick," 12 February 2014. [Online]. Available: http://chrismdp.com/2014/02/make-cucumber-features-more-readable-with-this-one-weird-trick/. [Accessed August 2015].

[24] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[25] A. Ghahrai, "BDD Guidelines and Best Practices," 18 December 2014. [Online]. Available: http://www.testingexcellence.com/bdd-guidelines-best-practices/. [Accessed August 2015].

[26] D. Kennedy, "Smelly Cucumbers," 13 January 2012. [Online]. Available: http://www.sitepoint.com/smelly-cucumbers/. [Accessed August 2015].

[27] H. Chen, "String Metrics and Word Similarity [Master Thesis]," University of Eastern Finland [School of Computing], 2012.

[28] D. Leffingwell, "Domain Modeling Abstract," 20 February 2014. [Online]. Available: http://www.scaledagileframework.com/domain-modeling/. [Accessed August 2015].

[29] S. Yeates, "What Is Version Control? Why Is It Important For Due Diligence?," 1 January 2005. [Online]. Available: http://oss-watch.ac.uk/resources/versioncontrol. [Accessed August 2015].

[30] T. McFarlin, "The Beginner's Guide to Unit Testing: What Is Unit Testing?," 19 June 2012. [Online]. Available: http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728. [Accessed August 2015].

Appendix A: Gherkin Grammar (Gherkin.xtext)

```
grammar uom.ac.uk.msc.cucumber.Gherkin with
org.eclipse.xtext.common.Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate gherkin "http://www.ac.uom/uk/msc/cucumber/Gherkin"

Feature:
      tags+=Tag*
      'Feature:'
      title=Title EOL+
      narrative=Narrative?
      background=Background?
      scenarios+=(Scenario | ScenarioOutline)+;

Background:
      backgroundKeyword=BackgroundKeyword
      title=Title? EOL+
      narrative=Narrative?
      steps+=Step+;

Scenario:
      tags+=Tag*
      'Scenario:'
      title=Title? EOL+
      narrative=Narrative?
      steps+=Step+;

ScenarioOutline:
      tags+=Tag*
      'Scenario Outline:'
      title=Title? EOL+
      narrative=Narrative?
      steps+=Step+
      examples=Examples;

Step:
      stepKeyword=StepKeyword
      description=StepDescription EOL*
      tables+=Table*
      code=DocString?
      tables+=Table*;

Examples:
      'Examples:'
      title=Title? EOL+
      narrative=Narrative?
      table=Table;

Table:
      rows+=TABLE_ROW+ EOL*;

DocString:
      content=DOC_STRING EOL*;

Title:
```

```
      (WORD | NUMBER | STRING | PLACEHOLDER) (WORD | NUMBER |
STRING | PLACEHOLDER | STEP_KEYWORD | TAGNAME)*;

Narrative:
      ((WORD | NUMBER | STRING | PLACEHOLDER) (WORD | NUMBER |
STRING | PLACEHOLDER | STEP_KEYWORD | TAGNAME)* EOL+)+;

StepDescription:
      (WORD | NUMBER | STRING | PLACEHOLDER | STEP_KEYWORD |
TAGNAME)+;

StepKeyword: STEP_KEYWORD;

BackgroundKeyword: 'Background:';

Tag: id=TAGNAME EOL?;

terminal NUMBER: '-'? ('0'..'9')+ ('.' ('0'..'9')+)?;

terminal STEP_KEYWORD: ('Given' | 'When' | 'Then' | 'And' |
'But') (' ' | '\t')+;

terminal PLACEHOLDER: '<' !('>' | ' ' | '\t' | '\n' | '\r')+ '>';

terminal TABLE_ROW: '|' (!('|' | '\n' | '\r')* '|')+ (' ' |
'\t')* NL;

terminal DOC_STRING: ('"""' -> '"""' | "'''" -> "'''") NL;

terminal STRING:
      '"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" |
'\\') | !('\\' | '"' | '\r' | '\n'))* '"' |
      "'" ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | "'" |
'\\') | !('\\' | "'" | '\r' | '\n'))* "'";

terminal SL_COMMENT: '#' !('\n' | '\r')* NL;

terminal TAGNAME: '@' !(' ' | '\t' | '\n' | '\r')+ ;

terminal WORD: !('@' | '|' | ' ' | '\t' | '\n' | '\r') !(' ' |
'\t' | '\n' | '\r')*;

terminal WS: (' ' | '\t');

terminal EOL: NL;

terminal fragment NL: ('\r'? '\n'?);
```