DETECTING AND CORRECTING DUPLICATION IN BEHAVIOUR DRIVEN DEVELOPMENT SPECIFICATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

By
Leonard Peter Binamungu

School of Engineering, Department of Computer Science

Contents

Word Count: 56184

List of Tables

List of Figures

Abstract

DETECTING AND CORRECTING DUPLICATION IN BEHAVIOUR DRIVEN DEVELOPMENT SPECIFICATIONS
Leonard Peter Binamungu
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2020

The Behaviour Driven Development (BDD) technique enables teams to specify software requirements as example interactions with the system. Due to the use of natural language, these examples (usually referred to as *scenarios*) can be understood by most project stakeholders, even end users. The set of examples also acts as tests that can be executed to check the behaviour of the System Under Test (SUT). Despite BDD's benefits, large suites of examples can be hard to comprehend, extend and maintain. Duplication can creep in, leading to bloated specifications, which sometimes cause teams to drop the BDD technique. Current tools for detecting and removing duplication in code are not effective for BDD examples. Moreover, human concerns of readability and clarity can rise. Previous attempts to detect and remove duplication in BDD specifications have focused on textually similar duplicates, not on textually different scenarios that specify the same behaviour of the SUT.

To fill this gap, this thesis does the following. **First**, we surveyed 75 BDD practitioners from 26 countries to understand the extent of BDD use, its benefits and challenges, and specifically the challenges of maintaining BDD specifications in practice. We found that BDD is in active use amongst respondents; and the use of domain specific terms, improving communication among stakeholders, the executable nature of BDD specifications, and facilitating comprehension of code intentions emerged as some of the main benefits of BDD. The results also showed that BDD specifications suffer the same maintenance challenges found in automated test suites more generally. We map the survey results to the literature, and propose 10 research opportunities in this area. **Second**, we propose and evaluate a framework for detecting duplicate scenarios

based on the comparison of characteristics extracted from scenario execution traces. We focus on the patterns of production code exercised by each scenario, and consider two scenarios to be duplicates if they execute the same functionality in the SUT. In an empirical evaluation of our framework on 3 open source systems, the comparison of execution paths of scenarios recorded more *recall* and *precision* than the comparison of full execution traces, public API calls, or a combination of public API calls and internal calls. Also, the focus on essential characteristics in the execution traces of scenarios improved the *recall* and *precision* of the duplicate detection tool. **Third**, we propose four principles describing BDD suite quality that can be used to assess which of a pair of duplicate scenarios can be most effectively removed. All of the four principles were accepted by at least 75% of the practitioners we surveyed. An empirical evaluation of the four principles on 3 open source systems shows that each principle gave acceptable remove suggestions, and thus the principles can be used to guide human engineers in removing duplicate scenarios from BDD specifications.

Declaration

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

Copyright

Acknowledgements

First, I would like to thank God for the health and protection throughout my PhD studies.

Second, I am grateful to my supervisors, Dr Suzanne M. Embury and Dr Nikolaos Konstantinou, for their invaluable guidance throughout my PhD studies. Thank you all for the thought-provoking thoughts that gave light to my research. In particular, I would like to thank Suzanne for believing in me and giving me an opportunity to pursue a PhD under your supervision at the University of Manchester. It has been a great opportunity to work under your guidance, and I will forever be grateful. Nikos, my appreciation to your unwavering support during the PhD is beyond what words can explain!

Third, I would like to thank the Department of Computer Science, University of Manchester, for funding a large part of my PhD studies, by paying school fees and funds for other research activities during the PhD. In connection to this, I would like to acknowledge the support of my employer, the University of Dar es Salaam, on this work. Specifically, I thank the previous Vice Chancellor, Prof Rwekaza S. Mukandala, and his entire university management team, as well as the current university management, for the financial and moral support they continuously provided during my PhD studies.

Fourth, I would like to thank my family (my wife Joyce, my son Lestyn, and my daughter Alinda) for their invaluable support throughout my PhD studies. This thesis is my reward to them for whatever they endured whenever I went missing.

Fifth, I would like to thank my uncle, Prof Jonathan Kabigumila, for all the support and inspiration throughout my academic life.

Sixth, my sincere appreciations go to the respondents and participants to the surveys and experiments reported in this thesis. Special thanks to members of various BDD practitioner communities for providing insights that informed this study. Anonymous reviewers who helped improve part or whole of this thesis are also immensely thanked.

Last but not least, I would like to thank all my relatives, friends, and fellow PhD students at the University of Manchester, as well as members of staff at the University of Dar es Salaam, for all the support they provided during my studies. It is not possible to mention all, but I am grateful to everyone who made me smile during my PhD journey.

Dedication

To my wife Joyce, my son Lestyn, and my daughter Alinda.

Chapter 1

Introduction

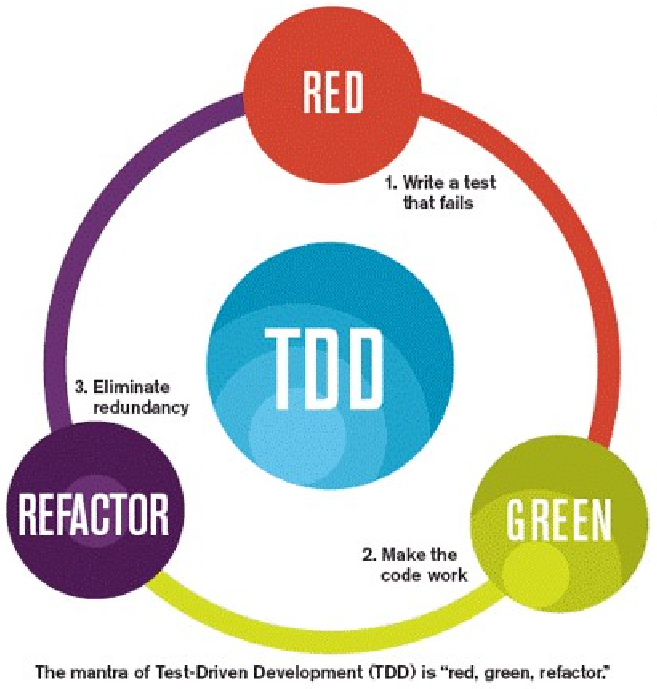
Owners and users of software systems expect the software to both satisfy business requirements and behave reliably when in operation. However, sometimes, software systems exhibit behaviours that deviate from business requirements, and they (software systems) experience failure during operation. To address this problem, software testing seeks to ensure that the software behaves as expected and to expose faults in the functioning of a software system [5, 6]. Testing is used extensively in many projects and can be responsible for up to 50% of the development costs [7, 8]. Intuitively, therefore, given the high costs involved in testing, it is important that software product owners appreciate the value brought by the investment in testing. One way to make software product owners appreciate the value of testing is to ensure that the tests created during the development process continue to be useful for as long as possible in the lifetime of a software system. However, during the lifetime of a software product, tests can become hard to maintain [9, 10], and thus becoming unfit for purpose.

Testing can be conducted manually, requiring human intervention every time the software has to be tested, or automatically in which the behaviour of the software system can be verified without human intervention [11, 12, 13]. *Inter alia*, automating the software testing process was introduced in a bid to lower the costs of testing [14]. Thus, test automation is particularly important in reducing the efforts required by teams to deliver software products within limited schedules and budgets [15], and in ensuring that code modification (for defect fixing or introduction of new feature) does not introduce defects into the software, and that the behaviour which code modification aims to introduce is exhibited by the software system [16]. Apart from reducing the cost and time required for testing, test automation minimises the threat of human errors associated with manual testing, fosters repeatability of tests, and facilitates efficient use of team members for other activities [17]. In Agile Software Development practices [18, 19], test automation is also used as a mechanism for getting quick feedback on the functioning of code submitted by different team members [20]. Despite the numerous benefits of automated testing, unfortunately, during the lifetime of the software, suites of automated tests can become hard to maintain due to design issues, among other reasons [9, 10, 21]. This can render the test suites ineffective, failing to serve the purpose they were created for.

In a software development project, the tests to be automated can be identified with or without customers. On the one hand, in the absence of customers, tests that are written during the development of a software product have a risk of ensuring that the produced code fulfils the intentions of developers, at the expense of precise business requirements. Because writing of tests requires understanding of specific development technologies, this kind of testing is often referred to as *technology-facing testing* and it produces tests that, in most cases, only members of the development team can understand. Typical examples of technology-facing testing can be found in teams that practice Test-Driven Development (TDD) [22] whereby, in each iteration, the following steps are followed by developers: writing a failing unit test; making the failing unit test pass by writing the required production code; and refactoring of production code, to improve its design. Figure 1.1 shows the typical TDD cycle.

On the other hand, when customers are involved in software testing, the main focus is usually on ensuring that the software satisfies business requirements. This supports the notion of *business-facing testing*, which encourages writing of software tests that customers can understand [23].

Acceptance Test Driven Development (ATDD) is an agile technique that facilitates business-facing testing by enabling a customer to collaborate with members of the development team to write acceptance criteria, which are then converted to functional test cases by testers and developers. Using ATDD, before a feature is developed, project team members create shared understanding by writing concrete examples with stakeholders, which are then converted into automated acceptance tests. This practice reduces information loss between a customer and members of the development team, and can ensure that the right software product is built [24]. During ATDD, implementing one acceptance test can involve the implementation of several unit tests, as shown


RED
1. Write a test that fails
TDD
3. Eliminate redundancy
REFACTOR
GREEN
2. Make the code work
The mantra of Test-Driven Development (TDD) is "red, green, refactor."
Figure 1.1: Typical TDD cycle (Source: [2])

in Figure 1.2; an acceptance test focuses on a high level functionality which can be associated with several low level implementation units (e.g., classes/methods), each of which is covered by a particular unit test. An acceptance criteria is converted into an executable acceptance test, which should fail because, at that moment, the required functionality is not implemented yet. After the failing acceptance test is in place, the required functionality is implemented by following the TDD cycle. When all the required functionality for a particular acceptance criteria has been implemented, the acceptance test will pass, signifying complete implementation of the required functionality. Then, the team selects the next acceptance criteria and follows the same process.

Related to TDD and ATDD is BDD (introduced in Section 1.1) which is another agile technique that enables teams to perform business-facing testing by producing software tests that all project stakeholders can understand. In BDD, tests are specified as requirements [25, 26] using a natural language. The tests expressed in a natural language are linked to the SUT using "glue code", and can be executed to verify if the SUT is behaving as expected. This approach has many advantages. Among other things, it improves communication and collaboration between project stakeholders, and it provides the living documentation that can be used to verify if the System Under Test behaves

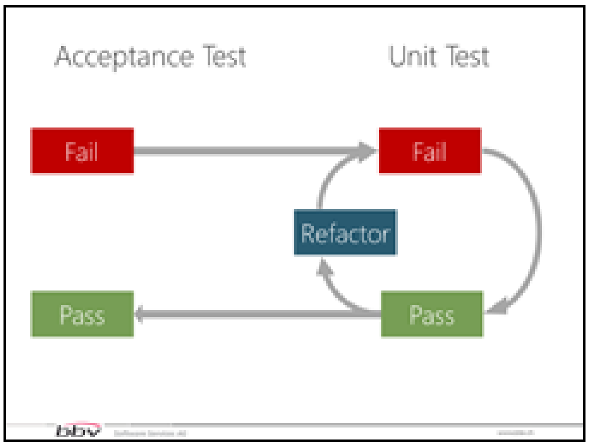


Figure 1.2: Typical ATDD cycle (Source: [3])

as per specification [27, 28].

The next section provides more information about the BDD technique, by introducing the key concepts required to understand the problem of focus in this thesis.

1.1 Behaviour Driven Development

This section introduces BDD, and the typical setup of a BDD project.

In Behaviour Driven Development [29, 30, 31], the behaviour of the required software is given as a collection of example interactions with the system, expressed using natural language sentences organised around a "Given-When-Then" structure (e.g., Listing 1.1), in a language called *Gherkin*. This gives a specification that is expressed in non-technical, domain-specific terms, that should be comprehensible by end-users. Importantly, the specification is also executable, thanks to "step definition functions" (e.g., Listing 1.2) that link the natural language sentences to the code that is being built. Each step in a scenario can be mapped to a step definition function for execution against the SUT. Thus, the set of examples acts both as a high-level specification of the requirements for the software and as a suite of acceptance tests that can verify whether the current implementation meets the specification or not.

More specifically, in BDD, the behaviour of a software system is described by a suite of *features*. Each feature is specified by a collection of *scenarios*. Each scenario describes an end-to-end interaction with the system, in terms of concrete data examples typical of what might be used in real system use. Scenarios are written in a form of structured

Listing 1.1: Example ATM Transactions Feature

```
Feature: ATM transactions
As an account holder, whenever
I perform ATM withdrawals, I want my
balance to be updated, and I should get apt messages

Scenario: Successful withdrawal from account
Given my account is in credit by $100
When I request withdrawal of $20
Then $20 should be dispensed
And my balance should be $80

Scenario: Unsuccessful withdrawal from account
Given my account has an initial balance of $80
When I request withdrawal of $100
Then nothing should be dispensed
And I should be told that I have insufficient funds
```

English, following a "Given-When-Then" pattern. Listing 1.1 shows a simple example feature, adapted from an open source specification for an ATM machine[1].

The feature with two contrasting scenarios show the BDD technique at work. Both the *Feature* and *Scenario* keywords are followed by a phrase describing the high level purpose of the feature or scenario that they represent. A scenario consists of a sequence of *steps*, each beginning with the reserved keywords *Given*, *When*, *Then*, *And* and *But*. The *Given* step (or steps) give the starting assumptions for the example. In the case of the first scenario in Listing 1.1, the *Given* step states that the user of the ATM is assumed to have an account with the bank that is in credit by a certain amount. The *When* steps give the series of actions that are presumed to be taken by the user (or an external trigger) in the example interaction. In the first scenario in Listing 1.1, this step states that the user requests the withdrawal of an amount of money from their account. The *Then* part describes a condition that should hold at that stage in the scenario. For the first scenario in Listing 1.1, the requested amount is dispensed and the balance is updated.

The second scenario in Listing 1.1 describes a similar interaction between a customer and the ATM, but with a different starting assumption. In this case, the customer does

[1] `github.com/krzysztof-jelski/atm-example`

not have enough in their account for the withdrawal to go ahead, and so a different outcome is to be expected. The relationship between various steps of a BDD scenario is analogous to a Finite State Machine [32] in which, given a particular state and input, a machine transitions to a new state, producing an appropriate output.

As well as documenting the business requirements, these examples can be executed to see which scenarios or steps have been implemented and which scenarios or steps have not been implemented. A BDD engine is used to execute the scenarios, by mapping steps in the scenarios to their corresponding step definition functions. The engine executes each step in turn, reporting both the steps that pass and the steps that fail. For example, in the Cucumber-JVM BDD execution engine, to indicate a failed step, the Then step might be coloured red, if the software returns an amount for the balance that is different from the expected one; however, if the returned amount is the same as the expected one, then the Then step might be coloured green.

To execute the scenarios against the current state of the system implementation, we must provide some *glue code*, telling the BDD execution engine what production code methods to execute for each scenario step. Listing 1.2 presents the glue code expected by Cucumber-JVM, a Java implementation of the the Cucumber execution engine, for the first ATM scenario. It can be seen that the glue code is a collection of methods, each of which is annotated with either `@Given`, `@When` or `@Then`, to indicate that the method is a *step definition* (i.e., it defines how a scenario step is executed). The annotation takes a string parameter, consisting of a regular expression. When the feature files are executed, the Cucumber engine matches the text of each step against the regular expressions paired with the step definition methods. The first step definition that matches is the one that is executed. Values from the step are extracted from the text by groups in the regular expression, and passed to the body of the method as parameters.

The body of the method then has the task of carrying out the intent of the step that it matches. *Given* steps describe the conditions that are assumed to hold when the scenario is executed, so the body of these methods must cause those conditions to become true for the software under test. This is achieved by invoking production code methods that affect the state of the software: creating a bank account, setting the balance of the account. For example, on line 5 in Listing 1.2, a bank account is created through an object of the production class called *BankAccount* and the account balance is set, to fulfil the condition of the starting state as specified by the *Given* step of the first scenario in Listing 1.1. The *When* steps are the ones that invoke the behaviour under

Listing 1.2: Glue code for scenario 1 in Listing 1.1

```
1  public class ATMWithdrawalSteps {
2    BankAccount account;
3  @Given("^my account is in credit by \$(\d+)$")
4  public void my_acc_is_in_credit_by_$(double amt) {
5    account = new BankAccount(amt);
6  }
7  @When("^I request withdrawal of \$(\d+)$")
8  public void i_request_withdrawal_of(double amt) {
9    account.withdraw(amt);
10 }
11 @Then("^\$(\d+) should be dispensed$")
12 public void should_be_dispensed(double ant) {
13   // cash dispensing code goes here
14 }
15 @Then("^my balance should be \$(\d+)$")
16 public void my_balance_should_be(double amt) {
17   assertEquals(amt,account.getAccountBalance());
18 }}
```

test: withdrawing money from an account, crediting money to an account, etc. Refer to line 9 in Listing 1.2 whereby a call is made to the *withdraw (double)* method of the *BankAccount* production class. *Then* steps check that the state of the system after the behaviour has been invoked is what we expect: checking the balance of the account, checking the log of the ATM, etc. See line 17 in Listing 1.2 whereby the expected balance in the bank account is compared with the actual balance retrieved using the production method called *getAccountBalance()*. Cucumber-JVM executes each step as a JUnit test, and so we use JUnit assertion methods, such as `assertEquals`, to make these checks.

We next discuss the setup of a typical BDD project.

Software developed using the BDD approach will typically consist of three types of artefact, as shown in Figure 1.3:

- A collection of BDD scenarios, in semi-structured natural language and readable by end users. Each scenario belong to a particular feature, and one feature file (saved using a *.feature* extension) can have one or more related scenarios. For example, feature *abc* in Figure 1.3 has two scenarios, *foo* and *bar*. Other features will have varying numbers of scenarios, depending on the aspects of the system represented by the individual features. A typical BDD suite consists of a collection of several feature files, as shown using the multidocument symbol on the left side of Figure 1.3.

- Glue code, which is a collection of step definition functions. The organisation and/or implementation of glue code depends on the programming language behind a BDD tool. In Cucumber-JVM, for example, step definition functions are usually housed under one or more glue classes; and a glue class consists of glue methods, each annotated with regular expressions matching the steps in the scenarios. For brevity, in Figure 1.3, we only show (using the arrows) the links between the individual steps in scenario *foo* to their corresponding glue methods.

- The production code classes and methods that (will) implement the behaviour described by the BDD scenarios. The glue code invokes only methods in the service-level API of the production code. Some of the required functionality will be provided by private classes and methods, that the glue code cannot access directly. For example, in Figure 1.3, calls to two different private functionalities, *IC1* and *IC2*, are made from the API class, *Class A*. Based on the arrows, while *IC1* is called from two different places in *Class A*, *IC2* is called from only one place in *Class A*; neither *IC1* nor *IC2* is called from *Class B*.

In summary, a BDD specification is formed by a collection of scenarios expressed in a natural language and organised around several features. The natural language scenarios are linked to the code for the SUT through the glue code, whereby, when a scenario is executed, regular expression matching is used to decide the glue methods to be executed as part of a scenario. We next discuss the problem faced by practitioners in maintaining large suites of BDD specifications, part of which is what this thesis addresses.

1.2 The Problem

Despite the advantages of BDD, it also raises problems. When BDD specifications grow large, they can become costly to maintain and extend. System functionality can

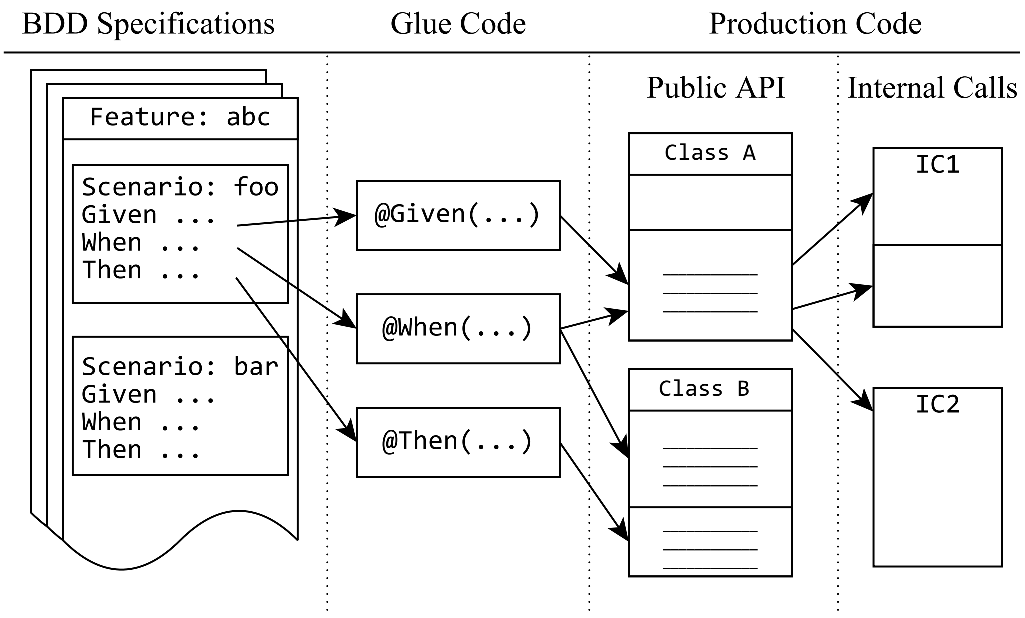


Figure 1.3: Organisation of a typical BDD project

become effectively frozen because of the costs and risks of changing the specifications. These maintenance challenges can be severe and could potentially cause teams to drop the BDD technique, returning to technology-facing testing to play the role of their BDD specifications. Moreover, the intellectual effort of understanding tens or hundreds of examples, and how they relate to one another and to the production code under construction, can be substantial. Tools to support developers in managing their suites of examples are still in their infancy, because BDD is still comparatively new. There could, therefore, be high costs for development teams when redundancy creeps into their BDD specifications. Execution times are likely to lengthen, increasing the delay between the entry of defects into an implementation and their detection by the BDD execution engine. This is especially possible for teams that rely on the execution of BDD specifications in order to detect defects in the System Under Test; in such cases, if test executions take longer, developers, too, might have to wait for too long before they discover the presence or absence of defects in the System Under Test. Waiting for too long before test execution completes might affect the ability of the development teams to plan, prioritise and perform various development tasks. More seriously, maintaining the quality and conceptual integrity of the specification could become significantly harder, increasing the risk that further redundancies, omissions and inelegances will be introduced.

Despite these challenges, existing tools perform poorly on this problem. Specifically, on the problem of detecting duplication in BDD specifications, existing tools can detect textual duplication between BDD scenarios [33], but not semantic duplication between BDD scenarios, particularly when different step phrases are used to express BDD scenarios that represent the same behaviour of the SUT. Besides, after duplicate scenarios are detected, the problem of managing them is non-trivial. For example, given a pair of functionally equivalent scenarios that are not syntactically equivalent, if the maintenance engineer wants to remove one scenario and keep the other, deciding which of the duplicate scenarios to remove is not easy. Naively speaking, given two duplicate scenarios, maintainers can remove any, since the scenarios are duplicates of each other. In practice, however, one scenario may be of better quality than the other. As such, removing a particular scenario may improve or impair the quality of a suite as a whole. There is, therefore, the need for a mechanism to guide maintenance engineers when making decisions on which duplicate scenarios to remove from the suite. To the best of our knowledge, such a mechanism does not exist yet.

1.3 Hypothesis and Research Questions

The central hypotheses in this work are that:

H1: Duplication is a cause for concern among BDD practitioners.

H2: Comparing how BDD scenarios exercise the production code can detect semantically equivalent BDD scenarios in a way that will outperform the detection of duplicate scenarios by existing approaches.

H3: We can use quality aspects of scenarios in a BDD suite to guide the removal of duplicate scenarios from the suite.

To confirm or refute these hypotheses, we posed the following research questions (RQs):

RQ1: To what extent is duplication a problem among BDD practitioners, and how do practitioners deal with duplication in BDD specifications?

RQ2: Can the comparison of how BDD scenarios exercise the production code detect semantically equivalent BDD scenarios in a way that outperforms the detection of duplicate scenarios by existing approaches?

RQ3: What are the characteristics of a "good" quality BDD scenario, and how can we assess the quality of a scenario relative to other scenarios in a suite?

1.4 Thesis Contributions

This thesis makes the following contributions:

1. **The challenges faced by BDD practitioners in maintaining BDD specifications:** We contribute to the understanding of the challenges faced by practitioners in maintaining BDD specifications. In particular, we report the results of the survey of BDD practitioners regarding the problems of duplication in BDD specifications as well as other challenges of maintaining BDD specifications. We learned that duplication in BDD specifications is among the maintenance challenges of concern for BDD practitioners, and that, for the most part, practitioners rely on manual techniques to detect and manage duplication in their BDD specifications. We also learned that BDD specifications suffer the same maintenance challenges found in automated test suites more generally.

2. **Research opportunities to address BDD maintenance challenges:** We identify the need to investigate the adaptation of techniques for maintenance of unit tests, production code and other software artifacts to the problem of maintaining BDD specifications; and the need incorporate maintenance concerns in the BDD workflow, tools, and training materials.

3. **Benchmark of duplicate BDD scenarios:** We develop a benchmark of semantically equivalent BDD scenarios, which can inform further research on BDD specifications.

4. **Understanding the limitation of existing duplicate detection tools on the problem of detecting duplicate BDD scenarios in which different step phrases are used to express the same behaviour of the SUT:** We observed that, when applied to the problem of detecting semantically equivalent but textually dissimilar BDD scenarios, the three mature duplicate detection tools we experimented with either missed the duplicates of interest or returned too many false positives.

This gives us insight into the limitation of tools developed for duplicate detection in program code on the problem of detecting semantically equivalent BDD scenarios that specify the same behavior using different step phrases.

5. **A framework to detect semantically equivalent BDD scenarios:** We propose a framework that differentiates between essential and accidental characteristics of a BDD scenario by analysing several execution traces of a scenario, and compares the essential characteristics to detect semantically equivalent BDD scenarios.

6. **Use of execution traces to detect semantically equivalent BDD scenarios:** During the evaluation of our framework on duplicate scenarios in 3 open source systems, the comparison of execution paths of scenarios detected more duplicate pairs than the comparison of public API calls, public API calls and internal calls, or their combination (execution paths, public API calls, and public API calls and internal calls). Also, the comparison of essential characteristics of scenarios was more effective at detecting semantically equivalent BDD scenarios than the comparison of the characteristics of scenarios observed when each scenario under consideration was executed only once.

7. **BDD Suite Quality Principles:** We propose four principles describing BDD suite quality that can be used to assess which of a pair of duplicate scenarios can be most effectively removed from a suite.

8. **Practitioner Support for the BDD Suite Quality Principles, and Other Quality Facets of BDD Suites:** We report the results of a survey of practitioner support for the four quality principles. All principles received support from the community (with at least 75% of respondents voting in support of each principle), though all of them also received a number of dissenting votes. We also report practitioners' opinions on other quality aspects of BDD suites that are not covered by the four principles.

9. **Operationalization of the BDD Suite Quality Principles:** We propose an approach to operationalize each principle, so that BDD specifications can be assessed automatically against it.

10. **Use of the BDD Suite Quality Principles in generating refactoring advice:** We use the operationalized principles to propose duplicate scenarios for removal, and evaluate the approach on 3 open source software systems. The results from

both the lab evaluation and evaluation with an industry practitioner suggest that the principles can give advice acceptable to human engineers on which duplicate scenarios to remove from the feature suites.

1.5 Publication Activity

Part of this work has been published in the following co-authored articles:

- Binamungu, Leonard Peter, Suzanne M. Embury, and Nikolaos Konstantinou. "Characterising the Quality of Behaviour Driven Development Specifications." In 21st International Conference on Agile Software Development, pp. 87-102. Springer, Cham, 2020.
- Binamungu, Leonard Peter, Suzanne M. Embury, and Nikolaos Konstantinou. "Maintaining behaviour driven development specifications: Challenges and opportunities." 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018.
- Binamungu, Leonard Peter, Suzanne M. Embury, and Nikolaos Konstantinou. "Detecting duplicate examples in behaviour driven development specifications." 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST). IEEE, 2018.

1.6 Thesis Organisation

The rest of this thesis is organised as follows:

Chapter 2 reports the survey of the existing literature on duplication in software systems, to determine whether existing approaches and techniques can detect and remove semantically equivalent BDD scenarios. It also surveys the existing literature on BDD suites analysis and improvement, and identifies the research gap, which this thesis aims to fill.

Chapter 3 presents the survey of BDD practitioners, to understand the challenges they face with respect to maintenance of BDD specifications. Based on the analysis of the literature related to the identified challenges, the research opportunities to support practitioners in maintaining BDD specifications are also presented.

Chapter 4 presents the development of a benchmark of known duplicate BDD scenarios. In particular, it covers the description of the three systems used to develop the benchmark, the approach used to develop the benchmark, as well as the resulting benchmark of known duplicate scenarios across the three systems we used.

Chapter 5 discusses the limitations of existing tools on the problem of detecting semantically equivalent BDD scenarios, and presents and evaluates the framework we propose for detecting semantically equivalent BDD scenarios.

Chapter 6 enunciates the principles we propose for preserving the quality of BDD suites, the support for the principles from the community of BDD practitioners, and the use of the principles to provide removal advise for semantically equivalent BDD scenarios.

Chapter 7 summarises the contributions of this thesis and discus future research directions.

Chapter 2

Literature Review

In this chapter, we want to discover whether the problem of detecting and removing semantic duplication in BDD feature suites has been solved. We first discuss the problem of duplication in software systems in general. Thereafter, through the review of the literature about detecting and removing duplication in software systems, as well as the literature on analysis of BDD feature suites, we demonstrate that existing approaches and techniques have not addressed the maintenance challenge of detecting and removing semantic duplication in BDD feature suites.

Specifically, we look for solutions and solution components in two areas of the literature. The first area is work on duplicate detection and/or removal in non-BDD contexts (for non-BDD artefacts). The second area is work on analysis and/or improvement of BDD feature suites. This includes studies that may or may not be about duplicate detection and/or removal.

We start by presenting, in Section 2.1, the terms used to talk about duplication in (and testing of) software systems. Thereafter, in Section 2.2, we present the general problem of duplication in software systems, evidence of its presence in real world software systems, reasons for its introduction in software systems, as well as its advantages and disadvantages. We then describe how duplication is dealt with in software systems, covering the different approaches for its detection (Section 2.3) and removal (Section 2.4) in software artefacts other than BDD feature suites. We then survey work on analysis and improvement of BDD feature suites (Section 2.5), covering aspects such as quality of scenarios in a suite, coverage of production code by scenarios in a suite, and maintenance of scenarios in a suite. Based on the analysed literature, the research gap addressed in this thesis is presented in Section 2.6. Finally, Section 2.7 summarises the chapter by stating the answers we found for the various questions that were of interest during literature review.

2.1 Terminology

This section introduces the terminology used in the literature to talk about duplication in software systems and the different types of duplication, measuring the performance of duplicate detection tools, as well as testing of software systems.

2.1.1 Terminology on Duplication

The definitions and types of duplication we present next are adapted from the literature on "software clones" (e.g., [34, 35, 36]).

Code fragment: A sequence of lines of code, with or without comments, forms a code fragment. It can be at the granularity of a function level, code block, statements sequence, or a complete statement.

Code duplicate: In a software system, two or more identical or similar code fragments are regarded as duplicates of each other.

Duplicate pair: A duplicate pair is formed by two code fragments which are identical or similar to each other.

Duplicate class: A duplicate class (also called a duplicate group) is formed by a group of identical or similar code fragments.

The "software clones" research community has categorised the different types of duplicates in software systems as follows [1, 35]:

Type 1: Identical code fragments with possible variations in comments and whitespaces.

Type 2: Code fragments with similar syntax but exhibiting differences in identifiers, literals, types, comments and layouts.

Type 3: Code fragments copied and modified to insert or delete statements, and to alter identifiers, literals, types and layouts.

Type 4: Functionally similar but textually different code fragments.

Table 2.1 gives the example pairs of duplicate code fragments for each of the four types of duplication.

Duplicate Type	**Example**	
	Code Fragment 1	**Code Fragment 2**

Type 1	if (a >= b) {
	c = d + b; // Comment1
	d = d + 1;}
	else c = d - a; //Comment2
		if (a>=b) {
		// Comment1'
		c=d+b;
		d=d+1;}
		else // Comment2
		c=d-a;

Type 2	if (a >= b) {
	c = d + b; // Comment1
	d = d + 1;}
	else
	c = d - a; //Comment2
		if (m >= n)
		{ // Comment1
		y = x + n;
		x = x + 1; //Comment3
		}
		else
		y = x - m; //Comment2'

Type 3	if (a >= b) {
	c = d + b; // Comment1
	d = d + 1;}
	else
	c = d - a; //Comment2
		if (a >= b) {
		c = d + b; // Comment1
		e = 1; // This statement is added
		d = d + 1; }
		else
		c = d - a; //Comment2

Type 4	int i, j=1;
	for (i=1; i<=VALUE; i++)
	j=j*i;
		int factorial(int n) {
		if (n == 0) return 1 ;
		else return n * factorial(n-1) ;
		}

Table 2.1: Example code fragments for different types of duplication in program code (adapted from Roy and Cordy [1])

Moreover, three terms are particularly common when evaluating the accuracy of duplicate detection tools [4]:

True Positives: Code fragments reported by a duplicate detection tool, and are actually duplicates of each other.

False Positives: Code fragments reported by a duplicate detection tool, but are actually not duplicates of each other.

False Negatives: Code fragments not reported by a duplicate detection tool, but are actually duplicates of each other.

The following metrics are commonly used to measure the performance of duplicate detection tools. Precision is a measure of the degree of accuracy with which an algorithm detects true duplicates, while recall is used to measure how good an algorithm is at detecting all the duplicates in a system [4]. Figure 2.1 illustrates the general relationship between the actual duplicates that might exist in a software artefact.

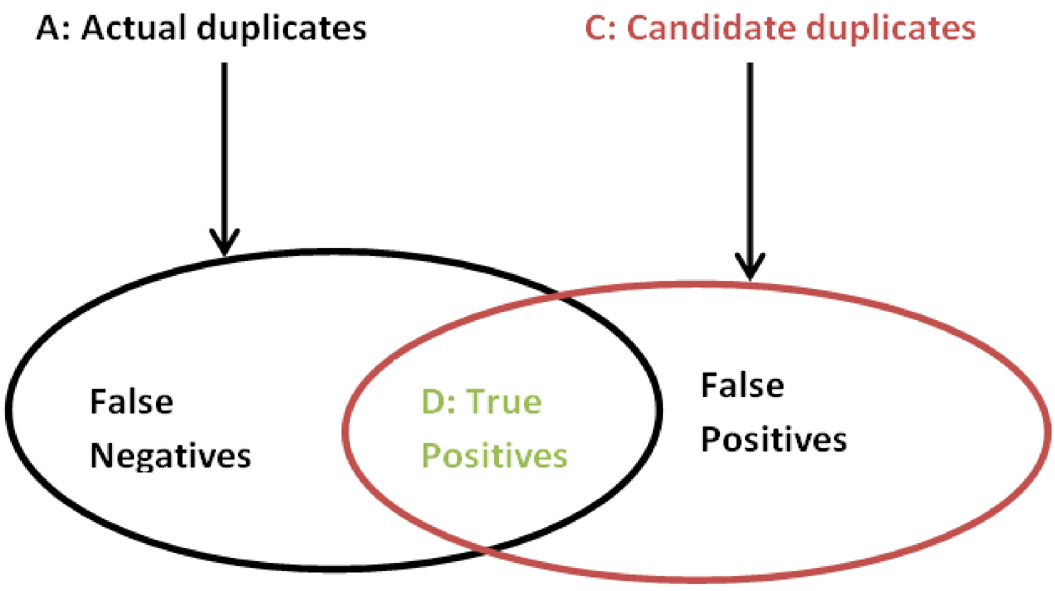


Figure 2.1: Possible distribution of duplicates and non-duplicates in a software artefact (Adapted from Roy and Cordy [4])

Adapting the definitions from the clone detection community, as described by Roy and Cordy [4, 37], and with reference to Figure 2.1, recall and precision for a duplicate detection tool T can be defined by equations 2.1 and 2.2.

$$recall = \frac{|Actual\ duplicates\ in\ S\ detected\ by\ T\ (\mathbf{D})|}{|All\ actual\ duplicates\ in\ S\ (\mathbf{A})|} \tag{2.1}$$

$$precision = \frac{|Actual\ duplicates\ in\ S\ detected\ by\ T\ (\mathbf{D})|}{|Candidate\ duplicates\ in\ S\ reported\ by\ T\ (\mathbf{C})|} \tag{2.2}$$

Further, the F-score (F1) metric gives the harmonic mean of precision and recall. Through it, we can get the balanced view of precision and recall. It is defined by equation 2.3—as summarized by Sasaki *et al.* [38]:

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{2.3}$$

2.1.2 Other Terminology

The following definitions are adapted from the Guide to the Software Engineering Body of Knowledge (SWEBOK) [39]:

> *Software testing:* Verification that a program behaves as expected on a finite set of test cases, which are selected appropriately from the domain that is typically infinite.
>
> *Defect:* Deficiency in a software that causes it (software) to not behave as required or specified. The term “defect” is sometimes used interchangeably to mean either a cause of software malfunction (fault) or an undesired effect observed in the service provided by the software system (failure).
>
> *Bug:* “A defect in source code. An incorrect step, process, or data definition in computer program. The encoding of a human error in source code.” ([39], p. 10-10)

2.2 Duplication in Software Systems

This section presents the problem of duplication in software systems, covering the availability of duplication in real world systems, how and why duplication is introduced, as well as the upsides and downsides of duplication.

2.2.1 Evidence of Duplication in Real World Systems

Duplication of code and/or functionality is commonplace in software systems [40, 41, 42, 43, 44]. It can be introduced intentionally or accidentally [45, 46]. To understand the manifestations of duplication in test code, Hasanain *et al.* analysed a software system which was in use in industry and found that 49% of test code was duplicated [47]. In a survey of developers to understand their experience with functionally similar code, Kafer *et al.* found that 91% of respondents had encountered functionally similar code in their professional practice; and 71% of respondents had experienced the problems caused by the presence of functionally similar code [40]. In an analysis of a repository of 68 Java projects, Suzuki *et al.* found functionally equivalent methods in 28 out of the 68 projects (41.2%) [48]. Li *et al.* found that 22.3% of code in the Linux kernel and 20.4% of code in FreeBSD was duplicated [41]. In a payroll system analysed as part of the duplicate detection study by Ducasse *et al.*, more than 50% of code in some files was duplicated [44].

2.2.2 Causes of Duplication in Software Systems

Reasons why duplicates get introduced into software systems are summarised in the works of Roy [36] and Rattan [35] as follows:

1. **Development strategy:** The use of a development strategy that encourages reuse of different software artefacts such as code, functionality, logic, design, architecture, etc. For example, copy/paste of code to guarantee syntactic and semantic reliability can increase the amount of duplication in a software system.

2. **Reduce development efforts and foster software reliability:** Duplication can be deliberately introduced to reduce development-time effort, and to keep the software reliable. For example, duplication can reduce both the costs of testing new code and risks of errors associated with new code.

3. **Overcoming development-time limitations:** Time limitations and difficulty in understanding an existing system can cause developers to write new functionality by using examples from existing code, thereby increasing the possibility of duplicating code. Also, the use of a language that lacks proper abstraction and inheritance mechanisms can cause developers to write the same code multiple times.

4. **Accidental duplication:** Different members working for the team at different periods of time can coincidentally implement the same functionality.

2.2.3 Upsides and Downsides of Duplication in Software Systems

Apart from the benefits of duplication that can be deduced from the reasons for its introduction (refer to Section 2.2.2), duplication in software systems offers the following additional benefits. It can help developers to overcome limitations of an implementation language, and shorten development times—for example, developers can learn how to use a particular API or library by mimicking existing code [45].

However, despite its benefits, duplication in software systems causes numerous problems:

1. **Increase in maintenance costs:** Although there is some disagreement among researchers on the impact of duplicate program code on software maintenance [49, 45], duplicated code has been reported to increase maintenance costs [50, 51, 52, 53]. Some studies on understanding the impact of duplication on the maintainability of program code have focused on measuring the effort required to understand and change duplicated code in the context of an evolving software system. For example, Mondal *et al.* [50] used hundreds of revisions of six open source systems written in Java, C and C# to estimate the effort required to understand and change duplicated and non-duplicated code. They estimated the effort spent on applying changes that had already taken place on methods, and predicted the effort that might be required to change the methods in the future. Duplicated code was found to require more maintenance effort than non-duplicated code. Specifically, their approach for estimating maintenance effort include the effort required to understand a particular method and its relatives[1], as well as the effort required to change a particular method and its relatives. To estimate the maintenance effort for the changes that have already taken place on methods, they measure two things: (1) time required to understand a particular method and other methods that either have a duplicate relationship with a method of focus or co-change with a method of focus; (2) number of tokens that were changed in a particular method and other methods that either have a duplicate relationship with a method of focus or co-change with a method of focus. To predict the effort that might be required to change a particular method in the future, they also measure two things: (1) time that might be required to understand a particular method and other methods that either have a duplicate

[1]Relatives of a method are methods that either have a duplicate relationship with a method of focus, or co-change with a method of focus during evolution.

relationship with a method of focus or co-change with a method of focus; (2) number of tokens that might have to be changed in a particular method and other methods that co-change with a method of focus. Due to the inability to know whether future changes will take place on duplicated or non-duplicated parts of a method, the number of tokens in duplicated methods were not included in the prediction of effort that might be required to change a method in the future.

Other studies on understanding the maintainability of duplicate program code have focused on measuring the effort required to fix bugs in duplicate code, as compared to the effort required to fix bugs in non-duplicate code. For example, in an experiment with developers, Chatterji *et al.* [51] measured the time taken by developers to fix bugs both when a bug was in duplicated code, and when a bug was in non-duplicated code; they also measured the probability of incorrect bug fixes both when a bug was in duplicated code, and when a bug was in non-duplicated code. They found that it was more difficult to correctly fix bugs in duplicated code than it was for bugs in non-duplicated code. Thus, the cost of providing correct fixes was higher in the presence of duplication than it was in the absence of duplication.

Another group of studies have focused on studying the stability of duplicated code, with the view that, if duplicated code changes too often than non-duplicated code, it (duplicated code) must be involving too much effort to maintain [54, 55, 56, 57]. For example, Lozano and Wermelinger [55] investigated the evolution of duplicate code over time, by analysing how methods with duplicated tokens had changed across different commits. They found that duplicate code was less stable than non-duplicate code, and posited that duplicate code has the potential of making human engineers spend more effort and time on making tweaks on duplicates, at the expense of implementing new software requirements. In another study, Monden *et al.* [53] analysed a 20-year old legacy industry software system and found out that modules with duplicated code were less maintainable than modules without duplicated code, because modules with duplicated code were found to have been revised more times compared to modules without duplicated code, to the extent that modules with duplicated code had become more complex to understand and change.

However, the view that duplicate code is less stable compared to non-duplicate code contradicts the view of other studies that suggest that duplicate code is more stable than non-duplicate code [58, 59]. An example of studies in which duplicated code is said to be more stable than non-duplicated code is the work of Krinke, who studied the changes that had been made on five open source software systems over the period of 200 weeks, and found that duplicated code had the lower average percentage of additions, deletions, or other modifications, compared to non-duplicated code [59]. Krinke's findings were further confirmed in a replication study conducted by Harder and Göde [58]. Such contradicting views might arise due to the differences in the studied software systems, as well as the differences in the methodologies employed by the different studies. However, it calls for further research to seek for a consensus.

2. **Propagation of bugs:** A bug present in a piece of code is likely to be present in all other places where that code is duplicated. So the more the duplication of pieces of code that contain bugs, the higher the chances that bugs will be propagated across the system [35]. For example, in software product-line engineering, a buggy component may be used in several products, increasing the possibility that a bug will be available wherever a buggy component is used [60]. To ensure consistency, bug fixing in one place has to be applied in all other places with duplicated code. Failure to do so can, at a later point, cause erratic behaviour in a system.

   Evidence of duplicate bugs in program code were reported in a study by Li *et al.* [41] in which 49 duplicate bugs in the Linux kernel and 31 duplicate bugs in FreeBSD were discovered. Majority of the duplicate bugs discovered through that study were confirmed by developers of the respective systems and were fixed in the subsequent releases of the systems. Li and Ernst [60] conducted an empirical study on four different projects (a software product line in industry, the Linux kernel, Git, and PostgreSQL) and found that, in a software product line, around 4% of bugs were duplicated across several products or files; they also discovered 282 duplicated bugs in the Linux kernel, 33 duplicated bugs in Git, and 33 duplicated bugs in PostgreSQL.

   Several studies have investigated different aspects of bug replication and proneness in duplicated code. In an empirical study conducted on thousands of revisions of four Java systems and two C systems, Islam *et al.* [61] found that replication of bugs because of code duplication was commonplace. Specifically, they found that: repeated bugs were in up to 10% of duplicated code; repeated bugs were found more in Type 2 and Type 3 duplicates compared to the extent to which they (repeated bugs) were found in Type 1 duplicates; buggy duplicate code groups tended to have very large percentages (mostly 100%) of bug repetitions; and repeated bugs constituted 55% of bugs in duplicated code. Moreover, Barbour *et al.* [62] analysed several revisions of two Java systems to determine the relationship between *late propagation* (a phenomenon commonly used to refer to how late the changes in a piece of code fragment are propagated across its duplicate counterpart(s)) and bug proneness in duplicated code. They found that late propagation increased the risks of bugs in software systems. Mondal *et al.* [63] conducted an empirical study on thousands of revisions of nine software systems (four in Java, three in C, and two in C#) and found that: Type 3 duplicates were more prone to bugs compared to Type 2 and Type 1 duplicates; Type 3 duplicates were most likely to be co-changed during bug-fixing; Type 3 duplicates experiencing bug fixing had higher chances of evolving while preserving similarities among duplicates, compared to Type 1 and Type 2 duplicates; and bug-proneness of duplicates was less related to late propagation. Besides, Xie *et al.* [64] analysed several revisions of three Java systems and, among other things, they found that: mutating duplicate groups of code to Type 2 or Type 3 duplicates increased the risk of bugs; the risk of bugs in a duplicate code group was directly proportional to the extent of differences between code fragments in a duplicate group.

Some of the research about bugs in duplicated code has attempted to detect duplicate bugs in program code. An example of such research is the work of Jiang *et al.* [65] who devised an approach to detect bugs in duplicate code, and used it to detect bugs that had been introduced through duplication in the Linux kernel and Eclipse. 41 bugs were detected in the Linux kernel and 21 bugs were detected in Eclipse. The idea behind their work is that, if two code fragments are duplicates of each other, then any unjustifiable difference between their control flows would indicate the presence of a bug related to inconsistent changes in the duplicate code fragments. Their approach first uses Deckard [66] to detect duplicates in program code. Then, the parse trees of the reported duplicates are compared to detect inconsistencies in the control flows of the reported duplicates. After that, to minimise the number of false positives in the list of what they report as bugs, they apply several heuristics to filter the detected inconsistencies. The filtering process produces a reduced list of inconsistencies, which is used to create bug reports that developers inspect to determine if they are bugs. Another example of research on detecting duplicate bugs in program code is the work of Li *et al.* [41] who proposed an approach that detects bugs in duplicated code based on the extent to which an identifier has been inconsistently changed in duplicated code. In their approach, they compute the extent to which an identifier has remained unchanged as the ratio of the number of times an identifier has remained unchanged to the total number of occurrences of an identifier in a given duplicated code fragment. The lower the ratio of unchanged identifier, the higher the chances that it could represent naming inconsistency, and thus indicate the presence of a bug. As stated earlier in the present section, this approach was used to detect 49 duplicate bugs in the Linux kernel and 31 duplicate bugs in FreeBSD. A similar approach that computes the ratio of identifiers that remain unchanged across pairs of duplicate code fragments was used by Inoue *et al.* [67] to detect bugs resulting from inconsistent changes in two Samsung mobile software systems. 25 true bugs were discovered in the first system, and 1 true bug was discovered in the second system. Li and Ernst proposed an approach that uses isomorphic graphs on the Program Dependency Graph (PDG) to detect semantically equivalent buggy code fragments [60].

Other research about bugs in duplicated code has focused on avoiding repetition of work that is likely to happen when fixing bugs in duplicated code. Nguyen *et al.* [68] conducted manual analysis of repeated bug fixes in five open-source software systems, and found that much of the repeated bug fixes occur in duplicated code. They subsequently developed a technique that utilises a graph representation of how objects are used in a software system to detect similar code and repeated bug fixes. Their approach also recommends other pieces of code to be modified based on bug fixes in their (pieces of code) duplicate counterparts. In another study, to facilitate consistent updates and fixing of bugs in duplicated code, Fish *et al.* developed a tool called CloneMap which analyses duplicates in different revisions of a software system and provides an interactive tree representation of source code, enabling developers to track the evolution of duplicates and get information about inconsistent updates of duplicates [69].

Other researchers have focused on detecting inconsistent bug fixing in duplicated code. For example, the work of Steidl and Göde [70] used features of duplicated code to construct a classifier for predicting inconsistent bug fixes in duplicate code. During empirical evaluation, their classifier attained a 22% precision, suggesting the need for more research on increasing the precision of prediction of inconsistent bug fixing in duplicated code.

3. **Producing poor software design:** Duplicate code can produce software with poor design. This is especially true in the absence of practices such as refactoring, inheritance, and abstraction [71]. Components of a poorly designed software can be difficult to reuse in the long run [36]. Ducasse *et al.* posit that, *"Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality."* ([44], p.1). Because duplication can degrade the design of a software system, its removal can improve the design, and thus the quality, of a software system. For example, to understand the impact of duplicate code removal on software quality, Fontana *et al.* [72] attempted to refactor code duplicates from five versions of Apache Ant and five versions of GanttProject. On almost all the studied source code quality metrics (for cohesion, complexity and coupling), refactoring of code duplicates had a positive impact on software quality, suggesting that the presence of duplicates produced poor design, which improved after the duplicates were removed. However, compared to the other problems of duplication which have been well-trodden in the literature, there is paucity of studies on exploring the question of whether or not the presence of duplicated code in a software system produces poor quality software design. This makes it hard to examine, by way of comparing and contrasting, both the quality and quantity of evidence in favour or against the position that duplication of code hampers the quality of system design.

4. **Increased consumption of technical resources:** Duplicate code can increase the cost of storage, compilation and execution of software, since the same piece of code has to be stored, compiled and executed multiple times [35]. Nonetheless, this claim is purely based on intuition because we could not find any study reporting empirical evidence on the relationship between duplicated code and the consumption of computational resources.

It is the need to address these disadvantages of duplication that have motivated much of the research aiming at detecting [73] and managing [74] duplication in software systems. Table 2.2 summarises work on evidence of duplication in software systems, as well as the advantages and disadvantages of duplication in software systems. After introducing the terminology and the different types of duplication, as used by the "software clones" research community (Section 2.1.1), presenting evidence that duplication exists in software systems (Section 2.2.1), causes of duplication (Section 2.2.2) as well as its pros and cons (Section 2.2.3), we next discuss how it is dealt with. In particular, we discuss the various approaches and techniques proposed in the literature for detecting (Section 2.3) and removing (Section 2.4) duplication in software systems; we also demonstrate that existing approaches and techniques for detection and removal of duplication in software systems suffer significant limitations on the problem of detecting and removing semantic duplication in BDD specifications.

In short, duplicates in terms of "software clones" have been widely studied, and approaches for the detection and management of duplicates have been proposed. For more details on various duplication detection and management approaches, readers can see the summaries by Min and Ping [73], Rattan *et al.* [35], Koschke [75] and Roy *et al.* [74].

S/n	Aspect	Description	Example Citations
1	Evidence of duplication in real world systems	Duplication in test code	[47]
		Duplication in production code	[48, 41, 44]
		Professional experience with duplication	[40]
2	Advantages of duplication in software systems	Reduce development efforts and foster software reliability	[36, 35, 45]
		Overcoming development-time limitations	
3	Disadvantages of duplication in software systems	Increases maintenance costs	[50, 51, 52, 53]
		Propagation of bugs	[61, 41, 60, 62, 64, 65, 67, 68, 69, 70]
		Producing poor software design	[71, 36, 44, 72]
		Increased consumption of technical resources	[35]

Table 2.2: Evidence, advantages, and disadvantages of duplication in software systems

2.3 Duplication Detection

This section reviews the state of the art in duplicate detection, to determine if existing approaches can detect duplicates in BDD specifications.

Existing duplicate detection approaches employ either static analysis, dynamic analysis, or a combination of both static and dynamic analysis.

2.3.1 Static analysis approaches

To detect duplication using static analysis approaches, programs are first converted into common intermediate representations such as text [76, 77], tokens [78, 79, 43, 80], syntax/parse trees [66, 81], Program Dependency Graphs (PDGs) [82, 83, 84], metrics [85] or a combination of these [86, 87]. The intermediate representations are then compared to detect duplicates. The following is a summary of the different types of duplicate detection techniques, as well as example studies in each type of duplicate detection techniques:

Text-based techniques: Segments of text are compared to detect duplication. Specifically, text-based techniques regard source code as a sequence of characters whereby, after whitespaces, comments, and newlines are removed from the source code, character sequences are compared, and code fragments that produce similar character sequences are regarded as duplicates of each other. For example, Ducasse *et al.* [77] proposed a duplicate detection technique in which, after uninteresting information such as comments and white spaces are removed from the source code, the resulting files are compared line-by-line to detect duplicates. When evaluated, their technique recorded the recall and precision values of more than 90%. Another example of text-based techniques can be found in the work of Roy and Cordy [76] who employed the Longest Common Subsequence (LCS) algorithm [88] to compare lines of source code text, to detect duplicate code fragments. Their technique had recall and precision values of up to 100% for some system, though on a small data set.

Token-based techniques: The source code of a software system is converted into a sequence of tokens, which are then compared to detect duplication. A token size is usually specified to limit the number of code fragments that get reported as duplicates based on matches between tokens. For example, in a technique proposed by Wang *et al.* [78] for detecting Type 3 duplicates, lexical analysis is performed to produce tokens for blocks of code, before comparing the resulting tokens to detect duplicates. During empirical evaluation, their tool had more than 80% of recall, precision, and F-score. Another example is the work of Sajnani *et al.* [79] who proposed a token-based duplicate detection technique that uses a language-aware scanner to create tokens for blocks of code, creates an index that maps tokens to their respective code blocks, and uses an index of tokens to search for similar blocks of code. When evaluated, their tool recorded the precision of 91% and the recall of more than 90%.

Tree-based techniques: Program code expressed in a language with a formal syntax is first converted into an Abstract Syntax Trees (AST) or a parse tree. The produced tree is then searched to detect similar sub-trees, and code fragments corresponding to similar sub-trees are reported as duplicates of each other. Jiang *et al.* [66] devised a tree-based duplicate detection technique that works as follows: it builds a parse-tree from the source code, generates vectors that represent structural information in the tree, and performs clustering of vectors; code fragments whose sub-trees produce vectors in the same cluster are regarded as duplicates of each other. During empirical evaluation, their tool was able to detect higher numbers of duplicates than state-of-the-art tools at the time. In another tree-based duplicate detection work, Falke *et al.* [81] proposed a technique in which program code is parsed into an AST, and then the suffix tree of an AST is analysed to detect similar sub-trees, and hence duplicate code fragments. Their tool had the recall of up to more than 70%.

Semantics-based techniques: The source code for a software system is transformed into representations of some aspect of its semantics (for example, program dependence graphs [82]). These representations are then compared, to locate candidate duplicate elements. Examples of semantics-based techniques in which PDGs are used are the work of Tajima *et al.* [89] who compare PDGs to detect functionally similar Java methods (precision of 71.7%), and the work of Krinke [83] in which similar sub-graphs on a PDG are used to detect duplicate code fragments (precision of 100%).

Other semantics-based techniques have employed the notion of *program slicing* in the process of detecting duplication in software [90, 91]. Given a variable *x* and its location in a program, a slice of *x* consists of other parts in a program that have affected x (backward slicing) or are affected by *x* (forward slicing) [92, 93]. For instance, by combining PDGs and program slicing, Komondoor and Horwitz use isomorphic subgraphs to identify duplicate code fragments, and detected a fairly good number of duplicates during empirical evaluation [90].

Metric-based techniques: Collections of code metrics, such as the number of calls to other functions and the number of decision points, are combined to form vectors for each code unit (e.g. class, method); the vectors of metrics are then compared to detect duplicate code fragments. For example, Mayrand *et al.* [85]

collected 21 source code metrics related to naming of identifiers, layout of source code, expressions and control flow, and code fragments with similar values of these metrics were regarded as duplicates of each other. A good number of duplicates was detected during evaluation.

Another example of metrics-based techniques is the work of Fang and Lam [94] in which, in test methods, assertions that are reachable through similar control flows are used to detect duplicate JUnit tests. For each JUnit test, they compute a metric called "assertion fingerprint", and test methods with similar values of assertion fingerprint are reported as duplicates. An assertion fingerprint for each assertion in a test method consists of the following information: number of branch counts from the beginning of a method before an assertion can be reached; number of merge counts from the beginning of a method before an assertion can be reached; number of possible exceptions after an assertion; a boolean value indicating whether or not an assertion is in a loop; and a boolean value indicating whether or not an assertion is in a catch block. All assertion fingerprints of a particular test method, ordered according to how they appear in test code, are combined to form a set of assertion fingerprints for a particular test method. Sets of assertion fingerprints for different test methods are compared to detect duplicate unit tests. When evaluated, their technique achieved a precision of 75%.

Hybrid techniques: Several of the above techniques are combined into a single duplicate detection method. For example, Kodhai *et al.* [86] combined metrics and text comparison to detect Type 1 and Type 2 duplicate code fragments, and achieved up to 100% as values of recall and precision. In the work of Gabel *et al.* [87], after sub-graphs in a PDG are converted to Abstract Syntax Trees, to detect duplicate code fragments, they search ASTs for similar sub-trees. When evaluated, their technique detected higher numbers of duplicates than state-of-the-art techniques at the time.

2.3.2 Dynamic analysis approaches

Dynamic approaches analyse runtime information to detect duplication. On the one hand, some duplicate detection techniques employing dynamic analysis have focused on the analysis of untransformed programs expressed in a high level language. For example, Su *et al.* [95] devised a dynamic analysis duplication detection technique in which Java programs are instrumented and executed, recording execution traces that represent runtime behaviours of the programs. Then, program execution traces are modelled as dynamic call graphs, in which isomorphic subgraphs are used to indicate the presence of duplicate code fragments. They employ a page ranking algorithm [96] to minimize the number of pairwise comparisons when searching for isomorphic subgraphs. During empirical evaluation with Java projects from Google code jam [97], the implementation of this technique recorded the precision of more than 90% in detecting groups of duplicate code fragments.

On the other hand, duplicate detection techniques employing dynamic analysis have focused on the analysis of transformed high-level language programs. Egele *et al.* [98] devised a technique called “blanket execution” for detecting duplicate functions in executable binary code. They perform repeated executions of a particular function in a particular environment (for example, at specific initial values of registers and memory locations), producing a vector of features observed at runtime (for example, memory accesses and performed system calls). Two functions producing similar feature vectors when executed under the same environment are considered to be duplicates of each other. The similarity between feature vectors is computed using Jaccard index [99]. A precision of 77% was achieved during evaluation of the tool.

Park *et al.* analysed runtime stack behaviour to detect similarly behaving software [100]. Their technique does the following: takes as input binary executables; executes binary executables and collects the traces; analyses the traces to identify the relations between different function calls made at runtime, thereby producing the usage patterns for a stack; and compares the usage patterns for the stack, to detect similarly behaving code. The similarity score between stack usage patterns is computed using LCS algorithm [88]. When evaluated, the ability demonstrated by this technique in detecting similarly behaving software was comparable to MOSS [101], the state-of the-art software similarity detection tool at the time.

2.3.3 Combination of static and dynamic analysis approaches

To minimize the amount of information to be processed at runtime, with the view of reducing the computational costs, some studies have combined both static and dynamic analysis approaches to detect duplicates. To detect duplicate code, techniques in this category have mainly focused on comparing inputs and outputs. In the work of Jiang and Su [102], static analysis is combined with dynamic analysis to detect duplicate code fragments based on if they have the same outputs for the same inputs. Their technique works as follows: First, function definitions in source code are parsed into statement sequences, and then each of the possible consecutive statement subsequences is regarded as duplication candidate. Second, variables that are used without being defined in a particular duplication candidate are regarded as input variables to the duplication candidate, and variables that are defined in a duplication candidate without being used in it are regarded as its output variables. Third, input generation is conducted, in which random values are generated for the input variables. Fourth, code fragments that are duplication candidates are executed on generated random input values, and code fragments that produce the same outputs on the same inputs are grouped together in a duplication class.

Using their approach, many duplicate code were detected in the Linux kernel 2.6.24. They also found that about 58% of the detected functionally similar code fragments were syntactically dissimilar, fortifying the need for duplicate detection techniques that focus on equivalence of functionality. They also recorded a precision of more than 67%.

Su *et al.* [103] devised a technique that compares inputs and outputs to detect duplicate Java methods. They first identify the outputs of a method as all variables that are written by a method and can be observed (e.g used by a developer) after a method finishes execution. Examples of outputs include the following: a variable passed to another method, a variable returned by a method, a variable written to a static or instance variable, and a variable written to an array which is a parameter to a method. After the outputs are identified, they use data and control flow analysis to determine all the input variables that affect each of the output variables. This produces input-output (IO) sets for the methods under analysis. Then, they perform bytecode instrumentation for the variables in the IO sets, executes the methods and records the actual values of IO variables at runtime. Finally, during similarity analysis, using the Jaccard index, methods that produce similar IO sets are regarded as duplicates of each other.

During empirical evaluation, their approach detected more than 800 duplicate methods in Google code jam projects [97], with 68% and 15% respectively as the proportions of true positives and false positives in the candidate sets.

The work of Elva [104] combines static and dynamic analysis techniques to detect semantically equivalent Java methods. In their approach, two methods are considered to be semantically equivalent if they have the same input and output, and affect the heap in the same way. The dynamic analysis part is preceded by the static analysis part. During the static analysis phase, only methods that have the same parameter types, the same return types, and can write to the same variables, are regarded as duplication candidates and are put into the same group. After groups of candidate duplicate methods are identified, depending on the type (instance or static) of each method in the duplicate candidate groups, receiver objects are created and random values of method parameters are generated. Thereafter, methods are invoked on created receiver objects or on individual classes (for static methods), recording the return values of methods as well as heap state at the end of method execution. Methods that produce the same output (return value and heap state after method execution) on the same input (parameter values and heap state at the time of method invocation) are considered to be semantically equivalent. Their approach detected 502 semantically equivalent methods in six open source Java software systems. They also found that, on average, static analysis reduced the required dynamic analysis by 91%.

Table 2.3 summarises the existing duplicate detection approaches.

2.4 Duplication Removal

As presented in section 2.3, there has been a lot of work on identifying duplicates in software systems, but considerably less work on advising stakeholders on how to reduce the amount of duplication. After duplicate pairs of code fragments have been detected in a software system, there can be numerous ways to manage the detected duplicates [74]. While in some cases it may be appropriate to find a suitable way to merge the duplicates, in other cases it may be appropriate to remove one of the duplicates in a pair and keep the other. In cases where one duplicate has to be removed while keeping the other, the process of deciding which of the duplicate to remove can be non-trivial; various factors, some of which could be related to the overall system quality, can come into play. We now discuss the attempts made to reduce duplication in software systems, as reported in the literature. In so doing, we demonstrate that the problem of advising users on which BDD scenario should be removed, given a duplicate pair of BDD scenarios, has not been addressed.

S/n	**Approach**	**Techniques**	**Example citations**
1	Static analysis	Text-based	[77, 76]
		Token-based	[78, 79]
		Tree-based	[66, 81]
		Semantics-based	[89, 83, 90, 91]
		Metric-based	[85, 94]
		Hybrid	[86, 87]
2	Dynamic analysis	Untransformed high-level language programs	[95]
		Transformed high-level language programs	[98, 100]
3	Combination of static and dynamic analysis	Input-Output comparison	[102, 103, 104]

Table 2.3: Summary of duplicate detection approaches

2.4.1 Refactoring of duplicates in production code

Research has been done into the refactoring of duplicate code in software systems, mainly focusing on software "clones" [105, 106, 107, 108, 109, 110, 111]. Some studies in this area have been about assessing the ability of duplicates to be safely refactored. Tsantalis *et al.* [109] proposed a technique that assesses the differences between duplicates to determine whether or not they can be merged without altering the program's behaviour. Their technique extracts data dependencies between statements in the duplicates, and checks whether these dependencies are preserved after the merging of duplicates. Evaluation of the technique showed that it provided acceptable assessment of the refactorability of duplicates, because most of the duplicates that were judged as refactorable by this technique could be refactored without causing test failures or compilation errors. Also, among other things, they found that duplicates in production code can be easily merged compared to duplicates in test code; and that duplicates in the same files, classes, or methods can be easily merged compared to duplicates in different files, classes or methods.

Chen *et al.* [112] proposed a tool called PRI (Pattern-based Clone Refactoring Inspection) which can advise developers on duplicates that can be removed and appropriate ways to remove them, as well as the duplicates that cannot be removed and the reasons for why they cannot be removed. It also does the following: summarises refactorings that have taken place on groups of duplicate code across various revisions of a software system, identifies instances in which duplicates in the same group have been inconsistently refactored and the reason for the inconsistency, advises developers on ways to safely remove duplicates in code, and ensures consistent removal of duplicates that belong to the same duplication group. Evaluation of PRI on 6 open-source software systems revealed that PRI can detect refactorings of duplication in code with an accuracy of 94.1%, and it can detect inconsistent refactoring of duplicates with an accuracy of 98.4%. Also, the qualitative evaluation of the tool with 10 students revealed that PRI simplifies inspection and monitoring of duplicate refactorings.

Other studies in this area have been about prioritising duplicates for refactoring, as well as guiding the actual refactoring. For example, to guide developers in differentiating duplicates that are more important for refactoring from duplicates that are less important for refactoring, Mandal *et al.* [108] devised a technique that suggests the refactoring of duplicates that have a tendency of changing together during software evolution. Their technique suggests for refactoring duplicates that co-change without losing similarity, across different revisions of a software system. Both the duplicates that co-change without losing their similarity and the relationships among co-changing duplicates are identified. Then, based on how likely to change the duplicates are, co-changing duplicates and their relationships are ranked for refactoring. Empirical evaluation of the technique on 13 software systems confirmed that co-changing duplicates were good candidates for refactoring.

To advise developers on what duplicates to refactor and how to do the refactoring, Yue *et al.* [106] proposed an approach that suggests duplicates to be refactored based on information from the present status of a software system as well as information from the past history of a software system. A total of 34 features about the duplicates were used to train the classifier that suggests the duplicates to be refactored. The 34 features are divided as follows: 11 features are about the code that constitutes the duplicates (examples include lines of code in a duplicate, number of tokens in a duplicate, and whether a duplicate is a test code or production code); 6 features are on the history of the duplicates (examples include percentage of developers who have maintained a particular file, and percentage of commits that changed the file); 6 features are on relative locations of duplicates (for example, whether or not the duplicates are in the same directory, file, class or method); 6 features are on the differences in syntax among the duplicates (for example, number of duplicates in the same duplication class, and proportion of naming differences for variables in a duplication class); and 5 features are about co-evolution of duplicates (for example, percentage of commits that change all duplicates in a particular duplication group).

Experimentation with this approach recorded an F-score of 83% for duplicates within the same project, and an F-score of 76% for duplicates across different projects. This outperformed the state-of-the-art approach described in the work of Wang and Godfrey [107], the classifier in which was trained based on 15 features only, and recorded the F-scores of 70% and 50% respectively for duplicates within the same project and duplicates across different projects.

In a work similar to ours, Fontana *et al.* [105] proposed a tool called DCRA (Duplicated Code Refactoring Advisor) for advising developers on best ways to refactor pairs of duplicate code in Java systems, focusing on improving the quality of system code. In DCRA, first, a tool called NICAD [76] is used for duplicate detection, and they augment each detected duplicate with information about its location, size, and type. Thereafter, using the information about the location of duplicates and the variables they contain, they suggest possible refactorings for the different duplicate pairs, and rank the suggested refactorings based on the quality of resulting code if the refactorings were applied. Quality is determined in terms of how the resulting refactoring would make good use of three OOP features: encapsulation, inheritance, and polymorphism. The tool also provides the summary of information about the duplicates, showing, among other things, the sizes of duplicates as well as the duplicates which should be most convenient to refactor. The evaluation of DCRA on four software systems gave advises on refactoring for more that 80% of known duplicate pairs, and the application of refactoring advises that were given by DCRA caused 66% of all duplicates across four evaluation systems to be properly refactored.

Tsantalis *et al.* investigated the use of lambda expressions [113] to support the merging of duplicates. Among other things, the use of lambda expressions was found to be more applicable for merging duplicates in test code than for merging duplicates in production code [110].

2.4.2 Removing duplication in test code

Various refactorings on test code were proposed by Van Deursen *et al.* [114]. For duplication in test code, in particular, they suggested the use of *Extract Method* [71], if duplication is in the same test class. For cross-class duplicated test code, however, they suggested mimicking the hierarchy of production classes in the hierarchy of test classes. For example, in some context, it might be helpful to move duplicated test code from two different test classes into a common class.

Various studies have applied the refactorings by Van Deursen *et al.* to refactor duplication in test code. Here, we give two examples of such application, one in the context of JUnit test suites and another one in the context of domain-specific test suites. Zhao [115] developed a tool called JTestParametrizer that refactors duplicates across pairs of JUnit test methods. First, a refactoring tool called JDeodorant [116] is used to map identical AST nodes and to identify the differences. Then, pairs of AST nodes in duplicate test code are mapped into a unified duplicate tree, and a parametrised utility method is used to combine the differences in behaviour, data and type. Thereafter, parameter values are passed to the extracted utility method to instantiate individual test cases. When evaluated on 5 open-source projects, with a total of 415 duplicate pairs, the tool was able to refactor 65% (268 pairs) of the duplicate pairs, and all the refactored Junit test methods could be compiled while 94% of the refactored test methods were able to pass when executed. Also, it was noted that using JTestParametrizer for refactoring produced precise test suites.

The removal of duplication in TTCN-3 (a test scripting language used in the telecommunication domain) was investigated in the work of Neukirchen *et al.* [117]. They proposed different refactorings for TTCN-3 test suites and implemented the various refactorings they proposed in a tool called TREX. Various refactorings for duplicated test code in TTCN-3 test suites were part of a broad array of refactorings proposed in their study. Among other things, they suggested the use of *Extract Method* to refactor duplicate templates in TTCN-3 test suites. A significant reduction in the number of lines of test code was observed when TREX was used to refactor three different TTCN-3 test suites; changeability of the evaluation test suites was also improved.

2.4.3 Test Suite Reduction

The body of work on Test Suite Reduction (TSR) [118], too, exploits duplication to determine a representative subset of tests that covers a whole test suite. The goal of TSR is to minimize both the size of the test suite and the time required to execute tests within the suite. Reduction of test execution time is especially important if fast feedback cycles are required by developers. The TSR problem can be summarised as follows:

> *Given a test suite T and a set of testing requirements R_n, the TSR problem is about finding $T_i \subset T$ such that T_i can still satisfy R_n.*

A testing requirement $r_i \subset R_n$ is often expressed in terms of the amount of coverage that is regarded as adequate for a particular component of the program under test. The result of a TSR process is a subset of a whole test suite in which duplicate test cases (multiple test cases covering the same testing requirements) have been removed.

Various TSR techniques proposed in the literature remove whole test cases from the test suites, and rely on the availability of coverage information. An example of TSR techniques that use coverage information to remove whole test cases from the suite is in the work of Smith *et al.* [119] who devised a TSR technique that works as follows: First, the code under test is instrumented so that the call tree associated with an execution of each test in a suite can be constructed when tests are executed. Then, tests are executed, creating a call tree, in which a node is created for each test case, and a path under each test node–formed by a sequence of method calls associated with the execution of a test–forms a unique test requirement for the program under test. Thereafter, a greedy heuristic algorithm [120] is applied on call trees to compute representative subsets of whole test suites. When applied on the evaluation case study, this technique reduced the tests in the suite by 45% and had a 82% reduction in execution time.

Realising that it can be time consuming and hard to collect, store and keep coverage information updated, Zhang *et al.* [121] proposed the TSR technique that works as follows to remove whole test cases from a suite: First, it constructs static call graphs to represent the relationships between tests in a suite and the system under test. Then, the greedy algorithm [120] is applied on static call graphs to identify tests that test the same part of the production code. This process enables the removal of duplicate whole test cases, producing a representative subset of an entire test suite.

Other TSR techniques in the literature seek to remove partial duplication among test cases in a suite. For example, by focusing on partial duplication among test cases, Vahabzadeh *et al.* [122] proposed a tool called TESTLER that performs TSR with an aim of removing duplicated statements in JUnit tests. Specifically, to avoid unnecessary loss of whole test cases while paying attention to the possibility of partial duplication among test cases in a suite, their technique analyse test execution information to model the relationships between statements in individual test cases and test states, enabling the re-arranging, identification and removal of duplicate statements in tests. When the technique was evaluated on 15 open-source projects, 43% of duplicate test statements were removed, which reduced tests with partial duplication by 52% and test execution time by up to 37%.

2.4.4 Removing duplication in natural language tests

To produce components that can be reused across multiple tests, Hauptmann *et al.* [123] used grammar inference algorithms to extract and group together test steps resulting from partial duplication between automated system tests expressed in a natural language [124]. Information such as the number of steps in a reuse component and the number of references to it are used by maintenance engineers to decide on suitable ways to refactor steps that are duplicated across several test cases. They first perform detection and annotation of all duplicates in a test suite. Then, by presenting test steps as sequences of tokens, they convert an entire test suite into input sequences for a grammar inference algorithm called Sequitur [125]. Modelling the provided input sequences by detecting recurring token sequences, Sequitur produces a grammar which is used to reconstruct the test suite in which steps duplicated across multiple test cases are grouped into reuse components. Applied on an industry case study, this approach helped human engineers in understanding how tests were internally structured and in removing duplicates from the evaluation test suite.

Devaki *et al.* [126] investigated the merging of Graphical User Interface (GUI) tests that contain identical steps. Their technique compares states induced by tests at runtime to detect partial duplication between steps from different test cases in a test suite. Then, the test cases with duplicated steps are merged into a single test case in which duplicated steps are executed only once. When the technique was evaluated on five web applications (four open source and one proprietary) with a total of more than

3300 tests that consisted of 19600 test steps, the number of test steps were reduced by 29% and the test execution time was reduced by 39%.

2.4.5 Assessing the quality of tests and requirements

Various studies have investigated metrics for assessing and improving the quality of test suites. Tengeri *et al.* [127] devised a method to assess and improve the quality of test suites. The method goes beyond the use of the traditional code coverage proportions when assessing the quality of a test suite. To use the method, one has to start by setting an improvement goal (e.g removing duplicate test cases, improving coverage of some parts of code, etc.). Then a granularity of focus is chosen; it can be coarse (e.g functional level) or fine (e.g statement level). After the granularity of focus is chosen, the tests are executed, different metrics are computed based on coverage information gathered through test execution, and the results of the metrics are used to inform the process of updating tests and code.

The study defined five different metrics that are computed from coverage information, and can be used to assess the quality of a test suite. The metrics are computed for pairs of test groups (a subset of test cases in a suite that tests a particular functionality) and code groups (code elements that implement a particular functionality). The first metric is about coverage of a test group for a particular code group: the ratio of the number of elements in a code group that are covered by tests in a test group to the total number of code elements in a code group. The second metric is about partition which differentiates elements of the code group based on tests that cover them; elements with the same coverage information are indistinguishable and thus they belong to the same partition. The third metric is about tests per program elements which is the ratio of the number of test cases in a test group to the number of elements in a code group; it tests the extent to which tests are unique and efficient. The fourth metric is about specialization of a test group and it is computed as the ratio of number of test cases in a test group that cover elements of a particular code group to the number of all test cases in the whole test suite that cover elements of a particular code group. The fifth metric is about uniqueness of tests for a particular code group which is computed as the ratio of the number of elements in a code group that are only covered by a test group to the number of all code elements covered by a test group. During evaluation, it was observed that the method did improve various metrics computed for the test suite of the evaluation system.

A study by Palomba *et al.* [128] found that test cohesion and test coupling are important criteria for increasing the quality of automatically generated test cases, and included these criteria in their algorithm for search-based test case generation. Meszaros [129] defines test cohesion and coupling as follows: Test cohesion refers to the simplicity of a test case–a highly cohesive test case should not be involved in the verification of a lot of functionality. Test coupling, on the other hand, measures the extent to which tests overlap with each other. Easily maintainable tests should have low cohesion values. Improvement in quality of automatically generated test cases was observed when the two criteria were incorporated in the algorithm for automatic generation of test cases using EvoSuite [130].

Daka *et al.* [131] used human judgement to develop a model for assessing the readability of unit tests, and then applied this model to generate readable unit tests. Crowd-sourced humans were used to rate the readability of tests on a five point scale (1 to 5). This formed a ground truth indicating the readability of various unit tests. After that, 24 various structural, complexity, and code density unit test features were selected and used to build the model. When applied on the human-judged ground truth for unit tests readability, the model was found to be in agreement with humans by 89%. Moreover, using the model to augment automatic generation of unit tests, it was found that more readable unit tests were generated, and the speed at which humans could answer questions about maintenance increased by 14% without losing accuracy.

There have been a small number of attempts to assess the quality of natural language tests and requirements through the notion of *smells*. Hauptmann *et al.* [132] used their experience of evaluating the quality of tests of industrial systems to propose a set of seven smells in manual tests expressed in a natural language. They also describe the activities (test maintenance, execution, or understanding) that are negatively impacted by each smell they propose. For example, for the smell about duplication in tests, they state that it negatively affects comprehension and maintenance of tests. They also devised approaches to detect each of the seven smells they proposed. For instance, they suggest the use of substring matching to detect duplication in tests. By applying their smell detection approaches, they were able to detect smells in seven industrial test suites.

The work of Femmer *et al.* [133] proposed nine smells in natural language requirements, and devised the methods for detection of the proposed smells. The detection methods were implemented in a tool called Smella which parses the requirements expressed in various formats (Microsoft Word, Microsoft Excel, PDF, plain text, and CSV) producing plain text; annotates the language used to express the requirements with meta-data that indicate, among other things, the roles of words (e.g adjective, possessive pronoun, etc.) in sentences; use annotations on requirements to detect smells; and presents the results of the smell detection process in a human-readable format. When evaluated on four systems (three industry and one academia), Smella detected smells in natural language requirements with the precision of 59% and the recall of 82%.

Table 2.4 summarises work on removing duplication in software systems, as well as assessing and improving the quality of test suites and software requirements.

S/n	Aspect	Example Citations
1	Removing duplication in production code	[112, 105, 106, 107, 108, 109, 110, 111]
2	Removing duplication in test code	[114, 115, 117]
	Test Suite Reduction	[119, 121, 122]
3	Removing duplication in natural language tests	[123, 126]
4	Assessing the quality of tests and requirements	[127, 128, 131, 132, 133]

Table 2.4: Approaches for removing duplication, as well as assessing and improving the quality of test suites and requirements

2.5 Analysis and Improvement of BDD Specifications

We also wanted to discover whether the detection and removal of duplicate BDD scenarios has been addressed by the existing studies on BDD. What we discuss next is the work on analysis of BDD specifications. In particular, we pay attention to studies on quality and maintenance of BDD feature suites.

There have been attempts to define quality in the context of BDD feature suites. Oliveira *et al.* [134] defined 8 quality attributes for BDD scenarios and devised a checklist with

12 questions to assist human engineers in evaluating the quality of BDD scenarios. This was a result of interviews with 18 BDD practitioners who gave opinions on quality attributes obtained from the literature on requirement engineering [135, 136] and user stories [137], as well as their (interviewees) other personal criteria for good quality BDD scenarios . Specifically, the following were found to be attributes of a good quality BDD scenario:

1. **Essential:** A BDD scenario should be concise enough by including only essential information in its steps' text.
2. **Focused:** A BDD scenario should be declarative, by stating what the scenario should do rather than how to do it.
3. **Singular:** A BDD scenario should have one clearly stated purpose–it should test only one thing.
4. **Clear:** A BDD scenario should be unambiguous and understandable to all project stakeholders.
5. **Complete:** Steps of a BDD scenario should have enough information that makes them easy to follow, and a set of scenarios in a feature should cover the feature as high as possible.
6. **Unique:** A BDD scenario should test something different from all other scenarios in a feature.
7. **Ubiquitous:** A BDD scenario should consistently use domain terms that are understandable to all project stakeholders.
8. **Integrous:** A BDD scenario should properly follow rules of a Gherkin language: A *Given* step should describe the pre-conditions, should come before *When* and *Then* steps, and should be in past tense. A *When* step should describe events, should come before *Then* step but after *Given* step, and should be in present tense. A *Then* step should describe the post-conditions, should come after *Given* and *When* steps, and should be in future tense.

As well, of the 12 questions on the checklist they proposed for evaluating the quality of BDD scenarios, 2 questions are for evaluating quality at a feature level, 6 questions are for evaluating quality at a scenario level, and 4 questions are for evaluating quality at a step level.

Moreover, a section of studies has focused on enforcing quality in software systems that employ BDD, by encouraging high coverage of production code by the associated BDD scenarios, as well as high reusability within BDD feature suites. Diepenbeck *et al.* [138] proposed an approach for increasing the coverage of BDD tests by generating BDD scenarios for uncovered production code. They first conducted an empirical study on several versions of two BDD projects, and found that coverage of production code by the projects' BDD scenarios decreased as the projects evolved over time. Based on this finding, they devised a technique to automatically generate BDD scenarios to cover the uncovered production code. They follow a two-stage process to generate BDD scenarios for uncovered production code: *global coverage* in which they ensure that all uncovered production methods are called by the glue code, and *local coverage* which focuses on covering lines in the production code that remain uncovered even after the *global coverage* stage. To create new BDD scenarios for uncovered production code, they introduce a notion of *feature graph*, a graph of all features and their associated scenarios in a BDD feature suite whose vertices represent system states and edges represent scenarios' steps. When generating scenarios for the uncovered production code, their approach first attempts to use the existing steps from the *feature graph* before resorting to generating new steps, particularly when the existing steps cannot cover specific part(s) of the production code.

To understand the potential for reuse amongst BDD feature suites, Irshad [139] used both the Normalised Compression Distance (NCD) algorithm and the Similarity Ratio (SR) algorithm to detect similarity between pairs of BDD scenarios. While the NCD algorithm was adapted from the literature [140], the SR was specifically proposed in their work. According to [140, 139], NCD is computed as shown in equation 2.4, where Z represents a compression algorithm, Z(x) and Z(y) respectively represent the sizes of datasets x and y after compression, Z(xy) represents the compressed size for the concatenation of datasets x and y, and min and max respectively are the functions for determining the minimum and maximum of given values.

$$NCD = \frac{Z(xy) - min\{Z(x), Z(y)\}}{max\{Z(x), Z(y)\}} \tag{2.4}$$

NCD values can be in the range [0, 1], where 0 indicates that two entities are completely identical, 1 indicates that two entities are completely different, and a value between 0 and 1 represents the distance between the two entities.

SR is computed as shown in equation 2.5. SR values are also in the range [0, 1] where 0 indicates that the two BDD scenarios do not have any step in common, and 1 indicates that all of the steps in the shorter BDD scenario are also in the longer BDD scenario.

$$SR = \frac{Number\ of\ similar\ steps\ in\ the\ two\ BDD\ scenarios}{Number\ of\ steps\ in\ the\ shorter\ BDD\ scenario} \tag{2.5}$$

When evaluated on BDD feature suites for two industry systems (one with 72 BDD scenarios, and another one with BDD 15 scenarios), SR slightly outperformed NCD in detecting similarity between BDD scenarios. SR could detect the false positives and false negatives that were observed when NCD was applied. Both approaches, however, produced the results that highly matched the opinions of domain experts. It was generally learned that there is a potential for reuse of steps and other facets in BDD feature suites.

In another work, Landhauer and Genaid [141] developed a tool called F-TRec (Functional Test Recommender) that builds a knowledgebase consisting of user stories, BDD scenarios, glue code, and production code, and the relationships between these artifacts, and then uses the knowledgebase to recommend steps for reuse when new BDD scenarios are being written. During empirical evaluation, their approach recorded a high recall (77.5%) but a low precision (8.68%).

Other authors have investigated how to improve the quality of BDD specifications by reducing maintenance costs that can be caused by such things as duplication and inconsistent updates in BDD projects. Yang *et al.* [142] studied the maintenance of traceability links between BDD feature files and the associated source code files. They devised a mechanism that can help maintainers to determine the feature files that should be modified when there is a modification in source code files. To detect co-changing patterns between feature files and source code files, they analysed a collection of file changing commits by comparing words from Gherkin feature files to words from Java source files. In particular, they used an NLP (Natual Language Processing) tool called Stanford CoreNLP [143] to extract from Gherkin feature files words that were either nouns or verbs. They focused on nouns and verbs because other types of words are not commonly used in Java [144] and they wanted to compare words from Gherkin feature files to words from Java files. A Java parser was used to identify and eliminate keywords that are specific to the Java language (for example, public, private, import, etc.). The words that remained after eliminating Java keywords formed a set of words for comparison with words extracted from Gherkin feature files. After keywords from each pair of co-changing files were identified, they computed the Cosine Similarity measures [145] for strings of words in pairs of co-changing files. A Cosine Similarity of above 95% was used to identify co-changing files. This approach was able to detect co-changing files with79% accuracy.

Moreover, based on the patterns of co-changes that were observed in feature files and source code files, they also proposed a Random Forest [146] model for predicting when to modify feature files before committing changes resulting from modification of source code. Their prediction model scored an AUC (Area Under the Curve) of 0.77. As well, they also found that the number of files of test code that are added as part of a commit, the number of other files that are modified as part of a commit, the number of files of test code that are renamed as part of a commit, and number of lines of code deleted from source code in a commit give best prediction of co-changes between feature files and source code files.

Suan [33] devised the techniques for detection and removal of textual duplication in BDD feature suites. They use identical text to detect exact matching of various elements (feature/scenario/scenario outline titles, backgrounds, steps, etc.) in Gherkin features. They also use a Dice Coefficient (DC) [147] algorithm to detect duplicates that are textually different. DC can be computed as shown in equation 2.6, where L1 is the number of terms in the first text, L2 is the number of terms in the second text, and C is the number terms that are common across the two texts, L1 and L2. For the purpose of the duplication detection algorithm, the terms in the text are expressed in the form of bigrams (consecutive units of words, letters, syllables, etc.)

$$DC = \frac{2*C}{L1+L2} \tag{2.6}$$

The study also suggested the following four different refactorings for the duplicates that their approach detects in BDD feature suites. The first one is about *renaming* of Gherkin elements, and it is applied on Gherkin elements that have unnecessary exact matching. For example, when two different scenarios in the same feature have the same title, it may be appropriate to rename the title for one of the scenarios. The second refactoring is about *removing* of Gherkin elements, which is applied on exact matches of Gherkin elements when one of them is clearly unwanted. For example, when a feature has two textually identical scenarios, one of then can be removed. Third is *migrating to Background* of certain Gherkin elements, and it is applied when certain *Given* steps are repeated across all scenarios of a particular feature. The fourth refactoring is about *combining* of some Gherkin elements, and it is applied on similar scenarios that differ only in terms of inputs and outputs. Such scenarios can be combined into a single scenario outline.

Of all the duplicates detected by SEED (the tool implemented as part of their study) during evaluation, 11% were also detected by human experts, 40% were the duplicates that should have been detected by human experts but there were no evidence that the human experts could also detect them as duplicates, and 49% were false positives returned by SEED. Also, 81.8% of the refactorings suggested by SEED were in agreement with the suggestions of human experts, and were mostly about removal and renaming of duplicates.

2.6 Research Gap

Essentially, the state-of-the-art techniques on duplication detection (refer to Section 2.3) have successfully located duplicates in program code. Detecting duplication in BDD specifications, however, poses specific challenges that naive application of these techniques do not address. As described in Section 1.1, BDD scenarios are expressed using a natural language, containing semi-structured elements and only minimal syntactic rules. The meaning of these semi-structured elements is defined by an accompanying body of code (so-called *glue code*) that must be supplied alongside them. Any attempt to detect duplicates in BDD specifications must take into account both these elements.

More radically, much duplicate detection in the context of program code is based around the idea of *clone detection*. While cloning is almost certainly used in the creation of BDD specifications, it is not the main issue of concern. BDD scenarios often exhibit high degrees of textual similarity—almost by definition. Within a feature file, each scenario should express a different aspect of the functionality, typically targeted at discontinuities in the functionality being specified: for example, defining the boundaries of what is acceptable behaviour, and the response of the system to erroneous inputs. These scenarios will typically contain very similar *Given* and *When* steps, with only the parameter values (the values matched by the regular expression groups in the step definitions) differing. Far from being problematic, such reuse of step definitions is important in keeping the BDD specification manageable and comprehensible. In BDD, we effectively create a language of common domain concepts that are important to the (eventual) users of the system. A coherent domain language cannot emerge from this process unless we are disciplined about refining and reusing step definitions wherever possible throughout the specification (except where the effect on readability negates the benefits of the reuse).

Therefore, the detection of clones in BDD scenarios is unlikely to be helpful. We need to be able to detect the kind of duplication that indicates redundancy in the BDD specification: when two or more scenarios describe the same piece of functionality. As discussed in Section 1.2, such redundancy has a high cost for teams using BDD, leading to slower QA feedback cycles, increased effort needed to comprehend and change the specification, and even in some cases to frozen functionality.

Considering the static analysis duplicate detection approaches (refer to Section 2.3.1), as observed by Juergens *et al.* [148] and Su *et al.* [95], static analysis approaches can be poor at revealing code with same behaviour since they solely depend on common intermediate program representations (for example, ASTs for tree-based techniques) for duplicate detection, when code with the same behaviour can have different intermediate representations. Consequently, this renders static analysis approaches unsuitable for detecting semantically equivalent BDD scenarios.

Furthermore, both the dynamic analysis approaches (see Section 2.3.2) and the approaches that combine static and dynamic analysis (see Section 2.3.3) generally rely on input-output comparison to detect duplication. Nevertheless, as noted by Su *et al.* [103, 95], deciding the extent of input-output similarity that is required to suggest duplication between programs is not easy. In Object Oriented Programming (OOP) languages, for example, domain-specific types can cause duplicate code to be immune to detection through input-output similarity analysis. As such, focusing on externally observable behaviours only (input and output) while ignoring information about internal execution behaviour of a program can miss important information required to indicate the presence of duplication.

As for the attempts to detect duplication in BDD specifications, the techniques proposed by Suan [33] and in the work of Irshad [139] can only detect textually similar duplicates. To the best of our knowledge, the work in this thesis is the first attempt to detect semantically equivalent BDD scenarios.

Regarding duplication removal, the following are the limitations we deduce from the literature. All of the techniques discussed in Section 2.4.1, Section 2.4.2 and Section 2.4.3, focus on removing duplicates in production code or test code expressed in typical programming languages, not duplicates expressed in a natural language, as is the case for BDD scenarios. Moreover, the contexts of the studies about the removal of duplicates in tests expressed in a natural language (refer Section 2.4.4) relate to ours in a sense that they were also working with tests expressed in natural language. However, these studies have focused on extracting repeated steps within tests, rather than working out which complete test is redundant and can be removed. They also do not offer a set of quality criteria that can be used to guarantee whole test suite quality when making decisions regarding which duplicate BDD scenarios to remove. Also, all of the options for refactoring duplicates in BDD feature suites, as proposed in the work of Suan [33] can only be applied on exact matches of texts in BDD scenarios. They cannot be applied on the context of removing semantically equivalent but textually different BDD scenarios from a feature suite.

More specifically, deciding which of the semantically equivalent but textually different BDD scenarios to remove can be a daunting task. It can require consideration of various aspects of a BDD specification, including the need to preserve the overall quality of a feature suite after the removal of a duplicate scenario. To the best of our knowledge, existing techniques do not offer that support, and ours is the first attempt to use quality metrics of BDD suites to advise human engineers on what duplicate scenarios should be removed.

2.7 Summary

Figure 2.2 summarises the literature we surveyed. As it can be seen from the sizes of the cycles in Figure 2.2, while there is a relatively large amount of work on duplicate detection and removal in software artefacts other than BDD, a relatively small amount of work has focused on analysing and improving the quality of BDD specifications. Further, of all the studies we found on analysing and improving the quality of BDD specifications, only two studies [33, 139] attempted to detect and remove duplication in BDD specifications. We next present the summary of answers for the different specific questions that were of interest during literature survey.

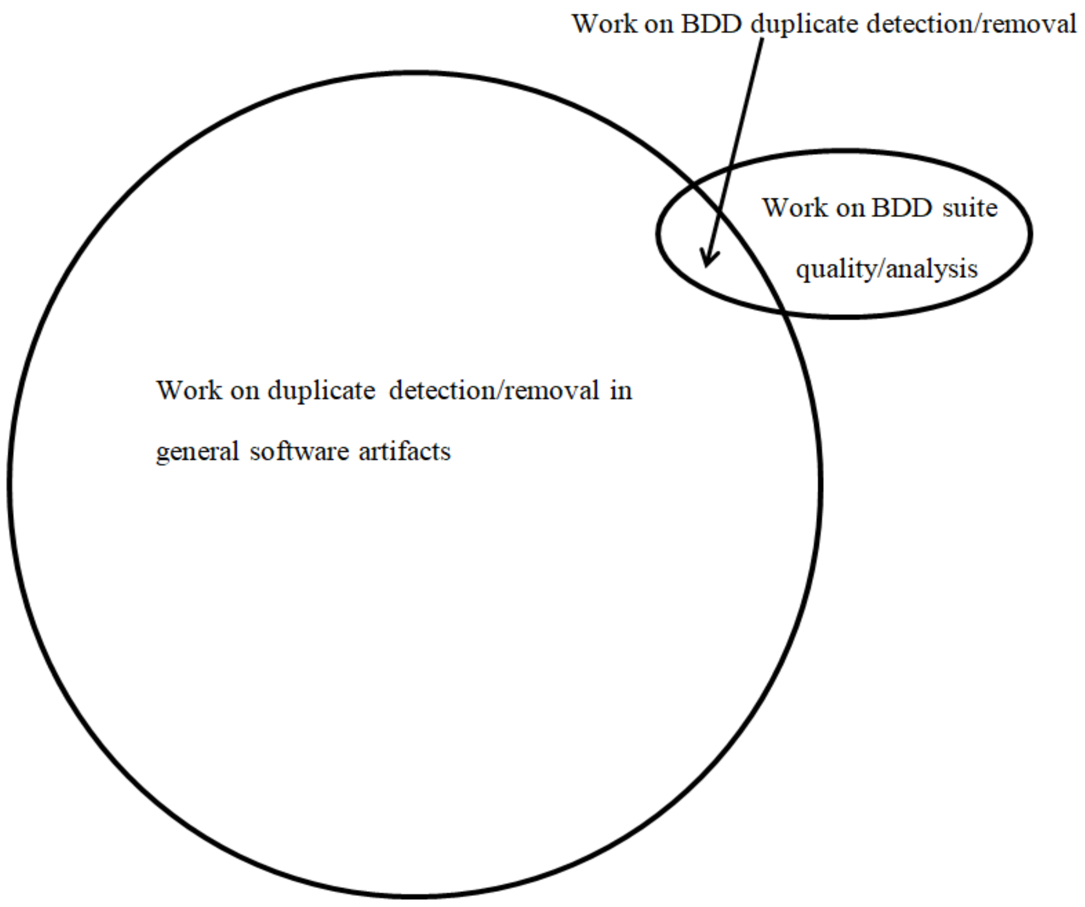

Work on BDD duplicate detection/removal
Work on BDD suite quality/analysis
Work on duplicate detection/removal in general software artifacts

Figure 2.2: Classification of the literature we surveyed

We have started by providing evidence that duplication exists in software systems, the reasons why duplication gets introduced into software systems, and the consequences of the presence of duplication in software systems. We found that duplication can be introduced deliberately into a software system, to serve specific design, development and maintenance purposes; but it can also be introduced accidentally due to various project and team factors. We observed that, irrespective of the motivation, the consequences of duplication can be serious, negatively impacting the maintenance of a software.

We then presented existing duplicate detection approaches and techniques, and discussed their limitations on the problem of detecting semantic duplication in BDD specifications. We noted that specialised techniques, that take into account the typical structure of a BDD project and how various artifacts are presented in a project, are required to effectively detect semantically equivalent BDD scenarios. We also reviewed the existing literature on duplicate removal in software systems, to determine if existing techniques can be used to advise which BDD scenario should be removed, after semantically equivalent BDD scenarios have been detected. We conclude that the decision on which duplicate BDD scenarios to remove should take into account various aspects of the BDD suite, but existing duplicate removal techniques do not offer that capability.

Finally, we surveyed the literature on analysis and improvement of BDD specifications, to determine, among other things, if the problem of detecting and removing semantic duplication in BDD suites has been studied. We found that existing research on duplicate detection in BDD has focused on detecting textual similarity in BDD feature suites and, because of that, semantically equivalent but textually dissimilar duplicates cannot be detected by the techniques proposed in the literature. Specifically, we found no study focusing on the detection and removal of semantically equivalent but (possibly) texually different BDD scenarios. We conclude that there is lack of techniques to detect and remove semantically equivalent BDD scenarios, to support the work of BDD practitioners.

Chapter 3

Challenges and Opportunities for Maintaining BDD Suites

3.1 Introduction

As the BDD technique has entered its second decade of use, a considerable body of experience has been built up by practitioners, and lessons have been learnt about both the strengths and the challenges involved in its practical application. Anecdotal evidence from the software engineers we have worked with suggest that the maintenance challenges, in particular, can be severe, and are leading some teams to drop the technique and to return to technology-facing automated testing to play the role of their BDD specifications. However, to the best of our knowledge, no empirical studies have been undertaken by the academic community to capture these lessons and to understand how research might be able to address some of the problems encountered by users of large BDD specifications over the long term.

In the present chapter, we want to understand if duplicates are a problem for maintainers of BDD specifications, and we want to discover whether that is the case before we spend much effort on solving it. To that end, we set out to answer the following research questions (RQs):

RQ1: Is BDD in a considerable active use in industry at present?

RQ2: What are the perceived benefits and challenges involved in using BDD?

RQ3: What are the maintenance challenges reported amongst the issues raised by users (and former users) of BDD?

RQ4: To what extent is the discovery and management of duplicates in BDD specifications seen as an unsolved problem by practitioners, and what techniques are being employed to deal with it?

To find answers to these questions, we surveyed 75 BDD practitioners from 26 countries across the world. This chapter presents the opinions of BDD practitioners regarding duplication and other challenges of maintaining BDD specifications. We also present research opportunities to address the identified BDD suite maintenance challenges.

The rest of this chapter is structured as follows: section 3.2 presents the design of the study, respondents, and the data analysis approach; section 3.3 presents the results of the study; section 3.4 discusses the significance of the results, providing answers to the research questions; section 3.5 enunciates the research opportunities, derived from the results of the survey; section 3.6 discusses the threats to the validity of the survey results, and the mitigation strategies; and section 3.7 summarises the chapter.

3.2 Study Design

This section presents the survey design, the process we used to recruit respondents, and the approach we used to analyse survey data.

3.2.1 Survey Design

To attract more responses and yet ensure that aspects of interest are well captured, we designed a short web survey with 18 questions. The first 14 questions covered the extent of use of BDD in different types of organisation, the benefits and challenges involved in using BDD, as well as the presence of duplication in respondents' BDD specifications and how it is dealt with. The other 4 questions were about respondents' demographics respectively: Name, Email, Job Title, and Organisation Name. Refer to Appendix A.1 for the full list of questions.

Whereas Q1-Q5, Q9, and Q11-Q13 were single choice questions, Q6-Q8 and Q10 were multiple choice questions. An "other(s)" option on all single choice and multiple choice questions allowed respondents to report other important information that might not have been well covered in the choices we gave. All demographics questions accepted free text responses, and were optional.

To avoid redundant and useless questions on the survey, and with an aim of producing a short survey that would attract more responses, the survey was designed in such a way that each non-demographic question would be used to solicit evidence for answering at least one of the four research questions posed in section 3.1. Figure 3.1 provides the mappings between the research questions (RQs) in section 3.1 and the non-demographic questions (Qs) on the survey in Appendix A.1. Specifically, responses to Q1-Q4 and Q6 were used as evidence for answering RQ1. (Though Q1 in the survey is a demographic-like question, the organisation types obtained through responses to Q1 were used alongside responses to other questions to understand the activeness of BDD use in different types of organisation.)

Moreover, responses to Q5, Q7-Q8 as well as some of the responses to Q14 were used as evidence for answering RQ2. Responses to Q10-Q13 were used as evidence for answering RQ4. To answer RQ3, both the free text responses obtained through the "other" option in Q8 and the free text responses to Q14 were analysed for evidence about the prominence of maintenance challenges amongst the issues reported by BDD practitioners. Q9 was used to gather evidence about typical sizes of BDD specifications that are likely to pose the maintenance challenges that we focus on in RQ3 and RQ4.

The survey was reviewed by a senior academic from our school who has experience in doing survey research. It was designed and deployed using SelectSurvey.NET[1], an instance of which is available for use on our university servers. Our respondents took an average of 10 minutes to complete the survey, and we received responses over the period of more than 2 and a half months from July 2017.

3.2.2 Recruitment of Respondents

Developers of BDD projects and other members of industry agile teams who had ever used BDD, or were using BDD at the time of our survey, were our targets. The survey was distributed through a convenience sample of online agile discussion groups and

[1]http://selectsurvey.net/

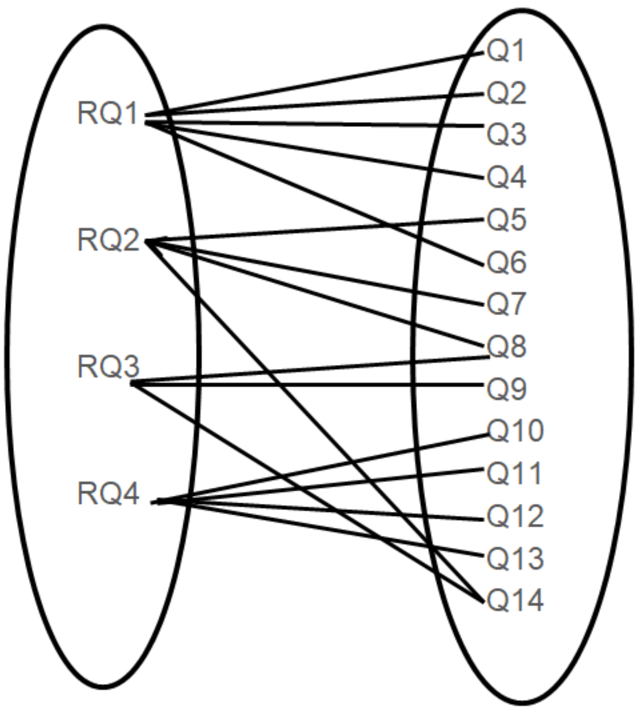


Figure 3.1: Mapping between the research questions in section 3.1 and the questions which are not about respondents' demographics on the survey in Appendix A.1

personal emails. Though it reduces generalizability of findings, convenience sampling is appropriate when random sampling is practically impossible [149, 150]. The survey was accessed and completed through a web link. We also encouraged respondents to pass the survey on to others, and so some of our respondents might have been recruited through snowballing. A similar method of recruiting survey respondents was used by Kochhar *et al.* [151], and Witschey *et al.* [150].

We began by posting the survey on on-line communities where BDD topics are discussed. The following Google groups were targeted: Behaviour Driven Development Google group, Cucumber Google group, and BDD Security Google group. After learning about the survey through one of these groups, the Editor in Chief of BDD Addict [152], a monthly newsletter about BDD, included the survey in the July 2017 issue, in order to reach a wider BDD community. The survey was also shared with the following twitter communities: Agile Alliance, Agile Connection, Agile For All, Scrum Alliance, Master Scrum, RSpec—BDD for Ruby, my twitter account and the twitter account for one of my supervisors.

We further identified email addresses of contributors to BDD projects on GitHub.com, and sent them personalized requests to complete our survey. Relevant projects were identified using the keywords “Behaviour Driven Development”, “Behavior Driven Development”, and “BDD”; we extracted the public email addresses of contributors from the resulting projects, up to and including the 5th page (10 results per page). (We selected this limit after manual examination of the usefulness of email addresses on later pages for a sample of projects. Pages 6 onward consisted of either trivial, demonstration, or empty project repositories, and thus owners of and contributors to these projects were considered to be less appropriate for this particular survey, and therefore their emails were deemed less useful.) We also searched for projects with keywords based on the names of the tools mentioned in the survey: namely, *Cucumber*, *FitNesse*, *JBehave*, *Concordion*, *Spock*, *easyb*, and *Specflow*. In total, 716 email addresses were identified and contacted about the survey.

3.2.3 Survey Respondents and Data Analysis Approach

This section presents the demographics of survey respondents, the different ways we use to present evidence required to answer RQ1, RQ2 and RQ4, and the approach we use to analyse free text responses required to answer RQ3.

Analysis of Data about Respondents

Of the 566 people who viewed the survey, 82 began to complete it out of whom 75 submitted responses to the main questions (questions not focusing on respondents' demographics, i.e., Q2-Q14 in Appendix A.1). We hereafter refer to the 75 respondents as R1 to R75. Further, 11 out of the 13 (84.6%) main questions were completed by all the 75 respondents. We used *IP Address Geographical Location Finder*[2] to approximate the geographical locations of respondents, and Table 3.1 shows the distribution of respondents by continent. A similar approach to identifying the locations of survey respondents was used by Garousi and Zhi whereby the analysis of IP addresses was used to determine the countries from which survey responses came from [8].

The types of organizations the respondents worked for were distributed as shown in Table 3.2. 44 participants gave the role they held in their organization, and most were in senior positions. However, the remaining 31 preferred not to state their roles, probably because we explicitly stated that identifying information was optional.

[2]http://www.ipfingerprints.com/geolocation.php

Continent	**No.**	**%**
Europe	49	65.3
North America	12	16.0
Asia	4	5.3
South America	5	6.7
Australia	3	4.0
Africa	1	1.3
Zealandia	1	1.3
Total	**75**	**100.0**

Table 3.1: Survey on BDD maintenance challenges: Distribution of respondents by continent

Organisation Type	**No.**	**%**
Public	26	35
Private	47	63
Sole Trader	1	1
Did not say	1	1
Total	**75**	**100**

Table 3.2: Survey on BDD maintenance challenges: Distribution of respondents' organisations

Table 3.3 shows the distribution of job roles for the respondents.

Analysis of Responses to Non-demographic Survey Questions

We received a total of 82 responses. We removed 7 responses in which respondents had completed only demographics data, leaving 75 valid responses. We used charts, tables, and summary of "other" comments on specific survey questions to present the responses.

To be able to identify the maintenance challenges amongst the issues reported by BDD practitioners, as required in RQ3, we used the thematic analysis guidelines by Braun and Clarke [153] to analyse free text data obtained through the "other" option in Q8

Role	**No.**	**%**
Software Engineers/Architects	20	26.7
Quality Assurance Engineers/Business Analysts	4	5.3
Team Lead/DevOps Tech Lead	10	13.3
Consultant	3	4.0
Chief Executive Officer (CEO)	2	2.7
Chief Technology Officer (CTO)	4	5.3
Researcher	1	1.3
Did not say	31	41.3
Total	**75**	**100.0**

Table 3.3: Survey on BDD maintenance challenges: Distribution of job roles of respondents

on the survey and all free text responses to Q14 on the survey. The thematic analysis approach presented by Braun and Clarke involves six phases [153, 154]:

1. **Familiarisation with the data:** This involves actively passing through the data, trying to make sense of them and to get an initial impression about the data.

2. **Producing initial codes:** At this stage, initial codes are produced, after an initial pass through in phase 1. *"Coding reduces lots of data into small chunks of meaning"* (Maguire and Delahunt [154], p.5).

3. **Looking for themes:** During this phase, related codes are sorted into possible themes. *"a theme is a pattern that captures something significant or interesting about the data and/or research question"* (Maguire and Delahunt [154], p.6). Extracts of data from which the codes were produced are also placed alongside the themes.

4. **Review of themes:** This phase involves the review of potential themes identified in phase 3, to determine, among other things, how well the identified themes are properly supported by the data extracts. It can involve merging of themes, creating new themes, or relocating codes and data extracts to other themes.

5. **Definition and naming of themes:** This involves further definition and refinement of themes identified through the previous phases. Both the core of each theme as well as facets of the dataset covered by each theme are identified. It also includes the analysis of the narrative of each theme, to determine how well each theme fits into the broader narrative to be told from the dataset, and how

well the broader narrative from the dataset fits into the research question.

6. **Report writing:** This involves reporting the outcomes of a thematic analysis, using a coherent and consistent story, and with the support of data extracts that help to strengthen the narrative.

There were 17 free text responses obtained through the "other" option in Q8, and 17 free text responses obtained through Q14. It is these 34 free text responses that were subjected to a thematic analysis. In particular, we conducted the *theoretical thematic analysis* [154] in which data analysis is guided by the research question. In our case, the research question was about the challenges involved in using BDD, from a practitioner's perspective.

Data coding started after we had gone through the data to become familiar with it and to draw some initial impressions about it. We coded everything in the text that related to BDD challenges. We used open coding—we had no predetermined codes. Table A.1 shows the codes we assigned to the various challenges reported by BDD practitioners who responded to our survey. After coding, we grouped related codes together to form the list of initial themes (see Table A.2). Thereafter, the initial themes and their associated codes were iteratively refined to produce the final list of themes, which we present as part of results (see Section 3.3.2 and Section 3.3.3). We considered a theme to be important if it had more than one code under it.

3.2.4 Ethical Considerations

At the time we undertook the survey, our University did not require ethical approval for surveys requesting only professional opinions from participants. We took a conservative approach, that went beyond these requirements, and designed the survey to ensure anonymity of participants and proper use of data collected through the survey. In particular, we kept any identifying information optional, and clearly stated this at the beginning of the survey, and in all survey completion invitations. We also stated clearly, on the survey and in the survey completion requests about the purpose of our research and how we intended to use survey data. In addition, participation in the survey was completely optional.

3.3 Results

This section presents the extent of BDD use amongst the BDD practitioners who responded to our survey, the perceived importance and benefits of BDD amongst respondents, and the challenges faced by respondents in using the BDD technique and maintaining BDD specifications. Here, we present the results for the different survey questions (Qs) as we obtained them, with the significance of results being discussed in Section 3.4.

3.3.1 RQ1. Extent of Active Use of BDD

Extent of BDD Use in Various Types of Organizations (Q1, Q2, & Q3)

Figure 3.2 shows the extent of BDD use in the organisations that respondents worked for. By comparing the extent of BDD use in different types of organisation, we found that, among the organisations the survey respondents worked for, BDD is used more in private organisations than in other types of organisation (see Figure 3.3). Figure 3.4 summarises responses as to whether BDD is a mandatory or optional tool for organisations.

Tools Used by Industry Teams to Support BDD and ATDD (Q6)

Respondents use the tools in Figure 3.5.

Plans to Use BDD in the Future (Q4)

Almost half of the respondents said that their organisations will use BDD as an optional tool on some projects in the future, while more than a quarter of the respondents said that it will be used as a key tool on all projects. Figure 3.6 summarizes the responses on planned future BDD use.

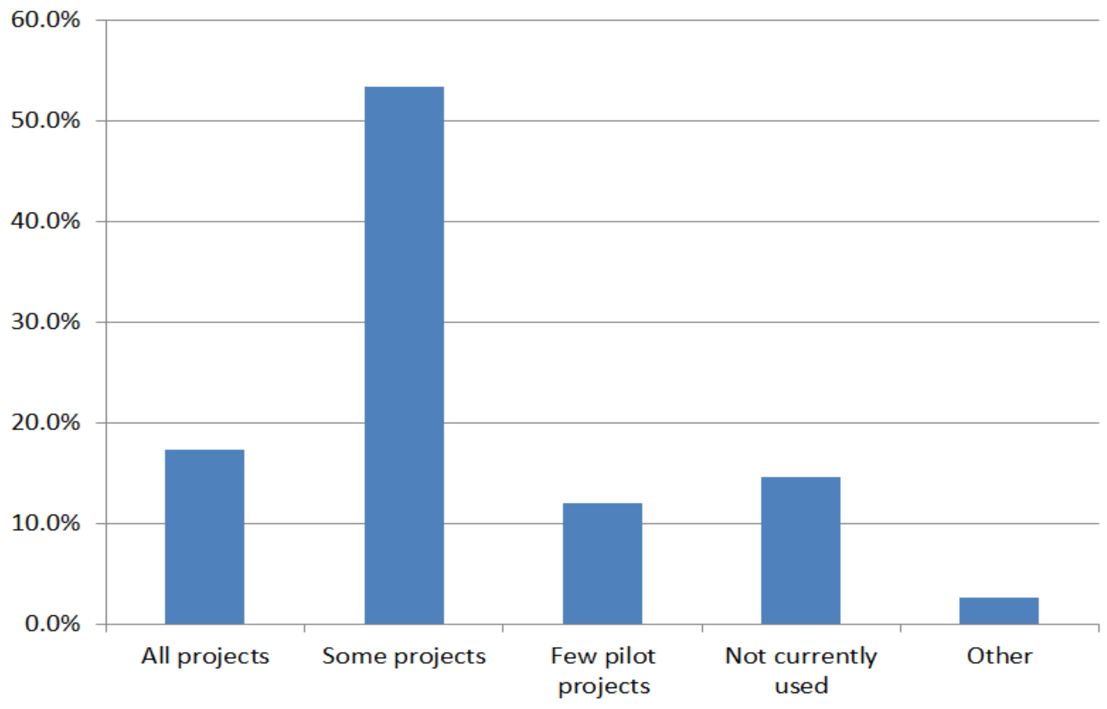


Figure 3.2: Extent of BDD use

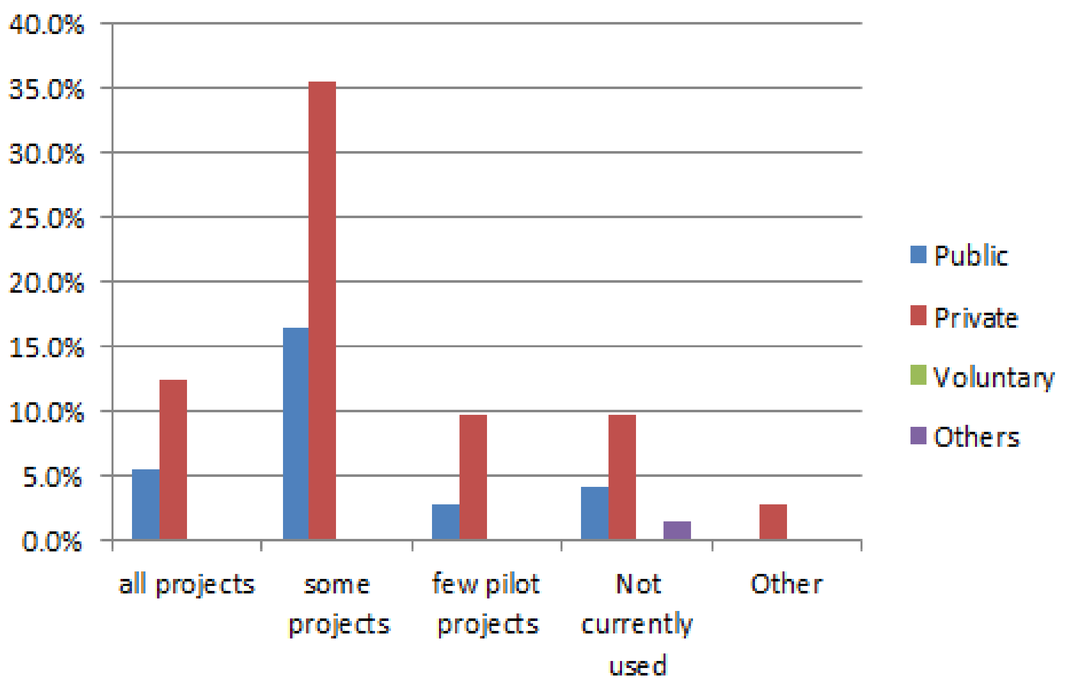


Figure 3.3: Extent of BDD use by type of organisation

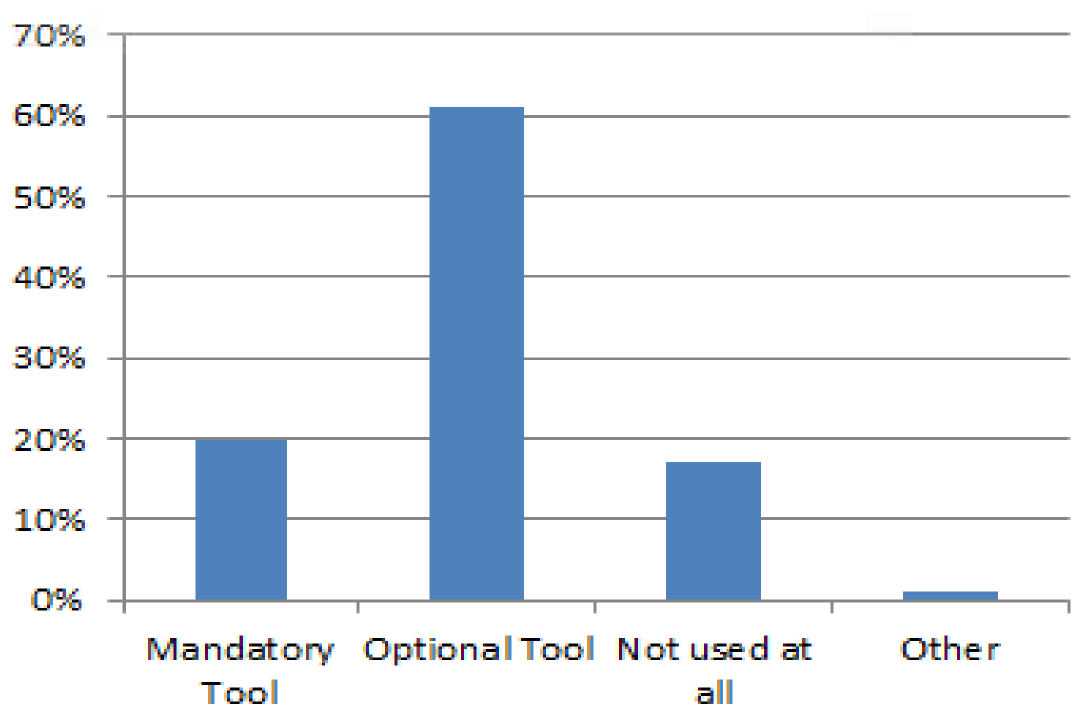


Figure 3.4: Extent to which use of BDD is mandated by organisations

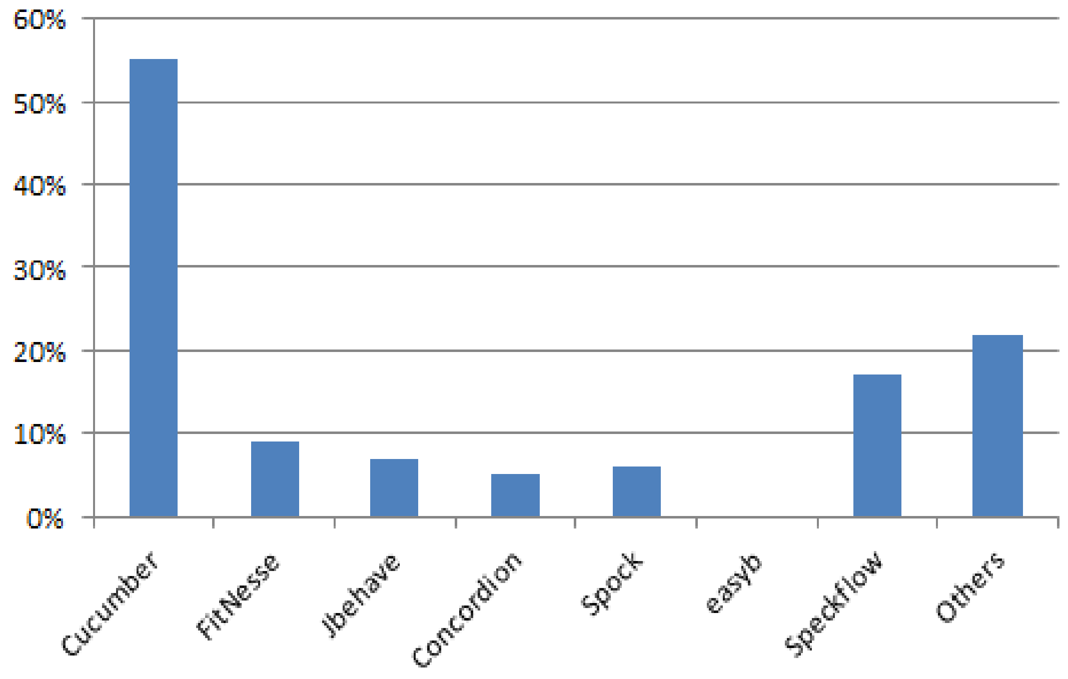


Figure 3.5: BDD Tools used by respondents

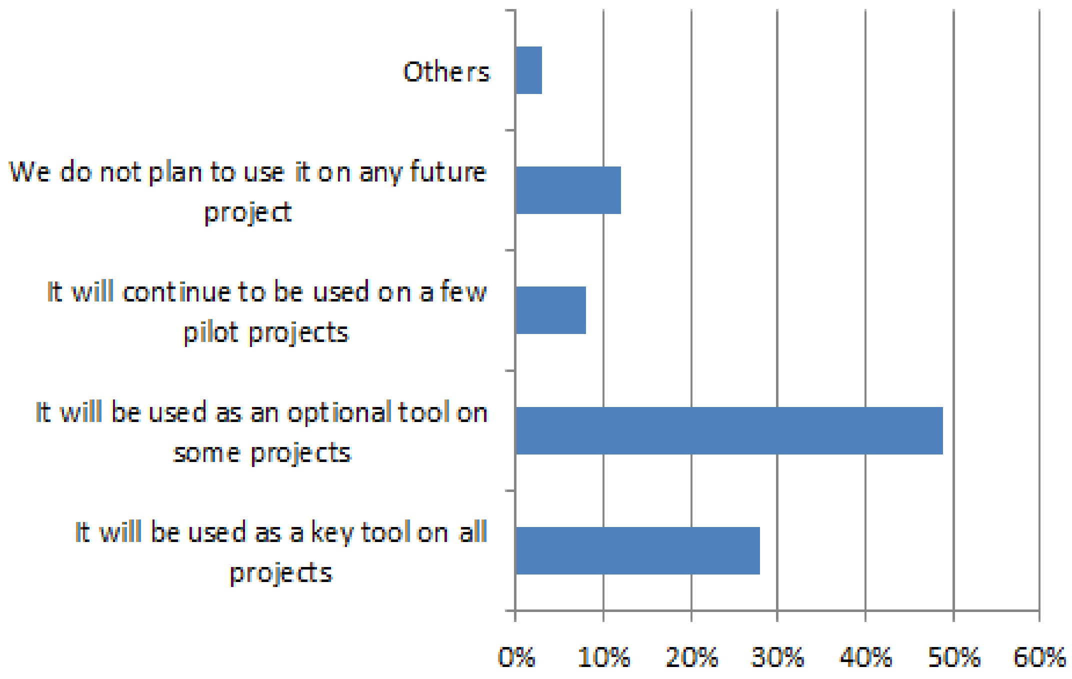


Figure 3.6: Plans by organisations to use BDD in the future

3.3.2 RQ2. Perceived Benefits and Challenges Involved in Using BDD

Perceived Importance and Benefits of BDD (Q5 & Q7)

Fig. 3.7 presents the perceived importance of BDD use by the respondents. The views given under "other" were:

- *"Personally I find it very important, my clients though have different opinions. Usually it requires a certain collaboration within the organization which is hard to establish. It is not the tool that is hard to use, but more the people to get into this work flow."*
- *"BDD enables teams to write standard tests that are more expressive."*

Respondents' opinions on the benefits of BDD are presented in Table 3.4. As the results show, respondents value the communication aspects of BDD, but also the benefits to developers in gaining early warning signals of problems.

Under "other", respondents listed the following additional benefits:

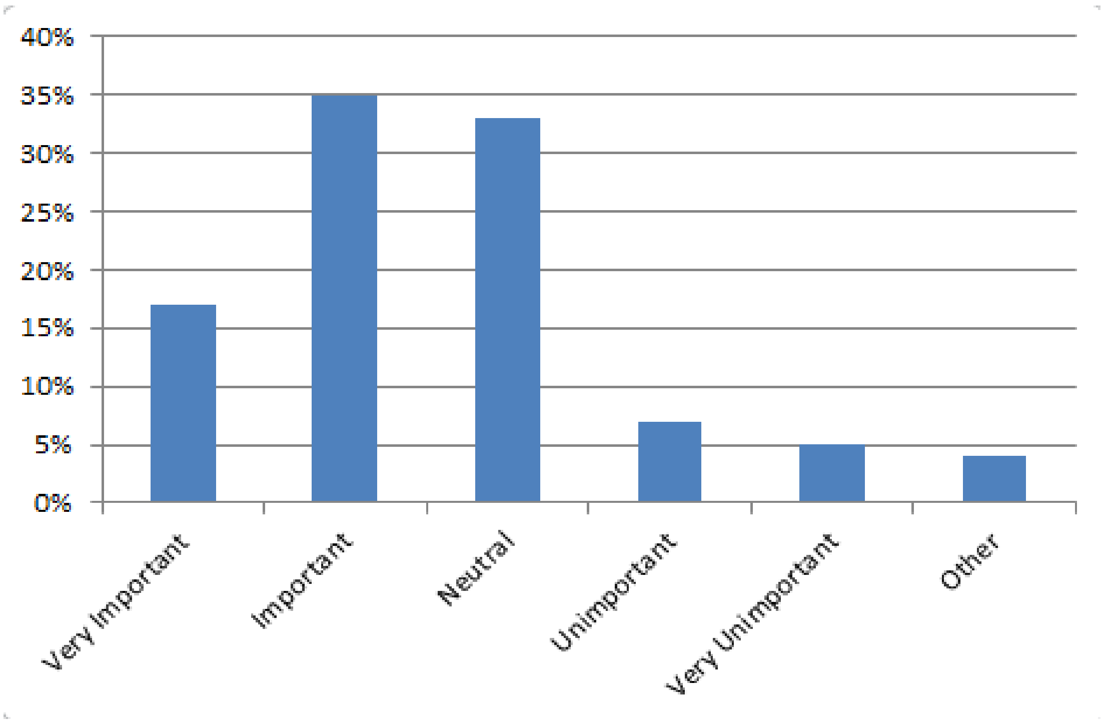


Figure 3.7: Perceived importance of BDD use

- BDD offers an improved way of documenting the software and the associated code:
  - *"Living documentation that evolves with the system over time"*
  - *"Documentation is a working code"*
- Simplifies and enriches software testing activities:
  - *"Helps QA team to write tests without code implementation details"*
  - *"Make possible fullstack tests, differently from unit tests."*
  - *"Reusable, finite set of steps used by test developers"*
- Improves software designs by facilitating domain knowledge capture:
  - *"primarily a design tool $\rightarrow$ it enables us to gain clarity about the domains at hand, especially at the seams"*

Challenges Faced by BDD Practitioners (Q8 & Q14)

Because of the strong emphasis in BDD on collaboration among all project stakeholders, there is a strong interaction between social and technical challenges involved in using the BDD technique. Because of this, we next present the results relating to both kinds of challenge together. However, depending on the context, readers can attempt to draw the line between social and technical BDD challenges.

The challenges faced by BDD practitioners, according to the respondents, are also given in Table 3.4. Respondents thought that the most challenging part of BDD is that it changes the usual approach to team software development. Other challenges reported under the“other” option in Q8 as well as under Q14 were the following (number of codes for each challenge are presented in brackets):

- The collaboration among stakeholders, as required in proper BDD workflows,

Benefits of BDD	**Rate (%)**
Software specifications are expressed in domain-specific terms, and thus can be easily understood by end users	67
Improves communication between various project stakeholders	61
Specifications can be executed to confirm correctness or reveal problematic software behaviour(s)	52
Code intention can be easily understood by maintenance developers	50
Could produce better APIs since it emphasizes writing testable code	28
Attention is paid to validation and proper handling of data	24
Other	7
Challenges of BDD	**Rate (%)**
Its use changes the team's traditional approach to software development, and that can be challenging	51
Its benefits are hard to quantify	35
It involves a steep learning curve	28
It can lower team productivity	20
Other	21

Table 3.4: Benefits and challenges of using BDD

can be difficult to establish, or sometimes ignored, leading to future problems or failures (9):

  – *"Main issue when applying BDD, it to find time to do the three amigos workshop, it is not a tool issue but more a people one"* (R67, Crafter & CTO)
  – *"Needing to involve Business and final users"* (R2)
  – *"It's a simple concept but can be hard to get right. Many people make the assumption it's about test automation and try to use like a scripting tool and the project ends in failure"* (R12, CEO)
  – *"Dsnger (Danger) of confusing the mechanics (automation, written specifications) with the intention (knowledge sharing, structured conversations, discovery of edge cases), focusing too much on the former."* (R24, Agile Principle Consultant)

- Writing or reading tests tends to be counter-intuitive for non-developer stakeholders (6):
  – *"Make other non-developers read tests. So far I have used BDD for couple of years and even though idea behind it [is] good, people who are not involved in testing are also not interested in test cases no matter how easy-to-read they are."* (R20)
  – *"it does not succeed at being legible to colleagues outside of software engineering departments"* (R19, Software Engineer)
  – *"BDD practices "by the book" often force domain experts to waste their time and insights trying to think like developers/testers, instead of expressing their needs. Real-world examples often have overwhelming details"* (R72, Principal Software Engineer)
- BDD specifications can be hard to comprehend, execute and maintain (10):
  – *"...Textual specs are too expensive to maintain long-term"* (R49, Senior software engineer)
  – *"BDD is often associated with slow suites. The difficulty of managing duplication is proportional to that slowness. Therefore, as BDD scales, in my opinion it is crucial to find ways to run slow scenarios fast, either by*

  reducing their scope, or by running them against multiple configurations of the system covered by the scenarios." (R28, Developer)

  - *"Some developers don't like the duplication that an (can) be created with having BDD separate to unit tests. BDD can also get out of hand and become far too technical and indecipherable by users"* (R73, Senior Test Engineer)
  - *"...tests were very brittle and manual QA types had limited ability to investigate."* (R18, Senior Software Engineer)
  - *"BDD add unnecessary layer of maintaining specification and make them still readable with clean code."* (R20)
  - *"All the usual challenges in getting automated testing running and maintained"* (R73, Senior Test Engineer)

- Scarcity of skills, training and coaching services about the BDD workflow (6):
  - *"requires design skills often absent or not valued"* (R65)
  - *"Its hard to find someone who really understand what should be tested by BDD therefore a bunch of developer has negative experience about it. Probably there is no a comprehensive material on the internet that can explain every aspect of BDD."* (R34, CTO)
  - *"As with other kinds of testing, the best way to learn is from somebody who has experience. Thus just by downloading a framework, reading a bit and trying, one can produce tests which value is disputable."* (R23, CEO)
  - *"there are very few sources about the structuring of *information* and the conveyance of semantic intent..."* (R65)

- Setup costs, availability and difficulty in using tools (8):
  - *"Productivity is initially (first 6 months) lowered. Beyond that productivity is increased. Every project has a greater setup cost, say 1-3 days to put BDD tooling in place. Hence it is not worth while for trivial projects..."* (R60, CTO)
  - *"...Writing the DSL is overhead on engineering team..."* (R18, Senior Software Engineer)

- *"Poor tooling"* (R45)
- *"...Using bdd tools that enable writing tests in human language end up being an overhead"* (R75)

3.3.3 RQ3. Challenges of Maintaining BDD Specifications

Based on the comprehensive list of challenges involved in using BDD, as presented in Section 3.3.2, this section presents the summary of challenges faced by respondents concerning the maintenance of BDD specifications. Thus, the maintenance challenges presented here are a subset of a comprehensive list of challenges faced by respondents in using BDD.

Size of BDD Suites (Q9)

As noted in Q9 in the survey, the typical sizes of BDD specifications that respondents work with are important in providing the context for the reported maintenance challenges. Clearly, the maintenance challenges reported are likely to be of less significance if typical suites contain numbers of scenarios (i.e., examples) that can be easily managed by hand. Fig. 3.8 shows what the respondents reported as the typical size of the BDD suites they work with. The majority are of the order of 1000 scenarios or less. While a significant minority are considerably large to make manual individual inspection of all scenarios a costly task, it is our view that if manual techniques are used, even a thousand scenarios (or a fraction of them) can be prohibitively expensive to comprehend, maintain and extend.

Maintenance Challenges

We now present a summary of BDD maintenance challenges that were reported by respondents.

As can be noted from the responses describing the general challenges of BDD, respondents mentioned that BDD feature suites suffer from the same kinds of maintenance challenge associated with any other forms of automated testing. Specifically, the maintenance challenges as presented earlier from the survey can be summarised as:

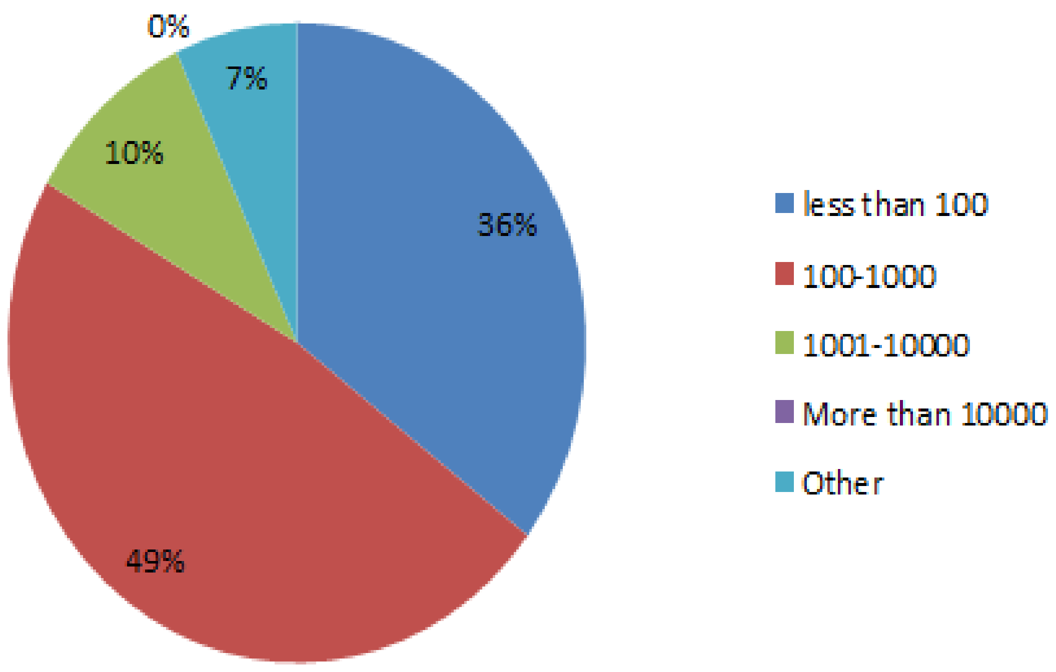


Figure 3.8: Number of scenarios in industry BDD projects (Q9)

- BDD feature suites can be hard to comprehend.
- It can be hard to locate sources of faults, especially in large BDD suites.
- It can be difficult to change BDD feature suites for the purpose of fault correction, accommodating new requirements, or adapting them to new environment.
- It can be hard to make slow BDD feature suites run faster.
- The need to maintain BDD feature suites in addition to unit tests.
- Coping with the possible complexity of BDD tools.
- Duplication detection in BDD specifications.
- Duplication management in BDD specifications.

3.3.4 RQ4. Duplication in BDD Suites

We now present the maintenance challenges caused by the presence of duplication in BDD feature suites, the extent to which duplication is present in respondents' BDD feature suites, and the current state of practice in the detection and management of duplication in BDD feature suites, as reported by the survey respondents.

Problems of Duplication (Q10)

61% of the respondents held the view that the presence of duplication in BDD specifications can cause the specifications to become difficult to extend and change (leading potentially to frozen functionality). Moreover, nearly half of the respondents (49%) said that the presence of duplication in BDD specifications can cause execution of BDD suites to take longer to complete than necessary, and 43% thought that duplication can make it difficult to comprehend BDD specifications. Under "other" (7%), the following problems of duplication in BDD suites were reported:

- The process of duplication detection and management can change the desired software behaviour:
  - *"Over refactoring features and scenarios to avoid duplication causes the requirements and their understanding to change from what the Product Owner wants."*
- Difficulty in comprehension and execution of specifications:
  - *"Contradicting specifications, if the duplication is not a result of the same team/individual working on it."*
  - *"Duplication in specs is usually a sign of incompletely or badly 'factored' behaviours, which can lead to overly complicated specs and difficult to set up system state."*
- Necessitates changes in several places in the suite during maintenance and evolution:
  - *"Changes required to be done in more than one place. I miss some 'include' keyword."*
- It is hard to use existing duplicate detection and management tools to detect and manage duplicates in specifications expressed in a natural language:
  - *"if the statements are in English prose basic refactoring tools / copy paste detection / renaming are difficult to catch and maintain."*

- BDD tests are end-to-end tests that are usually strongly connected to their unit tests, and that makes the process of detecting duplicate BDD tests difficult:
  - *"It's hard to detect duplication between BDD specs and unit-tests."*
- Difficulty in modelling how the scenarios are executed, and the scenarios can be very slow and brittle:
  - *"...criteria can hold at one level and cascade down - difficult to model *how* the scenarios are executed can be very slow and brittle (e.g. web tests) - hexagonal architecture please"*

Presence and extent of duplication (Q11 & Q12)

Figure 3.9 shows the proportions of respondents' BDD projects with duplication, Figure 3.10 shows the extent to which duplication is present in the respondents' BDD projects, and Figure 3.11 cross-compares the presence and extent of duplication in the respondents' BDD specifications.

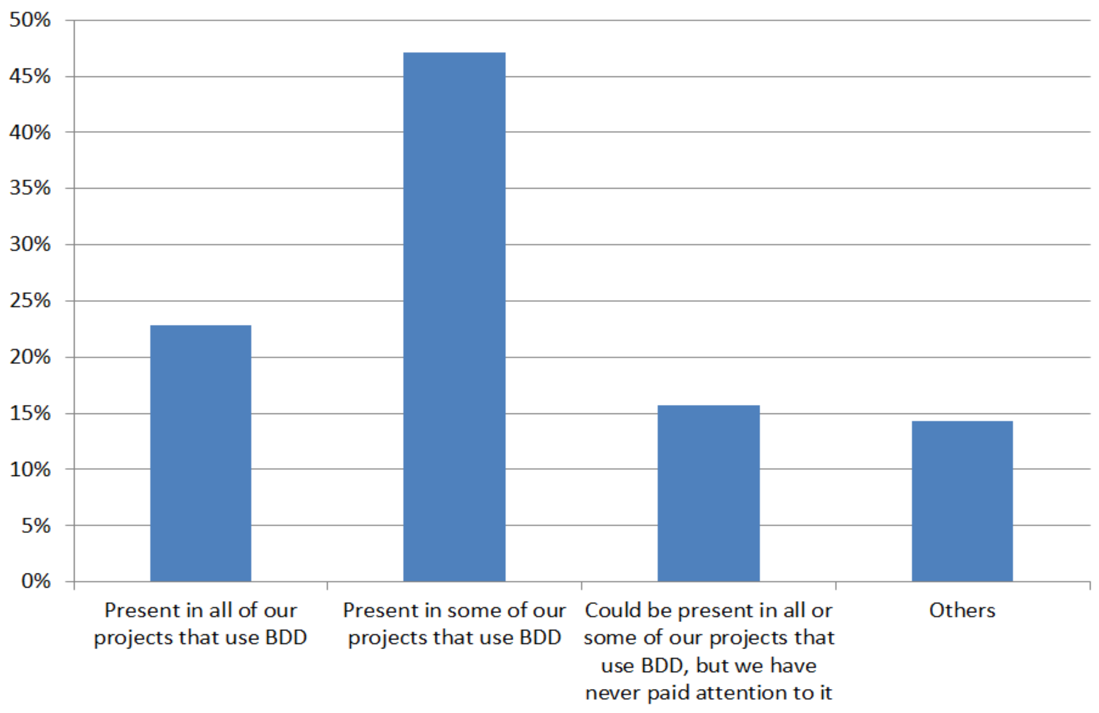


Figure 3.9: Presence of duplication in the respondents' BDD specifications

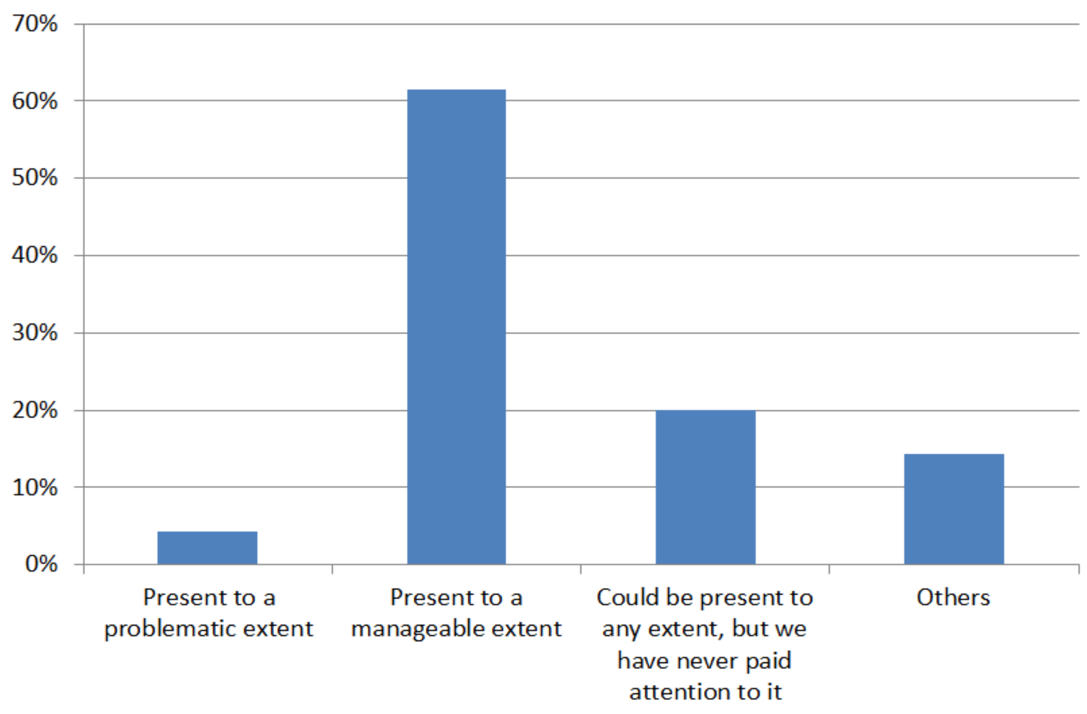


Figure 3.10: Extent of duplication in the respondents' BDD specifications

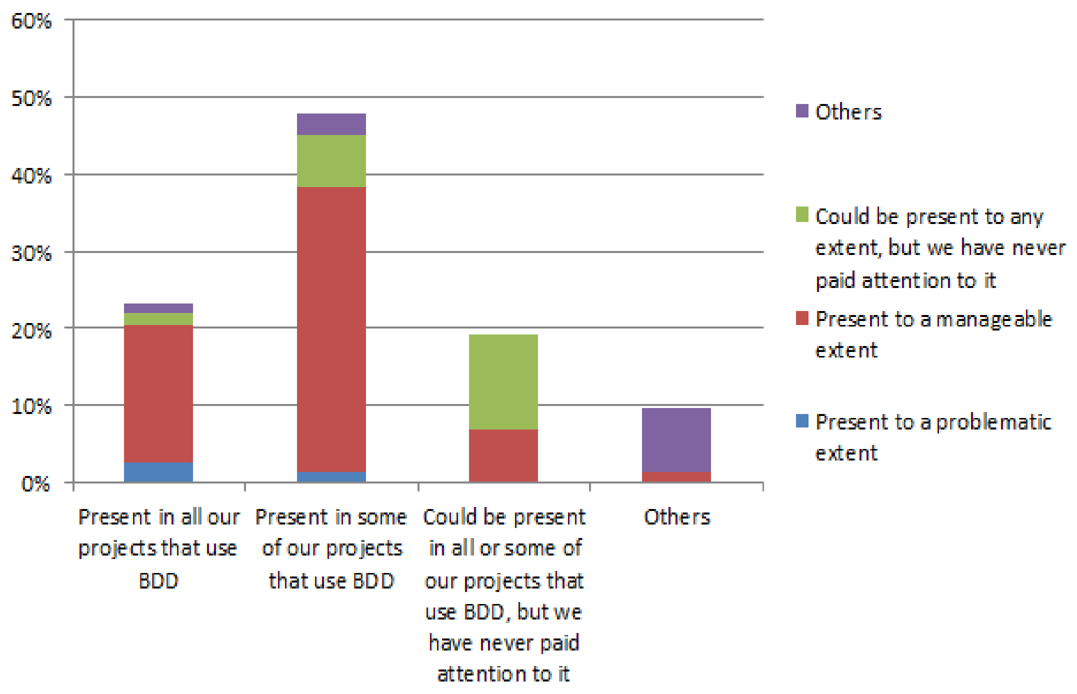


Figure 3.11: Presence and extent of duplication in the respondents' BDD specifications

Detection and Management of Duplication in BDD Specifications (Q13)

We now present the current state of practice in detecting and managing duplication in BDD specifications.

Figure 3.12 summarizes the different methods that respondents use to detect and manage duplication in their BDD specifications. These results suggest that duplication is manageable but costly for respondents. Most respondents who were concerned with duplication (40% of respondents) reported to have been detecting and managing it manually. Also, there was a significant proportion (17%) of respondents who had decided to live with duplication, given the complexity of its detection and management process. So, while about a quarter of respondents (26%) did not regard duplication as a problem, more than a half (57%) of the respondents were concerned with duplication detection and management. Moreover, Figure 3.13 relates the extent of duplication, suite size, and method of duplication detection reported by respondents.

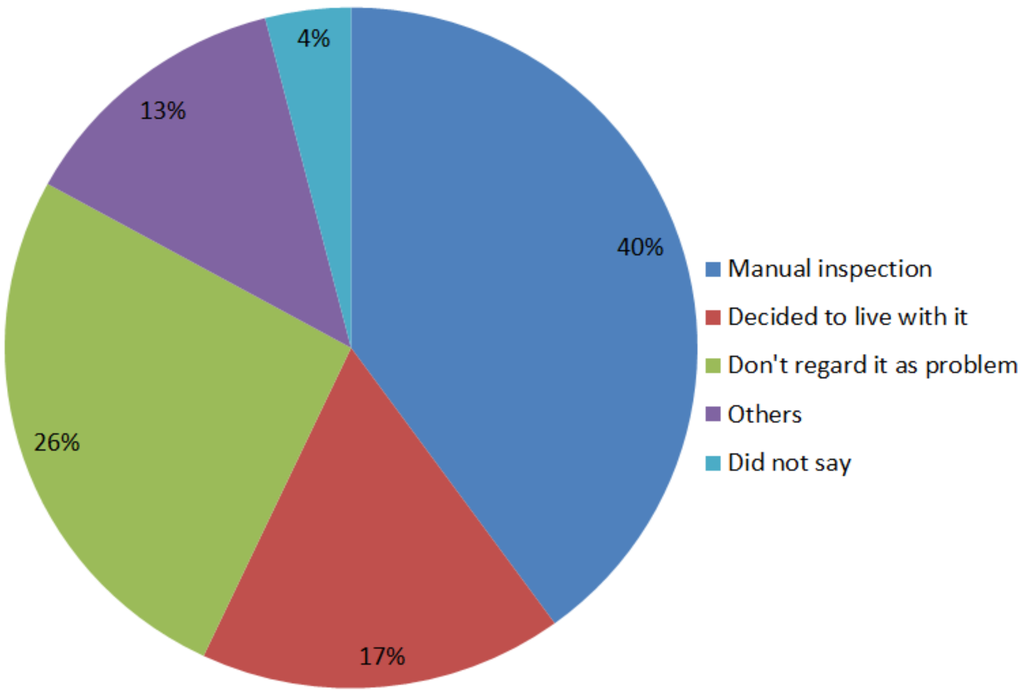


Figure 3.12: Duplicate detection and management methods

Under "other", some respondents had the following additional thoughts on how they approach duplication detection and management in their (respondents) BDD specifications:

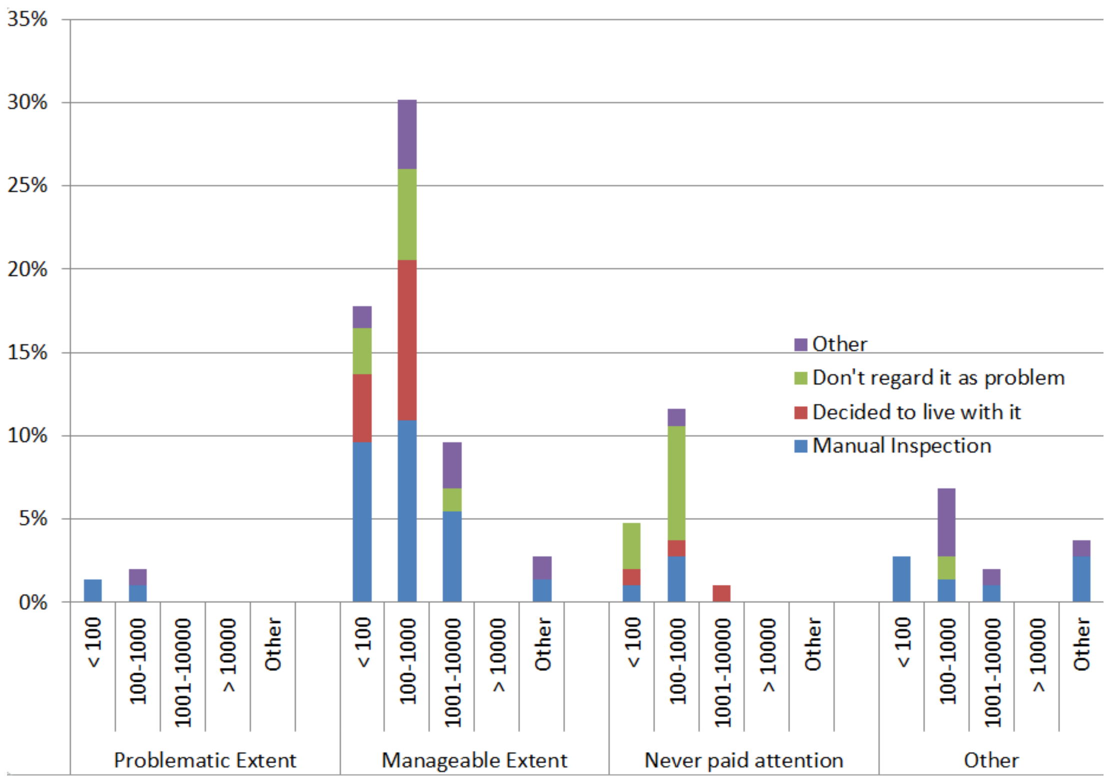


Figure 3.13: Extent of duplication, size of BDD specs, and duplication detection method

- *"We are looking at ways to automate at least part of the process of finding duplicates"*
- *"Treat the test code much like the production code. Refactor frequently to control duplication and make test intentions clear"*
- *"Pay attention to SRP during or after collaborative specification."*
- *"We organise the specifications specifically to prevent this (duplication). It would be one of the worst things to happen."*
- *"Using jbehave with 'givenScenario', we are able to reduce duplication by reusing steps."*

Since some of our respondents might have been previous BDD practitioners who no longer used BDD on any of their projects at the time of completing the survey, we also wanted to know the distribution of duplicate detection methods among active and non-active BDD practitioners who responded to our survey. An active practitioner in this regard is the one who uses BDD in either all projects, or some projects, or a few pilot projects. Understanding the distribution of duplicate detection and management methods among active BDD practitioners would give insight into the amount of potential beneficiaries of future automated approaches for detecting and managing duplicates in BDD specifications. Figure 3.14 shows the distribution of duplication detection methods among active and non-active BDD practitioners. Almost 60% of the respondents were active BDD practitioners who either: perform manual inspection to detect duplication in their BDD specifications and thereafter decide on how to manage it; or have decided to live with it, given the complexity of the detection and management process; or are currently looking for an automated solution to detect and manage it. Thus, in Figure 3.14, the group "manual, decided to live with it, or looking for solution" represents respondents who care about duplication detection and management in their (respondents) BDD specifications. On the other hand, the group "don't regard it as problem" represents respondents who do not care about the detection and management of duplicates in their (respondents) BDD specifications.

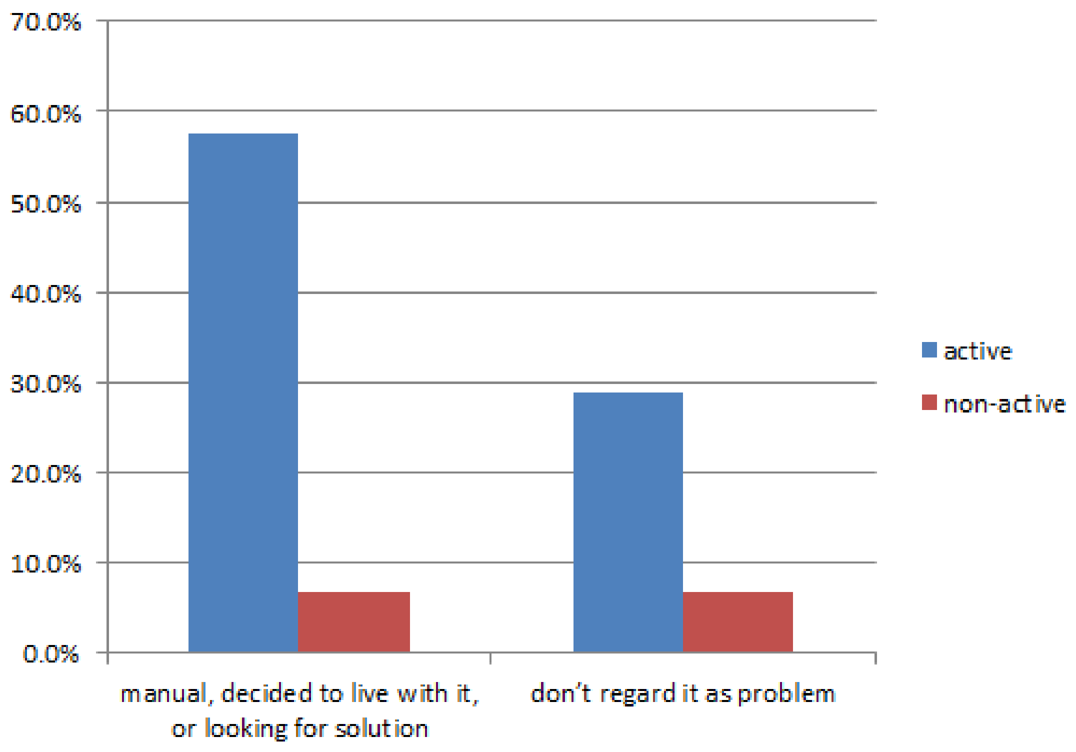


Figure 3.14: Duplication detection methods among active and non-active BDD practitioners

3.4 Discussion

This section discusses the significance of the results, providing answers to the research questions in section 3.1. The results' validity threats and mitigation strategies are discussed later in section 3.6.

RQ1: Is BDD in a considerable active use in industry at present? To explain the activeness of BDD use in industry, we use the theory of *vertical* (or explicit) and *horizontal* (or implicit) use by Iivari *et al.* [155, 156]. Vertical use expresses the degree of rigour with which a particular method is followed, eg., strict adherence to the method's documentation or partial adherence. Horizontal use, on the other hand, refers to the use of a method across multiple teams and projects in an organization after initial adoption, learning, and internalization.

In our study, the following is how we relate survey responses to the theory of vertical/horizontal use: with respect to *horizontal use*, for a range of organisation types, we pay attention to: whether BDD is used on all projects, some projects, a few pilot projects, or not used at all; whether BDD is used as a mandatory or optional tool; and plans by organisations to use BDD in the future. Also, we use *vertical use* to categorise issues reported by practitioners that are related to conformity or non-conformity with the BDD workflow.

Inline with the State of Agile Report [157, 158], we learn from the survey results that BDD is in active use in the industry. It can be seen from Figure 3.3 and Figure 3.4 that there is a substantial level of *horizontal use*, with some organizations using it on all projects, while others use it on some projects. Additionally, while there are organizations (20% of respondents) that have made BDD a mandatory tool, a significant proportion (61% of respondents) use it as an optional tool. This is to be expected as most organisations would use different software development techniques, for various reasons, including the dictates of a particular project. We can also expect that some organizations might use selected agile techniques, but not be committed users of every agile practice.

Answer to RQ1: *BDD is in active use in the industry, and it is used more in private organisations than in public organisations.*

Table 3.5 summarises the results of the specific survey questions related to the extent of horizontal use. For Q2, Active Horizontal Use combines use of BDD on all projects, some projects, and a few pilot projects; Inactive Horizontal Use represents responses that stated that BDD is not currently used. For Q3, Active Horizontal Use represents responses in which BDD is used as either a mandatory or an optional tool; Inactive Horizontal Use represents responses indicating that BDD is not in use by the respondents' organisations. For Q4, Active Horizontal Use combines plans to use BDD as a key tool on all projects, plans to use BDD as an optional tool on some projects, and plans to continue using BDD on a few pilot projects; Inactive Horizontal Use represents plans not to use BDD.

Survey Question	**Active Horizontal Use**	**Inactive Horizontal Use**
Q2: How would you describe the frequency of BDD use in your organisation currently?	82.6%	17.4%
Q3: Which of the following best summarises the use of BDD in your organisation currently?	81.0%	18.0%
Q4: How would you describe plans to use BDD in your organisation in the future?	85.0%	15.0%

Table 3.5: Activeness of BDD use in industry

Moreover, we note from the survey results that there are *vertical use* concerns whereby some practitioners do not observe BDD best practices, notably by avoiding or downplaying the collaboration aspects, resulting in future costs (refer to the results of survey questions Q8 and Q14 in Section 3.3.2). However, in the future, it would be interesting to investigate the extent to which teams that claim to use BDD strictly adhere to the dictates of the BDD workflow, such as ensuring that there are close collaborations between customers, developers and testers when creating BDD scenarios for specific projects. This would give more insights into the activeness and or inactiveness of vertical use of BDD in industry. That said, we posit that the observed extent of use, and the plans to continue using BDD (Figure 3.6) are sufficient to attract the attention of the research community in uncovering better ways to support BDD practitioners.

RQ2: What are the perceived benefits and challenges involved in using BDD? We use the following factors from the Agile Usage Model (AUM) [159, 160, 161] to explain the perceived importance, overall benefits and challenges of BDD. In the AUM, the following terms are used:

- **Relative advantage:** *"the degree to which the innovation is perceived to be better than its precursor"* ([161], p.2). This can be reflected in the ability of an agile method to offer benefits like improved productivity and quality, reduced cost and time, producing maintainable and evolvable code, improved morale, collaboration and customer satisfaction, etc [162].

- **Compatibility:** *"the degree to which agile practices are perceived as being consistent with the existing practices, values, and past experiences"* ([159], p.4).

- **Agile Mindset:** A mindset that perceives challenges as learning opportunities, building on optimism to grow over time, with effort in place [161].

- **Agile Coach:** An individual with both technical and domain knowledge who can point an agile team in right directions without imposing matters [161].

Generally, BDD has significant rating, with more that 50% of the respondents affirming its importance (refer to Figure 3.7, combining responses for **Important** and **Very Important**). The use of domain specific terms, improving communication among stakeholders, the executable nature of BDD specifications, and facilitating comprehension of code intentions are the benefits of BDD that were highly rated by respondents (Table 3.4). These all can be linked to the *relative advantage* factor in the AUM that BDD has over its precursor agile practice called TDD. Actually, it was the TDD's limitations in enabling teams to focus on implementing correct software behaviours that led to the birth of BDD [29]. However, comparative studies would be required to shed light on whether BDD's potential has been realised in practice.

The downside of BDD agreed to by most respondents has to do with changing the way teams used to approach software development (Table 3.4). This is in line with the *compatibility* factor in the AUM: new innovations are likely to face resistance by some adopters, especially when the adopters are slow at embracing changes. We are of the view that an *agile mindset* is important in addressing this challenge. Because approaching software development in a BDD way might be new to teams, willingness to learn and adopt new techniques is vital in the adoption of BDD. Furthermore, imprecise understanding of the BDD workflow, non-adherence to it, the scarcity of coaching services and training materials also hinder the adoption and continued use of BDD. It is our opinion that an *agile coach* with good understanding of the BDD workflow could help teams to navigate these challenges.

Answer to RQ2: *The use of domain specific terms, improving communication among stakeholders, the executable nature of BDD specifications, and facilitating comprehension of code intentions are the main benefits of BDD; changing a team's way of approaching software development is the main challenge of BDD.*

RQ3: What are the maintenance challenges reported amongst the issues raised by users (and former users) of BDD? From the survey responses, we learned that BDD feature suites suffer from the same maintenance challenges that test suites in automated testing face. Refer to Section 3.3.3 or Table 3.6 for an extended list of what survey respondents reported as challenges of maintaining BDD feature suites.To be of good quality, like the code under test, test suites must be maintainable [163, 164, 165, 114, 166]. As such, we hold the view that it is important to investigate, in the context of BDD, test maintainability aspects such as ease of change, bug detectability (for the bugs in both test and production code), test comprehensibility [164, 163], and other maintainability aspects, to support the work of BDD practitioners.

Answer to RQ3: *BDD feature suites suffer from the same maintenance challenges that test suites in automated testing face.*

RQ4: To what extent is the discovery and management of duplicates in BDD specifications seen as an unsolved problem by practitioners, and what techniques are being employed to deal with it? Apart from agreeing with the view that the presence of duplication in BDD feature suites could cause test execution, maintenance and evolution problems (refer to Section 3.3.4 on the responses to survey question Q10), most respondents think that, though present, duplication in their BDD specifications remains a manageable problem (see Figure 3.10 and Figure 3.11). However, duplication is still among the maintenance challenges of concern to some BDD practitioners. In fact, it has caused some practitioners to stop using the BDD technique: *"We decided to not use BDD any more because it was hard to maintain it... In the beginning we were checking for duplication, but at one point it has become very hard to manage them. Even though our tests were very much readable, our code underneath became less and less readable."* (R20)

Referring to Figure 3.12, for the most part, duplication detection and management is done manually (40% of respondents). Nevertheless, there is a significant proportion (17%) of respondents who have given up on the duplication detection and management process, because of its complexity. Combining these, more than a half (57%) of the respondents are concerned with duplication detection and management, except for those in the "other" category, who either explicitly expressed their need for an automated solution or mentioned their specific current manual approach to duplication detection and management. We thus specifically identified O_6 and O_7 in Table 3.6 as opportunities for the research community to investigate innovative ways to help BDD practitioners who either use the manual process, or have given up, or are likely to experience, in the future, serious duplication detection and management concerns.

Answer to RQ4: *Duplication in BDD suites is reportedly manageable. It is, however, still among the maintenance challenges of concern to some BDD practitioners and has, in some instances, caused practitioners to stop using the BDD technique. For the most part, detection and management of duplication in BDD suites is done manually.*

3.5 Research Opportunities

We now present the research opportunities, based on the challenges reported by survey respondents (refer to Section 3.3.2 and Section 3.3.3).

To understand how research might be able to address some of the problems encountered by users of large BDD specifications over the long term, we searched the existing literature for research areas in which problems similar to BDD challenges reported in our survey are being/have been investigated. Since the survey results revealed that BDD specifications suffer the same maintenance challenges found in automated test suites more generally, our literature search focused on research aiming to make automated test suites maintainable, comprehensive and extensible. Each of the identified challenges, summarised in Section 3.3.3, was mapped to the existing literature, resulting in 10 research opportunities which we propose in this area. In Table 3.6, we present the available research opportunities and link them to the relevant existing literature that covers similar problems in other areas, apart from BDD. The sample papers we list for each challenge (see column "Link to Related Literature" in Table 3.6) can serve as starting points for researchers wishing to investigate the problems addressed by the papers, in the context of BDD.

Specifically, the research under O_1 to O_7 would focus on making BDD specifications easy to understand, maintain and extend; and the research under O_8 to O_{10} would redress the process that is likely to result into BDD specifications with significant maintenance problems. *Inter alia*, we posit that a body of scientific evidence is required to provide answers to the following questions:

1. How could the BDD workflow be enhanced to produce more maintainable specifications? Specifically, it might be worthwhile investigating whether there are specific aspects of the BDD workflow that are prone to producing hard-to-maintain specifications, and how that could be redressed. For example, it might be interesting to compare the comprehensibility, maintainability, and extensibility of BDD feature suites developed collaboratively between all project stakeholders (i.e., following the proper BDD workflow), the comprehensibility, maintainability, and extensibility of BDD feature suites created by developers in a silo, and the comprehensibility, maintainability, and extensibility of BDD feature suites created without developers. If, for example, overwhelming scientific evidence suggest that BDD feature suites created by developers in a silo are easy to comprehend, extend, and maintain compared to BDD feature suites developed collaboratively between all projects stakeholders, then the BDD workflow might have to be adapted accordingly. Investigating appropriate ways to adapt the BDD workflow to accommodate new scientific evidence might be another possible research direction.

2. Better ways to adapt existing unit test maintenance techniques to the context of BDD tests. Or how could better techniques and tools specifically for the maintenance of BDD tests be developed?

3. Better ways to apply existing regression test suite reduction, selection and prioritization techniques (e.g. [118]) to address problems of slow suites due to the presence of duplication, and other concerns, in BDD specifications.

4. Characterization of duplication in the context of BDD specifications, and development of appropriate duplication detection techniques and tools.

5. How could the existing techniques and tools for detecting and managing duplication in program code (e.g. [35, 74]) be applied to the problem of duplication detection and management in BDD specifications?

3.6 Threats to Validity

The threats to the validity of our results are as follows:

1. We mainly depended on practitioners with online presence, either through GitHub

ID	Challenge	Opportunity	Link to Related Literature
O_1	Hard to comprehend BDD feature suites	Investigate BDD test smells, technical debt, and the adoption of test suite comprehension techniques to BDD specifications	[167, 114, 129, 163, 168, 169, 170, 171, 164, 172]
O_2	Difficulty of locating faults in large BDD suites	Investigate test fault localization techniques in the context of BDD specifications	[173, 174, 175]
O_3	Hard to change BDD suites	Investigate automated test repair for BDD specifications	[176, 177, 178, 179, 180]
O_4	Slow BDD suites	Investigate test minimization, selection and prioritization in the context of BDD	[118, 181, 182, 183]
O_5	The need to maintain BDD tests in addition to unit tests	Investigate integrated functional and unit test maintenance for BDD tests	[184, 180]
O_6	Duplication detection in BDD specifications	Investigate duplication detection in the context of BDD	[35, 1]
O_7	Duplication management in BDD specifications	Investigate duplication management in the context of BDD	[74]
O_8	Complexity of BDD tools	Investigate BDD tools selection guides, and how to reduce complexity in BDD tools	[185, 186]
O_9	Non-adherence to the BDD workflow	Investigate the incorporation of maintenance concerns at the core of BDD workflow and tools	[187, 170, 163]
O_{10}	Scarcity of coaching and material guidelines on BDD	Investigate the impact of coaching and guidelines on producing maintainable specifications	[188, 189, 161, 163]

Table 3.6: Challenges and research opportunities for maintenance and evolution of BDD specifications

or other online forums where BDD and other agile topics are discussed. Thus, we might have missed some in-house practitioners that are not easily reachable through any of those means. To mitigate the effects of this, we requested those who completed or saw the survey to refer it to others. Also, we sent survey completion requests to some practitioners who were known in person to the authors, and requested them to share the survey to others.

2. Some institutional rules and regulations might have determined whether or not participants responded in full. To mitigate the effects of this, we kept any identifying information optional, and clearly stated this at the beginning of the survey, and in all survey completion invitations.

3. Most of the respondents might have been using a particular BDD tool, so that our results could be valid for users of a specific BDD tool only. To cover practitioners using a variety of BDD tools, we followed the objective criteria mentioned in Section 3.2.2 to identify email addresses to which survey completion requests

were sent. We also posted the survey in general BDD and agile forums, in anticipation that respondents from those forums might be using different tools. Additionally, the survey included seven tools from which respondents could choose several tools, as well as an "other(s)" option.

4. The use of convenience sampling (in our case, depending on self-selecting respondents within the groups we contacted) might limit the ability to generalise from the survey findings. To mitigate the effects of this, we survey 75 respondents from 26 countries across the world, and some of the respondents were contributors to sizeable BDD projects in GitHub (see Section 3.2.2). Still, our results may not generalise to all BDD practitioners across the world. For example, our results do not represent BDD practitioners who are not proficient in English.

5. Our focus on duplication in the survey could have distracted respondents from mentioning other more damaging/significant maintenance challenges. But we needed specific information on duplication–a more generic survey might have meant respondents did not comment on duplication at all. To mitigate the effects of this, we specifically provided Q14 in the survey (see Appendix A.1) in which respondents were allowed to report other issues about BDD, be they related to maintenance or not. Nevertheless, one would still need to be careful when attempting to generalise from the findings of our survey.

3.7 Summary

Despite the benefits of BDD–such as producing customer-readable and executable specifications–management of BDD specifications over the long term can be challenging, particularly when they grow beyond a handful of features and when multiple team members are involved with writing and updating them over time. Redundancy can creep into the specification, leading to bloated BDD specifications that are more costly to maintain and use.

In this chapter, we have used quantitative and qualitative data collected using the survey of BDD practitioners, to report the activeness of BDD use in industry, its benefits, general and specific maintenance challenges, particularly regarding duplication. Based on the perspectives of an active BDD user community that we surveyed, we found that

BDD is in active use in industry, inline with the State of Agile Report [157, 158], and that it is used more in private organisations than in public and other organisation types. Some respondent organizations use it on all projects, while the majority of respondent organizations use it on only some of the projects. Also, while a few previous practitioners are not currently using it due to various challenges, some of which are maintenance related, the majority of respondents among currently active and non-active practitioners plan to use BDD in the future as either a key tool on all projects or as an optional tool on some projects. The following benefits of BDD were highly rated by respondents: the use of domain specific terms, improving communication among stakeholders, the executable nature of BDD specifications, and facilitating comprehension of code intentions. While respondents were of the opinion that changing the way teams approach software development was among the main downsides of BDD, a significant number of responses indicated that BDD specifications suffer from the same maintenance challenges found in automated test suites more generally. As well, we found out that duplication is among the maintenance challenges of concern for BDD practitioners, and, for the most part, its detection and management is done manually. Therefore, there exists a need to investigate novel ways to automatically detect and manage duplication in BDD specifications.

Despite the reported maintenance challenges, in this area, we are only aware of studies which are limited in both number and scope. We conclude that there is a scarcity of research to inform the development of better tools to support the maintenance and evolution of BDD specifications, and we propose 10 open research opportunities in this area (Table 3.6).

Chapter 4

Benchmark Development

4.1 Introduction

In this chapter, we present the process that was used to develop an experimental set of duplicate scenarios for detection, as well as the information about the resulting duplicates. Among other things, this set of duplicates is important for evaluating our duplicate detection approach that will be presented in Chapter 5. Specifically, we present our definition of a semantic duplicate scenario, the projects we used and the process used to identify them, as well as the categories of duplicates we obtained.

4.2 Semantically Equivalent Scenarios

BDD scenarios often exhibit high degrees of textual similarity—almost by definition. Within a feature file, each scenario should express a different aspect of the functionality, typically targeted at discontinuities in the functionality being specified: for example, at edge cases defining the boundaries of what is acceptable behaviour, and the response of the system to erroneous inputs. These scenarios will typically contain very similar *Given* and *When* steps, with only the parameter values (the values matched by the regular expression groups in the step definitions) differing. Far from being problematic, such reuse of step definitions is important in keeping the BDD specification manageable and comprehensible. In BDD, we effectively create a language of common domain concepts that are important to the (eventual) users of the system. A coherent domain language cannot emerge unless we are disciplined about refining and reusing step definitions wherever possible throughout the specification (except where the effect on readability negates the benefits of the reuse).

Therefore, we need to be able to detect the kind of duplication that indicates redundancy in the BDD specification: when two or more scenarios describe the same piece of functionality. The problem is that a legal BDD specification may contain two scenarios describing exactly the same functionality, while being completely textually dissimilar. Because steps are essentially described using natural language, we have the full scope and complexity of whatever language we are working in to describe the functionality. Even the division of the functionality into steps can be very different, with some core aspects being specified as several fine-grained *Given* steps in one scenario and as a single chunky *When* step in another.

How, then, can we define the concept of semantic duplication for BDD specifications, if even well-designed, non-redundant BDD specifications contain a high degree of textual and syntactic similarity? The key fact that marks two scenarios as being redundant is if they specify exactly the same production behaviour. This leads us to the following definition of semantically equivalent scenarios.

Definition 1 *Semantically equivalent scenarios: Two or more scenarios are semantically equivalent if they describe the same piece of functionality for the System Under Test. That is, they specify exactly the same production behaviour.*

4.3 Benchmark Development

4.3.1 Context

After defining what a semantic duplicate of a scenario is (Definition 1), we need an approach to detect it. But, in order to evaluate the effectiveness of our approach in detecting duplicate BDD scenarios, we need a benchmark against which to compare the results of our approach. In this section, we set out to develop such a benchmark.

Recall from Definition 1 that we consider BDD scenarios to be semantically equivalent if they describe the same piece of functionality for the System Under Test. In order to evaluate the tool for detecting scenarios that satisfy this condition, we needed access to BDD specifications with known duplicates, so that we could test the ability of our tool to detect these duplicates and only these duplicates. But because there were no benchmarks with known duplicate BDD scenarios, and because we did not have access to specific BDD systems as well as their development teams, we decided to look for open source BDD projects for use in developing a benchmark of known duplicate scenarios.

Moreover, manual discovery of known duplicates in a system is impractical because of, among other things, limited domain knowledge among judges, possible disagreements between judges on what should be regarded as a duplicate, and the size of the system may be too large making manual analysis prohibitively expensive [4]. Thus, developing data sets of duplicates in realistic systems is expensive, as it requires a group of participants with sufficient time and expertise to search manually [4]. We, therefore, decided to combine the following two approaches for developing a benchmark of known duplicate scenarios: manual discovery of duplicate scenarios in the selected evaluation BDD projects, as well as injecting duplicates into the evaluation systems.

In case of duplicates injection, given the potential for bias if we as designers of the duplicate detection method performed the injection ourselves, we recruited volunteers and asked them to inject duplicates for us. The process of obtaining these duplicates is outlined below. First, we describe the open source systems that we selected to be the subject of the experiment. Next, we present the process we followed in order to obtain duplicates from existing scenarios, and to inject duplicates.

4.3.2 The Host Systems

The following criteria were used to select the systems for our experiment:

1. The BDD features must be specified using Gherkin.
2. Glue code must be written in Java.
3. The system must contain a non-trivial number of features/scenarios.
4. The source code and associated files must be available in a public source code repository.
5. The domain of the system should be one which can be understood with only general knowledge, to allow our volunteers to specify reasonable scenarios.
6. The system should neither be a class assignment nor a small demo project.

7. The scenarios in the system should execute to completion, to produce the trace.

Based on these criteria, we searched three popular open source project repositories (GitHub[1], GitLab [2], and Bitbucket[3]) in two iterations. First, we searched the repositories for potential projects based on our criteria and came up with seven projects. The majority of the systems that were returned by our search were either not written in Java or trivial demonstration projects. Second, we tried to execute the seven projects, to make sure that they could be run and produce the traces before requesting our volunteers to work on them. The scenarios in four of the seven systems could not execute to completion, meaning that we could not trust their traces to be complete. This resulted in the selection of the following three projects, on which we based our experiment:

- jcshs[4]
- facad_services_cucumber_automated_test[5]
- atest_bf_app[6]

A look at the BDD features that describe the domain served by each of the three system suggests the following:

- **jcshs:** Describes features for use by telecom companies to facilitate, among other things, interactions between customers and the information stored in the company database. For example, customers can use the software to register with the company, as well as to send and receive text messages on various services offered by the the company.
- **facad_services_cucumber_automated_test:** Describes features for managing school information. Among other things, it facilitates the registration of pupils, manages teachers' profiles, and enables payment of various school dues by credit card.
- **atest_bf_app:** Describes features that, among other things, enable customers to perform online purchases.

[1]`https://github.com/`
[2]`https://about.gitlab.com/`
[3]`https://bitbucket.org/`
[4]`https://bitbucket.org/manv6/jcshs`
[5]`https://bitbucket.org/mohamrah/facad_services _cucumber_automated_test`
[6]`https://gitlab.com/alexandermiro/atest_bdd`

		Project		
S/n	**Item**	**System 1**	**System 2**	**System 3**
1	Number of features in a feature suite	23	8	14
2	Number of scenarios in feature suite	142	41	4
3	Number of scenario outlines in feature suite	0	0	23
4	Number of background steps in feature suite	21	0	8
5	Number of packages in the whole system	76	2	19
6	Number of production classes in the system	188	8	65
7	Number of glue code classes in the system	12	8	14

Table 4.1: Characteristics of selected evaluation software

We hereafter use *System 1* for jcshs, *System 2* for facad_services_cucumber_automated_test, and *System 3* for atest_bf_app. Table 4.1 summarises the characteristics of the three systems before duplicates injection. Based on the introduction to BDD (section 1.1), the contents of the table should hopefully be self-explanatory, except for the following: In Gherkin, Scenario Outlines allow many scenarios to be specified succinctly, using tables from which step parameters should be taken. Background elements are another piece of Gherkin syntax; they specify steps which should be included in all scenarios in the same feature file as the Background (and play a role similar to the *@Before* methods in xUnit test cases).

4.3.3 Duplicates between Original Scenarios

We first wanted a benchmark of duplicate pairs between original scenarios, for use in evaluating our duplicate detection approach before relying on injected scenarios. Guided by the meaning of a duplicate scenario in Definition 1, and our understanding of the domains for the evaluation systems, 13 pairs of duplicate BDD scenarios were manually identified from our three evaluation systems (Table 4.1). For each of the three evaluation systems, Table 4.2 shows the number of duplicate pairs between scenarios that existed before injection.

System	Number of Duplicate Scenario Pairs
1	9
2	1
3	3
Total	**13**

Table 4.2: Number of duplicate scenario pairs between original scenarios

4.3.4 Duplicate Injection

We recruited 13 volunteers for our study, each satisfying at least one of the criteria:

- At least 3 years experience in software testing.
- At least 3 years experience in analysis and design.
- At least 3 years experience in development.
- General understanding of one or all of the following: analysis and design of software systems, behaviour-driven development, test-driven development, and acceptance-test-driven development.

Whereas some volunteers were industry professionals at the time, some were PhD and MSc students with prior industry experience. We approached the volunteers face-to-face and through telephone, since we knew them personally. There was not a problem of bias in doing so, since the task was synthetic rather than evaluative–we were not asking them to make any value judgement for or against any part of our work.

The injection of duplicate BDD scenarios into the host software was conducted in two phases. The first phase was to allow us to trial the instructions and approach before we asked people to spend their time and effort. The second phase was for gathering more injected duplicates that we would use in the study.

In the first phase, we met face-to-face with 2 of the 13 volunteers and they injected duplicates in System 1 in three successive iterations. Because we needed an injection approach that could enable volunteers to produce useful duplicates with minimal or no supervision at all, we designed a series of tasks for enabling us to learn if this was the case. The tasks were executed in three iterations, covering different situations in which duplicates are likely to be created. In the first iteration, the volunteers were presented with a feature file in which some scenarios had been deleted. Only the titles of the missing scenarios remained and the volunteers were asked to come up with steps for each of those scenarios. After doing that, the volunteers were presented with the original feature file, with scenario titles removed but with all the scenarios' steps in place, including those that had been hidden during the first part of the experiment. The volunteers were asked to confirm that the scenario they had written was a full duplicate of one of the existing scenarios, and to record which one. Matching of duplicates was important for us to see if volunteers could recognise the duplicates of what they had created, and correct matching assured us that volunteers could create the kind of duplicates we expected them to.

In the second iteration, the volunteers were provided with general description of another feature (different from the first one), the Gherkin Background steps, and three scenarios from the given general feature description. They were then asked to add further scenarios based on the given feature description, covering aspects of the feature that were not covered by the initial set of scenarios. Again, at the end of the iteration, volunteers were asked to map the scenarios they had created to the original scenarios.

In the third iteration, a feature (different from the one used in the first and second iteration) with several scenarios was given, and volunteers were asked to study the feature and come up with completely new scenarios that they considered to be missing in that feature. Again, mapping of duplicate scenarios was done by the volunteers at the end of the iteration.

This gave us a starting point for our corpus of duplicate scenarios.

After the first phase, our duplication injection instructions were improved covering, among other things, the need for offline (not through a physical meeting between the researchers and volunteers) duplicates injection, while at the same time ensuring that we still got scenarios that duplicated, in a variety of ways, the behaviours of original scenarios in the evaluation systems. In the second phase, the improved duplication injection instructions were shared with the remaining 11 volunteers, and we continuously supported them through different online means as they worked towards injecting duplicates in the systems they were assigned to.

Volunteers were randomly assigned to the 3 systems. Table 4.3 presents the number of volunteers per system. In the second phase, 4 more volunteers (in addition to the 2 in the first phase) injected duplicates in System 1. All the duplicates in System 2 and

System	Number of Volunteers
1	6
2	5
3	3
Total	**13**

Table 4.3: Number of volunteers per system

System 3 were injected in the second phase. Moreover, in the second phase, of the 11 volunteers, one volunteer injected duplicates in two systems, System 2 and System 3. To attract more and diverse duplicate scenarios, the allocation of volunteers to systems prioritised the systems with more scenarios.

The choice of the scenarios to duplicate was based on two criteria: First, the scenario's ability to be easily understood by volunteers. Second, its ability to be executed to completion and produce the trace. Almost all scenarios across the 3 systems satisfied these criteria.

This process gave us 125 known duplicate scenario pairs, across the three systems. Table 4.4 shows the distribution of the duplicate pairs across the three systems. **IO** pairs represent duplicates between injected and original scenarios, and **II** pairs represent duplicates between scenarios that were injected by volunteers. For each of the 3 systems, we (the researchers) were able to form the II pairs by mapping different injected scenarios that targeted the same original scenarios. Including II pairs in the ground truth enabled us to classify every pair reported by our tool as either true or false positive.

	System 1	System 2	System 3	Total
Number of features involved in duplication	7	6	2	**15**
Number of IO duplicate scenario pairs	36	23	6	**65**
Number of II duplicate scenario pairs	39	17	4	**60**
Total number of duplicate scenario pairs (IO + II)	**75**	**40**	**10**	**125**

Table 4.4: Distribution of duplicate scenario pairs across the three evaluation systems

While preserving the overall functionalities of the original scenarios, each injected scenario was produced after either a complete re-write of an original scenario, or rewording of the steps of an original scenario, or reordering of steps of an original scenario, or merging or disaggregating steps of an original scenario, or a combination of these. Thus, the scenarios we obtained through the injection exercise represented possible manifestations of duplicate scenarios in real world systems.

To make the scenarios created by volunteers executable, we had to create the glue code for all the new steps. Because volunteers were guided to refrain from inventing new requirements, (i.e, by writing scenarios that were duplicates of existing functionality), we (the researchers) were able to create the glue code for all the new steps without extending the production code.

Through the duplicate injection process, we obtained versions of evaluation systems with 125 known duplicate pairs, the scenarios in which could be executed. Combining with 13 pairs of duplicates between original scenarios (OO) (Table 4.2), we obtained a benchmark of 138 known duplicate scenario pairs across the three evaluation systems.

4.4 Summary

Any approach for detecting semantically equivalent BDD scenarios would benefit from a benchmark of known duplicate scenarios pairs. But our search in both scientific and grey literature could not reveal such a benchmark. Thus, in this chapter, apart from defining what a semantic duplicate of a BDD scenario is, we have also presented the process we followed to develop a benchmark of known duplicate scenario pairs in three open source systems that were identified for that purpose. This resulted into a total of 138 known duplicate scenario pairs across the three systems, 13 of which representing duplicates between scenarios that existed before injection, and the remaining 125 pairs representing duplicate pairs obtained through the injection process.

Chapter 5

Detecting Duplicate Scenarios in BDD Suites

5.1 Introduction

In Chapter 3, we presented challenges faced by practitioners in maintaining BDD suites and, in particular, the problems of duplication. In the present chapter, we demonstrate that existing duplicate detection tools cannot effectively detect semantically equivalent scenarios in BDD feature suites, and, consequently, we present and evaluate our framework for detecting semantically equivalent scenarios in BDD feature suites.

Despite the benefits of the BDD technique, some of which were discussed in section 3.3.2, it also brings problems. Large BDD specifications can be hard to maintain and extend, and some teams report functionality becoming effectively frozen, or even dropping the use of the BDD technique, because of the costs and risks of changing the specifications (section 3.4). Furthermore, comprehension of large suites of BDD scenarios, the relationships between them and the relationships between the scenarios and the code they test, can be a daunting task. Also, because BDD is still relatively new, tools to support developers in managing their suites of scenarios are still in their infancy. Thus, it becomes costly for teams when redundancy creeps into their BDD specifications. The suites of scenarios can have long execution times, causing slow Quality Assurance and bug fixing cycles. Even worse, it becomes hard to maintain both the specifications' quality and conceptual integrity. This increases the risk of further redundancies, omissions and inelegances in the BDD specifications. Despite these challenges, the state-of-the-art tools for general duplicate detection in program code perform poorly on the problem of detecting duplication in BDD specifications.

In this chapter, we present a framework for detecting duplicate scenarios in BDD specifications based on the analysis of execution traces. We aim to establish when two scenarios exercise the code under test in the same way–that is, execute the same functional test over the production code. This is tricky, because we have to discriminate between important and unimportant differences in the scenarios' execution traces. In the present chapter, we attempt to answer the question of important and unimportant trace differences, and devise and evaluate a framework for detecting duplicate BDD scenarios.

Specifically, we attempt to differentiate between essential and accidental behaviours of a BDD scenario and, thereafter, we use identical essential behaviours to detect duplicate BDD scenarios. We suppose that a BDD scenario can have both essential and accidental properties. On the one hand, the behaviours that are inherent in the nature of a scenario are regarded as *essential*. These represent core functions of a scenario. On the other hand, subsidiary behaviours that result from attempts to express core functions of a scenario are regarded as *accidental*. These can change over time or can be removed without affecting the core functions of a scenario. Section 5.4.2 will provide more details about how we relate the notion of essences and accidents to the context of BDD scenarios. Based on the *essence-accident* notion, we consider to be semantically equivalent scenarios that exhibit the same *essential* behaviours.

We run a scenario several times to identify parts of its trace that change between runs, as well as parts of its trace that remain constant across several runs. We consider anything that changes between runs to be incidental and disregardable. Thus, parts of the scenario's execution trace that remain constant across several runs are deemed to represent its (scenario) essential characteristics, and parts of the scenario's execution trace that change between runs are considered to represent the accidental characteristics of a scenario.

The duplicate detection framework that we propose can detect duplicate scenarios with or without focusing on essential characteristics of scenarios. When the focus is not on essential characteristics of scenarios, each scenario is run exactly once, and whole or specific parts of the resulting traces for the individual scenarios are compared to detect duplicate scenarios. When the focus is on essential characteristics of scenarios, we regard scenarios that exhibit the same essential characteristics to be duplicates of each other.

The implementation of our framework works with BDD specifications expressed in the Gherkin[1] language, and the glue code written in Java using Cucumber-JVM conventions[2]. As well, the evaluation results suggest that the comparison of execution paths is the most promising effective way to detect duplicate BDD scenarios. In addition, the focus on essential traces of scenarios improved the ability to detect duplicate scenarios.

The key insight from this component of our research is to use *essential traces* (defined in section 5.4.3) to detect semantically equivalent BDD scenarios, instead of using *default traces* (defined in section 5.4.3) that might contain accidental information, causing duplicate scenarios to go undetected.

The contributions of this chapter are threefold:

- **Understanding the limitations of existing duplicate detection approaches and techniques on the problem of detecting semantically equivalent BDD scenarios:** Experimentation with three mature tools revealed that, when applied on the problem on detecting semantically equivalent BDD scenarios, the techniques and approaches for detecting duplication in program code either miss duplicates of interest or return too many false positives. This renders them unsuitable for the problem of detecting semantically equivalent BDD scenarios.

- **BDD duplicate detection framework:** We propose a duplicate detection framework that analyses various information in the dynamic traces of BDD scenarios to detect semantically equivalent scenarios in BDD specifications. Our duplicate detection framework attempts to detect duplicate scenarios with and without focusing on essential characteristics of scenarios. Without focusing on essential characteristics of scenarios, whole or specific parts of the traces produced when individual scenarios are executed only once are compared to detect semantically equivalent scenarios. We also propose an approach to determining an essential trace of a scenario. With an attempt to separate essential from accidental characteristics of scenarios, each scenario is executed several times, and trace parts that remain constant across several runs of a scenario are used to represent the scenario's essential characteristics; BDD scenarios that exhibit identical essential characteristics are regarded to be semantically equivalent.

[1] `cucumber.io/docs/reference`
[2] `github.com/cucumber/cucumber-jvm`

- **Use of execution traces to detect semantically equivalent BDD scenarios:** We used the proposed duplicate detection framework to detect duplicate BDD scenarios across three open source software systems, and learned that the comparison of execution paths of scenarios performs better at detecting semantically equivalent BDD scenarios, compared to the comparison of full traces, API calls, or the combination of API calls and internal calls. We also learned, based on two of the three evaluation systems, that the focus on essential characteristics of scenarios can improve the detection of semantically equivalent BDD scenarios.

The rest of this chapter is organised as follows: section 5.2 presents the duplicate detection problem in BDD specifications, section 5.3 highlights the limitations of existing duplicate detection tools on the problem of detecting duplicate BDD scenarios, section 5.4 presents our framework for detecting duplicate scenarios in BDD specifications, section 5.5 presents the evaluation of our approach, and section 5.6 concludes the chapter.

5.2 The Duplicate Detection Problem in BDD

BDD suites contain tens to several thousands scenarios (section 3.3.3). With suites at the larger end of the spectrum, maintaining existing scenarios or adding new steps or scenarios can be a daunting task. At present, there is lack of tools to facilitate effective management of collections of steps in BDD suites. This makes it difficult for teams to reuse steps when creating new scenarios and to find existing scenarios that are close to some new functionality that is being added to the suites. It is possible for a suite to have multiple scenarios or steps that specify the same functionality, with different step text. As stated previously, in the long run, specifying the same functionality using multiple step/scenario text can cause difficulties in comprehending, maintaining and extending BDD suites.

Listing 5.1 shows an example of three scenarios specifying the same functionality using different step text. The first two scenarios only differ in step text, but specify the same functional behaviour and have the same number of steps in Given, When and Then scenario parts. The third scenario not only exhibits variation in step text compared to the first two scenarios, but it also introduces new steps to cover aspects (checking card validity and checking availability of sufficient cash in an ATM) that

Listing 5.1: ATM Feature With Duplicate Scenarios

```
Feature: ATM transactions
As a bank account holder, whenever
I perform ATM withdrawals, I want my
account balance to be updated accordingly

Scenario: Successful withdrawal from an account
Given my account is in credit by $100
When I request withdrawal of $20 through the ATM
Then $20 should be dispensed
And the balance of my account should be $80

Scenario: Debit Account
Given an initial account balance of $100
When I request withdrawal of $20 through an ATM
Then I should receive $20 from the ATM
And the new account balance should become $80

Scenario: Withdraw cash from ATM
Given the balance in my account is $100
And I am using a valid card
When specify $20 as the withdrawal amount
And the ATM has cash that amounts to $20 or more
Then the ATM should dispense $20
And my account balance should be updated accordingly
```

were assumed in the first two scenarios. Despite the additions in the third scenario, in general, the same functional behaviour is specified by all the three scenarios in Listing 5.1. But, one would argue that the third scenario is functionally different from the first two because of the additional steps it contains. However, even in the face of that argument, in practice, there would still be a need for a mechanism to flag the three scenarios as duplicate candidates, given how functionally close the three scenarios are. That would help human developers to decide if indeed the three scenarios are duplicates of each other, and whether all the three scenarios have to be kept as part of the suite, or one/two of the three scenarios can be removed without losing the behaviours specified by the three scenarios. Importantly, if one scenario is subsumed by another scenario, it may be unnecessary to keep both scenarios in a suite.

Since the goal of reuse means that high degrees of similarity are to be expected between scenarios in BDD suites, it is challenging to detect the undesirable kind of duplication. But, intuitively, we would not want to have, in the same BDD suite, scenarios that belong to the same *equivalence class* [190].

It is important to note that, naturally, duplicate scenarios could appear next to one another in the same feature file like in Listing 5.1, or could appear between one or more scenarios in the same feature file, or could be scattered across multiple feature files in a BDD specification. So the artificially simple example in Listing 5.1 is just intended to illustrate the problem we are trying to address.

5.3 Existing Tools

We experimented with three mature clone detection tools, to assess their ability to detect duplicate BDD scenarios. In particular, we wanted to evaluate the ability of existing techniques on the problem of detecting duplicate scenarios, before proposing a new solution.

5.3.1 Tools Selection

After investigating the literature in the domain, we chose the tools that were available for download based on their semantic awareness, i.e., the duplicate detection techniques built in the tools incorporates some attempts to detect semantic duplication, as is the requirement in our case. We identified tools employing a variety of duplicate detection techniques discussed in Section 2.3. More specifically, tools employing text-based techniques were intuitively ruled out because behaviorally similar duplicates can be textually different [191, 148]. For token-based techniques, PMD/CPD represented the state-of-the-art, and so it was selected for our experiment. Both CloneDR [192] and DECKARD [66] represented mature tree-based tools, with different ways of manipulating trees to detect duplicates. We thus chose both of the two for our experiment as well.

The literature mentions the possibility of using PDG-based techniques to detect semantically equivalent code fragments. However, PDG-based techniques are discredited for their computational complexity, posed by the graph generation process. As well, apart from returning many false positives, these tools also perform better in detecting non-contiguous duplicates [193], when we wanted to be able to detect both contiguous and non-contiguous duplicates. For these reasons, plus the fact that PDG-based tools mentioned in the literature were not available for download and use, we did not try tools in this category. Besides, tools in other categories (for example DECKARD, which is tree-based) have semantic awareness [87], and so we could still meet the semantic awareness requirement without PDG-based tools. Eventually, PMD/CPD[3], CloneDR [192] and DECKARD [66] were chosen for our experiment.

5.3.2 Tools Configuration and Experiment Setup

After selecting the tools, 13 pairs of duplicates between original scenarios (Table 4.2) were used to assess the ability of existing duplicate detection tools on the problem of detecting duplicate BDD scenarios. Because the tools we experimented with can detect duplication in program code, we required code level representation of the duplicate scenarios that we wanted to detect. To obtain code level representation of each scenario, we performed manual unfolding of the glue code that would execute when each scenario was run. That is, we manually copied into one Java method glue code for all steps of a particular scenario. Eventually, each scenario was converted into one method, which we called a *scenario-representing-method*. The order of steps in a scenario dictated the order of glue code in a scenario-representing-method. Further, each pair of duplicate scenarios was represented by one class with two scenario-representing-methods.

The labour-intensive nature of manually unfolding the glue code made running the selected tools on all pairs of duplicate scenarios obtained through the duplicates injection experiment (section 4.3.4) prohibitively expensive. This is why we experimented with these tools only on a relatively small number of duplicates between original scenarios that had been manually identified from the three evaluation systems (section 4.3.3).

We ran the selected clone detection tools on the classes with scenario-representing methods. We used default configurations to run CloneDR and DECKARD since we did not come across any study with recommended values, and the alternatives we tried did not produce results with important differences. Also, after some trials, we decided to use the minimum token size of 10 for PMD/CPD because it is the one that could at least

[3]pmd.github.io

report duplicates matching whole scenarios, instead of partial code in the scenario-representing-methods. Some token sizes did not return any results, and others returned some results with too many false positives while missing the duplicates of interest.

A pair of duplicate scenarios was considered detected if their scenario-representing-methods were reported by the tool as duplicates. Scenario-representing-methods in which only part of their code was reported as duplicate meant that the duplicate scenarios they represented were considered undetected. In practice, reporting partial duplication between scenario-representing methods would imply that only parts of scenarios (and not whole scenarios) are reported as duplicates, when we were interested in semantic duplication between whole scenarios.

5.3.3 Results and Discussion

Table 5.1 shows the results of using existing tools to detect semantically equivalent BDD scenarios. "Known duplicates" represents the number of duplicate scenario pairs that we knew for each system, "Detected known duplicates" represents the number of known duplicate scenario pairs that were detected by each tool, "Candidates" represents the total number of duplication candidates reported by each tool, " Duplicates Between Whole BDD Scenarios" represents the number of pairs of duplicate candidates that reported duplication between whole scenarios, "Recall" represents the proportion of known duplicate scenario pairs that were detected by each tool, and "Precision" represents the proportion of candidates that reported duplication between whole scenarios. Whereas DECKARD was able to detect all but 3 of the duplicate pairs of scenarios, PMD/CPD and CloneDR missed almost all the pairs. Nevertheless, DECKARD reported many false positives, which, when applied to problems of a scale typical of commercial applications, would mean an extra cost of time and personnel for the team to judge the results.

We thus concluded that, in general, state-of-the-art techniques, which are specifically designed for detecting duplication in program code, are not suitable for the problem of detecting semantically equivalent BDD scenarios.

System	Tool	Known Duplicates	Detected Known Duplicates	Candidates	Duplicates Between Whole BDD Scenarios	Recall (%)	Precision (%)
System 1	PMD	9	0	12	0	0.00	0.00
	CloneDR	9	1	6	1	11.11	16.67
	DECKARD	9	9	589	112	100.00	19.02
System 2	PMD	1	0	7	0	0.00	0.00
	CloneDR	1	0	1	0	0.00	0.00
	DECKARD	1	0	43	0	0.00	0.00
System 3	PMD	3	0	8	0	0.00	0.00
	CloneDR	3	0	1	0	0.00	0.00
	DECKARD	3	1	951	48	33.33	5.05

Table 5.1: Results of experimentation with existing clone detection tools

5.4 Detecting Duplicate Scenarios

5.4.1 Hypothesis

Much existing work on duplicate detection in the context of program code is based around the idea of *clone detection*. While cloning is almost certainly used in the creation of BDD specifications, it is not the main concern. BDD scenarios often exhibit high degrees of textual similarity—almost by definition. Within a feature file, each scenario should express a different aspect of a software functionality, typically targeted at discontinuities in the functionality being specified: for example, at edge cases defining the boundaries of what is acceptable behaviour, and the response of the system to erroneous inputs. These scenarios will typically contain very similar *Given* and *When* steps, with only the parameter values (the values matched by the regular expression groups in the step definitions) differing. Far from being problematic, such reuse of step definitions is important in keeping the BDD specification manageable and comprehensible. In BDD, we effectively create a language of common domain concepts that are important to the (eventual) users of the system. A coherent domain language cannot emerge unless we are disciplined about refining and reusing step definitions wherever possible throughout the specification (except where the effect on readability negates the benefits of the reuse).

Therefore, the detection of clones in BDD scenarios is unlikely to be helpful. We need to be able to detect the kind of duplication that indicates redundancy in the BDD specification: when two or more textually distinct scenarios describe the same piece of functionality. The problem is that a valid BDD specification may contain two scenarios describing exactly the same functionality, while being textually dissimilar (Listing 5.1). Because scenario steps are described using natural language, we have the full scope and complexity of whatever language we are working in to describe the functionality. Even the division of the functionality into steps can be very different from one scenario to another, with some core aspects being specified as several fine-grained *Given* steps in one scenario and as a single chunky *When* step in another.

How, then, can we define the concept of semantic duplication of scenarios in BDD specifications, if even well-designed, non-redundant BDD specifications contain a high degree of textual similarity? The key fact that indicates two scenarios as being redundant is if they specify exactly the same behaviour of the software under test. This led us to the following hypothesis:

> **H:** We can detect semantically equivalent BDD scenarios by comparing how BDD scenarios exercise the production code.

5.4.2 Overall Duplicate Detection Framework

As a starting point for our work, we hypothesise that a good approach to detect semantically equivalent scenarios in BDD suites will require a combination of information from the natural language texts that are used to express the scenarios, plus more conventional analysis of the glue code and production code under test. We propose a framework with eight heuristics for detecting duplicate BDD scenarios. Each heuristic is inspired by a particular hypothesis about the run-time behaviours of semantically equivalent scenarios. As highlighted by the hypothesis in section 5.4.1, the main notion underpinning all the 8 heuristics is that *semantically equivalent BDD scenarios exercise the production code in the same way*, as summarised by the following definition:

Definition 2 *A scenario s_1 in a BDD suite Σ is a semantic duplicate of a scenario s_2 in Σ (where s_1 and s_2 are specified as two distinct scenarios in Σ) if: in every possible execution of Σ, the execution of both scenarios results in the invocation of the same sequence of production methods. The order of the method invocations must be the same in each case, and any repeated invocations of the same method must also be* *the same. The parameter types of each method call must also match, to allow for overloaded methods.*

The rest of this section presents an overview of the duplicate detection framework, Section 5.4.3 presents the definitions of key terms in our duplicate detection framework, Section 5.4.4 through Section 5.4.6 present the hypotheses that constitute our BDD duplicate detection framework, the combined duplicate detection algorithm, and the implementation of our duplicate detection algorithm.

Detecting semantic equivalence of programs is generally known to be an undecidable problem [194, 102]. But we need a way to approximate when two BDD scenarios specify the same behaviour of the Software Under Test, even if their steps may be phrased differently. To make a start in detecting scenarios that specify the same behaviour, we attempt to differentiate between essential and accidental properties of scenarios, and regard as semantically equivalent scenarios that have the same essential properties. In particular, we regard as essential characteristics the behaviours that are inherent in the nature of a scenario–that is, are important for the scenario to completely specify its fundamental functionality. For example, irrespective of many other things that different developers might add while writing the first scenario in Listing 5.1, it must check for the sufficiency of funds in a user's account and should only issue funds if a user has sufficient balance in their account. Essential characteristics are, thus, part and parcel of a scenario and without them specification of scenario's behaviour would be incomplete. On the other hand, accidental characteristics are behaviours that get introduced in the course of specifying the core functionality of a scenario, but are themselves not part of the core functionality of a scenario. For practical purposes, in our case, we consider anything that changes between runs to be incidental and disregardable. Admittedly, this imposes an obligation on a development team to avoid implementations where lots of things change. In fact, too much change might threaten the reliability of the test, making it non-deterministic. Too, poor quality tests might also yield a large number of false positives for the duplicate detection tool, since the non-determinism might cause lots of trace information to change. Our approach assumes that tests are of good quality.

Various information from the production code can be recorded as part of the scenario's execution trace. To make a start in exploring the information that might help to detect duplicate scenarios, we focused on information about the production methods that get exercised by a scenario. In particular, we wanted our trace to contain the information that can identify what production method was executed, what was its input(s) (e.g., the arguments it takes at runtime), what was its output (e.g., return value), as well as what statements inside the body of a method were involved in the computation of the output. Inspired by this view, for each production method that is involved in the execution of a scenario, we record the following information as part of a trace of a scenario:

- Name of a method
- Name of a production class to which a method belongs
- Parameter type(s)
- Return type
- Parameter value(s) of a method
- Return value of a method
- Access modifier of a method
- Executed statement(s) inside the body of a method

While somewhat limited in terms of expressing the actual runtime behaviours of a method, this combination of various information about a method can help us to explore both the observable behaviours of methods (e.g., through comparison of inputs and outputs) as well as the internal behaviours of methods (e.g., through comparison of the sequences of executed statements that produce certain outputs).

To detect duplicate scenarios, different heuristics in our duplicate detection framework compare all or parts of the information in the scenarios' traces. Our duplicate detection approach is guided by eight different but related hypotheses about the behaviours of duplicate scenarios. The heuristics for the first four hypotheses compare traces produced when each scenario is run exactly once to produce what we hereafter refer to as *default trace*. A default trace of a scenario is produced by running a scenario only once. As explained in hypotheses H1 through H4 in Section 5.4.4, the different heuristics compare different information from the default traces of scenarios to detect semantically equivalent scenarios.

The heuristics for the last four hypotheses detect duplicate scenarios by comparing *essential traces* of scenarios. An essential trace of a scenario consists of information that remain constant across several runs of the scenario. Different information from the essential traces of scenarios, as explained in hypotheses $H1'$ through $H4'$ in

Section 5.4.5, are compared to detect semantically equivalent scenarios. Figure 5.1 summarises the overall duplicate detection framework.

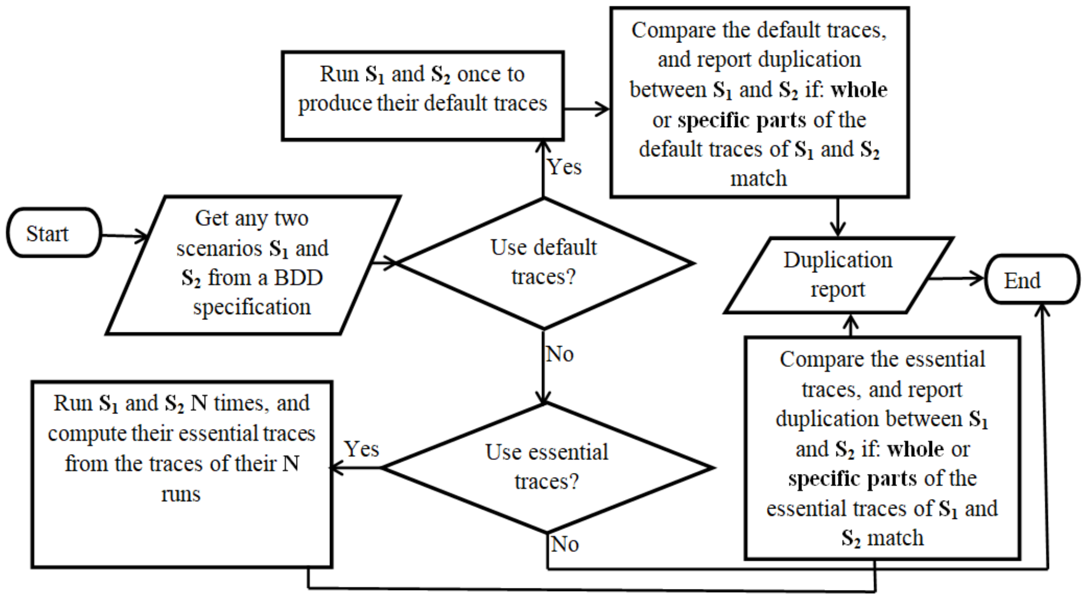

Start
Get any two scenarios S_1 and S_2 from a BDD specification
Use default traces?
Yes
Run S_1 and S_2 once to produce their default traces
Compare the default traces, and report duplication between S_1 and S_2 if: **whole** or **specific parts** of the default traces of S_1 and S_2 match
No
Use essential traces?
Yes
Run S_1 and S_2 N times, and compute their essential traces from the traces of their N runs
No
Compare the essential traces, and report duplication between S_1 and S_2 if: **whole** or **specific parts** of the essential traces of S_1 and S_2 match
Duplication report
End

Figure 5.1: Overall framework for detecting duplicate BDD scenarios

5.4.3 Definitions of Key Terms

Definition 3 *Full trace: A full trace of a scenario is a sequence of all information that are recorded for each production method that is exercised by a scenario.*

Refer to section 5.4.2 for the information we record for each method that is exercised by a particular scenario.

Definition 4 *Default full trace: A default full trace of a scenario is a full trace formed when a scenario is executed only once.*

Definition 5 *Default execution path: A default execution path of a scenario is a subset of the default full trace that is formed by executed statements only.*

Definition 6 *Default API calls: Default API calls of a scenario is a subset of the default full trace that is formed by API calls only.*

We use the information we collect about public methods to form a default trace of public API calls. In particular, since it should generally be possible to identify the API calls made during the execution of a scenario without knowing the actual statements that were executed inside the bodies of methods, we restrict ourselves to the following information when identifying API calls: class name, method name, parameter type(s), return type, parameter value(s), and return value.

Definition 7 *Default API and internal calls: Default API and internal calls of a scenario is a subset of the default full trace that is formed by both API and internal calls.*

We use the information we collect about both public and non-public methods to form a trace of default API and internal calls. As with public API calls, it should generally be possible to identify the internal calls made during the execution of a scenario without knowing the actual statements that were executed inside the bodies of methods. Thus, for each method in the default full trace, the following information, for both public and non-public methods involved in the execution of a scenario, is used to form the trace of API and internal calls: class name, method name, parameter type(s), return type, parameter value(s), and return value.

Definition 8 *Essential full trace: An essential full trace of a scenario is a full trace formed by information that remain constant across several runs of the same scenario.*

Definition 9 *Essential execution path: An essential execution path of a scenario is a subset of the essential full trace that consists of executed statements only.*

Definition 10 *Essential API calls: Essential API calls of a scenario is a subset of the essential full trace that consists of API calls only.*

Definition 11 *Essential API and internal calls: Essential API and internal calls is a subset of the essential full trace that consists of both API calls and internal calls.*

5.4.4 Using Default Traces to Detect Duplicate Scenarios

In order to identify duplicates, we record traces of candidate duplicate scenarios and compare them (traces) at different levels of granularity. These different levels of granularity are:

- Full trace
- Executed statements
- Public API calls

- API and internal calls

We now present the first group of four heuristics in which the duplicate detection algorithms use traces produced when each scenario is run exactly once. In this group of heuristics, we hypothesise that, when two semantically equivalent scenarios have no accidental differences, their traces should be identical, whether the scenarios are run once or several times. Thus, it should, ideally, be possible to detect semantically equivalent scenarios by comparing the traces produced when the scenarios are executed only once.

H1: Any differences in the full traces of two BDD scenarios indicate that they are not duplicates

We begin by assuming that semantically equivalent scenarios produce completely identical traces. In so doing, we wanted to learn how good the comparison of full traces can be at detecting semantically equivalent scenarios. Thus, this heuristic takes into account everything in the full traces when comparing traces for two BDD scenarios to determine if they are duplicates.

H2: Any differences in the execution paths of two BDD scenarios indicate that they are not duplicates

The comparison of default full traces, as in H1, could miss some duplicates. This can be true when the full traces of scenarios being compared differ in unimportant ways. For example, full traces for the two scenarios being compared could only differ in parameter values or return values of some production method(s); and the differences could result from deciding the parameter values in non-deterministic ways at runtime, e.g. when the system time is used as parameter value of a method. But, intuitively, runtime values that change depending on runtime conditions cannot entirely be used to rule out duplication. The two scenarios could still execute the same sequence of statements in the bodies of production methods that are exercised by the two scenarios, suggesting the possibility of duplication between them. Thus, instead of comparing the full traces of the scenarios, the present heuristic only compares the execution paths from the full traces of the scenarios.

H3: Any differences in the public API calls of two BDD scenarios indicate that they are not duplicates

If duplicate scenarios produce the same sequence of API calls, it may be unnecessary to pay attention to what statements are executed inside the bodies of production methods that are involved in the execution of duplicate scenarios. Only the comparison of the sequences of API calls can detect the duplicate scenarios. Thus, apart from the comparison of full traces and execution paths, as described in heuristics H1 and H2, another view on detecting duplicate BDD scenarios is that: *semantically equivalent BDD scenarios perform the same sequence of public API calls on the production code*. So, under this heuristic, the duplicate detection algorithm focuses on only public API calls from the default full trace.

H4: Any differences in the public API and internal calls of two BDD scenarios indicate that they are not duplicates

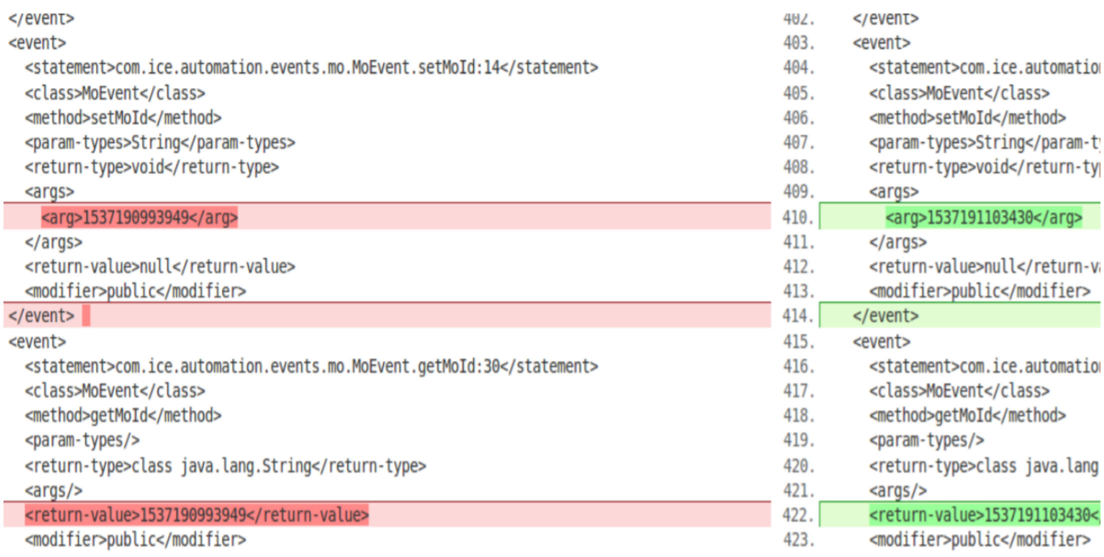
Sometimes, two scenarios with the same sequence of API calls can have important differences in internal calls. In such situations, a duplicate detection approach that focuses only on API calls, while ignoring internal calls, has the risk of increasing the number of false positives, because even scenarios that make different internal calls will be reported as duplicates on the basis of same sequence of API calls. To reduce the number of false positives that are likely to result from such an omission, it may be good to have a duplicate detection approach that combines API calls and internal calls. Thus, instead of considering public API calls only, as in H3, this heuristic advocates for the combination of public API calls and internal calls when comparing traces for two scenarios to determine if they are duplicates. To detect duplication, the algorithm for this heuristic focuses on public API calls and internal calls in the full traces of the scenarios.

5.4.5 Using Essential Traces to Detect Duplicate Scenarios

To avoid false negatives caused by things that change in each run of a scenario, several runs of a scenario are recorded and only things that stay the same are compared. This is motivated by the fact that a duplicate detection approach that performs naive comparison of whole or specific parts of default traces—without discriminating between essential and accidental parts of those traces, as in heuristics H1 through H4 in section 5.4.4—could miss some duplicate scenarios. This is possible when the differences in the traces being compared are only accidental. Thus, for the next group of four heuristics, each BDD scenario is run several times to produce multiple traces; and the traces for the different runs of a scenario are analysed to identify constant information across runs. It is this information that remains constant across several runs that forms an essential trace of a scenario. We first introduce how to determine an essential trace of a scenario, and thereafter present the heuristics that use essential traces to reason about the behaviours of semantically equivalent BDD scenarios.

Determining an Essential Trace of a Scenario

Figure 5.2 shows a part of traces for two runs of the same scenario (taken from our evaluation benchmark, described in section 4.3). The variation of some information in the traces for the two runs was caused by the differences in runtime conditions. A method *setMoId()* in the class *MoEvent* takes the system time in milliseconds as its argument, and that argument acts as a return value of the method *getMoId()*. This variation in two traces of the same scenario, caused by non-determinism in the method's parameter and/or return values, is an example of accidental differences that could manifest in the traces of semantically equivalent BDD scenarios.



```
</event>
<event>
  <statement>com.ice.automation.events.mo.MoEvent.setMoId:14</statement>
  <class>MoEvent</class>
  <method>setMoId</method>
  <param-types>String</param-types>
  <return-type>void</return-type>
  <args>
    <arg>1537190993949</arg>
  </args>
  <return-value>null</return-value>
  <modifier>public</modifier>
</event>
<event>
  <statement>com.ice.automation.events.mo.MoEvent.getMoId:30</statement>
  <class>MoEvent</class>
  <method>getMoId</method>
  <param-types/>
  <return-type>class java.lang.String</return-type>
  <args/>
  <return-value>1537190993949</return-value>
  <modifier>public</modifier>
```

```
</event>
<event>
  <statement>com.ice.automatio
  <class>MoEvent</class>
  <method>setMoId</method>
  <param-types>String</param-t
  <return-type>void</return-ty
  <args>
    <arg>1537191103430</arg>
  </args>
  <return-value>null</return-v
  <modifier>public</modifier>
</event>
<event>
  <statement>com.ice.automatio
  <class>MoEvent</class>
  <method>getMoId</method>
  <param-types/>
  <return-type>class java.lang
  <args/>
  <return-value>1537191103430<
  <modifier>public</modifier>
```

Figure 5.2: Example of variations in the traces of two runs of the same scenario

Moreover, an execution path of a BDD scenario can also vary, depending on the conditions at the time of execution. For example, one set of statements in the production code can be executed when one condition is true, while another set of production statements can be executed when another condition is true and, still, another set of production statements can be executed in any condition. This is especially possible if the production code involved in the execution of a scenario has a number of branching conditions. This kind of variation in the set of executed statements suggests that essential statements and accidental statements are a possibility for a BDD scenario. Statements that execute whenever a scenario is executed, irrespective of runtime conditions, can be regarded as essential to the behaviour of a scenario, while statements whose execution is affected by runtime conditions can be regarded as accidental to the behaviour of a scenario.

Thus, if we have two semantically equivalent BDD scenarios whose traces have accidental differences, it can be difficult to detect them as a duplicate pair, particularly when the duplicate detection approach performs comparison of default traces. As such, one way to make the traces for two scenarios comparable, for purposes of duplicate detection, is to identify, in the traces for the two scenarios, parts that remain constant across across several runs of each scenario.

Identifying the accidental parts in the trace of a scenario would suggest the need for it to be run several times, possibly under different conditions. But how many runs are enough to reveal the accidental parts in the trace of a scenario? There is no finite limit to the number of runs required to eliminate the incidental differences, but a practical limit needs to be identified. For purposes of experimentation in this work, five runs were chosen, after experimenting with several scenarios from our evaluation benchmark and learning that no new behaviours were observed in the scenarios' traces produced after the fifth run. So it was judged that five runs can give an approximately good picture of the possible variation points observed in the execution traces when the same scenario is run several times under different conditions (i.e. different system times). Thus, to produce an essential trace of a scenario that can be compared with the traces of other scenarios for purposes of duplicate detection, we follow the following three steps:

1. Run a scenario five times, collecting traces for the different runs.
2. Compare traces for the different runs of a scenario, to identify parts that vary across several runs as well as parts that remain constant across several runs.

Specifically, we compute diffs between traces for the different runs, to identify trace parts that vary across several runs, and trace parts that remain constant across several runs.

3. Regard as essential trace parts that remain constant across several runs of a scenario.

Heuristics on Essential Traces

After explaining the concept of an essential trace of a BDD scenario and how it is computed, we next present the heuristics that exploit essential traces to detect duplicate BDD scenarios. The comparison of essential traces is likely to do better at detecting duplicate scenarios than the comparison of default traces because accidental parts of the scenarios' traces are excluded from the comparison.

H1': Any differences in the essential full traces of two BDD scenarios indicate that they are not duplicates

In the present heuristic, we suppose that *semantically equivalent BDD scenarios produce the same essential full traces*. If the full traces for two duplicate scenarios have accidental differences, the comparison of default full traces, as in H1, can miss the duplicates. However, focusing on essential full traces only can increase the duplicate detection ability because accidental trace differences are excluded from the comparison.

H2': Any differences in the essential execution paths of two BDD scenarios indicate that they are not duplicates

If the default execution paths for duplicate scenarios have accidental differences, the comparison of default execution paths, as in H2, can fail to detect the duplicate scenarios. But, if the duplicate detection approach involves comparison of only the essential execution paths, the chances of detecting the duplicate scenarios increase because accidental parts of the execution paths of the scenarios are excluded from the comparison. Hence, focusing only on executed statements in an essential trace of a scenario, a possible perspective would be that *semantically equivalent BDD scenarios have the same* *essential execution paths*.

H3': Any differences in the essential API calls of two BDD scenarios indicate that they are not duplicates

If the default public API calls for duplicate scenarios have accidental differences, it is possible for the duplicate scenarios to go undetected, if comparison is done on default public API calls, like in H3. To increase the duplicate detection ability, it can be good to focus only on essential public API calls, which contain no accidental differences. So, focusing only on trace information about public API calls of a scenario, under the present heuristic, we propose that *semantically equivalent BDD scenarios exercise the production code in the same way by making the same essential public API calls*.

H4': Any differences in the essential API and internal calls of two BDD scenarios indicate that they are not duplicates

If default API and internal calls have accidental differences, the comparison of default API and internal calls can fail to detect duplicate scenarios. However, the focus on essential API and internal calls has the potential of increasing the ability to detect duplicate scenarios because accidental differences are excluded from comparison. Thus, focusing only on essential API and internal calls, we suggest that *semantically equivalent BDD scenarios make the same essential public API and internal calls on the production code*.

5.4.6 Algorithm and Implementation

We now present the duplicate detection algorithm that combines the 8 algorithms for the 8 different heuristics introduced in Section 5.4.4 and Section 5.4.5. We also give a summary of how an algorithm was implemented.

Algorithm

Algorithm 1 combines the heuristics for both default and essential traces. It is important to note that, for purposes of experimentation in this work, each heuristic was tested separately and the results were generated per heuristic, to allow for comparison of results for the different heuristics. But Algorithm 1 combines the algorithms for the different heuristics because it is assumed that, in real world, if a duplicate pair will not be detected by an algorithm for one heuristic, it would be detected by an algorithm for another heuristic. This removes a costly process of subjecting all candidate duplicates in a suite on each of the 8 heuristics.

Input description: Algorithm 1 takes the following three inputs:

- A BDD suite, hereafter referred to as Σ. For practical purposes, a complete BDD suite consists of a collection of scenarios, the production code for the System Under Test, and the glue code which connects the scenarios in a suite to the System Under Test.
- A set T of default full traces of the scenarios in Σ. As described in section 5.4.4, each scenario in Σ is executed once, producing a default trace, and this trace becomes a member of T. Thus, T is formed by the collection of default full traces of the scenarios in Σ. The information we record as part of the default full trace for each scenario is mentioned in section 5.4.2.
- A set E of essential full traces of the scenarios in Σ. As described in section 5.4.5, an essential trace of a scenario is formed by parts of the trace that remain constant across several runs. Thus, E is formed by the collection of essential full traces of the scenarios in Σ.

Process description: The following functions manipulate the inputs to produce a report of duplicate scenarios

- Given T and scenario name s as inputs, a function $defaultFullTrace$ searches T and returns from T the default full trace of s.
- Given E and scenario name s as inputs, a function $essentialFullTrace$ searches E and returns from E the essential full trace of s.
- Given two traces, t_1 (for scenario s_1) and t_2 (for scenario s_2), a functionn *equalOrSubseq(Trace t_1, Trace t_2)* checks for the equality or subsumption of one trace by another. If t_1 (for scenario s_1) and t_2 (for scenario s_2) are equal or one trace is subsumed by the other, then the two scenarios are considered to be semantically equivalent, and the information about the detected duplicate pair is added (using the *append* function) into the set *Pairs* that stores information about the

duplicate pairs of scenarios detected by the algorithm. In practice, the subsumption of one trace by another would represent a case in which the functionality represented by one scenario is subsumed by another scenario (for example, in Listing 5.1, the last scenario subsumes the first two scenarios). Further, as can be seen in Algorithm 1, the function *equalOrSubseq(Trace t_1, Trace t_2)* can take full traces or specific parts of full traces. To extract specific parts of a default full trace or an essential full trace, three different functions (*stmts(FullTrace t)*, *api(FullTrace t)*, and *apiAndInternal(FullTrace t)*) are used. The three functions respectively extract statements (representing execution paths), public API calls, and public API and internal calls, from either default full traces (t and t') or essential full traces (e and e').

Output description: Algorithm 1 produces a collection of pairs of duplicate scenarios detected from a suite Σ.

Asymptotically, Algorithm 1 is of $\theta(n^2) complexity.$

Implementation

Several BDD tools and language variants exist. We constructed our tool to work with BDD feature files written in the Gherkin language, currently the most widely used BDD language [195, 28], with glue code written in Java, using Cucumber-JVM conventions[4]. However, since we depend mainly on the ability to generate traces of selected classes and methods, the tool is easily adaptable to work with other BDD engines and languages whose program execution can be traced.

Each scenario is executed individually, and AspectJ[5] is used to generate trace information for the methods of the code under analysis. Other tools such as debuggers and profilers might be used for this task, but AspectJ was chosen based on the convenience and familiarity to the researcher, in addition to its suitability for the task as hand. The trace is produced as an XML file, which allows us to use XQuery [196] to search for duplicates. Since BDD scenarios typically number in the tens or hundreds [28], we can match the trace of one scenario against every other trace in order to search for duplicates, and still manage to get the results in reasonable time scales. Figure B.1 shows an example duplication report for some duplicate scenarios in evaluation System 1.

[4]`cucumber.io/docs/reference/jvm`
[5]`www.eclipse.org/aspectj`

Algorithm 1: Detecting semantically equivalent scenarios in BDD specifications

```
Input: A BDD suite $\Sigma$
       A set $T$ of default full traces of the scenarios in $\Sigma$
       A set $E$ of essential full traces of the scenarios in $\Sigma$
Output: Semantic duplicate scenario pairs
1  $Pairs \leftarrow \{\}$
2  foreach scenario $s$ in $\Sigma$ do
3      $t \leftarrow defaultFullTrace(T, s)$
4      $e \leftarrow essentialFullTrace(E, s)$
5      foreach scenario $s'$ in $\Sigma$ (where $s' \neq s$) do
6          $t' \leftarrow defaultFullTrace(T, s')$
7          $e' \leftarrow essentialFullTrace(E, s')$
8          if $equalOrSubseq(t, t')$ then
9              $Pairs.append(\{s, s'\})$
10         else if $equalOrSubseq(stmts(t), stmts(t'))$ then
11             $Pairs \leftarrow (\{s, s'\})$
12         else if $equalOrSubseq(api(t), api(t'))$ then
13             $Pairs.append(\{s, s'\})$
14         else if $equalOrSubseq(apiAndInternal(t), apiAndInternal(t'))$ then
15             $Pairs.append(\{s, s'\})$
16         else if $equalOrSubseq(e, e')$ then
17             $Pairs.append(\{s, s'\})$
18         else if $equalOrSubseq(stmts(e), stmts(e'))$ then
19             $Pairs.append(\{s, s'\})$
20         else if $equalOrSubseq(api(e), api(e'))$ then
21             $Pairs.append(\{s, s'\})$
22         else if $equalOrSubseq(apiAndInternal(e), apiAndInternal(e'))$ then
23             $Pairs.append(\{s, s'\})$
24         else
25             report nothing because $(\{s, s'\})$ is not a duplicate pair
26 return $Pairs$
```

Finally, since we store the traces for the scenarios in XML files, we leveraged the capabilities of the XMLUnit API[6] to compute the differences between the traces for several runs of a scenario. Identifying the differences between the traces for several runs of a scenario helps locate parts that vary in a trace of a scenario, as well as parts that remain constant across several runs. As described in section 5.4.5, an essential trace of a scenario is formed by trace parts that remain constant across several runs of

[6]https://www.xmlunit.org/api/java/master/index.html

a scenario.

5.5 Evaluation

In this section, we describe the design of the evaluation experiment we performed, present the results and discuss their significance.

We conducted an evaluation to test the hypothesis that comparing how scenarios exercise the production code can detect semantically equivalent BDD scenarios in a way that outperforms existing duplicate detection approaches. In particular, we wanted to provide answers to the following research questions:

> **RQ1:** How effective is each heuristic in detecting duplicate BDD scenarios, and which heuristics perform better than others?
>
> **RQ2:** Does the focus on essential information in the traces of BDD scenarios improve the detection of duplicate BDD scenarios?
>
> **RQ3:** How does the tool developed in the present study compare with existing duplicate detection tools?

5.5.1 Experiment Design

We used the benchmark of known duplicate pairs across the three evaluation systems (section 4.3) to test the ability of our tool to detect these duplicates and only these duplicates. As previously stated, we probably have to tolerate the fact that our tool produces some false positive results, given the difficulty of finding a definition of semantic duplication that can give accurate results in all cases. However, we would prefer the number of false positive results to be low, as each one represents a cost to the developer using the tool (in terms of the time required to investigate and reject the candidate duplicate). We can tolerate false negatives, too, to an extent, again given the difficulty of finding a definition of semantic duplication that can give accurate results in all cases. Having a way to find *some* duplicates in BDD specifications is still useful, even if others are left undetected. But, again, we would like the number of false negative matches to be low, to avoid the risk of important (costly) duplicates being missed.

However, as is well known, it is not usually possible to determine the number of false negatives using a pure injection approach due to the size of the data set. In our case, the BDD specifications we used for evaluation were comparatively small. But, as has been described in section 4.3.4, the people who volunteered to inject duplicates into our evaluation systems were not from the development team of the three systems, and so had no specific expertise in the intended meaning of the existing scenarios. We therefore had no reliable oracle that we could ask to provide information on known non-duplicates within the evaluation systems. Thus, it was necessary to assume that the only duplicates in the system were those developed in section 4.3. To obtain information about the false negative results, we can only ask how many of the known duplicates were *not* detected by our tool, and to set this as a lower bound on the actual number of false negatives. This is a normal practice in an area of software engineering that focuses on detecting clones in software systems [4, 37].

Apart from evaluating the ability of our tool to detect injected duplicates (Table 4.4), we also evaluated its ability to detect duplicates between original scenarios (Table 4.2). Thus, the same benchmark used to evaluate existing duplicate detection tools (Section 5.3) was used for this particular experiment. This, in turn, enabled us to compare our tool with existing duplicate detection tools on the problem of detecting semantically equivalent BDD scenarios.

A good performing heuristic should be able to detect as many known duplicate scenario pairs as possible from our benchmark. As well, the focus on essential traces of scenarios will be said to increase the duplicate detection ability if there exist duplicate pairs that are detected by a heuristic on essential traces but are not detected by a corresponding heuristic on default traces.

5.5.2 Results

We now present the results of the evaluation experiment, showing the detailed results as well as their summary through a plot of recalls, precisions and F-score.

Table 5.2 presents the results for the 8 hypotheses on the three systems. Other columns should be self-explanatory, except for the following:

- **Inj.** represents the number of known duplicate scenario pairs, obtained through injection.

- **Det.** represents the number of detected duplicate scenario pairs, out of the known ones (Inj.).
- **FN** represents the number of known duplicate pairs, injected in the system, that were not detected. FN is the lower bound of False Negatives because a particular system could have other duplicate pairs of scenarios that were not detected by our algorithm and were not part of our benchmark due to our inability to identify them. Thus, FN is constituted by only the known duplicates that went undetected, not all the duplicates that went undetected.
- **CS** represents the size of the candidate set, which is the number of scenario pairs that were reported as duplicates by an algorithm for a particular hypothesis (including those which were not in our set of injected pairs).
- Columns **OO-TP**, **OO-FP**, **IO-TP**, **IO-FP**, **II-TP** and **II-FP** show the distribution of the members of the candidate set for a particular heuristic. For example, column **OO** represents the number of members of a candidate set that report duplication between original scenarios (scenarios that existed before volunteers injected duplicates). These scenario pairs can either be True Positives (TP), classified as OO-TP; or False Positives (FP), classified as OO-FP.
- IO and II represent duplication between original and injected scenarios, and duplication between injected scenarios, respectively—as explained in section 4.3.4.

We manually inspected every pair in the candidate set to put it in a particular category.

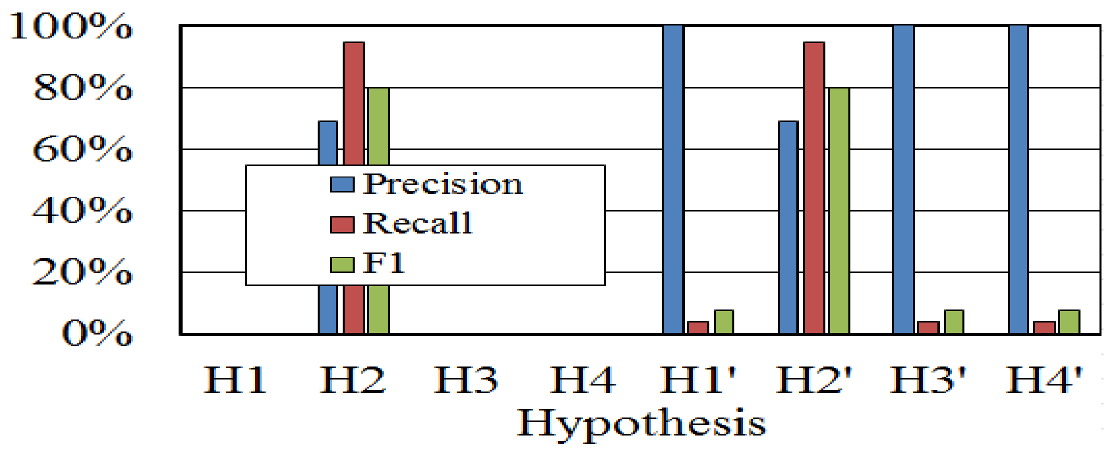
The charts in Figure 5.3 plot precision, recall and F-score for the results of the algorithms for each of the 8 hypotheses on the 3 systems. Recall is computed as the ratio of number of detected duplicate pairs (Det. in Table 5.2) to the number of known duplicate pairs (Inj. in Table 5.2). Precision is computed as the ratio of the number of True Positives (sum of OO-TP, IO-TP, and II-TP in Table 5.2) to the number of pairs in the candidate set (CS in Table 5.2).

Table 5.3 provides the comparison between our tool (SEED2) and the existing duplicate detection tools, on a benchmark of duplicates between original scenarios (Table 4.2). With reference to the column labels in Table 5.3, Recall is computed as the ratio of Detected Known Duplicates to Known Duplicates, while Precision is computed as the ratio of Duplicates Between Whole BDD Scenarios to Candidates. Because H2 and H2' are the best performing heuristics of our duplicate detection framework, and

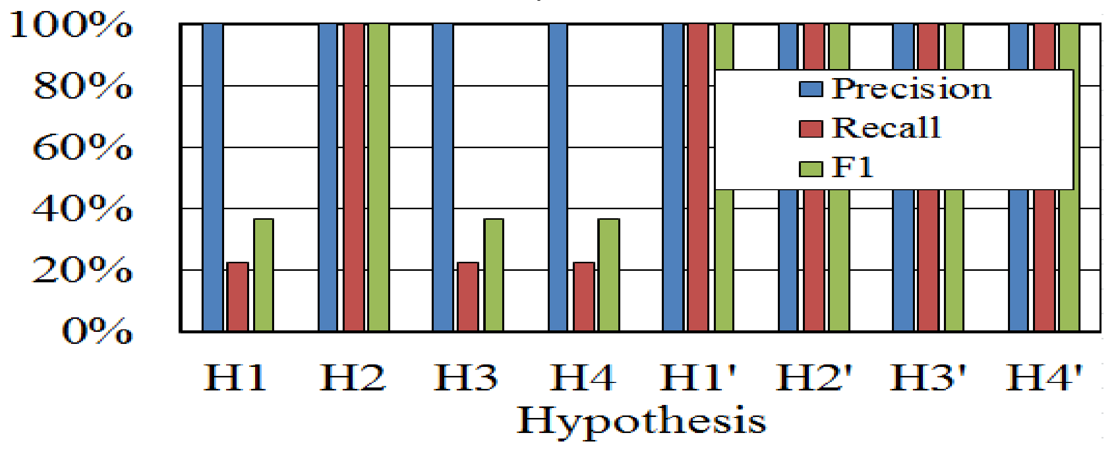
Hyp.	Syst.	Inj.	Det.	FN	CS	OO-TP	OO-FP	IO-TP	IO-FP	II-TP	II-FP
H1	1	75	0	75	0	0	0	0	0	0	0
	2	40	9	31	9	0	0	5	0	4	0
	3	10	0	10	0	0	0	0	0	0	0
H2	1	75	71	4	126	16	2	48	15	23	22
	2	40	40	0	40	0	0	23	0	17	0
	3	10	6	4	6	0	0	4	0	2	0
H3	1	75	0	75	0	0	0	0	0	0	0
	2	40	9	31	9	0	0	5	0	4	0
	3	10	0	10	0	0	0	0	0	0	0
H4	1	75	0	75	0	0	0	0		0	0
	2	40	9	31	9	0	0	5	0	4	0
	3	10	0	10	0	0	0	0	0	0	0
H1'	1	75	3	72	3	0	0	2	0	1	0
	2	40	40	0	40	0	0	23	0	17	0
	3	10	0	10	0	0	0	0	0	0	0
H2'	1	75	71	4	126	16	2	48	15	23	22
	2	40	40	0	40	0	0	23	0	17	0
	3	10	6	4	6	0	0	4	0	2	0
H3'	1	75	3	72	3	0	0	2	0	1	0
	2	40	40	0	40	0	0	23	0	17	0
	3	10	0	10	0	0	0	0	0	0	0
H4'	1	75	3	72	3	0	0	2	0	1	0
	2	40	40	0	40	0	0	23	0	17	0
	3	10	0	10	0	0	0	0	0	0	0

Table 5.2: Detailed results for the 8 different hypotheses on the 3 systems

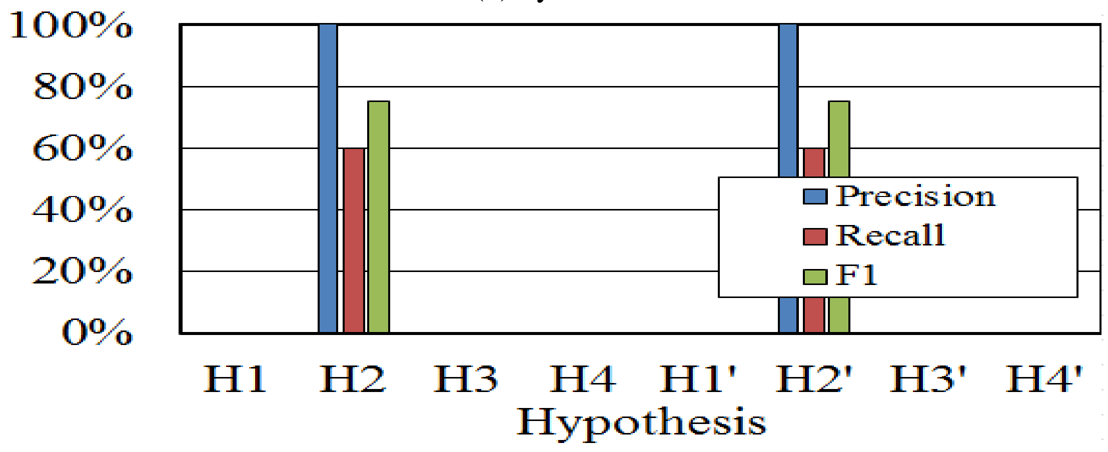
both H2 and H2' had equal performance during our evaluation (Table 5.2), we used the results of H2 when comparing the performance of our tool with the performance of the existing duplicate detection tools.



(a) System 1



(b) System 2



(c) System 3

Figure 5.3: Plot of precision, recall, and F1 score for the algorithms of 8 different hypotheses across the 3 systems

We used the results for H results we used to compare SEED2 with the existing tools

System	Tool	Known Duplicates	Detected Known Duplicates	Candidates	Duplicates Between Whole BDD Scenarios	Recall (%)	Precision (%)
1	PMD	9	0	12	0	0.00	0.00
	CloneDR	9	1	6	1	11.11	16.67
	DECKARD	9	9	589	112	100.00	19.02
	SEED2	**9**	**9**	**18**	**16**	**100.00**	**88.89**
2	PMD	1	0	7	0	0.00	0.00
	CloneDR	1	0	1	0	0.00	0.00
	DECKARD	1	0	43	0	0.00	0.00
	SEED2	**1**	**0**	**0**	**0**	**0.00**	**0.00**
3	PMD	3	0	8	0	0.00	0.00
	CloneDR	3	0	1	0	0.00	0.00
	DECKARD	3	1	951	48	33.33	5.05
	SEED2	**3**	**0**	**0**	**0**	**0.00**	**0.00**

Table 5.3: Comparison between our tool and the existing duplicate detection tools

5.5.3 Discussion

RQ1: On the effectiveness of each heuristic, and the identification of a best performing heuristic: The results for each heuristic are shown in Table 5.2 and Figure 5.3. Generally, the algorithms for H2 and H2' performed much better, on all the systems, compared to the algorithms for the rest of the heuristics. Based on these results, the comparison of execution paths (heuristics H2 and H2') of BDD scenarios revealed more duplicates across the three systems, than the comparison of full traces, API calls, or the combination of API and internal calls. This was so irrespective of whether or not the duplicate detection algorithm used default or essential traces of scenarios. More specifically, across the three systems, the comparison of both default and essential execution paths (H2 and H2') detected exactly the same duplicate pairs of scenarios. This could suggest that all of our evaluation systems had no runtime conditions that affected the choice of execution paths of the scenarios in the known duplicate pairs.

Additionally, the algorithms that compared execution paths also detected duplicate pairs of BDD scenarios that were not known to us beforehand. This was especially true for heuristics H2 and H2'on System 1 where 16 pairs of duplicate scenarios were reported between original scenarios, and confirmed to be true positives (see Table 5.2). 7 out of 16 (OO-TP) pairs of duplicates between original scenarios were not known to us beforehand; only 9 of them were known to us beforehand from Table 4.2.

Thus, in general, the comparison of execution paths gives better detection of semantically equivalent BDD scenarios.

Answer to RQ1: *Each heuristic can detect some duplicates, but the comparison of execution paths of scenarios is the most promising approach to detecting semantically equivalent BDD scenarios.*

RQ2: On whether the focus on essential traces of scenarios improves duplicate detection: From the results in Table 5.2 and Figure 5.3, when attempting to detect duplicate BDD scenarios, the focus on essential parts of full traces, API calls, or the combination of API and internal calls, can improve the duplicate detection capability. This can be seen by comparing H1 and H1', H3 and H3', and H4 and H4', for systems 1 and 2 in Figure 5.3. For system 1, in particular, in each of these comparable cases, the recall value improved to a relatively small extent. Nevertheless, the precision value in each of these comparable cases improved to 100%, meaning that each of the extra duplicate pair reported by the algorithms focusing on essential traces was a true positive. For system 2, there was a dramatic increase in all the three metrics—recall, precision and F1 score—for the three comparable cases: H1 and H1', H3 and H3', and H4 and H4' (refer to Figure 5.3). So, from these results, apart from execution paths, the comparison of essential traces of full traces, public API calls, or a combination of public API and internal calls, can also be used when relatively higher values of both recall and precision are preferred. For System 3, however, there was no evidence that the focus on essential traces improved the detection of semantically equivalent BDD scenarios.

Answer to RQ2: *The focus on essential traces of scenarios can improve the duplicate detection capability.*

RQ3: On how the tool developed in the present study compare with existing duplicate detection tools: From the results in Table 5.3, our tool has improved precision (88.89%) on System 1, compared to the precision of DECKARD (19.02%), the best performing tool out the existing duplicate detection tools we experimented with. However, similar to the existing tools, our tool, too, did not perform better on System 2 and System 3. This could partly be attributable to the size of the benchmarks of known duplicates between original scenarios of the two Systems (1 for System 2 and 3 for System 3), in addition to scenario/project-specific attributes. Thus, in the future, it would be good to test all the tools on reasonably large benchmarks of known duplicates between original scenarios, to ascertain their true abilities.

Answer to RQ3: *The tool developed in the present study recorded better precision than existing tools on a benchmark of pairs of known duplicates between original scenarios.*

5.6 Summary

Large BDD suites can be difficult to manage, especially when different parts of the same suites have been developed by different teams and/or people, probably over different periods of time. The specifications can contain duplication that can make them difficult to comprehend, maintain and extend.

In this chapter, we analysed the problem of detecting such duplication, discussed the limitations of existing tools for the problem, and have described our approach for detecting duplicate BDD scenarios. We have proposed to detect semantically equivalent BDD scenarios based on the way the production code is exercised by the individual scenarios. We analysed execution traces of methods in the production code to find duplication candidates. We evaluated the approach by asking 13 volunteers to inject duplicate scenarios into three open source software systems, forming the ground truth; and compared the ground truth with the duplicate candidates that our tool produced. The comparison of execution paths recorded better values of recall, precision and F-score, across the 3 evaluation systems, outperforming the comparison of full traces, public API calls, and a combination of public API calls and internal calls. Thus, on the basis of these results, comparison of execution paths is the most promising way to detect duplicate BDD scenarios.

To conclude, execution traces can help us find some duplicate scenarios in BDD specifications, but an appropriate abstraction over the trace is needed to give strong results in terms of precision and recall. However, individual attributes of projects can get in the way, leading to unforeseeable results. For example, the design and implementation details of individual systems could influence the duplicate scenarios that get detected by a generic approach.

Chapter 6

Quality-Oriented Removal of Duplicate Scenarios in BDD Suites

6.1 Introduction

In Chapter 5, we presented an approach to detect duplicate scenarios in BDD feature suites. In the present chapter, we present an approach to guide the removal of duplicate scenarios after they are detected. In particular, we want to explore the aspects of scenarios that can help us decide what scenario to remove, given a pair of duplicate BDD scenarios.

After pairs of duplicate scenarios are reported by duplicate detection tools, different strategies can be employed to manage the detected duplicate scenarios, just like it is done in managing duplicates that get detected in program code [74, 197]. Like the removal of duplicates in program code, among the possible actions for the removal of duplicate BDD scenarios are merging duplicate scenarios into one scenario or deleting one of the scenarios in a duplicate pair while keeping the other. Given a pair of duplicate scenarios, in case the decision is to delete one scenario and keep the other, the maintainer of the BDD suite is still left with the challenge of deciding which of the duplicated scenarios should be removed. Naively, since they are duplicates, one might say that either scenario in a pair could be removed without changing the semantics of the suite. In reality, however, some scenarios may be of better quality than others, while expressing the same example behaviour. Removing one of the scenarios in the pair might increase the quality of the suite as a whole, while removing the other might reduce it.

BDD suite quality has been heavily discussed amongst practitioners [152], but as yet no mature formal substantiated definitions have been given. Consequently, there are no formal quality guidelines that can help maintainers in removing duplicate scenarios from BDD feature suites. Existing studies on quality of BDD scenarios [198, 199, 134] are still in their infancy, and the metrics they provide can support the evaluation of the quality of individual scenarios within a feature suite. They, however, do not offer mechanisms to evaluate the quality of a scenario relative to the whole feature suite to which a scenario belongs. For example, both scenarios A and B from a particular feature suite can be generally poor according to the quality attributes by Oliveira *et al.* [134], but still scenario A can be preferable to scenario B when each is evaluated relative to the whole feature suite to which scenarios A and B belong. As such, faced with a need to make a decision on which scenario to remove, it would be preferable for the maintainer to remove B and keep A. We thus need a set of metrics to support this relative quality assessment, and the study we present in the present chapter set out to devise and evaluate the performance of such metrics on the problem of deciding which scenario to remove, given a pair of semantically equivalent BDD scenarios.

To understand how quality properties of a suite might be used to guide the removal of duplicate scenarios, we begin the assumption that:

> **H:** We can use quality aspects of scenarios in a BDD suite to guide the removal of duplicate scenarios from a suite.

To confirm or refute this hypothesis, we posed three research questions, which are answered in this chapter:

> **RQ1:** What are the properties of scenarios that affect the quality of a feature suite as a whole?
>
> **RQ2:** How are the quality properties in RQ1 perceived in practice?
>
> **RQ3:** To what extent are the scenarios' quality properties successful in guiding the removal of duplicate scenarios from the feature suites?

To answer these questions, it is necessary to have a clear idea of what is meant by quality of a scenario relative to the whole BDD feature suite. The attempts to define BDD quality in the literature can only facilitate the assessment of quality of an individual scenario in a suite, and not for assessing the quality of a scenario in relation to all other scenarios in a suite. We propose four principles describing BDD suite quality. Each of the proposed principles focuses on a particular property of BDD suites whereby, to increase the quality of a suite as a whole, some of the properties have to be maximised while some have to be minimised. The first principle encourages reuse of step phrases across scenarios in a feature suite. The second principle encourages reuse of domain vocabulary across scenarios in a feature suite. This is inline with the agile practice of *Ubiquitous Language* [200] and the *ubiquitous* attribute in the list of quality attributes for BDD scenarios as proposed by Oliveira *et al.* [134]. The third principle discourages the use of implementation level terms that, in most cases, only members of the development team can understand, when, in principle, a BDD specification should be understood by all project stakeholders, including those outside the development team. The fourth principle encourages consistency of abstraction level when expressing different scenarios of a particular feature suite.

We surveyed the community of BDD practitioners to gauge the level of support for the principles. All principles received support from the community (with at least 75% of respondents voting in support of each principle), though all of them also received a small number of dissenting votes. We also report on practitioners' opinions on other BDD quality aspects not covered by the four principles.

We further demonstrate how the proposed principles can be used to guide maintainers of BDD suites in removing duplicate scenarios from the suites. During the evaluation of the ability of the principles to guide the removal of duplicate scenarios, each of the four principles gave a reasonable number of acceptable remove suggestions. Thus, the results suggest that the four principles can be used to assess which scenario in a pair of duplicate scenarios can be most effectively removed.

This chapter makes the following contributions:

- **BDD Suite Quality Principles:** We propose four principles describing features expected of a high quality BDD specification.
- **Practitioner Support for the BDD Suite Quality Principles, and Other Quality Facets of BDD Suites:** We report the results of a survey of practitioner support for the principles. We also report on practitioners' opinions on other quality aspects of BDD suites that are not covered by the four principles.
- **Operationalization of Principles:** We propose an approach to operationalize each principle, so that BDD specifications can be assessed automatically against

it.

- **Use of Principles in Generating Refactoring Advice:** We use the operationalized principles to propose removal of duplicate scenarios, and evaluate the approach on 3 open source software systems. The results from both the lab experiment and the experiment with a BDD practitioner from the industry suggest that the four principles can be used to support quality-preserving removal of duplicate scenarios from BDD feature suites.

The rest of this chapter is structured as follows: section 6.2 presents the quality principles we use to guide the removal of duplicates in BDD suites; section 6.3 presents practitioners' opinions about the proposed BDD quality principles, as well practitioners' opinions on other aspects of BDD suite quality not covered by the four principles; section 6.4 describes our operationalisation of the principles; section 6.5 presents the empirical support we gathered for the principles; section 6.6 presents the threats to the validity of the results for the work described in the present chapter; and section 6.7 concludes the chapter.

6.2 BDD Suite Quality Principles

Given a pair of duplicate scenarios, we need a set of criteria to decide which scenario to keep and which scenario to remove. The mere fact that two scenarios are duplicates does not mean that deleting either of them will have the same effect on the overall quality of the BDD suite. One of the duplicated scenarios may be of higher quality than the other. In order to advise owners of a BDD suite regarding which of the duplicate scenarios to remove we need a definition of "good quality" for BDD suites that is precise enough to be used by an automated tool. While there is a lot of discussion about BDD quality (and some well known rules of thumb) [152], we could find no validated precise definition that would be suitable for helping in the removal of duplicate scenarios. Oliveira *et al.* [134] attempted to define the characteristics desirable in a good BDD scenario, but those characteristics are not precise enough to facilitate the assessment of one scenario in relation all others in a suite. We set about defining a number of BDD quality principles, based on our own experience and understanding of writing BDD suites, informed by the opinions of practitioners as reported in blogs and and websites.

In this section, we first present the process used to produce the principles, and then we describe the four principles in their general form. In section 6.4, we will operationalise the principles, to give versions that can be used in a refactoring advisor tool.

6.2.1 Aspects of Quality in BDD Specifications

We now detail the process used to produce the four BDD suite quality principles.

To understand what constitutes good quality in BDD suites, we first searched the literature for attempts to define quality in BDD specifications. This gave us only the work of Oliveira *et al.* which suggested that good BDD scenarios should be *essential*, *focused*, *singular*, *clear*, *complete*, *unique*, *ubiquitous*, and *integrous* [134, 198, 199]. Refer to section 2.5 for the description of each of these attributes. However, these attributes define, in more general terms, the characteristics expected of a good scenario, but are not precise enough to facilitate the assessment of the quality of one scenario in relation to all other scenarios in a suite. Thus, given a pair of duplicate scenarios in which both scenarios evaluate to "good" (for example) based on the quality attributes by Oliveira *et al.* [134], the maintainer wanting to remove one scenario and keep the other would still face the challenge of deciding which scenario to remove. Removing one scenario may preserve the overall quality of a BDD feature suite, while removing the other may impair it. We thus need more precise definitions of BDD quality that can be used to assess the quality of a scenario relative to all other scenarios across a feature suite. Such definitions would support the process of making broadly informed decisions when removing duplicate scenarios from a BDD feature suite.

To obtain attributes that are suitable for assessing the quality of a scenario relative to all other scenarios across a feature suite, we borrowed ideas from the quality attributes in the work of Oliveira *et al.* [134] and complemented these ideas with other practitioners' opinions on quality in BDD feature suites. To obtain practitioners' opinions on quality in BDD feature suites, we analysed articles from the BDD Addict Newsletter [152], a monthly online newsletter about BDD, which publishes articles about various aspects of BDD from the perspective of BDD practitioners. Articles from 32 issues of the newsletter (from February 2016, when the first issue was released, to December

2018) were analysed for quality facets in BDD suites. We then searched StackOverflow[1] and StackExchange[2] for any additional BDD quality facets that might not have been covered in the BDD Addict Newsletter. Table 6.1 summarises the quality facets we obtained through this process.

S/n	**Quality Aspect**
1	A good quality scenario should be concise, testable, understandable, unambiguous, complete, and valuable
2	Reuse of steps across scenarios can improve suite quality
3	Declarative (high level) steps are preferred to imperative (low level) steps
4	Business terminology should be consistently used across the specification
5	Scenarios should focus on the benefit they offer to users, if implemented
6	Scenarios should use the terminology understood by all project stakeholders
7	Each scenario should test one thing
8	Scenario titles should be clear
9	Scenario descriptions should be focused
10	Personal pronoun “I” should be avoided in steps
11	Too obvious and obsolete scenarios should be avoided in the suite
12	Scenario outlines should be used sparingly
13	Scenarios should clearly separate Given, When and Then steps
14	Use past tense for contexts (Given), present tense for events (When), and “should” for outcomes (Then)

Table 6.1: BDD quality aspects from scientific and grey literature

But, in the interest of time, we could not produce precise and automatable definitions of BDD quality from all the quality facets by Oliveira *et al.* [134] as well as the quality facets in Table 6.1, for use in the removal of duplicate scenarios. Hence, we wanted a subset of these quality facets, which is concrete enough for use in guiding the removal of duplicate BDD scenarios from the suites. We, therefore, conducted a thought experiment in which we attempted to apply the quality facets in Table 6.1 on the problem of deciding which scenario to remove, given a pair of duplicate BDD scenarios.

[1]https://stackoverflow.com/
[2]https://stackexchange.com/

More specifically, informed by quality facets from the literature [134] and practitioners' opinions (Table 6.1), and our own experience and understanding of writing BDD suites, we analysed 9 randomly selected pairs of duplicate scenarios from the 3 evaluation systems (described in section 4.3.2), aiming to determine which scenario to remove in each pair and the reasons for the decisions we made. By assessing the reasons for the choices we made for each pair, we learned the following: across the 9 pairs, we suggested keeping the scenario whose steps either had more reuse across the specification, or used more domain terms, or used less implementation level terms, or had an abstraction level that broadly resembled with the abstraction levels of other scenarios in the specification it belonged to, or a combination of two or more of these aspects. We thus summarise, in four quality principles, the aspects that stood out in our experiment of deciding which duplicate scenarios to remove. These principles are described in detail in the next four sections.

6.2.2 Principle of Conservation of Steps

This principle seeks to maximise the use of existing steps across the specification, and tries to avoid having too many steps that are used only in one scenario or a small number of scenarios. To illustrate this idea, focusing on the first scenario in Listing 1.1, suppose we need to write a scenario for when the bank customer tries to withdraw more money than is in their account; we should reuse the step phrases from the existing scenarios rather than inventing new ways of phrasing the same idea (e.g. "my account is in credit by \$10" rather than "my account balance is \$10").

The rationale is as follows. The steps in a BDD suite form a vocabulary for talking about the functionality of the system. The `Given` and `Then` steps describe aspects of the system state, while the `When` steps describe all the actions the system's users and stakeholders should be able to take. Expressing a variety of system functionalities using a small number of steps reduces the comprehension effort needed to understand the whole specification, and reduces the chance that duplicated or subtly inconsistent scenarios will be added in future.

Based on this intuition, this principle advocates for the introduction of new steps only if the requirements to be specified in those steps cannot be expressed by existing steps. So, given a pair of duplicate scenarios, this principle suggests the removal of the duplicate scenario whose steps are used least across the specification. (In the ideal case, this would remove the only appearance of a step, allowing it to be removed from the BDD suite altogether.)

6.2.3 Principle of Conservation of Domain Vocabulary

Any organisational process that is supported by software will typically accrue over its lifetime a set of terms or phrases describing the key ideas in the domain of the process, that are used by the people involved to communicate about and advance the state of the work. The agile practice of a *Ubiquitous Language* requires the software team to use the same terms wherever possible, in the artefacts that describe and constitute the system [200]. This is also true within BDD scenarios.

With this in mind, the Principle of Conservation of Domain Vocabulary seeks to maximise the value of each domain term or phrase that used in the BDD suite. Inevitably, in any human endeavour, different terms may be used for the same concept (and the same term may have different meanings). But each additional term increases the cognitive load for readers and writers of scenarios. We therefore consider a suite to be of high quality if it can express the required semantics clearly with the minimum number of domain terms and phrases. This is also inline with the *ubiquitous* aspect in the list of quality aspects for BDD scenarios proposed by Oliveira *et al.* [134]. So, given a pair of duplicate scenarios, this principle suggests the removal of the scenario which contains more less frequently used domain terms.

6.2.4 Principle of Elimination of Technical Vocabulary

Since BDD scenarios are meant to be readable by all project stakeholders (including end users), the use of technical terms that, in most cases, only the development team can understand, is discouraged. As such, scenarios that use domain terms are generally preferred to scenarios that use technical terms. This principle, therefore, focuses on minimising the use of technical terms in the steps of BDD scenarios. So, given a duplicate pair of scenarios, the Principle of Elimination of Technical Vocabulary will keep the scenario with no or fewer technical terms.

6.2.5 Principle of Conservation of Proper Abstraction

One challenging aspect in the creation of a BDD feature suite is to select an appropriate level of abstraction for the scenarios, and in particular for the steps. Higher level steps convey more semantics, so that scenarios can be expressed using fewer steps, and are often closer to the domain concepts that end users are familiar with. But they require more extensive glue code to be written, with more embedded assumptions, so that sometimes the meaning of the suite cannot be understood with precision without reference to the glue code. Lower level steps describe more fine-grained aspects of system state and state change. Scenarios using them will typically be longer, requiring more steps to express the same semantics than when using higher level steps. But lower level steps require smaller simpler glue code to implement them. Feature suites written using very low level steps can often be very brittle, breaking easily when small implementation details change, and can be too procedural, resembling traditional testing scripts, rather than end-user focussed declarative examples.

Intuitively, therefore, a BDD feature suite in which scenarios are written at a consistent level of abstraction will be easier to understand, extend and maintain. On the contrary, if the feature suite has a mix of scenarios expressed at a low level of abstraction and scenarios expressed at a higher level of abstraction, it can be difficult for a maintenance engineer to decide on the level of abstraction to use in expressing a new scenario. Moreover, there is likely to be duplication of steps and glue code, and the test harness code will also be at inconsistent levels of abstraction, adding to the comprehension and maintenance burden.

This principle attempts to capture this notion of feature suite quality. As such, given a pair of duplicate scenarios, adherence to this principle would recommend keeping the scenario whose abstraction level is largely consistent with the abstraction levels of other scenarios in the specification, and deleting the scenario with steps that are at a contrasting abstraction level with the bulk of the suite.

6.3 Community Support for the Quality Principles

We conducted a survey to gather practitioners' opinions on the four principles. We wanted to discover whether the principles resonated with practitioners as accurate descriptors of facets of BDD suite quality, and whether there were important quality facets we had overlooked.

6.3.1 Survey Design

The survey questions covered respondents' demographics, respondents' views on the four principles and respondents' opinions on quality aspects not covered by the principles. The four questions on demographics were:

> *Q1: Which of the following best describes your job?* (Options: Developer; Tester; Business Analyst; Chief Technology Officer (CTO); Researcher; Other, please specify)
>
> *Q2: What is the size of your organisation?* (Options: 1-20 employees; 21-99 employees; 100-1000 employees; more than 1000 employees; Other, please specify)
>
> *Q3: Which of the following best describes your experience of working with BDD?* (Options: less than 1 year; 1-5 years; 6-10 years; over 10 years; Other, please specify)
>
> *Q4: What country are you based in?* (Free text)

To mitigate the potential for bias and allow respondents to think and respond in a natural way, the exact principles were not disclosed in the survey. Instead, we sought respondents' degree of agreement with informal statements of the principles:

> *Q5: When adding new scenarios to a BDD specification, we should strive to reuse existing steps wherever that is possible without compromising readability of the scenario.*
>
> *Q6: When writing the BDD scenarios for a particular domain, we should strive to make use of a minimal set of domain terms in our scenario steps. That is, we prefer to write steps that reuse domain terms already used in other steps, rather than introducing new terms, wherever that is possible without compromising readability of the scenario.*
>
> *Q7: When adding new scenarios to a feature suite, we should prefer to use steps that are expressed using domain terms over steps that are expressed using more technical language, whenever we have a choice.*

> *Q8: Within a feature suite, the abstraction levels of steps in one scenario should be largely consistent with the abstraction levels of steps in other scenarios in the suite.*

A 5-point likert scale (Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree) was used to record the respondents' degree of agreement with each of the given statements in questions 5 to 8. For questions 1 to 3 and questions 5 to 8, an "other" free text option allowed respondents to provide alternative responses or provide qualified degree of agreement. Question 8 was supplemented by 2 example scenarios, clarifying what we meant by "abstraction levels".

Lastly, question 9 allowed free text for respondents to describe BDD quality aspects not covered by the statements in questions 5 to 8:

> *Q9: Please give us any other thoughts on how to keep scenarios and steps of a BDD specification readable, easy to extend, and maintainable.*

The survey was pretested on a BDD practitioner from industry. Like the survey in Chapter 3, this particular survey was developed and deployed using SelectSurvey.NET on our university's servers. Respondents completed the survey over a period of one month from December 2018.

6.3.2 Ethical Considerations

The same ethical considerations in section 3.2.4 apply to this survey as well.

6.3.3 Respondents and Data Analysis

The same sampling mechanism and email contacts used for the survey in Chapter 3 (section 3.2.2) were used to distribute this survey as well. In addition to the BDD Google groups that were used to distribute the survey in Chapter 3, this particular survey was also posted on the following Google groups of BDD practitioners: Specflow, Concordion, and Serenity BDD.

The survey was viewed by 129 people, of whom 56 submitted responses to the key questions on BDD suite quality. In the remainder of this chapter, all discussions of survey results refer to this subset of 56 respondents. We randomly assigned them numbers 1 to 56, and so we will hereafter refer to them as R1 to R56. The number of responses to questions on the four principles were as follows: Conservation of Steps (55), Conservation of Domain Vocabulary (54), Elimination of Technical Vocabulary (55), and Conservation of Proper Abstraction (56). 31 people responded to Q9 by mentioning other quality aspects of BDD suites.

The distribution of respondent roles was as shown in Table 6.2, and the sizes of respondent organisations were as shown in Table 6.3. The respondents' experience of working with BDD is shown in Figure 6.1, and the geographical distribution of respondents was as shown in Figure 6.2.

Role	Percentage of Respondents
Developer	60.7%
Tester	12.5%
Consultant	7.1%
Chief Technology Officer (CTO)	5.4%
Researcher	3.6%
Business Analyst	1.8%
Other	7.1%
Did not say	1.8%
Total	**100.0%**

Table 6.2: Survey on BDD quality principles: roles of respondents

Size	Percentage of Respondents
1-20 employees	26.8%
21-99 employees	16.1%
100-1000 employees	26.8%
More than 1000 employees	21.4%
All sizes	7.1%
Did not mention	1.8%
Total	**100.0%**

Table 6.3: Survey on BDD quality principles: sizes of respondent organisations

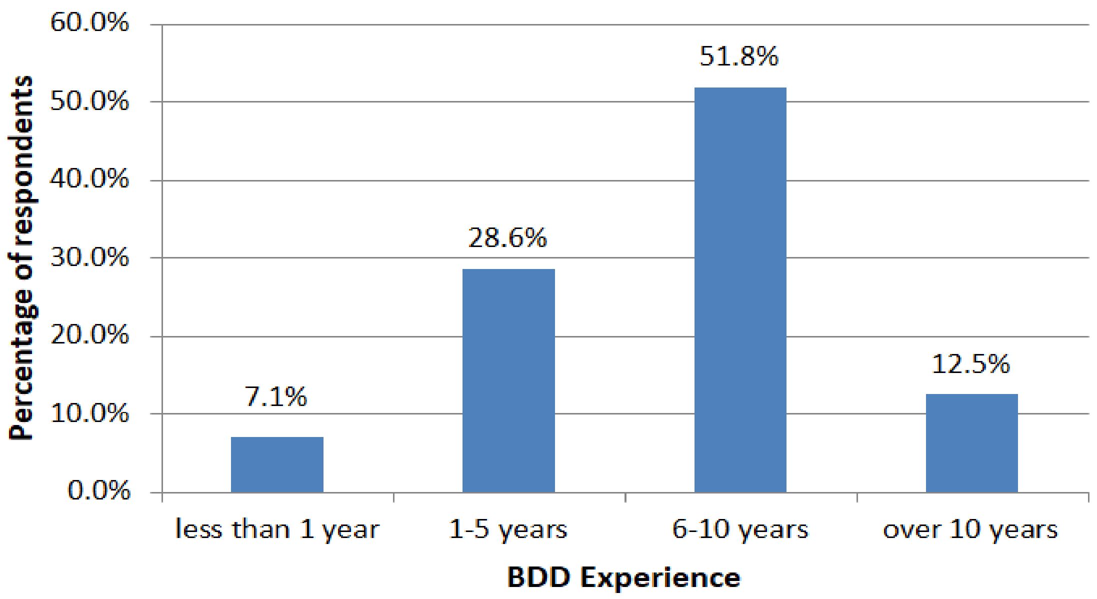


Figure 6.1: Survey on BDD quality principles: Respondents' experiences of working with BDD

Also, the same mechanism used to analyse survey data in section 3.2.3 was used to analyse the data for this survey as well. Different from section 3.2.3, the theoretical thematic analysis in this regard was guided by the research question: how to keep BDD suites comprehensible, extensible and maintainable?

6.3.4 Results

Figure 6.3 shows the respondents' degree of agreement with each principle. Each principle was accepted by at least 75% (Strongly Agree + Agree) of the respondents who answered the question about it and clearly indicated their degree of agreement.

Other comments on the respective principles were:

1. **Conservation of Steps:**
    - Steps should also be expressed in general terms:
        - *"Agree somewhat, but that's not the most important consideration. You should also consider generalizing existing steps. It's not a big deal, though, to create new steps and have them call common code under the step definition layer."* (R2, Consultant)

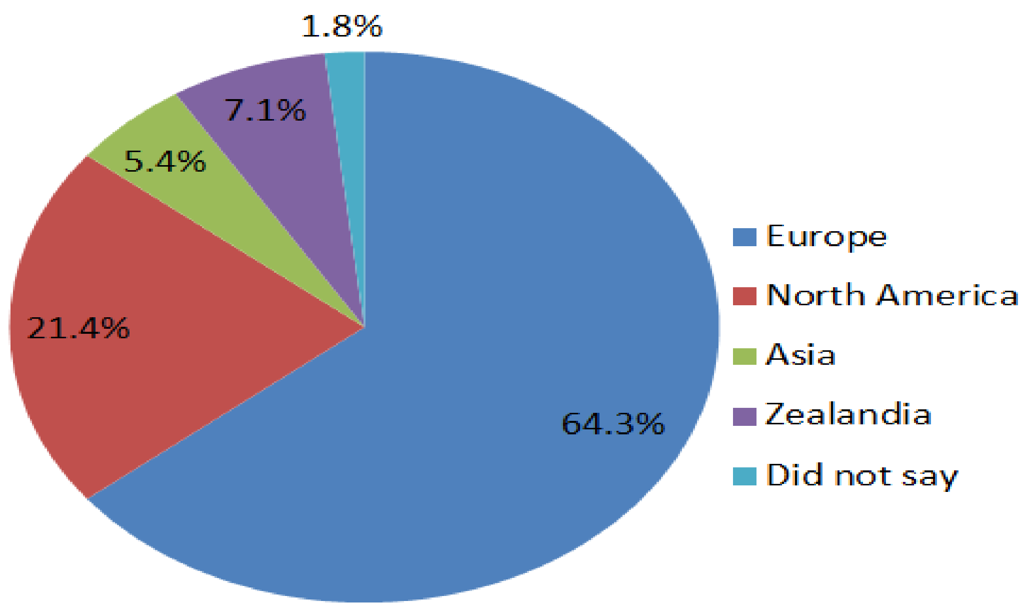


Figure 6.2: Survey on BDD quality principles: Geographical distribution of respondents

- Sometimes it can be a good idea to focus on writing clear steps that serve the purpose, and then fix the design later:
  - *"Well, I would approach to famous phrase: First make it work. Then make it right. (Kent Beck?). I agree you should try to not duplicate steps, but one also must be careful not to force reuse of existing steps. Sometimes is maybe just much better to have 2 very similar steps which doing almost same thing (but actually different), then have 1 step which hides some logic under the hood."* (R7, Developer)
  - *I try to express myself as clear as I can at the moment. Any reuse is secondary. I also know that I will learn more about how the system should in work in due time. That will effect how I might want to rephrase myself later.* (R4, Developer)
- The main focus should be on the readability, and reuse of steps can affect the readability and maintainability of the specification:
  - *"Readability is the most important consideration here. If through the light of the new spec, formulations of previously existing specs come outdated, those should be considered for an update as well."* (R55,

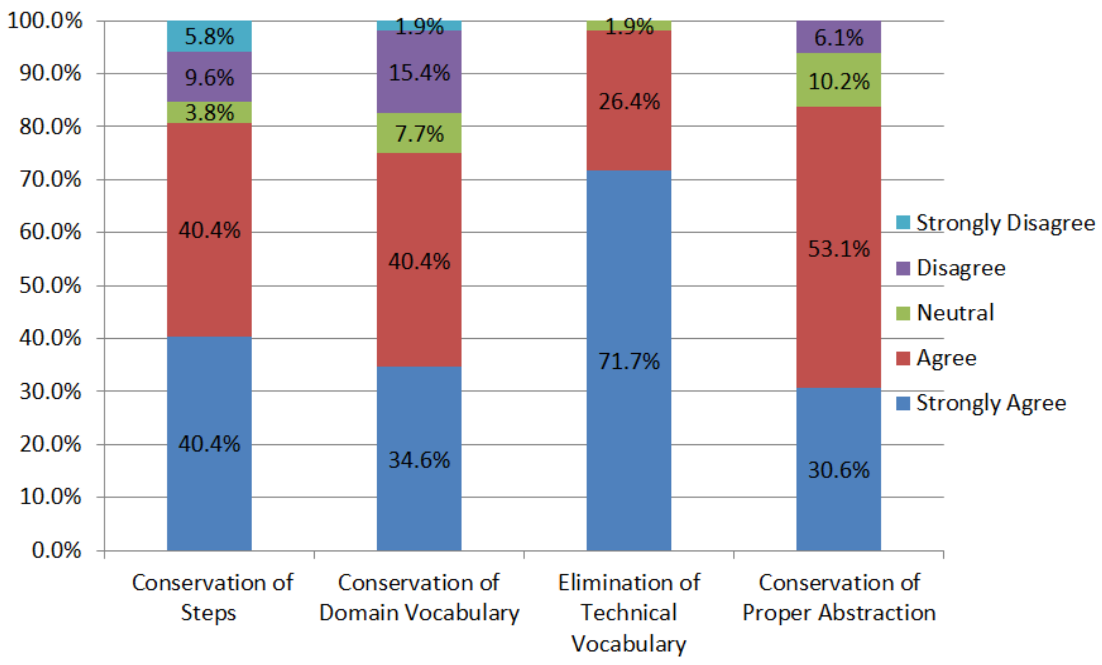


Figure 6.3: Acceptability of each BDD quality principle by respondents

Developer)

- *"Duplication of steps is wasteful but using steps as reusable building blocks has a high risk of compromising readability and accessibility for business stakeholders."* (R8, Consultant)

2. **Conservation of Domain Vocabulary:**

   - It should be possible to use new domain terms whenever necessary, provided that the specification remains readable to customers:

     - *"You should reuse domain terms where appropriate and introduce new domain terms where appropriate. The scenarios should sound natural to the widest community in the business."* (R2, Consultant)

     - *"The scenarios should, IMO, reflect the real domain language (as practiced in DDD-like projects); hence I think we should introduce domain terms from the ubiquitous language as they're used..."* (R20, Developer)

     - *"The understanding and the language of the domain usually evolve and refine over the course of a project. Therefore, if new terms arise* *as the model evolves, those new terms must be used everywhere where applicable (refer to ubiquitous language, E. Evans, Domain Driven Design). It would be a reprehensable omission to stick to outdated or unclear terms just for the sake of simplicity of consistency with slightly related areas of the domain."* (R55, Developer)

3. **Elimination of Technical Vocabulary:**

    - Implementation words can sometimes be used, depending on the product owner and expected readers of the specification:

        - *"I would say I would normally strongly agree, but BDD tests can be used to test the so-called non-functional requirements as well. Product owners with some technical knowledge (the best type of product owner) can understand technical terms and how tech affect overall behaviour through non-functionals. So yes, why we should try to express business behaviour with domain terms, non-functionals may make more sense to use technical terms and we should not prohibit that."* (R13, Developer)

        - *"I would agree but it depends on who reads the scenarios and how the domain terms differ from the technical ones."* (R40, Developer)

        - *"The short answer is it depends .Basically it depends on the people who will be reading the scenarios."* (R42, Tester)

    - Sometimes, it can be challenging to translate domain words used in scenarios into implementation details:

        - *"We ran into the issue where steps expressed using domain terms were on occasion too vague to know what the implementation details were supposed to be, as there was no spec other than the BDD tests (specification by example), and if the wrong implementation was chosen we would receive a bug report. So while I think steps expressed using domain terms are more readable and understandable at a higher level for non-technical staff, we had an implementation where we needed more technical details to be specified to explain, for example, how the system could get into a certain state instead of just specifying that it was the case, and there were cases where the Given steps described an* *impossible situation that lead to a decent discussion to get it erased."* (R23, Developer)

4. **Conservation of Proper Abstraction:**

   - The abstraction levels should be determined by capturing correct requirements, and producing scenarios that are readable to customers:
     - *"It depends. The abstraction level should help readability and it should be based on the intent of the scenario... "* (R25, Test Architect)
   - Lower abstraction levels can be appropriate if scenarios carry data:
     - *"Well... Not an easy question. I had situations when I have many scenarios in feature, and my outputs are based on my inputs. So I used 2nd approach, with lower level abstraction because that way I could change my inputs from feature file."* (R7, Developer)
   - Sometimes, one can use different abstraction levels for Given, When, and Then steps:
     - *"I always prefer writing the 'Given' part in most abstract way, then the 'When' section should be more detailed. and the 'Then' should be detailed but not in technical (like UI) language"* (R25, Test Architect)
   - The abstraction level of a scenario should depend on the behaviour being specified:
     - *"The level has to fit the behaviour the example is illustrating. It should be mention all the necessary parts and leave out all technical parts"* (R55, Developer)

Additionally, in response to Q9 in the survey, other opinions from respondents on how to keep BDD suites readable, easy to extend and maintain, were as summarised in Table 6.4 and Table 6.5.

6.3.5 Conclusions from the Survey

In general, the majority of the respondents supported the principles as acceptable descriptors of facets of BDD suite quality (see "Strongly Agree" and "Agree" responses in Figure 6.3). The written comments stressed the importance of reuse within BDD,

S/n	Theme	Frequency	Sample Excerpts
1	Specification should act as readable business documentation	11	-*"The key is to have a multi-layered approach; the gherkin scenarios should focus on being readable as business documentation..."* (R8, Consultant)
2	Clear description of business goals using examples	5	-*"Describe the business goal and the steps on how to achieve them as clearly as you understand at the moment."* (R4, Developer) -*"Focus on clean specifications that are consistent within the bounded context"* (R6, IT Consultant)
3	Use of common domain concepts and terms across the specification	5	- *"...I like the idea of a glossary of terms from the Writing Great Specifications book..."* (R6, IT Consultant) -*"Use the same domain language and terminology as the rest of your organisation/customers/industry"* (R26, Chief Technology Officer) -*"...Have a glossary with important domain concepts, primary term and possible synonyms."* (R44, Principal Software Architect)
4	Focus on capturing comprehensive requirements for all project stakeholders	5	- *"BDD specification should satisfy both business analyst and developer as much as possible."* (R11, Developer) - *"Everything around BDD and Specification by example is around creating a shared understanding. That is the core reason to do examples in the first place; the help us uncover hidden assumptions..."* (R18, BDD Coach)
5	Specification should be easy to understand based on general domain knowledge	4	-*"Test them on other people not involved in the project. Can they understand what they mean? Can they determine the intent of each scenario?... "* (R2, Consultant) -*"Where possible, involve less technical stakeholders and team members in the process of scenario development..."* (R46, Developer)
6	Share specs with stakeholders for reference and correction, and perform regular maintenance of specs	4	-*"I believe the key would be to periodically revisit them and keep updated, if necessary rewrite or reword older ones. I find it very useful to also publish scenarios using ci tools somewhere so business people can read specs and spot inconsistencies"* (R47, Developer) -*"At the very least, have the specs available for reference by the project stakeholders."* (R46, Developer) -*"...Refactoring also applies to BDD scenarios..."* (R44, Principal Software Architect)

Table 6.4: Other recommended practices for all project stakeholders on how to keep BDD specifications readable, easy to extend, and maintainable

S/n	Theme	Frequency	Sample Excerpts
1	Write reusable and yet focused steps and step definitions	11	-*"...the gherkin scenarios should focus on being readable as business documentation, and map to reusable steps in the step definitions. It is the DSL code in the step definitions where the real reusability benefits occur"* (R8, Consultant) -*"It's best to re-use steps either by referring to them directly (Using Given, And...), or creating a new step definition using the underlying API, not calling one step definition from another"* (R24, Software Engineer in Test)
2	Aim for more stateless scenarios	4	-*"The scenarios should be stateless, in the sense that they should store as few data as possible."* (R50, Developer)
3	Proper use and order of Given, When, and Then steps; and careful choice and use of framework-specific BDD features	4	-*"Ensure that WHEN's only perform actions and THEM's only assert (do not modify the SUT state) and are expressed as such"* (R43, Tester) -*"...Choose good titles (scenario/feature) 9) Don't send test data from feature file, (but examples of behavior are allowed)10) Less is More 11) Limit one feature per feature file. This makes it easy to find features. 12) Hide data in the automation 13) Steps order must be given/when/then - not other order'"* (R25, Test Architect)
4	Miscellaneous: Keeping an inventory of all steps in a project; clear separation of customer-readable tests from glue code and the underlying API; and leveraging the full capabilities of underlying BDD framework and regular expressions	3	-*"I'm not aware if this is already possible but it would be helpful to produce a dictionary of all the steps used in a project by extracting them from the feature suites."* (R1, Developer) -*"The key is to have a multi-layered approach; the gherkin scenarios should focus on being readable as business documentation, and map to reusable steps in the step definitions. It is the DSL code in the step definitions where the real reusability benefits occur"* (R8, Consultant) -*"Make full use of the underlying BDD framework / regular expressions and craft the step definitions like a powerful text-based API."* (R33, Developer)

Table 6.5: Other recommended practices for testers and developers on how to keep BDD specifications readable, easy to extend, and maintainable

but put most emphasis on readability and clarity of the resulting specifications. Finally, the validity threats and mitigation strategies for this survey are the same as those in section 3.4.

6.4 Operationalization of BDD Quality Principles

Since the four principles had received practitioner support, the next step was to attempt to operationalize each principle in some way, to allow it to be automatically assessed against a feature suite and glue code. The principles themselves are quite abstract, and deal with concepts that are not amenable to full automation. We have therefore tried to find pragmatic approximations to the principles that can be automated. The algorithms we have experimented with are described below in terms of a comparison of a pair of duplicated scenarios, and all assess the relative effect that removing each member of the pair has on the quality of the BDD suite.

6.4.1 Assessing Conservation of Steps

This principle implies that we should prefer scenarios that use step phrases that are frequently used by other scenarios over ones that use unique step phrases or phrases that are only rarely used. To assess this, we statically analyse the feature suite to discover how many times each step phrase is used. This data for individual step phrases must be aggregated together to provide a score for whole scenarios. Several aggregation options are possible. We elected initially to sum the usage score for all the steps in a scenario to arrive at its score, but realised that usage scores will be the same for steps that are identical between the scenarios being compared. We therefore sum the usage scores of step phrases in the scenario that are not used in its duplicate pair. The scenario in the pair of duplicates with the lowest usage score is the one that should be recommended for deletion (see Algorithm 2). Asymptotically, Algorithm 2 is of $\theta(n) complexity.$

Algorithm 2: Assessing Conservation of Steps

```
Input: A BDD suite $\Sigma$
       A set $D$ of known duplicate pairs of scenarios in $\Sigma$
Output: A set of remove suggestions for all known duplicate pairs in $D$
suggestions ← ∅ //an initially empty set of remove suggestions
Get a list of all scenarios (sc) and steps (sct) in a suite:
sc ← scenarios //sc is a list of all scenarios in a suite
stc ← steps(sc) //sc is a list of all scenario steps in a suite
foreach (s, s') ∈ D do
    Get a list of steps that are unique to each scenario in a duplicate pair:
    st ← steps(s) / steps(s') // st is a list of steps that are unique to scenario s
    st' ← steps(s') / steps(s) // st' is a list of steps that are unique to scenario s'
    Find the number of times each unique step of a particular scenario
    in a duplicate pair is used across the suite:
    usage = sum([ 1 | st ∧ stc ])
    usage' = sum([ 1 | st' ∧ stc ])
    Suggest the removal of either scenario if s and s' have the same usage count;
    otherwise, suggest the removal a scenario with lower steps' usage count:
    if usage = usage' then
        suggestions ← suggestions ∪ remove((s, s'), { s, s' })
    else if usage > usage' then
        suggestions ← suggestions ∪ remove((s, s'), { s' })
    else
        suggestions ← suggestions ∪ remove((s, s'), { s })
return suggestions
```

6.4.2 Assessing Conservation of Domain Vocabulary

To assess how far a BDD suite satisfies the principle of Conservation of Domain Vocabulary, we need to be able to determine whether individual steps make use of terms and phrases that end users/stakeholders would recognise and use themselves when talking about the business processes the BDD suite describes. This is not something that can be easily assessed automatically with precision. However, all development teams struggle with the need to use correct terminology when talking with end users. It is common for such teams to build up glossaries of terms or to make use of enterprise models of the business in determining what things and actions in the domain should be called.

We therefore assume the availability of a list of preferred domain terms that we can use to assess the appropriateness of wording used in a BDD Suite. We combine this with a list of common *stop words* such as "at" and "when", which are ignored. We give each step a score based on the number of recognised domain terms/phrases the step contains. Again, the scores for the steps are aggregated into a score for each scenario being compared. We sum the scores for each step. The scenario suggested for removal is the one with the lower aggregated score (Algorithm 3). We decided to try the simple scenario-based score first–more sophisticated approaches are clearly possible and should be explored in future work. Asymptotically, Algorithm 3 is of $\theta(n) complexity.$

Algorithm 3: Assessing Conservation of Domain Vocabulary

```
Input: A BDD suite $\Sigma$
       A set $D$ of known duplicate pairs of scenarios in $\Sigma$
       A set $dwords$ of terms used in the domain of $\Sigma$
Output: A set of remove suggestions for all known duplicate pairs in $D$
suggestions ← $\emptyset$ //an initially empty set of remove suggestions
foreach (s, s') $\in D$ do
    Get a set of words from the steps of each scenario in a duplicate pair:
    w ← { words(st) : st $\in$ steps(s) } // w is a set of words in s
    w' ← { words(st) : st $\in$ steps(s') } // w' is a set of words in s'
    Get a number of domain words used by each scenario in a duplicate pair:
    kw = | w $\cap$ dwords | //kw is a number of domain words in s
    kw' = | w' $\cap$ dwords | //kw' is a number of technical words in s'
    Suggest the removal of either scenario if s and s' have the same amount
    of domain words; otherwise, suggest the removal a scenario with a lower
    amount of domain words:
    if kw = kw' then
        suggestions ← suggestions $\cup$ remove((s, s'), { s, s' })
    else if kw > kw' then
        suggestions ← suggestions $\cup$ remove((s, s'), { s' })
    else
        suggestions ← suggestions $\cup$ remove((s, s'), { s })
return suggestions
```

6.4.3 Assessing Elimination of Technical Vocabulary

Automating assessment of this principle is also challenging, since for a full solution we need an engine that can decide whether a step phrase contains technical wording or not. Again, we fall back on a pragmatic approach of creating a list of technical words to be avoided in steps. Since these words are, by definition, *not* domain specific, we can potentially create one such dictionary for use across all projects in an organisation, seeded from an initial list provided along with our tool. Since for some applications, the domain words may themselves be technical, the algorithm for assessing scenarios against this principle would ideally have access to the dictionary of domain terms created for the previous principle, so that terms appearing in both lists can be considered as domain terms for the purposes of the calculation.

In this case, the score per step is a simple count of the number of words in the technical dictionary that appear within it (and which are not in the domain dictionary). Aggregation is again performed by summing the scores for each step in the scenario. In a duplicate pair, the scenario with the highest score is recommended for deletion (Algorithm 4). Asymptotically, Algorithm 4 is of $\theta(n) complexity.$

Algorithm 4: Assessing Elimination of Technical Vocabulary

```
Input: A BDD suite Σ
       A set D of known duplicate pairs of scenarios in Σ
       A set twords of technical vocabulary to avoid
Output: A set of remove suggestions for all known duplicate pairs in D
1  suggestions ← ∅ //an initially empty set of remove suggestions
2  foreach (s, s') ∈ D do
3      Get a set of words from the steps of each scenario in a duplicate pair:
4      w ← { words(st) : st ∈ steps(s) } // w is a set of words in s
5      w' ← { words(st) : st ∈ steps(s') } // w is a set of words in s'
6      Get a number of technical words used by each scenario in a duplicate pair:
7      tw = | w ∩ twords | //tw is a number of technical words in s
8      tw' = | w' ∩ twords | //tw' is a number of technical words in s'
9      Suggest the removal of either scenario if s and s' have the same
10     amount of technical words; otherwise, suggest the removal a scenario
11     with a higher amount of technical words:
12     if tw = tw' then
13         suggestions ← suggestions ∪ remove((s, s'), { s, s' })
14     else if tw > tw' then
15         suggestions ← suggestions ∪ remove((s, s'), { s })
16     else
17         suggestions ← suggestions ∪ remove((s, s'), { s' })
18 return suggestions
```

6.4.4 Assessing Conservation of Proper Abstraction

This principle is arguably the most challenging to automate, since it centres on a subtle and largely implicit aspect of BDD suites. But we still need a computable measure that can act as a proxy for the more sophisticated and hard-to-compute principle. Thus, we must again fall back on a pragmatic approach, if we are to automate it. Our solution is based around the observation that steps at a higher level of abstraction typically require more extensive glue code to implement than lower level steps. For example, the glue code for the step “a customer with a balance of -\$100” will likely involve only one or two API calls to set up such a customer. A higher level step, “a customer who is a persistent debtor”, will require more substantial glue code, setting up a trail of transactions over a period of time.

We wanted to explore whether this tendency could be exploited in order to automate assessment of this principle. We take the simplifying assumption that the abstraction level of a step is proportional to the number of glue code statements that implement it. We compute the following metrics:

1. *Average Statements per Step, for the whole specification (ASPS):* This is the ratio of the number of all glue statements in the specification to the number of all steps in the specification. We use ASPS to represent an overall estimate of the abstraction level of an entire specification. For a particular BDD feature suite, ASPS can serve as a benchmark for recommended abstraction level across the suite, and can be used to guide the maintenance and extension of a suite (for example, by giving a rough figure of expected number of steps in each new scenario).

2. *Average Statements per Step for individual scenarios in a duplicate pair:* For the first scenario in a duplicate pair, ASPS1 is computed as the ratio of the number of glue statements in the first scenario to the number of steps in the first scenario. An analogous quantity (ASPS2) is computed for the second scenario in the pair.

3. *Abstraction Distance (AD) between individual scenarios in a duplicate pair and the entire specification:* We use AD to approximate the extent to which the abstraction level of a particular scenario in a duplicate pair deviates from the abstraction level of the entire suite. For the first scenario in a pair, AD1 is computed as absolute(ASPS - ASPS1). An analogous quantity (AD2) is computed for the second scenario in the pair.

Using this principle, we propose the scenario with the highest Abstraction Distance be removed, since this suggests it deviates most from the abstraction level of the entire suite (Algorithm 5). Clearly, this is a simple case, and it might be worth considering putting a threshold on suggestions, such that if two scenarios differ only by a negligible value of AD, then there may not be much reason to recommend the removal of one scenario over the other. Future work might explore sophisticated approaches for this recommendation problem. Asymptotically, Algorithm 5 is of $\theta(n) complexity$.

Algorithm 5: Assessing Conservation of Proper Abstraction Levels

```
Input: A BDD suite Σ
       A set D of known duplicate pairs of scenarios in Σ
       A list G of glue statements for the scenarios in Σ, grouped by step
Output: A set of remove suggestions for all known duplicate pairs in D
1  suggestions ← ∅ //an initially empty set of remove suggestions
2  foreach (s, s') ∈ D do
3      Compute scenario-specific metrics
4      num_steps = | { st | sc ∈ Σ and st ← steps(sc) } | //num_steps is a number of
         all steps in a suite
5      num_gstmts = | G | // num_gstmts is a number of all glue statements for the
         scenarios in a suite
6      avg_stmts = num_gstmts / num_steps //avg_stmts is the average number of
         glue statements per step, for the whole suite
7      sc_avg_stmts = | gl | gl ← (glue(s) ∈ G) | / | steps(s) | // sc_avg_stmts is the
         average number of glue statements per step, for scenario s
8      sc'_avg_stmts = | gl | gl ← (glue(s') ∈ G) | / | steps(s') | // sc'_avg_stmts is
         the average number of glue statements per step, for scenario s'
9      sc_abs_dist = | avg_stmts - sc_avg_stmts | //sc_abs_dist is an abstraction
         distance between s and the suite
10     sc'_abs_dist = | avg_stmts - sc'_avg_stmts | //sc'_abs_dist is an abstraction
         distance between s' and the suite
11     Suggest the scenario to remove based on the values of abstraction distance:
12     if sc_abs_dist = sc'_abs_dist then
13         suggestions ← suggestions ∪ remove((s, s'), { s, s' })
14     else if sc_abs_dist > sc'_abs_dist then
15         suggestions ← suggestions ∪ remove((s, s'), { s })
16     else
17         suggestions ← suggestions ∪ remove((s, s'), { s' })
18 return suggestions
```

6.4.5 Implementation

We wrote code for extracting various data from a BDD feature suite. This pre-processing stage helped to organise data on which the recommendation algorithms (section 6.4) could be run. For each BDD feature suite under analysis, the following three types of data were extracted automatically:

1. A list of all steps in the suite, mapped to the scenarios in which they appear and partitioned according to whether they are `Given`, `When`, or `Then` steps.
2. A list of glue statements mapping to respective scenario steps in a suite.
3. A list of all known duplicate pairs of scenarios in a feature suite.

These three types of data constituted all the inputs we needed for assessing the Principle of Conservation of Steps and the Principle of Conservation of Proper Abstraction.

We also used a semi-automatic approach to create the following two more types of data that were part of the inputs required to assess the Principle of Conservation of Domain Vocabulary and the Principle of Elimination of Technical Vocabulary:

4. A list of technical words: The following procedure is used to create the list of technical words:
   - First, we adapted a list of 18 technical words from *cuke_sniffer*, a tool to identify smells in Gherkin feature files [201]. The words in that list have been judged by independent experts to be implementation level words that should generally be avoided when writing BDD scenarios. This formed a seed list of our technical terms.
   - Second, we used Apache Lucene-6.6.0 [3] to create the list of all words from the steps of scenarios in a system under analysis.
   - Third, the list of all words from the steps of the feature suite under analysis was manually analysed to identify additional technical words that were not part of the seed list. Depending on the software domain, it can be particularly hard to differentiate technical from non-technical words in a feature suite. However, the fact that we worked with software systems whose domain can be understood based on general knowledge enabled us to navigate this challenge. Even so, our list of technical words may still not be perfect.

[3] https://lucene.apache.org/core/6_6_0/

But it was generally okay for purposes of experimentation. Moreover, six systems were used to create our list of technical terms. Of the six systems, three were described in section 5.5.1 and were used for the evaluation of the present work, and the other three systems were used only for identification of more technical words, in a bid to have a broad list of technical words from a variety of systems. We were able to use the three additional systems for the purpose of identifying technical words because the process we used to generate the list of words for a particular system did not require execution of scenarios, a criteria the additional three systems could not meet during the duplicate injection experiment described in 5.5.1.

This process gave us a total of 74 implementation words. Since the technical terms are (by definition) not domain-specific, one technical list can suit many domains. That is why we used one list of technical words for all the three evaluation systems.

5. A list of domain words: After the list of technical words had been obtained, the list of domain terms for a particular system was obtained by removing all technical terms from the list of all words in a feature suite.

For a particular system, both the list of domain words and the list of steps in the scenarios are used to assess the Principle of Conservation of Domain Vocabulary, and both the list of technical terms and the list of steps in the scenarios are used to assess the Principle of Elimination of Technical Vocabulary.

Lists 1, 2, and 3 above are stored in XML files, while lists 4 and 5 are supplied as input lists respectively to the algorithms for the Principle of Elimination of Technical Vocabulary and the Principle of Conservation of Domain Vocabulary. We use XQuery[4] to query the XML lists, to obtain scenario removal suggestions for the different principles' algorithms. Figure C.1 shows sample remove suggestions for some duplicate scenarios in evaluation System 1. We can get results in reasonable time scales since typical BDD scenarios number in tens to hundreds [28], with steps numbering in tens to a few thousands. However, the performances of both the Principle of Conservation of Domain Vocabulary and the Principle of Elimination of Technical Vocabulary can be affected by the sizes of the list of domain words and the list of technical words respectively. But typical lists numbering in tens to several hundreds should generally

[4]https://www.w3.org/TR/xquery-31/

be able to produce results in reasonable time scales.

6.5 Evaluation

This section presents the evaluation of the use of the proposed BDD Quality Principles in guiding the removal of duplicate scenarios.

6.5.1 Context and Research Questions

We conducted an experiment to answer the following question:

Q: *How does an automated solution perform in terms of giving acceptable scenarios removal suggestions?*

To answer this question properly, we need to test our hypothesis that the operational forms of the quality principles we proposed are able to make acceptable suggestions for which of a pair of duplicates to remove. Here "acceptable" means consistent with what a human expert might advise. We can run the tool on the TP duplicates from our evaluation in section 5.5 but we need a "ground truth"–we need to know, for each duplicate pair, which is the one that should be removed to maintain/increase suite quality. This is challenging because there is no way to compute the ground truth or even to acquire it. The closest to a ground truth we could get to would be to ask the developers of the systems we worked with for their preferences. But the evaluation systems we used were open source and had no developer contacts, and so it was impossible to get original developers of the three systems. This forced us to look for other alternatives. Two options were available: to find human experts who could act as a proxy for original developers, and to find a computable ground truth. We explored both options.

For an option of using human experts, the people we used to inject the duplicates could act as a proxy for original developers, but were too likely to be biased in favour of their scenarios. So we had to find a neutral third party. Ideally, we would have preferred to get more than one human experts because decisions on which scenarios to remove are subjective and so human experts might disagree, especially on borderline cases. In case of disagreements between human experts, a somewhat reliable ground truth of acceptable recommendations could be produced by focusing on cases where several human experts agree. But we only managed to get one BDD expert. Again, our industry collaborator had limited time to spend on our experiment, and manual preparation of the ground truth is known to be prohibitively expensive [4].

For an option of finding a computable ground truth, we employed the pragmatic approach based on the premise that: injected scenarios are most likely to violate the proposed BDD quality principles. This is because scenario injectors (1) had limited understanding of the domain served by the software, which limited their ability to properly reuse steps, domain keywords, and other reusable aspects of the BDD specifications they worked with when injecting duplicate scenarios; (2) were somewhat less likely to write scenarios that used terms and steps consistently with the rest of the specification, since we asked them to focus on individual scenarios, and they may not have ever looked at large parts of the rest of the specification. It is therefore reasonable to expect that, given a pair of duplicate scenarios (IO) in which one scenario (O) existed before injection and another scenario (I) was injected by a volunteer, the algorithm should suggest the injected scenario (I) for removal. For this to be true, a further assumption is that writers of the original scenarios (Os, in this regard) fairly observed, possibly unknowingly, the principles we have proposed for preserving the quality of BDD specifications. Besides, these features were not an issue for our experiment on duplicates injection, where we needed only duplicate semantics, not high quality of scenarios.

It is important to note that the idea of using the original/injected status of the TP duplicates is not something we assume it is correct, nor something we want to get evidence for or against. It is an easily computable value that we have reason to believe that it might act as a suitable proxy for the aground truth. So, like any ground truth proxy, we cannot draw strong conclusions from its use but any broad/strong patterns in the results can be explored, to see if they might have significance in the context where the particular features/limitations of the proxy are known and taken into account.

We used the two weaker forms of ground truth in the hope that they would balance/-compliment each other, since we did not have a single good source of ground truth. We will also be conservative in what claims we make on the basis of the experiments we were able to run. More specifically, we combined the following two approaches to evaluate the effectiveness of the proposed principles in guiding the removal of duplicate scenarios:

1. **Lab Experiment:** For this particular evaluation, we explored the expectation of recommending injected scenarios for removal. So Q can be further refined to:

> **Q1:** *How does an automated solution perform in terms of suggesting the removal of injected scenarios?*

The success criteria in this regard is that an automated solution should suggest the removal of injected scenarios in all (or majority of) IO pairs across the systems used in the benchmark. Q1 was further broken into the following questions:

> **Q1.1:** *What is the performance of each principle in terms of suggesting injected scenarios for removal, and what affects the performance?*
>
> **Q1.2:** *What is the sensitivity of the operationalised principles to changes to the contexts of the vocabularies?*
>
> **Q1.3:** *What proportion of scenarios injected by each volunteer is suggested for removal by each principle, and what affects the performance of each principle on the scenarios developed by the individual volunteers?* This question is based on the premise that there might have been traits inherent in the duplicates injected by specific individuals, and we wanted to explore how those traits might have affected the recommendations made the algorithms for the proposed BDD quality principles.
>
> **Q1.4:** *How does the combination and prioritisation of the four principles perform, in terms of suggesting injected scenarios for removal?*

2. **Industry Experiment:** For this evaluation, we worked with an expert BDD practitioner to build the proxy ground truth of remove suggestions for a sample of known pairs of duplicate BDD scenarios. Specifically, this experiment sought to answer the following questions:

> **Q2:** *To what extent are the injected scenarios suggested for removal by a human expert?* Answers to this question would qualify the results of the pragmatic approach used in the lab experiment.
>
> **Q3:** *To what extent do the suggestions of the tool agree with the suggestions of a human expert?*

In the subsequent sections, we present experiment design, results and discussion for both the lab and industry experiment.

6.5.2 Lab Experiment: Design, Results and Discussion

Experiment Design

To answer Q1, we used 65 IO pairs (Table 4.4) of duplicate BDD scenarios from the benchmark created for the duplicate detection experiments. The characteristics of these systems, the process through which they were identified, and the process through which the duplicate scenarios were injected in them were detailed in section 5.5.1. To recall from section 4.3.4, the number of duplicate pairs between original and injected scenarios (IO) were as follows: System 1 (36), System 2 (23), System 3 (6). For each of the 3 systems, we used the process described in section 6.4.5 to extract the data required to run the algorithms for the proposed quality principles.

Furthermore, to study the sensitivity of the operationalised principles to changes to the context of the vocabularies, for each of the three evaluation systems, we created different categories of words, in order to get different domain word sets that we could run the experiment against. In particular, for each system, we used the following process to create different sets of domain words. To obtain an initial set of domain words, for each system, we regarded all contents of the Lucene index file (except names of the indexed files which, by default, are also reported as part of Lucene index) to be domain words. Other sets of domain words were obtained by excluding numbers and technical words from the initial set, as well as by including words in a particular set based on their frequency of occurrence in the Lucene index file. The following categories were identified to contain words from our evaluation systems:

1. *All words*–everything in the lucene index file, except the file names.
2. *All domain words*–as in set 1, excluding numbers and technical words.
3. *Domain word frequency greater than 1*–all words in set 2 that occurred more than 1 time in the lucene index.
4. *Domain word frequency greater than 2*–all words in set 2 that occurred more than 2 times in the lucene index.
5. *Domain word frequency greater than 3*– this set, which should hopefully be self-explanatory, however, only had words for System 1 and System 2. System 3 had no words in this category.

Up to this point, we were able to answer Q1.1, Q1.2, and Q1.3. To answer Q1.4, however, we combined the four principles, prioritising them in the following order: Conservation of Steps, Conservation of Domain Vocabulary, Elimination of Technical Vocabulary, and Conservation of Proper Abstraction. This order was chosen based on the sum of the scores of each individual principle in the “Agree” and “Strongly Agree” categories in the survey responses (Conservation of Steps (80.8%), Conservation of Domain Vocabulary (75.0%), Elimination of Technical Vocabulary (98.1%), and Conservation of Proper Abstraction (83.7%)) and on the results for Q1.1. Thus, apart from the popularity criteria, our ordering prioritised the principles that had the potential of giving higher numbers of acceptable remove suggestions. Hence, while the Principle of Elimination of Technical Vocabulary had more support than the rest in the popularity category, to produce any results, it largely depended on whether technical words were used in the scenarios’ steps for the specific evaluation systems, and on our ability to identify the implementation words from the steps of the evaluation systems. This is why the Principle of Elimination of Technical Vocabulary was preceded by both the Principle of Conservation of Steps and the Principle of Conservation of Domain Vocabulary in our order.

Results and Discussion

We now present and discuss the results of our lab experiment.

Q1.1: Performance of each principle: Figure 6.4 shows the performance of each principle when used individually to suggest injected scenarios for removal.

In general, most of the injected scenarios across the three systems were suggested for removal when steps, or domain words, or both were conserved, compared to when the focus of our experiment was on elimination of technical vocabulary or conservation of proper abstraction (see Figure 6.4). Based on the results for each principle, we conducted manual analysis of the scenarios in each IO duplicate pair across the three systems, to discover what might have influenced the recommendations made by the different principles. For each step in a duplicate pair of scenarios, we queried the list of all steps of a particular specification (section 6.4.5) to determine the number of times it (step) had been used across the suite. A step was considered to have been reused if it appeared more than once across the specification; otherwise, it was considered not reused. The following were revealed:

Conservation of Steps: For System 1 and System 3, there was extensive reuse of steps

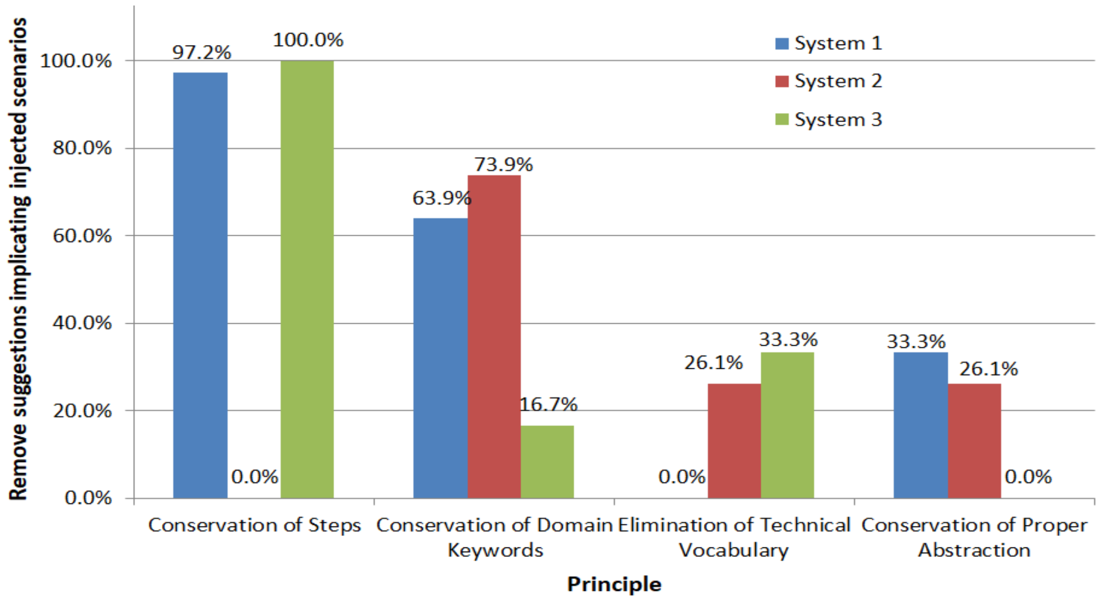


Figure 6.4: Proportions of IO duplicate pairs in which injected scenarios were suggested for removal by the specific principles

across the original scenarios in the feature suites. Since most of the steps introduced by volunteers involved rewording, merging or splitting of the original steps, and the injected steps were generally not reused across the suite, the Principle of Conservation of Steps suggested the removal of injected scenarios in almost all pairs.

For System 2, however, there was not a single pair in which an acceptable remove suggestion (remove an injected scenario and keep an original scenario) was given. Out of the 23 known duplicate pairs, for 17 pairs, the suggestion was to remove either scenario and, in 6 pairs, the suggestion was to remove the original scenario and keep the injected scenario. This was because there was generally no reuse of steps across the feature suite. This was true for both original and injected scenarios. Specifically, for the 17 pairs in which the suggestion was to remove either scenario, each step in both original and injected scenarios was used only once across the feature suite. For the other 6 pairs in which the suggestion was to remove an original scenario and keep an injected scenario, volunteers had split original steps into several new steps, making injected scenarios have more steps than their original counterparts. Because neither original steps nor new steps were reused across the feature suite, the Principle of Conservation of Steps recommended the scenarios with fewer steps (original ones, in this regard) for removal. So the principle might work well when the suite is generally of high quality, but breaks down when it (suite) does not follow the principle in the first place.

Conservation of Domain Vocabulary: While most of the injected scenarios for System 1 and System 2 had fewer domain keywords, compared to their original counterparts, majority of the injected scenarios in System 3 used slightly more domain vocabulary than their original counterparts. That is why in most duplicate pairs in System 3 the recommendations resulting from the Principle of Conservation of Domain Vocabulary were to keep injected scenarios and remove original ones (see Figure 6.4).

Elimination of Technical Vocabulary: The performance of this principle largely depended on the ability to identify, beforehand, the list of known technical words in the scenarios. Of the 74 technical words in our reference list, identified through the process detailed in section 6.4.5, only two words were from System 1. Because there was not a single pair of duplicate scenarios in System 1 in which an injected scenario had used the two words more than its original counterpart, in no pair in System 1 did the Principle of Elimination of Technical Vocabulary suggest an injected scenario for removal (see Figure 6.4). This result suggests that the Principle of Elimination of Technical Vocabulary cannot help us if the suite is already of consistently high quality. Further, in 3 of the 36 pairs of duplicate scenarios in System 1, the Principle of Elimination of Technical Vocabulary made suggestions to remove original scenarios and keep their injected counterparts. This was because, in the 3 pairs, the two technical words were used more number of times in the original scenarios than they were used in the injected scenarios. Moreover, in 4 other pairs, the two implementation words had been used the same number of times in both injected and original scenarios. Thus, the suggestion in each of the 4 pairs was to remove either scenario. Besides, the two words were not used at all in the other 29 pairs, and thus the suggestion in each of these pairs was to remove either scenario.

Nevertheless, this principle suggested for removal injected scenarios in more than a quarter of known duplicate pairs in System 2, and a third of known duplicate pairs in System 3. This was because some of the volunteers who injected scenarios in System 2 and System 3 had used a variety of technical words that were part of our reference list. This demonstrated the importance of the principle–our volunteers had added scenarios that reduced the quality of the suite, and the Principle of Elimination of Technical Vocabulary acted to correct the problem.

Conservation of Proper Abstraction: Because the abstraction levels of the injected scenarios were largely consistent with the abstraction levels of other scenarios across the feature suites for all the 3 systems, only a few injected scenarios (System 1 and System 2) and none from System 3 were suggested for removal by the Principle of Conservation of Proper Abstraction (refer to Figure 6.4). For most of the pairs in which injected scenarios were suggested for removal, to form injected scenarios, steps in original scenarios had been either split into several steps or merged into fewer steps. This caused injected scenarios to have abstraction levels that deviated from the average abstraction levels of the specifications in question.

Q1.2: Sensitivity Analysis on Domain Vocabulary: Figure 6.5 shows the sensitivity analysis results for including and excluding words from the list of domain words.

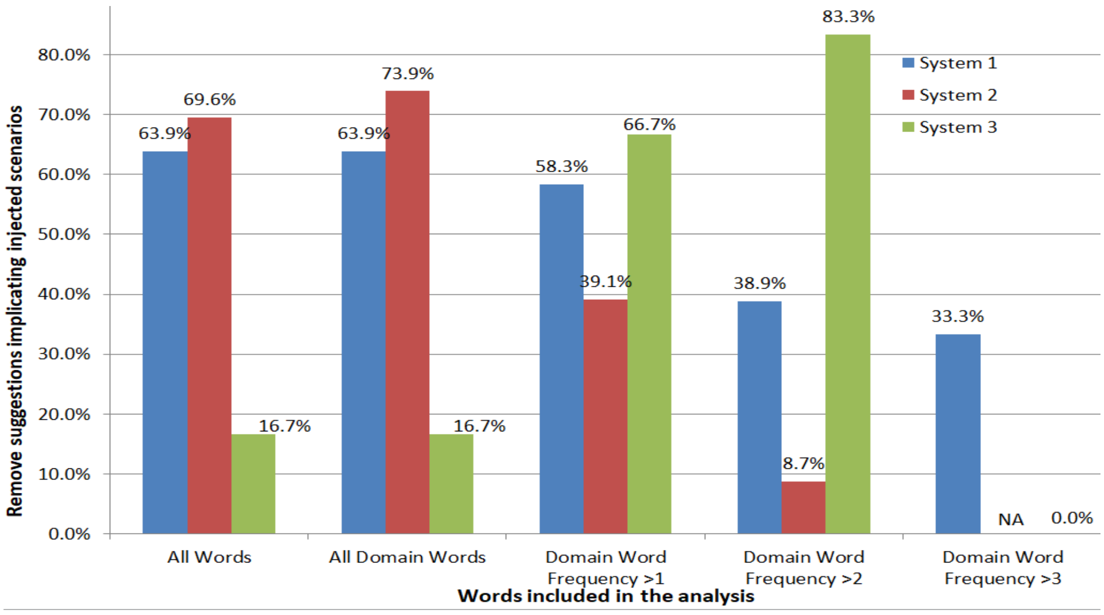


Figure 6.5: Suggestion of injected scenarios for removal: Sensitivity Analysis for the Principle of Conservation of Domain Vocabulary

Referring to Figure 6.5, for System 1 and System 2, the number of remove suggestions implicating injected scenarios decreased with an increase in the frequency of the words included in the list of domain words. This was due to the fact that frequent domain words were mostly used in injected scenarios as well, causing the algorithm to suggest the removal of either injected scenarios or original scenarios, or the removal of original scenarios, keeping injected scenarios.

For System 3, however, the number of remove suggestions implicating injected scenarios was directly proportional to the increase in frequency of words included in the list, for the frequency categories up to and including *Domain Word Frequency* >2. This was because frequent domain words were least used in injected scenarios, causing them to be suggested for removal in most pairs. The category *Domain Word Frequency >3* had no word in it for System 3.

In general, while System 2 and System 3 were too sensitive to changes in the lists of domain words, System 1 was relatively less sensitive to changes in domain words. Since manual analysis of the domain words used across the three systems revealed that there was a relatively high reuse of domain words in System 1 compared to the other two systems, the sensitivity analysis results suggest that the Principle of Conservation of Domain Vocabulary can be useful in situations where a suite has achieved high quality through reuse of domain words.

Q1.3: Removal suggestions for scenarios injected by specific volunteers: Table 6.6 shows the proportion of injected scenarios suggested for removal by each principle, for each of the 13 volunteers.

Volunteer	**System**	**Injected**	**Suggested for Removal**							
			Conservation of Steps		**Conservation of Domain Keywords**		**Elimination of Technical Vocabulary**		**Conservation of Proper Abstraction**	
			Suggested	**%**	**Suggested**	**%**	**Suggested**	**%**	**Suggested**	**%**
1	1	5	5	100.0	5	100.0	0	0.0	4	80.0
2	1	4	3	75.0	3	75.0	0	0.0	0	0.0
3	1	9	9	100.0	9	100.0	0	0.0	5	55.6
4	1	9	9	100.0	9	100.0	0	0.0	0	0.0
5	1	5	5	100.0	5	100.0	0	0.0	0	0.0
6	1	4	4	100.0	4	100.0	0	0.0	4	100.0
7	2	4	0	0.0	0	0.0	0	0.0	0	0.0
8	2	4	0	0.0	0	0.0	0	0.0	3	75.0
9	2	6	0	0.0	0	0.0	0	0.0	0	0.0
10	2	5	0	0.0	0	0.0	5	100.0	0	0.0
11	2	4	0	0.0	0	0.0	1	25.0	3	75.0
11	3	2	2	100.0%	2	100.0	0	0.0	0	0.0
12	3	2	2	100.0%	2	100.0	0	0.0	0	0.0
13	3	2	2	100.0%	2	100.0	2	100.0	0	0.0

Table 6.6: Performance of the four principles on the scenarios injected by each of the 13 volunteers

While almost all scenarios injected into System 1 by volunteers 1 through 6 were suggested for removal when both steps and domain keywords were conserved, generally, the scenarios which were injected by volunteers 7 through 11 in System 2 were not suggested for removal when both steps and domain keywords were conserved (see Table 6.6). For the Principle of Conservation of Steps, this was due to the same reasons stated before in the discussion of the results for Q1.1. However, for the Principle of Conservation of Domain Vocabulary, for 6 pairs from System 2, the use of domain words was slightly more in injected scenarios than in original scenarios, causing original scenarios to be suggested for removal. In the other 17 pairs, domain words were used the same number of times in both original and injected scenarios, causing the algorithm to suggest the removal of either scenario in each of the pairs.

Additionally, while the majority of injected scenarios across the three systems were not suggested for removal by the Principle of Elimination of Technical Vocabulary, mainly because the majority of volunteers did not use the technical words in our list, only the scenarios injected by 5 of the 13 volunteers were suggested for removal by the Principle of Conservation of Proper Abstraction. Manual examination of the injected scenarios suggested for removal by the Principle of Conservation of Proper Abstraction revealed the following. For System 1, volunteers who injected scenarios had merged one or more steps of the targeted original scenarios, raising the abstraction levels slightly higher, and thus deviating from the abstraction level of an overall specification. For System 2, however, volunteers had introduced new steps that ended up having no glue counterparts in the original scenarios. Thus, the extra number of steps with no glue statements affected the metric ASPS (described in section 6.4.4), making the abstraction levels of the injected scenarios deviate more from the average abstraction level of an entire specification. This resulted into higher values of abstraction distance, causing injected scenarios to be suggested for removal.

Q1.4: Performance of the algorithm that combines and prioritises the four principles The percentages of injected scenarios that were suggested for removal by the combined algorithm, for each of the 3 systems, are shown in Table 6.7.

	System 1	**System 2**	**System 3**
# of IO scenario pairs	36	23	6
# of IO pairs in which injected scenarios were suggested for removal	35	10	6
% of injected scenario remove suggestions	97.2%	43.5%	100.0%

Table 6.7: Suggestions of injected scenarios for removal: combined algorithm

Inspecting the results for the combined algorithm revealed that the remove suggestions for System 1 and System 3 were made by the first principle in the order (Conservation of Steps). This is why the results of the combined algorithm on System 1 and System 3 are exactly the same as the results of the Principle of Conservation of Steps in Q1.1 (refer to Table 6.7 and Figure 6.4). Also, the results of the combined algorithm on System 2 revealed that combination of the algorithms for the respective principles allowed other principles to make suggestions where earlier principle(s) in the order could not. Specifically, different from Figure 6.4 where the Principle of Conservation of Steps could not make, on System 2, any of the removal suggestions we expected, the combined algorithm allowed other principles to make acceptable remove suggestions for this system (see Table 6.7).

In general, based on the results for Q1, the idea that injected scenarios are most likely to violate the BDD quality principles was acceptable for some principles, for some systems, and for some volunteers.

6.5.3 Industry Experiment: Design, Results and Discussion

Experiment Design

To answer Q2 and Q3, we worked with an industry collaborator who had 8 years of BDD experience, and held the position of Test Lead in their company. The collaborator acted as an independent judge for suggesting scenarios for removal in selected pairs of duplicate scenarios. As stated earlier, access to original developers was not possible, even though it would have been a much better option for us. But even if we accessed original developers, still, we could not know for definite whether our expert would give the same, better or worse suggestions than the original developers. Working with a BDD practitioner from the industry was the best we could do under the circumstances we had. Additionally, though this judge was not from the development teams of the evaluation systems, the fact that we chose evaluation systems whose understanding did not require specialised domain knowledge (as mentioned in section 4.3.2) meant that any experienced BDD practitioner would easily understand the scenarios in the evaluation systems and make plausible remove suggestions.

Because our industry collaborator had limited time, we could not get remove suggestions for all the 65 known duplicate pairs across the 3 systems. Instead, we randomly sampled 14 duplicate pairs of scenarios from System 1. We chose System 1 because it was the largest of the 3 systems, and had more known duplicate scenario pairs (the ones obtained through our experiment on duplicates injection) compared to the other two systems.

To cover scenarios with diverse characteristics, the sampled duplicate pairs were randomly selected from 6 different features, and the injected scenarios in the 14 duplicate pairs were developed by 4 different volunteers. Moreover, to avoid the possibility of influencing the recommendations made by our industry collaborator, we did not disclose either the proposed quality principles or the information on whether a particular scenario in a given pair was original or injected. Instead, we gave our collaborator pairs of duplicate scenarios (one pair at a time) and asked them to make recommendations on which scenarios should be removed. Thus, using the "think aloud" technique [202], for each of 14 pairs, they were asked to state the scenario they would remove and why. In most cases, they were able to make suggestions by just looking at the scenarios, but, occasionally, they had to inspect glue code before they could make decisions. It took about 1 and a half hours to get remove suggestions for all 14 pairs of duplicate scenarios.

We also wanted to compare the results of the pragmatic approach used in the lab experiment with the results of an expert, with respect to suggesting injected scenarios for removal. Thus, for the sample of 14 pairs of duplicate scenarios, we analysed the remove suggestions made by our tool, to obtain the number of pairs in which injected scenarios were suggested for removal. The results of this process were then compared with the number of pairs in which injected scenarios were suggested for removal by the human expert.

Besides, to determine the extent to which the suggestions made by our tool would agree with the suggestions of a human engineer, for a sample of 14 pairs, we regarded as proxy ground truth the suggestions made by the industry expert, and determined the number of pairs in which both the tool and industry expert made the same suggestions.

Results and Discussion

We now present and discuss the results of the industry experiment.

Q2: Human expert's suggestions of injected scenarios for removal: For each of the 14 pairs of duplicate scenarios from System 1, Table 6.8 shows the expert opinions on the scenarios to remove and the reasons. What is presented in the column labelled

"Reason" is our own summary of the reasons given for the recommendations made on each pair.

Pair	Remove	Reason
1	I	O has more information–I is subsummed by O
2	I	O looks much better
3	I	O has more steps that are not in I
4	O	I has more steps compared to O
5	I	O has more information in steps compared to I
6	I	O has more steps and more information compared to I
7	I	O does more more in glue code compared to I
8	I	O does more more in glue code compared to I
9	O	I has a clearer title than O
10	I	I misses some information which is covered in O
11	Cannot decide	Keep both because they look different
12	I	I has more wording explaining the same thing
13	I	O has more information in the steps than I
14	I	O has a clearer scenario title; and I has more wording with no extra value

Table 6.8: Expert opinions: scenarios suggested for removal and the reasons

Based on the evidence in Table 6.8, generally, the expert preferred to keep scenarios that were readable, with more steps, and rich in information; and it turned out that majority of injected scenarios fell short of these properties. Thus, as can be seen in Table 6.8, the majority of injected scenarios (11 out of 14) were suggested for removal by an expert. Additionally, from the results in Figure 6.6, the expert suggested injected scenarios for removal in more pairs than the pragmatic approach used in our lab experiment, except for the Principle of Conservation of Steps in which the pragmatic approach suggested injected scenarios for removal in slightly more pairs than was the case for an expert. This strengthened our intuition that injected scenarios are most likely to violate the quality principles and, therefore, should be suggested for removal.

Figure 6.6 shows the proportions of System 1 duplicate pairs in which injected scenarios were suggested for removal by both the pragmatic approach used in our lab experiment and the expert-involving approach. Of the 14 duplicate scenario pairs used in our evaluation with the human expert, it was in 11 pairs only (78.6%) in which injected scenarios were suggested for removal. The same expert's value (78.6%) across all the principles in Figure 6.6 is because we used expert recommendations as a proxy ground truth for all the principles. In comparison with the lab experiment, for the Principle of Conservation of Steps, in only 13 of the 14 pairs (92.9%) were the injected scenarios suggested for removal during our lab experiment. This means, in 2 more pairs, injected scenarios were suggested for removal during the lab experiment. In each of the other three principles, the proportion of the 14 pairs in which injected scenarios were suggested for removal during our lab experiment was less than the proportion of pairs in which an expert suggested the removal of injected scenarios. For example, in only 10 out of the 14 pairs (71.4%) used in expert evaluation did the Principle of Conservation of Domain Vocabulary suggest the removal of injected scenarios.

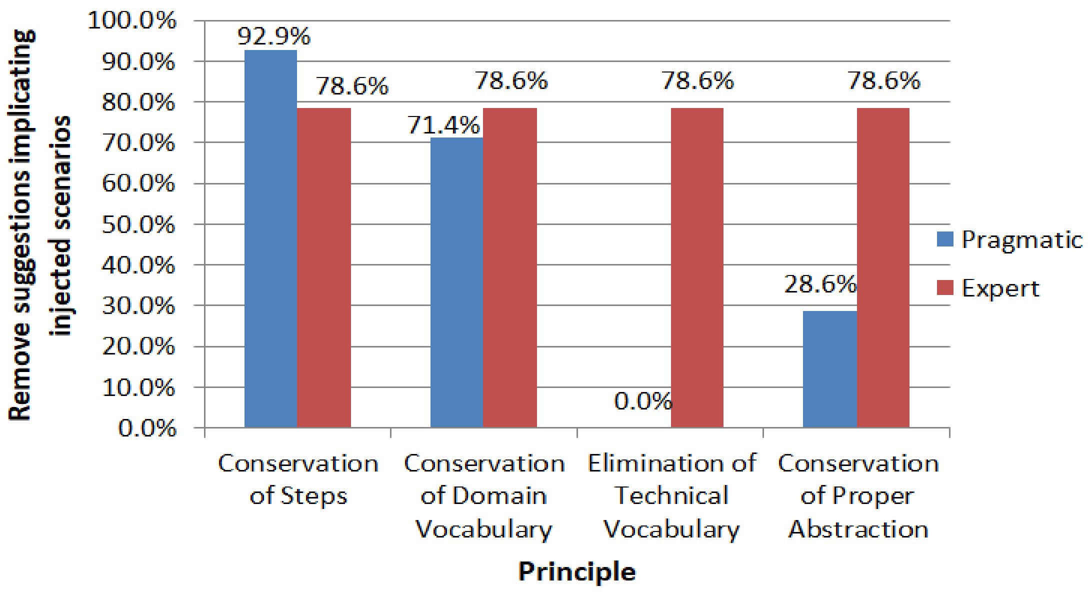


Figure 6.6: Pragmatic vs expert: suggestion of injected scenarios for removal

Q3: Performance of our tool on the expert proxy ground truth: Figure 6.7 shows the performance of our tool on the proxy ground truth created based on the recommendations of the human expert who worked with 14 duplicate scenario pairs from System 1.

From Figure 6.7, we can see that the majority of remove suggestions by both Conservation of Steps and Conservation of Domain Vocabulary matched with the suggestions of the expert. In addition, Conservation of Proper Abstraction agreed with the expert in more than a quarter of the duplicate pairs. This could be attributed to the fact that many developers would determine an abstraction level of the scenario based on

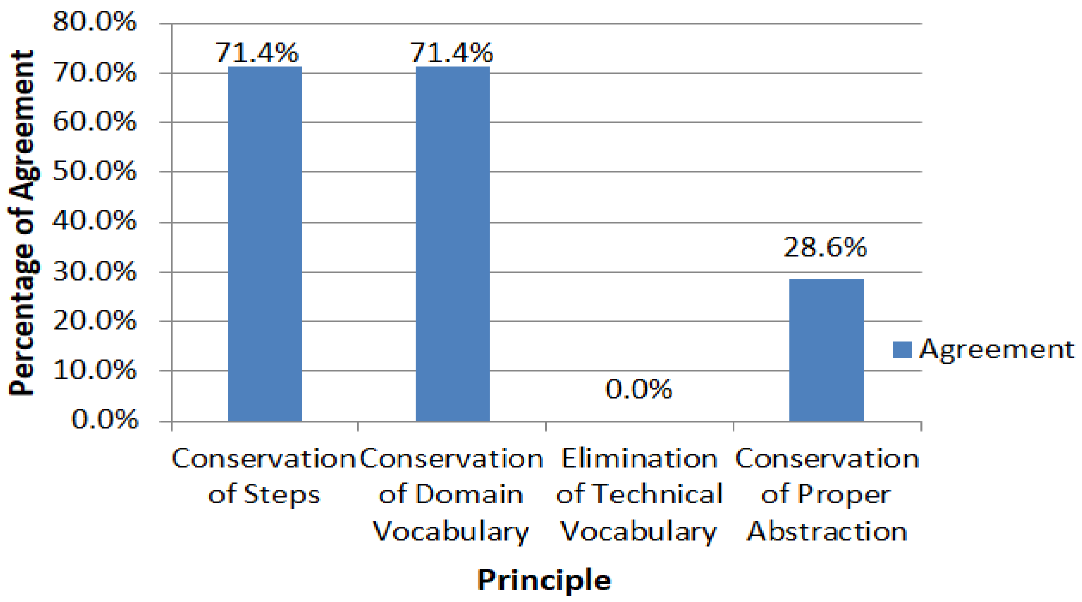


Figure 6.7: Degree of agreement between our tool and expert

its (scenario) intent rather than the requirement to be consistent with the rest of the specification, as discussed in section 6.2.5. More investigation into the best ways to measure appropriate abstraction levels for scenarios would improve the correctness of remove suggestions by automated solutions.

Besides, as stated in the discussion for Q1.1, the fact that only 2 out of 74 words in the list of technical terms came from System 1 largely affected the remove suggestions by the Principle of Elimination of Technical Vocabulary. Thus, the effectiveness of this principle can be better assessed when technical terms are frequently used in a specification.

Answer to Q: *An automated solution that implements the proposed quality principles can give acceptable remove suggestions when a BDD suite has quality violations. However, the principles could be less helpful when a BDD suite is consistently of good or poor quality.*

6.6 Threats to Validity

The validity threats associated with the survey presented in section 6.3 about practitioners' support of BDD quality principles are:

1. Validity threats 1 and 2 in section 3.6, as well as their respective mitigation strategies, also apply to the survey in section 6.3 on practitioners' support of the BDD quality principles.

2. Most of the respondents might have been using a particular BDD tool, so that our results could be valid for users of a specific BDD tool only. To cover practitioners using a variety of BDD tools, we followed the objective criteria mentioned in section 3.2.2 to identify email addresses to which survey completion requests were sent. We also posted the survey in general BDD forums, in anticipation that respondents from those forums might be using different tools.

3. The use of convenience sampling (in our case, depending on self-selecting respondents within the groups we contacted) might limit the ability to generalise from the survey findings. To mitigate the effects of this, we survey respondents from at least 4 continents across the world (see Figure 6.2), and some of the respondents were contributors to sizeable BDD projects in GitHub (see section 3.2.2). Still, our results may not generalise to all BDD practitioners across the world. For example, our results do not represent BDD practitioners who are not proficient in English.

As well, the following are the validity threats associated the experiment in section 6.5 on evaluation of the use of BDD quality principles in suggesting duplicate scenarios for removal:

4. Part of our list of technical words (which also affected the list of domain words we used for individual systems) was identified through intuitive reasoning. This was mitigated in part by the fact that we were working with systems from different domains that required general knowledge to be understood. As such, it was fairly easy to differentiate domain words from technical words, in all the evaluation systems. Also, we adopted a list which was already independently judged as technical words by several practitioners who developed the *cuke_sniffer* tool [201]. This constituted almost a quarter of our list of implementation words. However, the risk still remains, even if mitigated.

5. Some injected scenarios might comply with the BDD quality principles more than their original counterparts, thereby affecting the validity of what we regard as acceptable suggestions in our lab experiment. To mitigate the effect of this, we conducted manual analysis of all the pairs in which injected scenarios were

suggested for removal, across the 3 systems, and our assumption was true in majority of the cases.

6. Our approach for deciding on duplicate scenarios for removal was evaluated on 3 open source systems only, which might affect our ability to generalise from the results. To mitigate the effect of this, we worked with sizeable systems, independently developed for 3 different domains. Also, our evaluation used 65 pairs of duplicate scenarios across the 3 systems, and those duplicate pairs were developed by 13 different volunteers, each of whom duplicated several scenarios in different feature files. This enabled us to use scenarios with diverse characteristics. Nevertheless, in the future, we will evaluate our approach on more industry software systems, involving practitioners for judging the validity of removal suggestions given by our tool.

7. Since the original scenarios are far more likely to be consistent with the wider scenario set than the injected ones, the use of original-injected pairs when deciding which scenarios to remove is likely to suggest more injected scenarios for removal since injected scenarios may not reflect how duplicate scenarios might appear in real projects. However, despite this limitation, the approach enabled us to evaluate the ability of the proposed principles in identifying scenarios of relatively poor quality. Nevertheless, in the future, it would be good to evaluate the principles on real duplicates.

8. The implementation of glue code for injected scenarios, as discussed in section 4.3.4, might have affected quality assessment using the Principle of Conservation of Proper Abstraction. However, since duplicate injection did not create brand new scenarios, glue code for the duplicated scenarios was built naturally by copy-pasting from existing code. This was intended to limit this threat to validity as far as possible.

6.7 Summary

BDD is currently used by industry teams to specify software requirements in a customer understandable language [28]. This produces a collection of examples that act as executable tests for checking the behaviour of the SUT against the specifications. However, large volumes of BDD suites can be hard to understand, maintain and extend.

Duplication, for example, can be introduced by members joining the teams at different points in time. Further, existing work on the removal of duplication in BDD feature suites has mainly focused on the removal of syntactic duplication [33]. However, to the best of our knowledge, no prior work has focused on removing semantically equivalent scenarios from BDD feature suites.

In this chapter, we have proposed and operationalised four quality-preserving principles for guiding the removal of semantically equivalent BDD scenarios. Each principle was supported by at least 75% of the practitioners we surveyed. Also, the four principles gave many acceptable removal suggestions, during our empirical evaluation with 3 open source systems into which duplicate scenarios were injected by volunteers.

To respond to the main hypothesis in the present chapter and the associated research questions:

- The following hypothesis was investigated in this chapter: *we can use quality aspects of scenarios in a BDD suite to guide the removal of duplicate scenarios from a suite*. Evidence presented in this chapter point to the fact that focusing on specific quality aspects of scenarios can provide guidance on which of a pair of duplicate scenarios to remove. Specifically, the work in this chapter has presented evidence that there is some value in the proposed quality principles–they can give reasonable advice in cases where quality breaches covered by the principles are found inconsistently across the suite. But the principles are less helpful when suites are consistently good or of poor quality, and when suites contain more subtle quality issues. The principles cannot judge "readability" of a scenario, for example–a property that was rated highly by the survey respondents, and that probably trumps all over other quality principles. Respondents would prefer to keep the scenario that breaks our rules if it is the more readable one. This suggests a future work idea, looking for general metrics of text readability, to see if they can be applied to BDD suites. This might help to answer the question of whether and how text readability corresponds to the quality of Gherkin specifications.
- **RQ1: On the properties of scenarios that affect the quality of a feature suite as a whole:** The following properties of scenarios were found to relate to the quality of a BDD suite as as whole, and were supported by the majority of BDD practitioners that we surveyed: the reuse of steps in several scenarios across the suite, the use of common domain terms to express the steps of individual

scenarios across the suite, the use of little or no technical terms when expressing steps of scenarios in a suite, and expressing scenarios at a consistent abstraction level across the suite. Other quality aspects of a BDD suite that are not covered by the four proposed principles are summarised in Table 6.5 and in Table 6.1.

- **RQ2: On how the proposed quality properties are perceived in practice:** All the four principles were generally highly supported by practitioners as acceptable facets of quality in BDD feature suites. Each principle received at least 75% votes of support from respondents of the survey of BDD practitioners (sum of "Strongly Agree" and "Agree" in Figure 6.3).

- **RQ3: On the extent to which the proposed BDD scenarios quality properties are successful in guiding the removal of duplicate scenarios from the feature suites:** In general, the Principle of Conservation of Steps and the Principle of Conservation of Domain Vocabulary gave more acceptable removal suggestions, compared to the Principle of Elimination of Technical Vocabulary and the Principle of Conservation of Proper Abstraction. All the four principles were especially observed to give acceptable removal recommendations when the quality aspects they advocate for were breached in a suite.

Chapter 7

Conclusions and Future Work

BDD is now used by many software teams (section 3.3) to allow them to capture the requirements for software systems in a form that is both readable by their customers and detailed enough to allow the requirements to be executed to check whether the production code implements the requirements successfully or not. The resulting feature descriptions, as sets of concrete scenarios describing units of required behaviour, provides a form of *living documentation* for the system under construction (as compared to the passive documentation and models familiar from other approaches to requirements engineering [31]). Unfortunately, management of BDD specifications over the long term can be challenging, particularly when they grow beyond a handful of features and when multiple development team members are involved with writing and updating them over time. Redundancy can creep into the specification, leading to bloated BDD specifications that are more costly to maintain and use. Despite these challenges, studies investigating BDD maintenance challenges are few in number and limited in scope.

7.1 Summary of Research Contributions

1. **Identification of BDD maintenance challenges and research opportunities:** Using quantitative and qualitative data collected through the survey of BDD practitioners, we have gained evidence of the activeness of BDD use in industry, its perceived benefits, and the maintenance challenges BDD practitioners have

encountered, including the challenge of duplication, which is of particular interest to us in this thesis (Chapter 3). We mapped the identified challenges to the literature, to identify those which are still open and which have solutions. From this, we produced a list of 10 open research challenges that have support from the practitioners in our survey (Table 3.6).

2. **A framework for detecting semantically equivalent scenarios in BDD suites:** We have proposed a dynamic analysis based framework in which execution traces are analysed to differentiate between essential and accidental characteristics of BDD scenarios; and, thereafter, the essential characteristics of scenarios are compared to detect semantically equivalent BDD scenarios (Chapter 5). The framework was evaluated on 3 open source systems into which a total of 125 pairs of duplicate scenarios were injected by 13 volunteers. The comparison of scenarios' execution paths was found to detect more duplicate scenarios than the comparison of public API calls, public API calls and internal calls, or their combination (execution paths, public API calls, and public API calls and internal calls). As well, the comparison of essential of only the essential traces of scenarios was found to detect more duplicate scenarios than the comparison of every piece of information in the execution traces of scenarios.

   The tool and data for this work can be accessed here[1] [2] [3] [4].

3. **BDD suite quality principles and their use to guide the removal of semantically equivalent BDD scenarios from suites:** We have proposed four quality principles for BDD suites. These principles encourage the reuse of steps and domain vocabulary, and the use of consistent abstraction levels for steps in various scenarios of a BDD specification; and discourage the use of technical terms in step phrases, to produce scenarios that can be understood by all stakeholders (a very important BDD foundational element) (Chapter 6). We surveyed 56 BDD practitioners about the proposed principles, and each principle was supported by at least 75% of the respondents. In addition, we report on practitioners' other opinions on how to keep BDD suites understandable, extensible and maintainable. Finally, we have proposed algorithms that operationalise the four principles

[1] https://gitlab.cs.man.ac.uk/mbaxrlp2/SEED2/
[2] https://gitlab.cs.man.ac.uk/agile-research/system1-res
[3] https://gitlab.cs.man.ac.uk/agile-research/System2-res
[4] https://gitlab.cs.man.ac.uk/agile-research/system3-res

in the context of a tool to guide the user in the removal of semantically equivalent BDD scenarios. Based on the proxy ground truths we used in our lab and industry evaluation (section 6.5), the principles gave acceptable remove suggestions across three systems. In particular, the Principle of Conservation of Steps and the Principle of Conservation of Domain Vocabulary gave more acceptable removal suggestions, compared to the Principle of Elimination of Technical Vocabulary and the Principle of Conservation of Proper Abstraction. Nevertheless, all the four principles gave acceptable removal recommendations when the suite breached the quality aspects espoused by the respective principles. Thus, the proposed principles can give advice to engineers on which duplicate scenarios to remove from the feature suites.

In general, we learned that it is hard to appreciate the value of the four principles when working with specifications of high quality. However, there seems to be some value in the four principles we proposed, particularly when working with specifications of low quality. But, other quality aspects of BDD suites remain to be covered–especially readability.

The study presented in this thesis has improved our understanding of how BDD is perceived and practiced in the software industry, as well as the challenges facing BDD practitioners. It has also improved our understanding of how duplicate BDD scenarios can be detected and removed from BDD specifications.

7.2 Answers to the Research Questions

RQ1: To what extent is duplication a problem among BDD practitioners, and how do practitioners deal with duplication in BDD specifications? Most respondents thought that, though present, duplication in their BDD specifications remains a manageable problem. However, duplication is still among the maintenance challenges of concern to some BDD practitioners, and, in some instances, it has scared away some practitioners from using BDD. Further analysis of the survey results revealed that about 57% of the respondents were concerned with duplication to an extent–they performed manual inspection to detect and manage duplication, or they had given up the detection and management of duplication due to the complexity of the duplicate detection and management process (refer to Fig. 3.13). We therefore argue that there is a need for more research about automated techniques and tools to support practitioners in detecting and managing duplication in BDD feature suites.

RQ2: Can the comparison of how BDD scenarios exercise the production code detect semantically equivalent BDD scenarios in a way that outperforms the detection of duplicate scenarios by existing approaches? The comparison of the execution paths of the scenarios in BDD feature suites can detect semantically equivalent scenarios with better precision that existing tools. However, even with the comparison of execution paths (which detected more duplicate scenarios than other methods in our experiments), still, we were not getting 100% precision, and on all projects. Also, focusing on only the information that remain constant across several runs of a scenario can be more effective at detecting semantically equivalent BDD scenarios, than the comparison of traces without due regard to essential parts of the trace.

RQ3: What are the characteristics of a "good" quality BDD scenario, and how can we assess the quality of a scenario relative to other scenarios in a suite? We proposed, and managed to gather limited evidence in support of our proposal, that a "good" quality BDD scenario should do the following, without compromising the readability of scenarios across the feature suite:

1. maximize the reuse of step phrases in the suite as far as possible;
2. maximize the use of the ubiquitous language across the suite [200];
3. minimize the use technical terms;
4. use steps whose abstraction levels are consistent with the abstraction levels of steps in other scenarios of the specification.

Thus, given two scenarios *A* and *B* in a suite *S*; if *A* is assessed to be better than *B* on one or more aspects of choice from this list, then *A* can be considered to be of better quality than *B*. Therefore, if *A* and *B* were semantically equivalent, and we were required to remove one of them from *S*; we would remove *B* since it is deemed to be of poorer quality than *A*.

7.3 Future Work

To extend the work in this thesis, there are several possible future research directions. In this section, we categorise future work as follows: future work on duplicate detection, future work on quality of BDD specifications, and future work for BDD as a whole, in the context of the whole software project life cycle and its artefacts.

7.3.1 Future work on duplicate detection in BDD

Future research on duplicate detection in BDD specifications could focus on the following:

- Investigate the best ways to combine the dynamic approaches discussed in Chapter 5 with approaches based on textual and syntactic similarity, perhaps to give more helpful reporting of duplicates that includes suggestions for how the duplication can be resolved most elegantly.
- Evaluate the proposed duplicate detection and removal approaches on more systems, particularly industry systems. This would provide the opportunity to experiment with systems of diverse characteristics that contain real duplicate scenarios, as oracled by actual developers and or domain experts. At the time when we started this work and were creating the benchmark, very few systems with BDD specifications had been made public, and many were toy systems rather than real production systems. This has now changed, based on our recent observations on several software repositories hosting systems (e.g., Github, Gitlab, and Bitbucket). So, we would have the chance to evaluate on a much wider range of systems. Moreover, the system of injecting duplicates had both strengths and weaknesses. Now that we have a version of the duplicate detection tool that we can run, we can adopt a different approach–by sending a report of the duplicates we find to the actual developers and giving them the option of responding to the tool's output. This could counterbalance the strengths and weaknesses of our duplication detection approach.

  In addition, it would also be good to investigate the extent and impact of duplication in industry systems. In particular, it would be good to know about the proportions of scenarios that are duplicated in a variety of real systems, and how that duplication impacts the maintenance of BDD specifications.

7.3.2 Future work on quality of BDD specifications

- On the duplicate scenario removal suggestions using the Principle of Conservation of Steps, it will be good to investigate which scenario to remove when one scenario in the duplicate pair has more steps, but all steps in the duplicate pair of scenarios are used only once in the feature suite. This will address both the possibility of keeping many steps unnecessarily, if a scenario with fewer steps is selected for removal; and the challenge of diminishing the reuse potential of the steps in the specification, particularly if a scenario with more steps is selected for removal, suggesting that, in the future, developers might have to rewrite the steps removed because of the suggestions of the duplicate removal algorithm using the Principle of Conservation of Steps.

- Investigate novel ways to help practitioners to manage steps, terms and abstraction levels of scenarios in the specifications. In particular, it will be good to investigate the following: best ways to merge steps in duplicate scenarios, or scenarios with many steps in common, without altering the scenarios' behaviours or compromising the coverage of the specification; novel ways to inform maintenance engineers about steps existing in the specifications they work with, to avoid unnecessary introduction of new steps; novel ways to guide the use of proper terms and abstraction levels when writing new steps for specific BDD specifications. In connection to the abstraction levels of steps in scenarios, it would be good to investigate the appropriate trade off between complexity in the scenario, step definitions, and the SUT.

- Use more industry collaborators to evaluate the effectiveness of the proposed BDD suite quality principles in guiding the removal of duplicate scenarios. Particularly on the development of the benchmark of known poor quality scenarios in known duplicate scenario pairs, it would be good to have each pair judged by more than one BDD expert, to increase the reliability of the benchmark. It will also be good if the known duplicate scenario pairs will be distributed across reasonably many systems, with diverse characteristics.

- Explore the possibility of forming other BDD suite quality principles out of the quality aspects not covered by the four proposed principles. This would include further exploration of the aspects summarised in Table 6.5 and other comments given on each of the four principles as summarised in section 6.3.4. For example, it would be good to conduct further study on what makes BDD suites more readable, the metrics that can be used to assess readability of scenarios in a BDD suite, and how readability of scenarios can be used to guide the removal of duplicate BDD scenarios.

- It might also be interesting to contextualise the advantages and disadvantages of duplication in BDD specifications. For example, it might be interesting to investigate whether duplication of steps in BDD scenarios increases the readability of a feature suite. Thus, the trade off between the benefits and costs of duplication in BDD specifications might be worthy an investigation.

7.3.3 Future work for BDD as a whole

- In the future, it would be good to investigate the degree of conformity to the BDD workflow by the industry teams. Specifically, for the teams that use BDD on all projects, some projects, or a few pilot projects, it would be good to know the extent to which they strictly observe the dictates of the BDD technique.
- Table 3.6 summarises some of the opportunities for future research in BDD. For example, it would be good to do the following: study the kinds of smells that occur in BDD suites; identify the patterns of change that occur in BDD based systems by analysing commits in public systems, propose refactorings for them, and automate the refactorings for use in development tools.

As yet, BDD techniques have not been studied in great depth by the research community, and we need to gain more data and a better understanding of their strengths and limitations in practice, to allow better tools to be developed.

Bibliography

[1] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[2] http://hanwax.github.io, “The mantra of test-driven development is “red, green, refactor”,” 2019, last accessed 17 July 2019. [Online]. Available: http://hanwax.github.io/assets/tdd_flow.png

[3] U. Enzler, “Acceptance test driven development,” 2012, last accessed 17 July 2019. [Online]. Available: https://www.planetgeek.ch/2012/06/12/acceptance-test-driven-development/

[4] C. K. Roy and J. R. Cordy, “Benchmarks for software clone detection: A ten-year retrospective,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 26–37.

[5] G. Fraser and J. M. Rojas, “Software testing,” in *Handbook of Software Engineering*. Springer, 2019, pp. 123–192.

[6] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[7] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[8] V. Garousi and J. Zhi, “A survey of software testing practices in canada,” *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[9] S. Berner, R. Weber, and R. K. Keller, “Observations and lessons learned from automated testing,” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 571–579.

[10] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 201–209.

[11] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? a multi-vocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, 2016.

[12] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 2007, pp. 261a–261a.

[13] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, "Trade-off between automated and manual software testing," *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, pp. 114–125, 2011.

[14] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual gui testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016.

[15] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[16] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "15 years of software regression testing techniquesa survey," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 05, pp. 675–689, 2016.

[17] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Impediments for software test automation: A systematic literature review," *Software Testing, Verification and Reliability*, vol. 27, no. 8, p. e1639, 2017.

[18] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[19] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

[20] E. Collins, G. Macedo, N. Maia, and A. Dias-Neto, "An industrial experience

on the application of distributed testing in an agile software development environment," in *2012 IEEE Seventh International Conference on Global Software Engineering*. IEEE, 2012, pp. 190–194.

[21] C. Liu, "Platform-independent and tool-neutral test descriptions for automated software testing," in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 713–715.

[22] K. Beck, *Test-Driven Development: by Example*. Addison-Wesley Professional, 2003.

[23] E. F. Collins and V. F. de Lucena, "Software test automation practices in agile development environment: An industry experience report," in *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 57–63.

[24] M. Gärtner, *ATDD by example: a practical guide to acceptance test-driven development*. Addison-Wesley, 2012.

[25] E. Bjarnason, M. Unterkalmsteiner, M. Borg, and E. Engström, "A multi-case study of agile requirements engineering and the use of test cases as requirements," *Information and Software Technology*, vol. 77, pp. 61–79, 2016.

[26] E. Bjarnason, M. Unterkalmsteiner, E. Engström, and M. Borg, "An industrial case study on test cases as requirements," in *International Conference on Agile Software Development*. Springer, 2015, pp. 27–39.

[27] L. Pereira, H. Sharp, C. de Souza, G. Oliveira, S. Marczak, and R. Bastos, "Behavior-driven development benefits and challenges: reports from an industrial study," in *Proceedings of the 19th International Conference on Agile Software Development: Companion*. ACM, 2018, p. 42.

[28] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Maintaining behaviour driven development specifications: Challenges and opportunities," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 175–184.

[29] D. North, "Introducing BDD," Better Software Magazine, 2006.

[30] C. Solis and X. Wang, “A study of the characteristics of behaviour driven development,” in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 2011, pp. 383–387.

[31] M. Wynne and A. Hellesoy, *The Cucumber Book*. Pragmatic Programmers, LLC, 2012.

[32] G. ORegan, “Automata theory,” in *Guide to Discrete Mathematics*. Springer, 2016, pp. 117–126.

[33] S. Suan, “An automated assistant for reducing duplication in living documentation,” Master’s thesis, School of Computer Science, University of Manchester, Manchester, UK, 2015.

[34] M. Mondal, “Analyzing clone evolution for identifying the important clones for management,” Ph.D. dissertation, University of Saskatchewan, 2017.

[35] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[36] C. K. ROY, “Detection and analysis of near-miss software clones,” Ph.D. dissertation, Queens University, 2009.

[37] C. K. Roy and J. R. Cordy, “A mutation/injection-based automatic framework for evaluating code clone detection tools,” in *IEEE International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 157–166.

[38] Y. Sasaki *et al.*, “The truth of the f-measure,” *Teach Tutor mater*, vol. 1, no. 5, pp. 1–5, 2007.

[39] B. Pierre, *Richard E.(Dick) Fairley, 2014. SWEBOK: Guide to the Software Engineering Body of Knowledge v3. 0*. IEEE.

[40] V. Käfer, S. Wagner, and R. Koschke, “Are there functionally similar code clones in practice?” in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. IEEE, 2018, pp. 2–8.

[41] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “Cp-miner: Finding copy-paste and related bugs in large-scale software code,” *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[42] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[43] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[44] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360).* IEEE, 1999, pp. 109–118.

[45] C. J. Kapser and M. W. Godfrey, "cloning considered harmful considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[46] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Cloning practices: Why developers clone and what can be changed," in *2012 28th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2012, pp. 285–294.

[47] W. Hasanain, Y. Labiche, and S. Eldh, "An analysis of complex industrial test code using clone analysis," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS).* IEEE, 2018, pp. 482–489.

[48] M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos, "An exploratory study of functional redundancy in code repositories," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM).* IEEE, 2017, pp. 31–40.

[49] D. Chatterji, J. C. Carver, and N. A. Kraft, "Code clones and developer behavior: results of two surveys of the clone research community," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1476–1508, 2016.

[50] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *2017 IEEE 11th International Workshop on Software Clones (IWSC).* IEEE, 2017, pp. 1–7.

[51] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of cloned code on software maintainability: A replicated developer study," in *2013 20th Working Conference on Reverse Engineering (WCRE).* IEEE, 2013, pp. 112–121.

[52] A. Lozano and M. Wermelinger, “Assessing the effect of clones on changeability,” in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 227–236.

[53] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, “Software quality analysis by code clones in industrial legacy software,” in *Proceedings Eighth IEEE Symposium on Software Metrics*. IEEE, 2002, pp. 87–94.

[54] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, “Is cloned code really stable?” *Empirical Software Engineering*, vol. 23, no. 2, pp. 693–770, 2018.

[55] A. Lozano and M. Wermelinger, “Tracking clones’ imprint.” *IWSC*, vol. 10, pp. 65–72, 2010.

[56] M. Mondal, C. K. Roy, and K. A. Schneider, “An empirical study on clone stability,” *ACM SIGAPP Applied Computing Review*, vol. 12, no. 3, pp. 20–36, 2012.

[57] ——, “An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study,” *Science of Computer Programming*, vol. 95, pp. 445–468, 2014.

[58] J. Harder and N. Göde, “Cloned code: stable code,” *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1063–1088, 2013.

[59] J. Krinke, “Is cloned code more stable than non-cloned code?” in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 57–66.

[60] J. Li and M. D. Ernst, “Cbcd: Cloned buggy code detector,” in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 310–320.

[61] J. F. Islam, M. Mondal, and C. K. Roy, “Bug replication in code clones: An empirical study,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 68–78.

[62] L. Barbour, F. Khomh, and Y. Zou, “Late propagation in software clones,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 273–282.

[63] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug-proneness and late propagation tendency of code clones: A comparative study on different clone types," *Journal of Systems and Software*, vol. 144, pp. 41–59, 2018.

[64] S. Xie, F. Khomh, and Y. Zou, "An empirical study of the fault-proneness of clone mutation and clone migration," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 149–158.

[65] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 55–64.

[66] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[67] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, "Experience of finding inconsistently-changed bugs in code clones of mobile software," in *2012 6th International Workshop on Software Clones (IWSC)*. IEEE, 2012, pp. 94–95.

[68] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 315–324.

[69] A. Fish, T. L. Nguyen, and M. Song, "Clonemap: A clone-aware code inspection tool in evolving software," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0368–0372.

[70] D. Steidl and N. Göde, "Feature-based detection of bugs in clones," in *2013 7th International Workshop on Software Clones (IWSC)*. IEEE, 2013, pp. 76–82.

[71] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[72] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering*, vol. 2013, 2013.

[73] H. Min and Z. L. Ping, “Survey on software clone detection research,” in *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*. ACM, 2019, pp. 9–16.

[74] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (keynote paper),” in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 18–33.

[75] R. Koschke, “Survey of research on software clones,” in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[76] C. K. Roy and J. R. Cordy, “Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[77] S. Ducasse, O. Nierstrasz, and M. Rieger, “On the effectiveness of clone detection by string matching,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37–58, 2006.

[78] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, “Ccaligner: a token based large-gap clone detector,” in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1066–1077.

[79] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcerercc: scaling code clone detection to big-code,” in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[80] N. Göde and R. Koschke, “Incremental clone detection,” in *Software Maintenance and Reengineering, 2009. CSMR’09. 13th European Conference on*. IEEE, 2009, pp. 219–228.

[81] R. Falke, P. Frenzel, and R. Koschke, “Clone detection using abstract syntax suffix trees,” in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[82] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, “Incremental code clone detection: A pdg-based approach,” in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 3–12.

[83] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on.* IEEE, 2001, pp. 301–309.

[84] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2006, pp. 872–881.

[85] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *icsm*, vol. 96, 1996, p. 244.

[86] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics," in *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on.* IEEE, 2010, pp. 241–243.

[87] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering.* ACM, 2008, pp. 321–330.

[88] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[89] R. Tajima, M. Nagura, and S. Takada, "Detecting functionally similar code within the same project," in *2018 IEEE 12th International Workshop on Software Clones (IWSC).* IEEE, 2018, pp. 51–57.

[90] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium.* Springer, 2001, pp. 40–56.

[91] H. W. Alomari and M. Stephan, "Towards slice-based semantic clone detection," in *2018 IEEE 12th International Workshop on Software Clones (IWSC).* IEEE, 2018, pp. 58–59.

[92] H. W. Alomari, M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi, "srcslice: very efficient and scalable forward static slicing," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 931–961, 2014.

[93] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 109–120.

[94] Z. F. Fang and P. Lam, "Identifying test refactoring candidates with assertion fingerprints," in *Proceedings of the Principles and Practices of Programming on The Java Platform*. ACM, 2015, pp. 125–137.

[95] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: detecting similarly behaving software," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 702–714.

[96] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[97] Google, "Google code jam," 2016, last accessed 27 September 2019. [Online]. Available: https://codingcompetitions.withgoogle.com/codejam

[98] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components." USENIX, 2014.

[99] Y. Ishii, T. Watanabe, M. Akiyama, and T. Mori, "Clone or relative?: Understanding the origins of similar android apps," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 2016, pp. 25–32.

[100] J. Park, D. Son, D. Kang, J. Choi, and G. Jeon, "Software similarity analysis based on dynamic stack usage patterns," in *Proceedings of the 2015 Conference on research in adaptive and convergent systems*. ACM, 2015, pp. 285–290.

[101] A. Aiken, "Moss: A system for detecting software plagiarism," *http://www. cs. berkeley. edu/˜ aiken/moss. html*, 2004.

[102] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 81–92.

[103] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally

similar code in complex codebases," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[104] R. ELVA, "Detecting semantic method clones in java code using method ioe-behavior," Ph.D. dissertation, University of Central Florida Orlando, Florida, 2013.

[105] F. A. Fontana, M. Zanoni, and F. Zanoni, "A duplicated code refactoring advisor," in *International Conference on Agile Software Development*. Springer, 2015, pp. 3–14.

[106] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 115–126.

[107] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 331–340.

[108] M. Mandal, C. K. Roy, and K. A. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 114–123.

[109] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[110] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 60–70.

[111] N. Volanschi, "Stereo: editing clones refactored as code generators," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 595–604.

[112] Z. Chen, Y.-W. Kwon, and M. Song, "Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution," *Journal of Software: Evolution and Process*, vol. 30, no. 10, p. e1951, 2018.

[113] R.-G. Urma, M. Fusco, and A. Mycroft, *Java 8 in action*. Manning publications, 2014.

[114] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[115] J. Zhao, "Automatic refactoring for renamed clones in test code," Master's thesis, University of Waterloo, 2018.

[116] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: clone refactoring," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 613–616.

[117] H. Neukirchen, B. Zeiss, J. Grabowski, P. Baker, and D. Evans, "Quality assurance for ttcn-3 test specifications," *Software Testing, Verification and Reliability*, vol. 18, no. 2, pp. 71–97, 2008.

[118] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[119] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 539–540.

[120] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 270–285, 1993.

[121] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Jtop: Managing junit test cases in absence of coverage information," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 677–679.

[122] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," 2018.

[123] B. Hauptmann, E. Juergens, and V. Woinke, "Generating refactoring proposals to remove clones from automated system tests," in *Proceedings of the 2015*

IEEE 23rd International Conference on Program Comprehension. IEEE Press, 2015, pp. 115–124.

[124] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.

[125] C. G. Nevill-Manning, "Inferring sequential structure," Ph.D. dissertation, Citeseer, 1996.

[126] P. Devaki, S. Thummalapenta, N. Singhania, and S. Sinha, "Efficient and flexible gui test execution via test merging," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 34–44.

[127] D. Tengeri, Á. Beszédes, T. Gergely, L. Vidács, D. Havas, and T. Gyimóthy, "Beyond code coveragean approach for test suite assessment and improvement," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–7.

[128] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 130–141.

[129] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[130] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[131] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 107–118.

[132] B. Hauptmann, M. Junker, S. Eder, L. Heinemann, R. Vaas, and P. Braun, "Hunting for smells in natural language tests," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1217–1220.

[133] H. Femmer, D. M. Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *Journal of Systems and Software*, vol. 123, pp. 190–213, 2017.

[134] G. Oliveira, S. Marczak, and C. Moralles, "How to evaluate bdd scenarios' quality?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. ACM, 2019, pp. 481–490.

[135] IIBA, *A Guide to the Business Analysis Body of Knowledge (Babok Guide)*. International Institute of Business Analysis, 2015.

[136] K. Brennan *et al.*, *A Guide to the Business Analysis Body of Knowledge*. Iiba, 2009.

[137] M. Cohn, *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[138] M. Diepenbeck, M. Soeken, D. Grobe, and R. Drechsler, "Towards automatic scenario generation from coverage information," in *2013 8th International Workshop on Automation of Software Test (AST)*. IEEE, 2013, pp. 82–88.

[139] M. Irshad, "Assessing reusability in automated acceptance tests," Ph.D. dissertation, Blekinge Tekniska Högskola, 2018.

[140] L. Amar and J. Coffey, "Measuring the benefits of software reuse-examining three different approaches to software reuse," *Dr Dobbs Journal*, vol. 30, no. 6, pp. 73–76, 2005.

[141] M. Landhauer and A. Genaid, "Connecting user stories and code for test development," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 33–37.

[142] A. Z. Yang, D. A. da Costa, and Y. Zou, "Predicting co-changes between functionality specifications and source code in behavior driven development," in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 534–544.

[143] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[144] M. Pawlan, "Essentials of the java programming language," *Sun Developers Network Tutorials & Code Camps*, 1999.

[145] B. Li and L. Han, “Distance weighted cosine similarity measure for text classification,” in *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2013, pp. 611–618.

[146] R. Forests, “by leo breiman,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[147] G. Kondrak, “N-gram similarity and distance,” in *International symposium on string processing and information retrieval*. Springer, 2005, pp. 115–126.

[148] E. Juergens, F. Deissenboeck, and B. Hummel, “Clone detection beyond copy & paste,” in *Proc. of the 3rd International Workshop on Software Clones*, 2009.

[149] R. D. Fricker Jr, “Sampling Methods for Online Surveys,” *The SAGE Handbook of Online Research Methods*, p. 162, 2016.

[150] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann, “Quantifying developers’ adoption of security tools,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 260–271.

[151] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[152] specsolutions, “Bdd addict newsletter,” 2019, last accessed 25 June 2019. [Online]. Available: https://www.specsolutions.eu/news/bddaddict/

[153] V. Braun and V. Clarke, “Using thematic analysis in psychology,” *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[154] M. Maguire and B. Delahunt, “Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars.” *AISHE-J: The All Ireland Journal of Teaching and Learning in Higher Education*, vol. 9, no. 3, 2017.

[155] J. Iivari and M. Huisman, “The relationship between organizational culture and the deployment of systems development methodologies,” *MIS Quarterly*, pp. 35–58, 2007.

[156] J. Iivari and J. Maansaari, “The usage of systems development methods: are we stuck to old practices?” *Information and software technology*, vol. 40, no. 9, pp. 501–510, 1998.

[157] CollabNet VersionOne, "13th Annual State of Agile Report," 2019, last accessed 6 October 2020. [Online]. Available: https://stateofagile.com

[158] ——, "14th Annual State of Agile Report," 2020, last accessed 6 October 2020. [Online]. Available: https://stateofagile.com

[159] M. Senapathi, M. Drury, and A. Srinivasan, "Agile usage: Refining a theoretical model." in *PACIS*, 2013, p. 43.

[160] M. Senapathi and M. L. Drury-Grogan, "Refining a model for sustained usage of agile methodologies," *Journal of Systems and Software*, vol. 132, pp. 298–316, 2017.

[161] M. Senapathi and A. Srinivasan, "An empirical investigation of the factors affecting agile usage," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 2014, p. 10.

[162] L. Vijayasarathy and D. Turk, "Agile software development: A survey of early adopters," *Journal of Information Technology Management*, vol. 19, no. 2, pp. 1–8, 2008.

[163] D. Bowes, T. Hall, J. Petrić, T. Shippey, and B. Turhan, "How good are my tests?" in *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*. IEEE Press, 2017, pp. 9–14.

[164] D. Gonzalez, J. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan, "A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 391–401.

[165] B. Zeiss, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski, "Applying the iso 9126 quality model to test specifications," *Software Engineering*, vol. 15, no. 6, pp. 231–242, 2007.

[166] M. Greiler, A. Van Deursen, and A. Zaidman, "Measuring test case similarity to support test suite understanding," *Objects, Models, Components, Patterns*, pp. 91–107, 2012.

[167] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On

the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 5–14.

[168] G. Samarthyam, M. Muralidharan, and R. K. Anna, "Understanding test debt," in *Trends in Software Testing*. Springer, 2017, pp. 1–17.

[169] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 547–558.

[170] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the 26th International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[171] M. S. Greiler, "Test suite comprehension for modular and dynamic systems," 2013.

[172] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 387–396.

[173] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 101–110.

[174] R. Ramler, M. Moser, and J. Pichler, "Automated static analysis of unit test code," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 2. IEEE, 2016, pp. 25–28.

[175] M. Waterloo, S. Person, and S. Elbaum, "Test analysis: Searching for faults in tests (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 149–154.

[176] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 207–218.

[177] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 433–444.

[178] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. ACM, 2011, pp. 24–29.

[179] M. Hammoudi, G. Rothermel, and A. Stocco, "Waterfall: An incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 751–762.

[180] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 33.

[181] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, p. 29, 2017.

[182] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang, "A survey on test suite reduction frameworks and tools," *International Journal of Information Management*, vol. 36, no. 6, pp. 963–975, 2016.

[183] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, no. 3, pp. 445–478, 2013.

[184] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 220–229.

[185] A. Rodrigues and A. Dias-Neto, "Relevance and impact of critical factors of success in software test automation lifecycle: A survey," in *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*. ACM, 2016, p. 6.

[186] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 337–346.

[187] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 196–205.

[188] D. Kulak and H. Li, "Getting coaching that really helps," in *The Journey to Enterprise Agility*. Springer, 2017, pp. 197–209.

[189] A. Baah, *Agile Quality Assurance: Deliver Quality Software-Providing Great Business Value*. BookBaby, 2017.

[190] M. Sidman, "Symmetry and equivalence relations in behavior," *Cognitive Studies*, vol. 15, no. 3, pp. 322–332, 2008.

[191] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 78–87.

[192] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[193] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[194] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the dynamic detection of functionally similar code fragments," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 299–308.

[195] A. Okolnychy and K. Foegen, "A Study of Tools for Behavior-Driven Development," Software Construction Research Group, Faculty of Mathematics, Computer Science, and Natural Sciences, Rwthaachen University, Germany, Tech. Rep. FsSE/CTRelEng 2016, February 2016.

[196] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "Xquery 1.0: An xml query language," 2002.

[197] S. Bazrafshan and R. Koschke, "An empirical study of clone removals," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 50–59.

[198] G. Oliveira and S. Marczak, "On the empirical evaluation of bdd scenarios quality: preliminary findings of an empirical study," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2017, pp. 299–302.

[199] ——, "On the understanding of bdd scenarios quality: Preliminary practitioners opinions," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2018, pp. 290–296.

[200] J. Shore *et al.*, *The Art of Agile Development: Pragmatic guide to agile software development*. " O'Reilly Media, Inc.", 2007.

[201] R. Cochran, C. Vaughn, R. Anderson, and J. Patterson, "cuke_sniffer," https://github.com/r-cochran/cuke_sniffer, 2012.

[202] J. Nielsen, T. Clemmensen, and C. Yssing, "Getting access to what goes on in people's heads?: reflections on the think-aloud technique," in *Proceedings of the second Nordic conference on Human-computer interaction*. ACM, 2002, pp. 101–110.

Appendix A

A Survey on BDD Use by Software Engineering Teams and the Challenges of Duplication in BDD Specifications

A.1 Survey Questions

Q1: How would you characterise your organisation? (Options: Public; Private; Voluntary (eg Charities); Other, please specify)

Q2: How would you describe the frequency of BDD use in your organisation currently? (Options: Used on all projects; Used on some projects; Used on a few pilot projects; Not used on any current project; Other(s), please specify)

Q3: Which of the following best summarises the use of BDD in your organisation currently? (Options: Used as a mandatory tool; Used as an optional tool; Not used at all; Other(s), please specify)

Q4: How would you describe plans to use BDD in your organisation in the future? (Options: It will be used as a key tool on all projects; It will be used as an optional tool on some projects; It will continue to be used on a few pilot projects; We do not plan to use it on any future project; Other(s), please specify)

Q5: In general, how does your organisation regard BDD as a tool for software *development?* (Options:Very Important; Important; Unimportant; Very Unimportant; Other, please specify)

Q6: If you currently use BDD/ATDD (Acceptance Test-Driven Development), what tools does your organisation use? (tick all that apply) (Options: Cucumber; FitNesse; JBehave; Concordion; Spock; easyb; Specflow; Other(s), please specify)[1]

Q7: In your opinion, what do you think are the benefits of using BDD in a software project? (tick all that apply) (Options: Software specifications are expressed in domain-specific terms, and thus can be easily understood by end users; Specifications can be executed to confirm correctness or reveal problematic software behaviour(s); Code intention can be easily understood by maintenance developers; Attention is paid to validation and proper handling of data; Could produce better APIs since it emphasizes writing testable code; Improves communication between various project stakeholders; Other(s), please specify)

Q8: In your opinion, what do you think are the challenges of using BDD to develop software systems? (tick all that apply) (Options: Its use changes the team's traditional approach to software development, and that can be challenging; It involves a steep learning curve; Its benefits are hard to quantify; It can lower team productivity; Other(s), please specify)

Q9: What is the typical number of scenarios on the individual systems you work with? (Options: less than 100; 100-1000; 1001-10000; More than 10000 ; Other, please specify)[2]

Q10: In our research, we are looking at the issue of duplication in BDD specifications, and the problems and challenges this causes for stakeholders. By duplication, we refer to BDD specifications in which some functionality is specified multiple times, by different scenarios. Select all the statements below that you think could be associated to the presence of duplication in BDD specifications. (Options: Execution of BDD suites take longer to complete than necessary; Specifications can become difficult to understand in full; Specifications can

[1]Strictly speaking, some of these are more properly termed *Acceptance Test Driven Development (ATDD)* tools, but they were included because of the close relationship and interaction between BDD and ATDD.

[2]To provide context for the maintenance challenges reported, we asked for information about the typical sizes of the BDD suites used and managed by the respondents.

become difficult to extend and change (leading potentially to frozen functionality); Other(s), please specify)

Q11: How would you describe the presence of duplication in any of your BDD specifications? (Options: Present in all our projects that use BDD; Present in some of our projects that use BDD; Could be present in all or some of our projects that use BDD, but we have never paid attention to it; Other(s), please specify)

Q12: Please indicate the extent to which duplication is present in any of your BDD specifications (Options: Present to a problematic extent; Present to a manageable extent; Could be present to any extent, but we have never paid attention to it; Other(s), please specify)

Q13: How do you detect and manage duplication in your BDD specifications? (Options: We perform manual inspection to detect duplication, and thereafter decide how to manage the duplicates we detect; We appreciate the need to detect and manage duplication, but we have decided to live with it, given the possible complexity of the detection and management process; We do not regard the presence of duplication as a problem; Other(s), please specify)

Q14: Please tell us about any other issues regarding your use of BDD that have not been covered in the earlier questions (Free Text)

Q15: Name (Free Text)

Q16: Email (Free Text)

Q17: Job Title (Free Text)

Q18: Organisation Name (Free Text)

A.2 Codes and Themes from the Challenges Reported by BDD Practitioners

This appendix presents the codes assigned to the various BDD challenges reported by respondents (Table A.1), and the sorting of codes to form themes (Table A.2).

Table A.1: Codes for the challenges facing respondent BDD practitioners

Respondent	Reported BDD Challenge	Code
R2	*“ Needing to involve Business and final users”*	Involvement of domain experts and end users
R10	*“Difficult to have the just enough. Difficult to write clear Gherkin Scnarios (Scenarios)”*	Adequate and clear scenarios
R12	*“It’s a simple concept but can be hard to get right. Many people make the assumption it’s about test automation and try to use like a scripting tool and the project ends in failure”*	Improper practice of BDD
R19	*“it does not succeed at being legible to colleagues outside of software engineering departments”*	make non-developers read tests
R20	*“Make other non developers read tests. So far I have used BDD for couple of years and even though idea behind it good, people who are not involved in testing are also not interesting in test cases no matter how easy_to_read they are.”*	make non-developers read tests
R23	*“As with other kinds of testing, the best way to learn is from somebody who has experience. Thus just by downloading a framework, reading a bit and trying, one can produce tests which value is disputable.”*	-Lack of training and coaching -Improper practice of BDD
R24	*“Dsnger (Danger) of confusing the mechanics (automation, written specifications) with the intention (knowledge sharing, structured conversations, discovery of edge cases), focusing too much on the former.”*	Improper practice of BDD

R29	*"Once the gherkin syntax is well known, stakeholders tend to skip ahead, reducing the benefits of the specification workshop"*	Improper practice of BDD
R34	*"Its hard to find someone who really understand what should be tested by BDD therefore a bunch of developer has negative experience about it. Probably there is no a comprehensive material on the internet that can explain every aspect of BDD."*	-Improper understanding of BDD - Scarcity of BDD training material
R37	*"BDD test scenarios still have to be a valid code"*	Additional code
R49	*"Stakeholders don't write specs. Textual specs are too expensive to maintain long-term"*	-stakeholders non-involvement -Hard to maintain BDD specs
R60	*"Productivity is initially (first 6 months) lowered. Beyond that productivity is increased. Every project has a greater setup cost, say 1-3 days to put BDD tooling in place. Hence it is not worth while for trivial projects."*	Initial setup costs
R65	*"requires design skills often absent or not valued"*	- Absence or inapplication of skills
R70	*"Sometimes writing the fixtures requires huge effort and cost"*	High setup efforts and costs
R72	*"BDD practices "by the book" often force domain experts to waste their time and insights trying to think like developers/testers, instead of expressing their needs. Real-world examples often have overwhelming details"*	Impose structured thinking on domain experts and end users
R73	*"All the usual challenges in getting automated testing running and maintained"*	All challenges in automated test suites

R75	*“In my experience, product, business analyst dont even read bdd test, let alone modify, write them in user stories. Using bdd tools that enable writing tests in human language end up being an overhead. We just write functional tests using JUnit by using readable names in business language.”*	-make non-developers write and read tests -Tools overhead

R18	*“BDD tests are heavyweight for developers to use: well written tests are almost always easier to read and run unless you have a very extensive DSL. Writing the DSL is overhead on engineering team so it frequently falls to QA roles. This can result in a poorly written DSL but more importantly it means DSL knowledge moves to the QA team. Eventually it becomes another silo, and a bad one at that. ”* *“My past BDD experience was in another company in which QA was all manual testers with no programming background. We tried to formalize the manual testing checklist using Salad (Lettuce + a webdriver DSL) with the intent to simplify webdriver testing to the point that QA could write their own tests. Over time I became less confident that this could work, tests were very brittle and manual QA types had limited ability to investigate. ”* *“I have read success stories where managers wrote BDD tests and I think this is actually the critical piece: Organisation leadership must show tangible buy-in in the form of regular interaction/writing. Thoughtless delegation to QA results in long-term rot, delegate to engineering results in acrimony as engineers wonder why they can’t use a real language. Without manager buy-in BDD is a pointless layer to test through.”*	-Extra load for developers in creating DSL -Creating a DSL understandable to both developers and QAs -Limited programming skills for some QAs -Brittle and hard to maintain tests -support from management

R20	*"BDD add unnecessary layer of maintaining specification and make them still readable with clean code."*	-Additional layer of code that requires maintatance
R24	*"Extension of answer to 10: Stakeholders "writing" scenarios on their own for "improved efficiency", thereby completely missing the main benefit of BDD - establishing shared understanding."*	Collaboration between stakeholders
R27	*"Stability of user-interface testing frameworks cause problems, and a lack of established standards to handle translating between specification actions..."* *"Additionally, the difficulty of getting junior developers to understand XPath tends to make adoption difficult, especially when the immediate costs are more apparent than the long term benefits"*	-Unstable frameworks -Lack of standards to guide translation of natural language specs to implementation details -Lack of skills -Initial setup costs
R28	*"BDD is often associated with slow suites. The difficulty of managing duplication is proportional to that slowness. Therefore, as BDD scales, in my opinion it is crucial to find ways to run slow scenarios fast, either by reducing their scope, or by running them against multiple configurations of the system covered by the scenarios."*	- Longer suites execution times -duplication management difficulty -Lack of mechanisms to reduce suites execution times
R30	*"Writing full specs is to much for non developers. And developers prefer to write testcases as code, not gerkin. So we end up not doing bdd at all"*	-Make non-developers write specs -Developers writing specs as code
R36	*"...the complexity of the test software needed to support BDD is often as high as the software under test, and the business unit owners never end up writing tests anyway..."*	-Complexity of software to support BDD -Diffulty in making non-developers write specs

R39	*"Writing feature specs at the correct level of abstraction"*	deciding scenario abstraction levels
R45	*"Poor tooling"*	Poor tooling
R65	*"there are very few sources about the structuring of *information* and the conveyance of semantic intent (e.g. e-prime / crispness index). secondly the limitations between using natural language to describe certain analytical problems (the use of labels)."*	-Scarcity of training material -Limitation of natural language on some analytical problems
R67	*"Main issue when applying BDD, it to find time to do the three amigos workshop, it is not a tool issue but more a people one"*	Collaboration between stakeholders
R73	*"Some developers don't like the duplication that an be created with having BDD separate to unit tests. BDD can also get out of hand and become far too technical and indecipherable by users"*	-Additional layer of tests -Hard to understand and maintain
R74	*"Not everyone likes having their specs in such strong conjunction with the sourcecode, some business people rather be vague but personally I think that is because they either are not sure or want to change their mind later. Telling them that changing their mind later is fine, but now I need precise specs, doesn't always work."*	specs in strong conjuction with code

Table A.2: BDD challenges: themes with related codes

S/n	Theme	Code
1	Collaboration between stakeholders	Involvement of domain experts and end users
		Improper practice of BDD
		Improper practice of BDD
		Improper practice of BDD
		Improper practice of BDD
		stakeholders non-involvement
		Ignoring stakeholders collaboration
		Collaboration between stakeholders
		Creating a DSL understandable to both developers and QAs
		Developers writing specs as code
2	Make non-developers read and write tests	make non-developers read tests
		make non-developers read tests
		Impose structured thinking on domain experts and end users
		make non-developers write and read tests

		Make non-developers write specs
		Diffulty in making non-developers write specs
3	Training and coaching	Lack of training and coaching
		Improper understanding of BDD
		Scarcity of BDD training material
		Absence or inapplication of skills
		Scarcity of training material
		Lack of skills
4	Comprehensibility, extensibility and maintainability of tests	Hard to maintain BDD specs
		All challenges in automated test suites
		Brittle and hard to maintain tests
		Longer suites execution times
		duplication management difficulty
		Lack of mechanisms to reduce suites
		Hard to understand and maintain
		Additional layer of tests

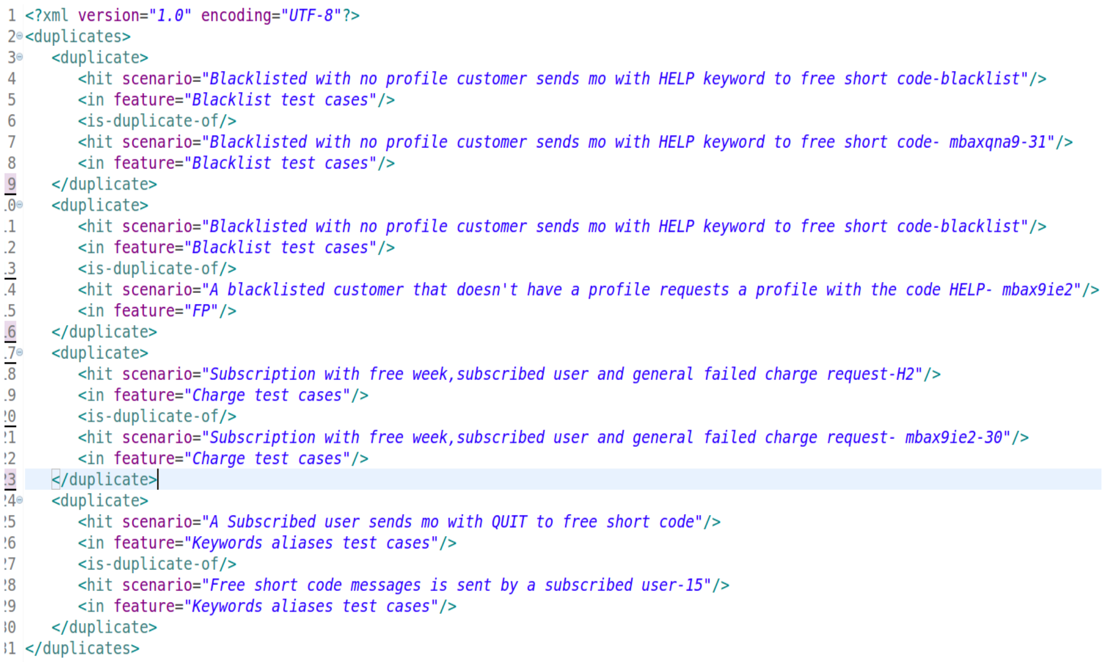
		Additional code
		Additional layer of code that requires maintatance
5	Setup costs	Initial setup costs
		High setup efforts and costs
		Initial setup costs
		Extra load for developers in creating DSL
6	Tools and Sotware	Tools overhead
		Unstable frameworks
		Complexity of software to support BDD
		Poor tooling
7	Miscellaneous	Adequate and clear scenarios
		Limited programming skills for some QAs
		support from management
		Lack of standards to guide translation of natural language specs to implementation details

	deciding scenario abstraction levels
	Limitation of natural language on some analytical problems
	specs in strong conjuction with code

Appendix B

Sample Duplicate Report

B.1 Example duplication report



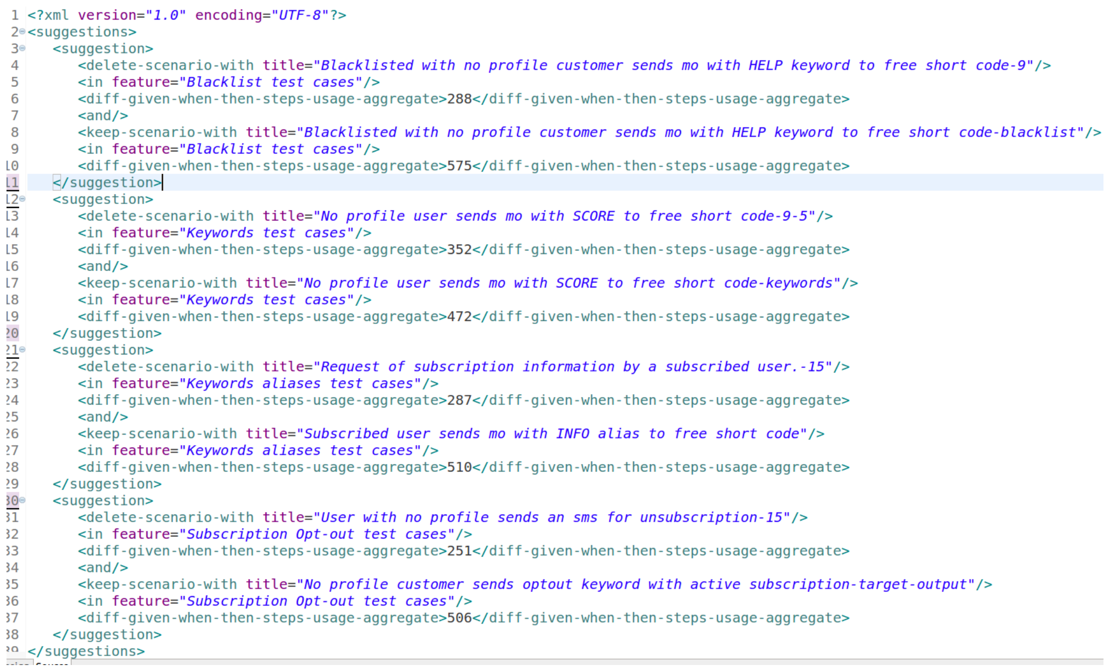
```xml
<?xml version="1.0" encoding="UTF-8"?>
<duplicates>
    <duplicate>
        <hit scenario="Blacklisted with no profile customer sends mo with HELP keyword to free short code-blacklist"/>
        <in feature="Blacklist test cases"/>
        <is-duplicate-of/>
        <hit scenario="Blacklisted with no profile customer sends mo with HELP keyword to free short code- mbaxqna9-31"/>
        <in feature="Blacklist test cases"/>
    </duplicate>
    <duplicate>
        <hit scenario="Blacklisted with no profile customer sends mo with HELP keyword to free short code-blacklist"/>
        <in feature="Blacklist test cases"/>
        <is-duplicate-of/>
        <hit scenario="A blacklisted customer that doesn't have a profile requests a profile with the code HELP- mbax9ie2"/>
        <in feature="FP"/>
    </duplicate>
    <duplicate>
        <hit scenario="Subscription with free week,subscribed user and general failed charge request-H2"/>
        <in feature="Charge test cases"/>
        <is-duplicate-of/>
        <hit scenario="Subscription with free week,subscribed user and general failed charge request- mbax9ie2-30"/>
        <in feature="Charge test cases"/>
    </duplicate>
    <duplicate>
        <hit scenario="A Subscribed user sends mo with QUIT to free short code"/>
        <in feature="Keywords aliases test cases"/>
        <is-duplicate-of/>
        <hit scenario="Free short code messages is sent by a subscribed user-15"/>
        <in feature="Keywords aliases test cases"/>
    </duplicate>
</duplicates>
```

Figure B.1: Example duplication report from our tool

Appendix C

Sample Remove Suggestions

C.1 Sample remove suggestions



```xml
<?xml version="1.0" encoding="UTF-8"?>
<suggestions>
   <suggestion>
      <delete-scenario-with title="Blacklisted with no profile customer sends mo with HELP keyword to free short code-9"/>
      <in feature="Blacklist test cases"/>
      <diff-given-when-then-steps-usage-aggregate>288</diff-given-when-then-steps-usage-aggregate>
      <and/>
      <keep-scenario-with title="Blacklisted with no profile customer sends mo with HELP keyword to free short code-blacklist"/>
      <in feature="Blacklist test cases"/>
      <diff-given-when-then-steps-usage-aggregate>575</diff-given-when-then-steps-usage-aggregate>
   </suggestion>
   <suggestion>
      <delete-scenario-with title="No profile user sends mo with SCORE to free short code-9-5"/>
      <in feature="Keywords test cases"/>
      <diff-given-when-then-steps-usage-aggregate>352</diff-given-when-then-steps-usage-aggregate>
      <and/>
      <keep-scenario-with title="No profile user sends mo with SCORE to free short code-keywords"/>
      <in feature="Keywords test cases"/>
      <diff-given-when-then-steps-usage-aggregate>472</diff-given-when-then-steps-usage-aggregate>
   </suggestion>
   <suggestion>
      <delete-scenario-with title="Request of subscription information by a subscribed user.-15"/>
      <in feature="Keywords aliases test cases"/>
      <diff-given-when-then-steps-usage-aggregate>287</diff-given-when-then-steps-usage-aggregate>
      <and/>
      <keep-scenario-with title="Subscribed user sends mo with INFO alias to free short code"/>
      <in feature="Keywords aliases test cases"/>
      <diff-given-when-then-steps-usage-aggregate>510</diff-given-when-then-steps-usage-aggregate>
   </suggestion>
   <suggestion>
      <delete-scenario-with title="User with no profile sends an sms for unsubscription-15"/>
      <in feature="Subscription Opt-out test cases"/>
      <diff-given-when-then-steps-usage-aggregate>251</diff-given-when-then-steps-usage-aggregate>
      <and/>
      <keep-scenario-with title="No profile customer sends optout keyword with active subscription-target-output"/>
      <in feature="Subscription Opt-out test cases"/>
      <diff-given-when-then-steps-usage-aggregate>506</diff-given-when-then-steps-usage-aggregate>
   </suggestion>
</suggestions>
```

Figure C.1: Sample duplicate removal suggestions based on conservation of steps