The Impact of Test Code Summary to Understand the System Behaviour

Master's Thesis

Tejaswini Dandi

The Impact of Test Code Summary to Understand the System Behaviour

MASTER'S THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Tejaswini Dandi
born in Andhra Pradesh, India

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

The Impact of Test Code Summary to Understand the System Behaviour

Author: Tejaswini Dandi
Student id: 4414845
Email: T.Dandi@student.tudelft.com

Abstract

In the recent past, a new agile methodology, Behaviour-Driven Development (BDD) has been developed which aims to describe a system in terms of behaviour, which helps stakeholders understand the system behaviour and in communication with project members. However, existing projects do not have the advantage of explaining the system in terms of behaviour as in BDD, instead are often done through documentation. Through our paper, we propose a tool to automatically generate high-level context summaries of the test classes describing the behaviour in existing projects. The paper describes how we developed a tool that the behaviour of the system can be summarized from the test cases. In evaluating our approach, we found that the automatically generated summary from a test class 1) is helpful to the stakeholders in order to understand the behaviour of a part in a system, and 2) partially approaches a BDD scenario.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Behaviour-Driven Development (BDD) is a new agile software development approach, that aims to deliver the highest possible value of the product to the customer. Unlike other agile methodologies like Test-Driven Development (TDD) or Acceptance Test-Driven Development (ATDD), where the focus is to improve software quality and productivity [4, 5] by verifying the system's state, instead of the system's behaviour. This 'behaviour' is the value of the product in BDD. BDD focuses on system's behaviour and thus helps in the communication between stakeholders and project members [1]. This communication happens through examples describing the expected system behaviour and the examples are then converted into executable tests, which test the behaviour of the system. These test cases are written in a high-level language in such a way that the stakeholders should be able to understand them [7].

In contrast, existing projects (which are not developed using BDD approach) do not have the benefit of having these BDD tests and thus communication with the stakeholders is often relegated to using other forms of documentations. With documentation, communication with the stakeholders and project members is not as effective as in BDD [1]. This is because documentation speaks about how the system will work for a given input [36], but likely cannot effectively explain for each user story or scenario, how the system will behave. The explanation about the system behaviour with the help of a user story or scenario may help the stakeholders better understand the system. Hence, our objective is to provide summaries explaining the behaviour of the system in the form of scenarios to the stakeholder.

In this research, we aim to present the behaviour of the existing system to the stakeholders by generating BDD like test case descriptions through the summarization of existing lower-level (functional) test cases. More specifically, we will summarize low-level test cases to the use case level and apply BDD scenario template to the summary. Thus, these summaries can be useful to stakeholders in understanding the system behaviour and also to check if the system is exhibiting the expected behaviour.

There are many studies [39, 57, 34, 30, 42, 35] which generate comments for source code. However, there are some approaches like [50, 56] which try to generate high-level summaries for the source code. These approaches aim to summarize the code to mainly help developers, understand the logic behind the source code. These approaches are developed to summarize source code but do not summarize test code. The approach [8] summarizes the test class, giving the description about each statement in the test code. All these approaches are very useful to a developer or a tester to know what does the code do and fix bugs in the code, but these kind of summaries with technical details about the code will not be useful to the stakeholders, as they mainly focus on how a system works. In contrast, in our study, we want to generate high-level summaries for the test code with less technical details which may be useful to the stakeholders.

1.1 Research Questions

To generate high-level summaries for the test classes describing the behaviour of the system, applying high-level summarization techniques to the test class, will provide a better understanding of the test code to the stakeholders. First, we will implement a tool that extracts test classes that represent the behaviour of a system. To accomplish this goal, we detect the integration tests from a test suite, as they show the major parts of a system that work together, providing paths between different parts of the module. To obtain these integration tests, we have to separate them from pure unit tests. This leads to following question:

Research Question 1. How can we generate high-level summarization of the test cases that is enough to bridge the gap between low-level test cases and use case level?

To make the analysis more concrete, we define the following set of sub research questions:

Research Question 1.1. How to find the test classes which can represent the system behaviour?

Research Question 1.2. How to extract integration tests from a test suite?

The next objective is to produce high-level summarization for the integration tests. It is important to determine which information from the test classes should be extracted to fulfil this objective.

Research Question 2. What information from a test class should be used in order to create a high-level summary?

Once we have a high-level summary, it is important to analyse how useful these summaries of the test classes are for the stakeholders to understand the behaviour of the system and how close the summary is to a BDD scenario.

Research Question 3. How much do these summaries help the stakeholders understand the system behaviour?

Research Question 4. How well does these summaries approach a BDD scenario in terms of preciseness, in having unnecessary information and in types of missing information?

1.2 Overview of chapters

To answer these research questions, we study the available literature on this subject in the next chapter. To answer RQ1 and RQ2, we implement a tool which generates behaviour summary in Chapter 3. Chapter 4 discusses about the experimentation procedure and set up. In Chapter 5, we discuss the results about the generated summary answering the RQ3 and RQ4, and discuss possible threats to validity in our study. We review the related literature in chapter 6. Finally, conclusion and implication of our study along possible future research are discussed in Chapter 7.

Chapter 2

Background

This chapter begins with a brief introduction about Behaviour Driven Development along with an example in Section 2.1, followed by discussion about different techniques to summarize methods and classes in the source code in Sections 2.2 and 2.3 respectively. At the end, we give an overview of what techniques can be useful to implement our tool.

2.1 Behaviour Driven Development

Behaviour Driven Development (BDD) is about expressing a requirement in the form of expected behaviour. BDD is a process of exploring, discovering, defining, and then finding out the desired behaviour of a software system [20]. This process is done with the help of conversations, concrete examples to understand the problem that has to be solved for the stakeholders. Then, the examples are refined into automated tests, to describe the desired behaviour of the system. The conversations in BDD process will take place in the Specification or Discovery Workshops. These workshops mainly involve the stakeholders, developers and testers and thus also called "Three Amigos meetings" [7]. During the meeting, the stakeholders come up with a problem, which becomes a "user story", and developers and testers ask for concrete examples about the problem to find out the constraints and requirements to form "scenarios" which will result in a clear understanding of how the system should behave.

2.1.1 Understanding BDD with an example

Let's look at a simple example to know how BDD is performed using user stories and scenarios.
Let the 'User story' be:
As a Student
I request a process to square the number
To gain a faster calculation

And the 'Scenario' be:
Given a variable x with value 30

When I multiply x by 30
Then x square should equal 900

Once we have a scenario with steps, we need to define step definitions to test the scenario steps. Assume that the step definitions are written in `Java`, and will look as in Listing 1.1.

```
public class NumberSquaringSteps {
    int x;

    @Given("a variable x with value $value")
    public void givenXValue(int value) {
    x=value;
    }

    @When("I multiply x by $value")
    public void whenImultiplyXBy(int value) {
    x = x * value;
    }

    @Then("x should equal $value")
    public void thenXshouldBe(int value) {
    if (value != x)
        throw new RuntimeException("x is" + x + ", but should be "
        + value);
    }

}
```

Listing 1.1: Motivating example

After we implement the step definitions, we test the scenarios and see if they pass the tests. When they fail the test, we implement the code that is sufficient to pass the tests. Once the scenario passes the test, it can be said that the expected behaviour of the application has been achieved.

2.1.2 Plain Text Description with User Story and Scenario Templates

BDD provides pre-defined templates and these templates are used during user stories and scenarios. The user story is defined using following template [1]:

Story Title (One line describing the story)

As a **Role**
I request a **Feature**
To gain a **Benefit**

This template gives a clear view on what feature a system should support and why it needs to be supported. The 'story title' represents the activity done by the user. In that specific activity, a user has a given 'role', and the 'feature' provided helps the user to complete the activity and then the user gains a 'benefit'.

The user scenario specifies how a system should respond with an outcome when it is in a particular context and event. The user scenario is defined using the following template [1]:

Scenario 1: **Scenario Title**

Given **Context**
And **Some more contexts**...
When **Event**
Then **Outcome**
And **Some more outcomes**...

For both the defined templates above, the description for the highlighted words should be done using ubiquitous language which is defined for that project, which means that in the later phases of the project these words will be mapped to class and method names.

2.2 Summarization Approach for Methods

To summarize the `java` methods, literature [30, 34, 39] suggests different techniques and strategies. We discuss about the two strategies [34, 30] in this section as they provide relevant information about the techniques in summarizing a method, which we believe that, they would be useful during the development of our tool.

Let us consider the strategy proposed by McBurney et al. [34] as 'Strategy A' and the strategy proposed by Sridhara et al. [30] as 'Strategy B'. Figure 2.1 shows an overview of Strategy A and Figure 2.2 shows an overview of Strategy B.

2.2.1 Step 1: Pre-processing and Software Word User Model

In Strategy A, during the pre-processing phase, it uses *PageRank* to identify the most important methods from the given methods in the context. *PageRank* uses an effective strategy to model a software program as a 'call graph' in which the nodes are methods and the edges are calling relationship between the methods. The methods which are invoked many times

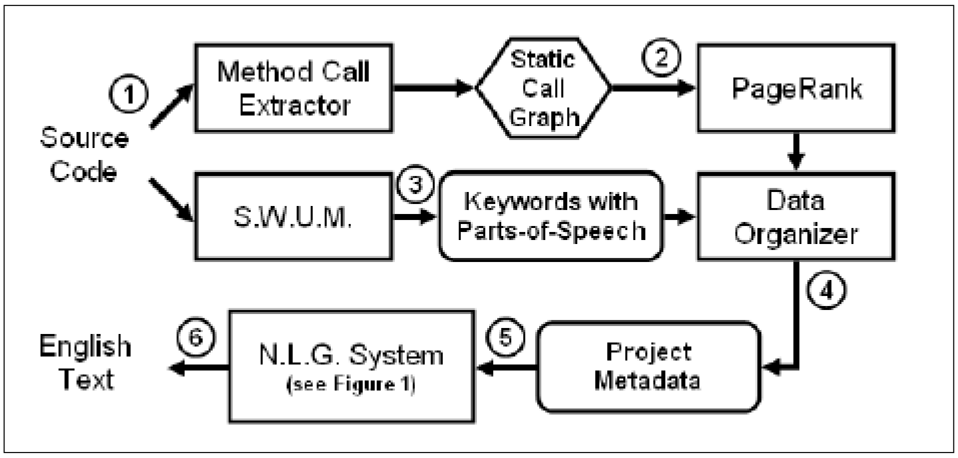


Figure 2.1: Overview of Strategy A [34]

by other methods or the methods which are invoked by other important methods are called as more important methods. So the important methods have more edges compared to the rarely called methods. Hence, the call graph is used to determine the invoked methods in the method's context and the comments are generated for these methods.

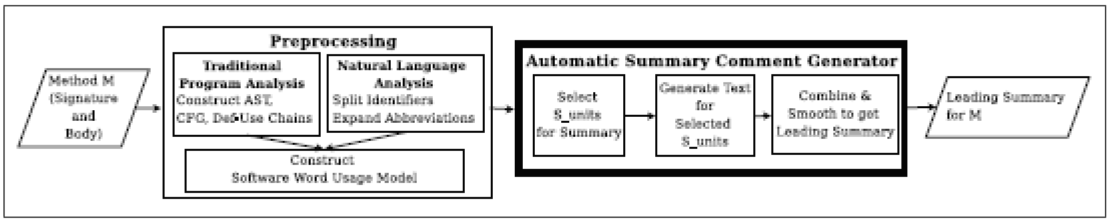

Method M (Signature and Body)
Preprocessing
Traditional Program Analysis
Construct AST, CFG, Def-Use Chains
Natural Language Analysis
Split Identifiers
Expand Abbreviations
Construct Software Word Usage Model
Automatic Summary Comment Generator
Select S_units for Summary
Generate Text for Selected S_units
Combine & Smooth to get Leading Summary
Leading Summary for M

Figure 2.2: Overview of Strategy B [30]

In the pre-processing phase of strategy B, every method is given as input and the identifiers in them are split into component words so that they can be analysed for text generation. The identifiers are split using camel case splitting, where the splitting is done based on capital letters, underscores, and numbers. Sometimes the variable names have abbreviations (e.g., `Button butSelectAll, MouseEvent evt`), which when used in the summary can decrease the readability. So techniques are used to automatically identify and expand abbreviations in the code before generating text.

The above two strategies in this step, commonly use the Software Word User Model (SWUM) [40], which extracts the keywords in the code and identify them as phrasal concepts. To automatically generate the comments, it is necessary to identify the linguistic elements like the *action, theme, and the secondary arguments* in the method. SWUM exactly does the same by identifying the linguistic elements. SWUM captures the words in the code along with their linguistic information and structural relationships. In the form of phrasal concepts, it captures the knowledge expressed through natural language and programming language structure and semantics. It represents the statements in the program as *verbs, nouns, prepositional phrases*. Semantics captured by *action-theme* relationship can be combined with natural language to generate text phrases that represent the code.

2.2.2 Step 2: Data collection

In strategy A, the data from call graph, the identified keywords from the SWUM and the source code of the project are collected. The *Data Organizer* finds the statements that make a call in the code for every method call in the call graph. These statements are collected to provide a concrete example to the programmer. The *Project Metadata* is created by combining the example statements in the call graph and SWUM keywords using Data Organizer.

In strategy B, it defines s_unit selection which is aimed to choose, the important lines of code which can be included in the summary. It defines the following heuristics for selecting s_unit statements:

1. **Identifying major s_unit candidates**: The characteristics that can be used to choose good s_unit statement for the method's summary are Ending s_units, Void-Return s_units, Same-Action s_units, Data-Facilitating s_units, and Controlling s_units.

2. **Filtering out Ubiquitous operations**: The operations such as exceptional handling, logging operations or method's cleanup operations, which are less specific to a method's computational intent are unnecessary to be described in the summary. These can be identified by using AST and by checking if an s_unit is within catch or finally block, or by looking at the *action* and *theme* identified by SWUM for keywords like log, error, debug, close etc.

3. **S_unit Selection Process**: The selection process for s_unit summary has three phases. In the initial phase, Ending, Void-Return and Same-Action s_units are identified and then added to the summary set. Then for each s_unit, Data-Facilitating s_units are added in the summary set. Later, the Controlling s_units are added to the summary set.

2.2.3 Step 3: Summary Generation

In strategy A, the *Project Metadata* from the previous step is given as input to the *Natural Language Generator* (NLG). The NLG has the following steps:

1. **Content Determination**: The information about a method's context is represented using different types of messages. The first one is *Quick Summary Message* which shows a brief and high-level action to summarize the entire method. The second type of message is *Importance Message* which is aimed to provide clues to the programmers on time to spend on reading a method. The third one is *Output Usage Message*, which provides information about the method's output. The next type of message is

Use Message which shows the programmer how to use a method, using an example in the code.

2. **Document Structuring**: All the messages obtained from the previous step are organized into a single document with the message order: Quick Summary Messages, Output Used Messages, Called Messages, Importance Messages, and then Use Messages.

3. **Lexicalization**: This step decides the phrases that can be used to describe the message types in the step Content Determination. These phrases are not complete sentences, but these phrases will be grouped with other phrases later in the Aggregation step and in the last step Realization, they are formed into sentences.

4. **Aggregation**: During this phase, phrases generated during the lexicalization phase are used to create more complex and readable phrases. The sentences are formed by grouping the phrases of the messages which are the patterns of message types the system looks for.

5. **Surface Realization**: This strategy uses an external library, *Simplenlg* [41] to form complete sentences from the phrases in the previous step.

In strategy B, in the "Text Generation" step , the summary content from set of s_units needs to be converted into understandable natural language phrases. The text generator initially constructs the subphrases for the arguments using lexicalization and all the subphrases are then concatenated for the entire s_unit.

Lexicalization of Variables: The general information about the variable is known from the variable's type name and specific information from the variable name. So to generate an English noun phrase for the variable, the type name is appended to variable name. For example, `Document current` would be lexicalized as "current document". But when the type name is an adjective, then the phrase will be constructed as type name followed by variable name. The basic s_unit has a single method call, and the text generation strategy for the single method call is used in returns, nested method calls, composed method calls, assignment and loop and conditional expressions.

2.2.4 Overview

The table 2.1 provides an overview of different method summarization techniques. The initial step these techniques propose is Pre-processing and a common model called SWUM model. Then in the second step, each study proposed different ways of collecting the information like statements, branching conditions in the method. Once the important information has been extracted, then in the third step, the extracted information is used to generate summaries for the methods.

Steps	Technique
Pre-processing	Collecting important methods [34]; creating abstract syntax tree for every method [30]; using SWUM model [34, 30]
Data Collection	Collecting keywords from source code [34]; different s_units from source code [30]
Summary Generation	Lexicalization of keywords and Aggregation of text phrases [34, 30]

Table 2.1: Overview of method summarization techniques

2.3 Summarization Approach for Classes

Moreno et al. [42] proposed a technique for summarization of classes. The initial step is to identify the elements in the class which describe the class like interfaces, parent classes, attributes and methods. Then, these are divided into two sets based on the analysis required, where the first set contains interface names, parent class names, and inner class names and the second set contains attributes and method names. This approach only considers individual classes and does not consider the communication between the classes during summarization.

In object oriented programming, there are two types of responsibilities for classes: (i) generic responsibilities which are domain independent and (ii) specific responsibilities which are domain dependent. So to describe the responsibility of a class, class stereotypes are used, are explained in the Section 2.3.1 [44]. Since the domain specific information is given by identifiers of fields and methods, the quality of the summaries depends on the identifiers. But this strategy lacks in analysing the word relations, i.e., domain analysis or domain artifacts like domain vocabulary, that determine the quality of the identifiers, so the summaries quality implicitly depends on the identifiers. The summary has 4 parts and are explained in the following sections:

1. A description based on the parent class, interfaces, or class stereotypes.
2. The description of the structure based on stereotypes.
3. The description of the behaviour based on counting the relevant methods.
4. The list of existing inner classes.

2.3.1 Step 1: Stereotype identification

1. **Method stereotypes:** The responsibility of a method within the class is described using method stereotypes. There are 15 types according to [45, 46] and they are divided into 4 categories:

- **Structural methods**: There are two types of structural methods. They are: 1) Accessors, methods return some information about the object through a parameter, but do not change the state of the object and 2) Mutators, methods change the object state and returns a value.
- **Creational methods**: These methods destroy or create the object.
- **Collaborational methods**: These methods define the communication between the objects.
- **Degenerate methods**: These methods does not read or write to the object's state directly or indirectly.

2. **Class stereotypes**: The intent of the class in the system is represented using class stereotypes. According to [44] there are 13 different stereotypes namely, Entity class, Minimal entity class, Data provider class, Commander class, Boundary Communicator class, Factory class, Controller class, Pure controller, Large class, Lazy class, Degenerate class, Data class and Small class.

2.3.2 Step 2: Heuristics for Content Selection

Once the classes are determined using class stereotypes, then the methods which should be included in the summary need to be identified. To identify these methods, two types of filters are used on the set of all methods in the target class:

1. **Stereotype-based filter**: This filter removes the methods which are irrelevant to the class stereotypes by its definitions. For each class stereotype, one heuristic is applied.
2. **Access-Level filter**: This filter depends on the access levels permitted by modifiers of the method. The main focus is on the most visible responsibility of a class, so the methods are removed from least visible to more visible following the stop rules.

2.3.3 Step 3: Text Generation

This step provides relevant information about the class found in the previous step in a readable text description. The templates are defined for the four parts of the summary which is discussed above.

1. **General Description**: The summary will start by describing the type of objects which represent the class. So parent class names and interface names are used as qualifiers to represent the object.
2. **Stereotype Description**: The stereotype definitions are enhanced with information like classes used in the target class, the represented object, or presence of certain kind of methods.

3. **Behaviour Description**: To describe the behaviour of the class relevant methods are filtered using different filtering techniques as discussed above. The behaviour description is divided into 3 parts, where the first block and second block say about accessor and mutator methods, and the last block says about the remaining methods in the class. To provide a readable description about the method in natural language, lexicalization of fields and phrase generation for methods is used. The lexicalization of fields is done in the same way as described in Strategy B in summarization of methods.

2.3.4 Step 4: Inner class enumeration

This summarization step is optional as it is only available when the class declares an inner class.

2.4 Summarization Approach for Test cases

Panichella et al. [8] proposed a tool *TestScribe* which is aimed to generate automatic summaries for the JUnit test cases and some part of the target code that is being tested. The approach for generating test case summaries has the following 4 steps and is shown in Figure 2.3:

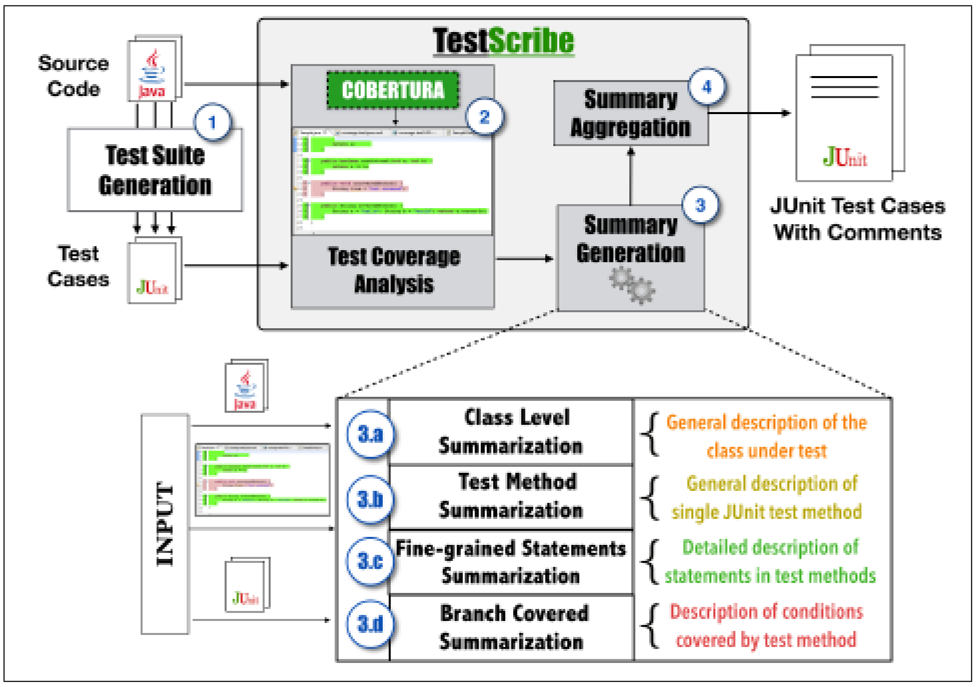

TestScribe
Source Code
Test Suite Generation
Test Cases
COBERTURA
Test Coverage Analysis
Summary Aggregation
Summary Generation
JUnit Test Cases With Comments
INPUT
Class Level Summarization
General description of the class under test
Test Method Summarization
General description of single JUnit test method
Fine-grained Statements Summarization
Detailed description of statements in test methods
Branch Covered Summarization
Description of conditions covered by test method

Figure 2.3: Overview of TestScribe [8]

2.4.1 Test Suite Generation

In this initial step, the tool *TestScribe* uses EvoSuite [29] for automatically generating the JUnit test cases for code written in Java. EvoSuite uses an algorithm that generates individual test suites depending on the coverage criterion where the search was conducted using the fitness function [29] for the test targets like statements, branches. In order to provide test cases which are more clear, the test suite is post processed by reducing its size and by maintaining the maximum code coverage. The assertions are added to the test cases by generating them using mutation based heuristic. At the end of the generated test suite, testers can revise the assertions manually.

2.4.2 Test Coverage Analysis

To figure out the statements and branches that are being tested by individual generated test case, *TestScribe* uses Cobertura [38]. In order to generate the coverage summaries, information related to the elements in covered code statements like method calls, attributes, branches, etc. is required. So *TestScribe* builds a parser on the top of Cobertura to collect the information about: (i) attributes and methods in the code which are directly or indirectly called by the test case; (ii) the statements which are executed, attributes used, and calls to other methods from the called methods in the code; (iii) the boolean values (true/ false) from the branch statements to determine which part of the code is verified. This step produces the list of code elements and lines of code covered by every test case.

2.4.3 Summary Generation

This step provides a higher-level view of the code that is going to be tested by each test case. To produce that view, *TestScribe* uses Software Word Usage Model which selects the natural language phrases/words from the statements that are being covered by the test cases. To generate a summary, there are 3 steps that are being implemented: 1)Pre-processing, 2)Part-of-speech tagging 3)Summary generation at class level, method level, branch level and for statements.

2.4.4 Summary Aggregation

The summary aggregator provides the natural language summaries and descriptions along with the JUnit test class. The summaries are presented as: (i) a block of comments before the test class describing the code under the test; (ii) a block of comments before each test method body describing the statements with coverage scores for that method; and (iii) inline comments for the corresponding statements with fine grained descriptions.

2.5 Discussion

The above discussed techniques can be useful during implementation of our tool. Since we want summarize a test class and its test cases by our tool, we find the techniques: identifying important methods using a call graph and using Cobertura to detect the covered classes during pre-processing phase, SWUM model to extract the keywords in the code in data collection phase, and lexicalization and aggregation in summary generation phase useful.

Chapter 3

Implementation

In this chapter, we explain how we have implemented a tool, which generates a behaviour summary from a test class and answering the research questions 1 and 2. In the section 3.1, an overview of the approach to implement the tool is described an the later sections explain each step we followed to attain the behaviour of a test class.

3.1 Overview of the approach

As said in the introduction, the main aim is to generate a summary where we will summarize low-level test cases to the use case(high) level and apply a BDD test case template to the summary. In order to provide a high-level summary, there are two main steps to be followed: (i) we need to find test classes through which we can represent the behaviour of the system and (ii) we summarize these classes to provide a behaviour summary of the system. These two steps are highlighted in the Figure 3.1 along with the sub-steps that are necessary to be attained are shown in the dotted boxes.

Since the first step is to find test classes from the existing system which capture the behaviour of the system, one way to find out these classes would be identifying the integration tests (IT). The reason to consider integration tests is, they show the major parts of a system that work together, providing paths between different parts of the module. These paths allow us to figure which classes are being covered or called by an integration test. These covered or called classes can provide us information about background functionality of the system. Therefore determining integration tests in the system will be our initial step.

To find the integration tests in a system, it is necessary to distinguish integration tests from unit tests. Unit testing is testing the smallest execution unit while integration testing is interfacing among the units to demonstrate that the units are collectively operable [2]. A unit test can be a class or a single method as they are fine grained and errors can be found at lower level. According to Feathers [3], there are few rules to indicate test is an integration test and not a unit test:

- a test uses a database

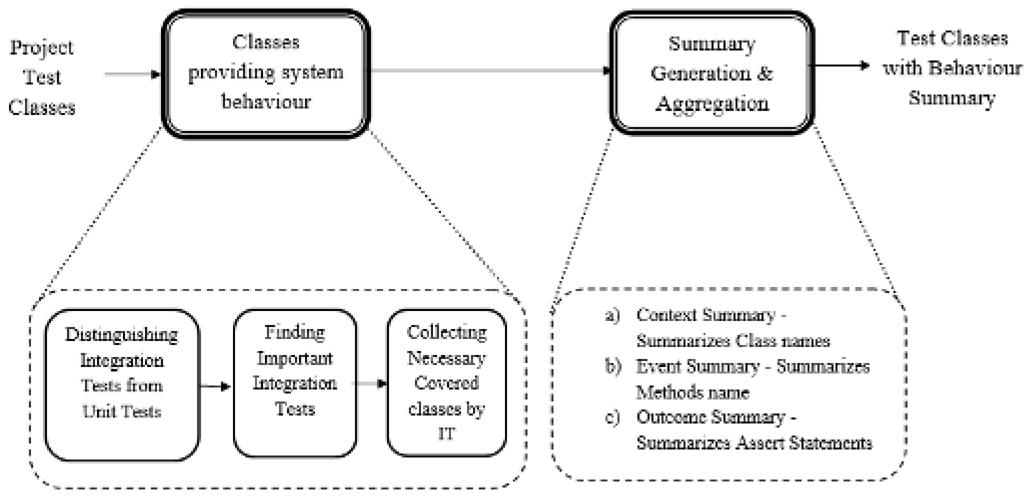

Project Test Classes
Classes providing system behaviour
Summary Generation & Aggregation
Test Classes with Behaviour Summary
Distinguishing Integration Tests from Unit Tests
Finding Important Integration Tests
Collecting Necessary Covered classes by IT
a) Context Summary - Summarizes Class names
b) Event Summary - Summarizes Methods name
c) Outcome Summary - Summarizes Assert Statements

Figure 3.1: Approach Overview

- a test communicates over the network
- a test read from or writes to files or perform other I/O operations on file system
- a test uses external system (e.g a queue or a mail server)
- a test cannot run at the same time as other unit tests

JUnit is a testing framework for the Java programming language, which helps programmers to write and run their test cases written in Java. The JUnit test cases are considered during this research because of its widespread use as a testing platform in Java applications. In JUnit, the *test command* is a test method and is annotated with `@Test`. The test command contains one or more *assertions*, which evaluates an expression. One or more test commands are written for a public method of a class and the test commands are grouped together if they share common setup, forming a test class.

Once we have integration tests, the next step would be finding the coverage information for each test. This coverage information will be useful in determining which parts of a system work together. These covered parts consists of classes, which represent the system behaviour. To obtain the coverage information, we use the tool *Cobertura* [1] which is based on *jcoverage* [2].

We then summarize the integration tests with maximum coverage, as these type of tests can help us provide communication between the source classes, which in turn help us know the behaviour of the system. To summarize them, we get the information such as the class

[1]`http://cobertura.github.io/cobertura/`

[2]`http://java-source.net/open-source/code-coverage/jcoverage-gpl`

name, methods names from the covered classes. We rely on a tool TestDescriber [8] which automatically generates test class summaries. The summaries follow the BDD template, which is as follows [1]:

Given **Context**
And **Some more contexts**...
When **Event**
Then **Outcome**
And **Some more outcomes**...

The highlighted text 'Context, Event' and 'Outcome' are filled with the text representing behaviour of a class and is discussed in the later sections of this chapter.

To capture the classes providing the system behaviour, three steps are to be followed: 1) Distinguishing integrations tests from unit tests, 2) Finding important integration tests and 3) Collecting necessary covered classes by important integration tests. These steps are explained in the Sections 3.2, 3.3 and 3.4 respectively.

3.2 Distinguishing unit and integration tests

A test is considered to be a unit test when it uses *mock objects* to replace the real objects like network, file system adapters, and real database. A mock object is a substitute implementation to stimulate the behaviour of a real object in controlled ways [6]. The advantage of using mock objects is the isolation of the unit tests from influences like network time-outs, file system errors or slow database connections that are not problems with the unit itself. Generally, we consider a test as an integration test when the mock object is replaced by the real implementation of that object. We can differentiate the tests, if we can find the class under test, all the objects called from a test and the mock objects, are explained in the below sections.

3.2.1 Finding the called classes

Parsing the test class

In order to register which objects are being called by a test method, we used the technique to parse through the test method in a test class. In a test method, each method call of a class is detected using a regular expression while parsing through the code. The method being called and the class of the method are stored as a result along with the test method. We collect a list of all the classes and their respective methods that are called for each test method.

We register the method calls from five different methodTypes, each identified by the JUnit annotation:

- `BeforeClass`, which indicates that the method is run once, mostly for initialization of the test class.
- `Before`, which indicates that the method is called before every test method.
- `AfterClass`, which indicates that the method is run once when all the test methods in a test class have been run.
- `After`, which indicates that the method is executed after every test method.
- `Test`, which indicates a test method in a test class.

We detect each test method in a test class, if there is `Test` annotation in front of it and then parse through each test method to identify the method call in them. Considering the test methods with `Test` annotation, will also remove the method calls(if any) present in between any two test methods. Even though the other methods with annotations as `BeforeClass, Before, AfterClass`, and `After` are not test methods, since these methods are being used by the test method, classes that are being called from these methods will be added to the list of called classes. For example, in Figure 3.4 we collect all the underlined sentences in `@Before, @After and @Test` as a method call along with the class name. Since we depend on the JUnit annotations, we require the tests to be written in JUnit version 4 or up.

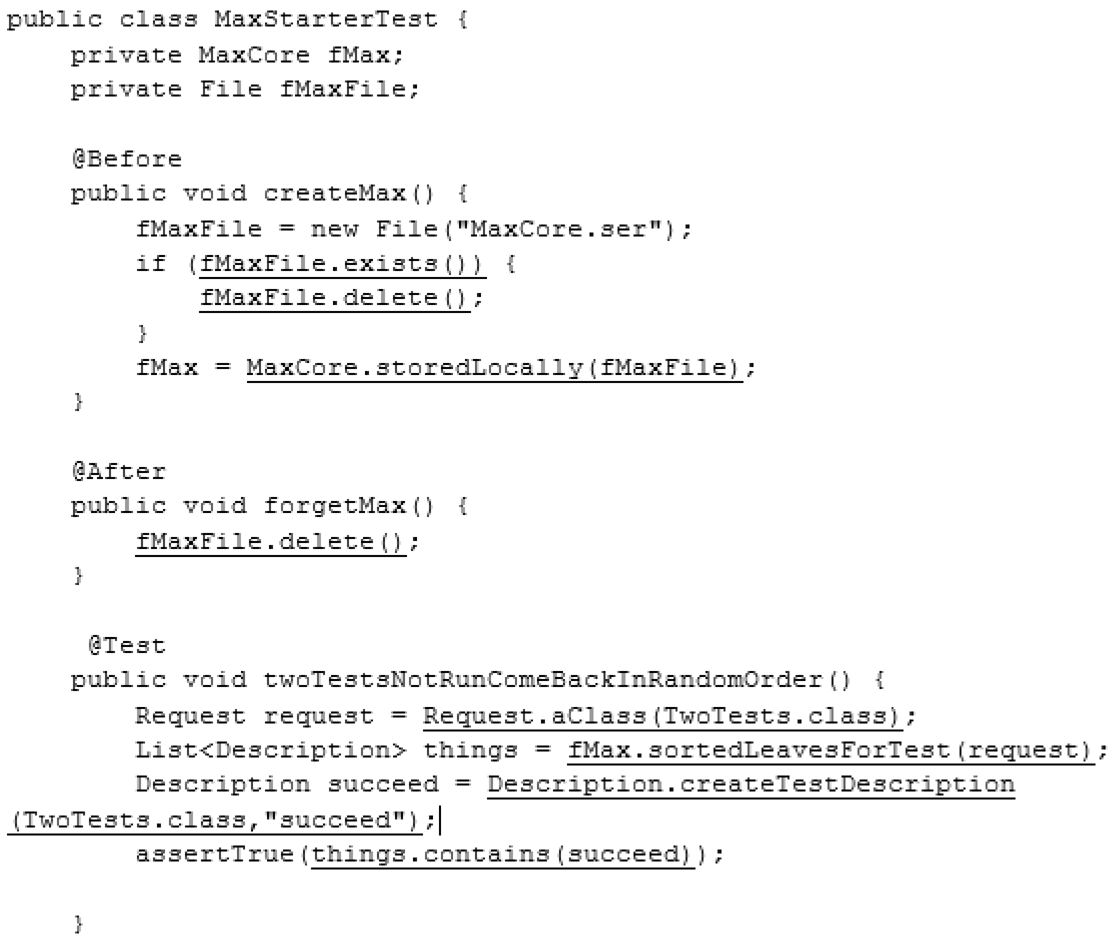
Next we remove the unwanted classes and methods through filtering techniques.

Filtering

We do not like to consider all called classes in the list, as usage of some called classes will not render a test as an integration test. For example, call to a core Java class method need not be considered, as it will not help us in detecting what kind of test it is. We created two lists namely “blacklist” and “whitelist”, to filter the unwanted list of classes and methods.

1. **Blacklist**

   The initial way to limit the called classes is to create a blacklist. This list contains several core Java classes and their methods which are considered as low level and will not help us in differentiating the type of test. However, there are few methods in the core Java classes which depend on the system configuration. For example in Figure 3.3, `java.util.Locale` has a method `getDefault()` which depends on the current system settings, so if a test uses this method, it is potentially not an unit test. So we remove this method from the blacklist. Similarly, we remove similar other methods such as `getDefault, getDefaultRef` in `java.util.TimeZone` class, `java.util.Date` with `Date` object, `java.lang.System` with `currentTimeMillis` etc which use current system settings.



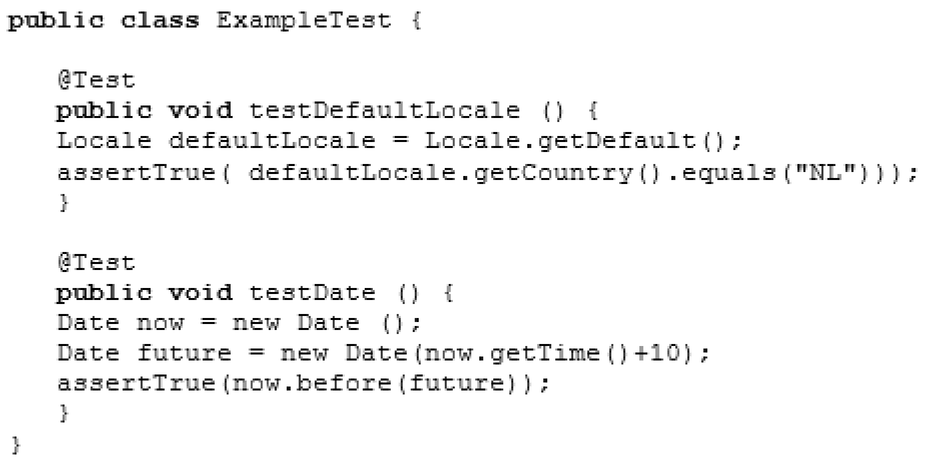
```
public class MaxStarterTest {
    private MaxCore fMax;
    private File fMaxFile;

    @Before
    public void createMax() {
        fMaxFile = new File("MaxCore.ser");
        if (fMaxFile.exists()) {
            fMaxFile.delete();
        }
        fMax = MaxCore.storedLocally(fMaxFile);
    }

    @After
    public void forgetMax() {
        fMaxFile.delete();
    }

     @Test
    public void twoTestsNotRunComeBackInRandomOrder() {
        Request request = Request.aClass(TwoTests.class);
        List<Description> things = fMax.sortedLeavesForTest(request);
        Description succeed = Description.createTestDescription
(TwoTests.class,"succeed");
        assertTrue(things.contains(succeed));

    }
```

Figure 3.2: Detected called methods from the code

2. **WhiteList**

   While a good number of low level and unwanted classes are removed from the called classes list by the blacklist, there can still be other classes from the used frameworks or external libraries that a project can depend on. We create a whitelist which defines the list of classes that are used by the test methods from the dependencies. This list contains listing of all the single classes and entire packages from the external libraries or frameworks used by the project.

   This list can also be directly added to the blacklist, but creating a separate whitelist will enable the user to change this list according to the project and its dependencies, making the whitelist project dependent. For example, if we consider the JUnit test suite, listing the JUnit framework as whitelist will have a negative effect on the results.



```
public class ExampleTest {

    @Test
    public void testDefaultLocale () {
    Locale defaultLocale = Locale.getDefault();
    assertTrue( defaultLocale.getCountry().equals("NL")));
    }

    @Test
    public void testDate () {
    Date now = new Date ();
    Date future = new Date(now.getTime()+10);
    assertTrue(now.before(future));
    }
}
```

Figure 3.3: Blacklisted called methods from the code

3.2.2 Finding the class under test

We assume that a test class has a only a single class, `XClass` as class under test, if it fulfils some of the following conditions:

1. if the test class name is `XClassTest`, `TestXClass` or `XClassTests`;
2. if the test class has only a single called class, `XClass`;
3. if the test class uses class, `XClass` which is present in the list of called classes.
4. if the (sub)package of the test class is same as the `XClass`;
5. if the inner classes of `XClass` are used.

If a test class is fulfilling the first three conditions every time, then we can atleast consider it as an unit test. If it fulfils all the above conditions, we can almost say that it is an unit test [14]. This gives us a list of test classes that are unit tests and we can separate these test classes from the list of the test classes in a project. The remaining list of test classes in the project can now be considered as integration tests. Even though we cannot exactly say that all the remaining test are integration tests, but we can atleast ensure that the list will not contain any unit test.

3.2.3 Finding mock objects

Mock objects can be created using two ways: (i) creating mock objects manually and (ii) using mocking framework like Mockito or EasyMock. We detect the manually created mock objects if there is usage of “Mock” in the class name. We then blacklist the class with “Mock” in their names to remove them from the list of called classes. The second method allows a way not to collect classes that are being mocked, by whitelisting all the relevant methods from the mock framework.

Project Name	**Unit**	**Int.**	**Total**
Apache Common Digester	10	190	200
Apache Common Collections	209	796	1005
Maven Core	8	229	237
Maven Model	81	67	148

Table 3.1: Classification of tests as unit tests and integration tests

3.2.4 Unit or Integration Test

After following the above discussed steps, a test class is determined as an unit test or integration test depending on the number of classes in list of called classes after filtering using blacklist and whitelist and removing the class under test. If the test method has no called classes in the list, then we say it is an unit test. If the test method has a list which contains other classes, then we say it is an integration test.

We implemented our tool to distinguish integration tests and unit tests on 4 open source projects: Apache Common Digester, Apache Common Collections, Maven Core and Maven Model. In the table 3.1, the number of integration tests and unit tests detected from the 4 projects are presented.

3.3 Determining important Integration Tests

From step described in Section 3.2, we have a list of integration tests from which we need to detect the important tests. As mentioned earlier in the Section 3.1, to detect the behaviour of a system, we will consider the integration tests which test maximum classes. In order to find an integration test which tests most classes, we came up with two techniques: (i) using a call graph to find the important test and (ii) using Cobertura to find the test coverage. We explain in the below sections how we implemented these techniques.

3.3.1 Using Callgraph Technique

We initially came up with the idea of using the call graph to identify important methods. The call graph is an effective strategy, in which the nodes are the methods and the edge is the calling relation between the methods. According to [34], the methods which are invoked many times or which are invoked by other important methods are called as more important methods than methods which are called rarely. Thus, the important methods have more edges compared to the rarely called methods. Using the call graph, we count the number of incoming calls and outgoing calls to a method. For example, *r2* node in Figure 3.4 is considered as important node as it is the most invoked node. We used Java agent, a `jar` file which contains an agent class having some specific methods to support instrumentation. We run Java agent on the set of classes with methods in order to track their invocations. With the help of Java agent, we produce a caller-callee relationship, along with number of calls.

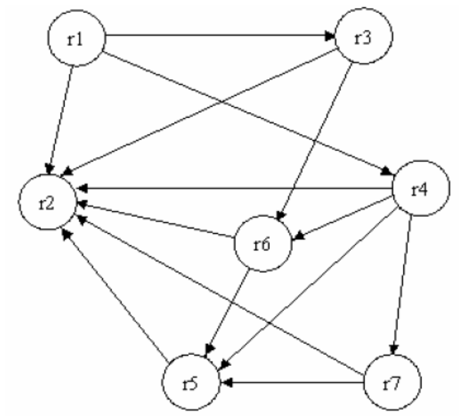


Figure 3.4: Callgraph

The drawback of this technique, was about finding the threshold of the incoming and outgoing calls. It was difficult to conclude with what number of incoming and outgoing calls we can decide a method as important. It can be possible that a method can only have few incoming or outgoing calls but still can play a very important role in a project. The next drawback is, while defining a call graph above, we said the importance of a method depends on the number of times it is invoked. However, we argue that if a method is called many times, it may be a low level implementation and we claim that the method with more outgoing calls will be important as it distributes its work to other low level methods. Hence, we moved to another method (in Section 3.3.2) which does not deal with the caller-callee relationship but depends on the number of classes a single test class is covering.

3.3.2 Using Cobertura tool

In this technique, we used Cobertura to obtain coverage information of a test class. In order to get the classes a test class is covering, we need to perform three main steps:

1. instrumenting the compiled classes of the test class,
2. run the JUnit test case and
3. generate an XML report containing the coverage information.

In order to implement the above steps, we created a class, `CoberturaRunner`, which executes them. From this class, we run the Cobertura instrumentation script to instrument the compiled classes. Then, we run each JUnit test case in the test class using Cobertura as profiling tool. After running the test class, we generate an XML file with the coverage information using Cobertura coverage script. The coverage information contains details about all the packages and their classes in the project. Each package, class and method will have its own line and branch coverage along with their names. The methods will have the

Test Class Name	**No. of Covered classes**
MaxStarterTest	156
ParameterizedTestTest	99
Junit38SortingTest	96
Junit38ClassRunnerTest	90
AnnotationTest	90
SingleMethodTest	87
ForwardCompatibilityTest	86
SuiteTest	86
TestRuleTest	83
CategoryTest	82

Table 3.2: Number of covered classes by the test classes from JUnit project

return value along with their names, providing information about lines (with line numbers) that are covered and branches if present. The attribute `hits` of a method will provide the coverage of a line; if it has a value equal to 1 then, the line is covered and if the value is 0 then, the line is not covered.

Now we have the coverage information for all the classes in an XML file, we need to parse the file in order to extract the necessary information like class names and lines which are covered. We parse through the XML file and get the information about the class name if `line coverage` is greater than 0 and the line numbers if `hits` is greater than 0. This results in all the classes that are covered by a single test class. We follow the same procedure for all the test classes to determine their coverage information. The importance of a test class depends on the number of classes covered. The higher the number of covered classes, the higher the importance of the test class. The following table 3.2 shows the data of highest number of covered classes by the test classes in JUnit project.

figure about the listing the example of no.of covered classes

3.4 What type and how many covered classes to be summarized

After obtaining the important integration tests, we need to determine the number of covered classes that can be used for generating the behaviour summary. To achieve this, we created a parameter *level*, which gives the depth of the covered classes. This means, the main class under test is considered as *first level* of the covered classes, and to derive the covered classes for the *second level*, we look at the lines covered in main class, find the method calls or objects instantiated in it. For the *next level* of covered classes, again we consider only the lines covered with method calls or objects instantiate in the *second level* classes and so on. So first, we need to decide the value of *level* of a covered class that could be considered to be summarize a test class, is discussed in Section 3.4.1. It is necessary to identify the *level* of covered classes because if we summarize all the covered classes by a test class, it would not be a reasonable context and length for a behaviour summary and the classes at the bottom level could be lower-level classes.

3.4.1 Determining the threshold value of level of a covered class

To determine the threshold for *level*, we have hypothesized two ways:

i finding the number of classes until x level, or

ii finding the covered classes which communicate more frequently than other covered classes.

In the first case, we have pragmatically parsed through all the covered classes at each level until fourth level for top 10 out of all the important integration tests. We conducted this experiment on the JUnit framework. While parsing through each level of covered classes, we found that the number of covered classes from level 1 to 4 are in-between 15 to 20. We discovered that the covered classes at third and fourth level are commonly covered by most test classes. We also found that the covered classes are mostly low-level classes at fourth level. The above two reasons were the cause not go into deeper levels and stop at level four to determine the covered classes.

In the second case, we also parsed through the covered classes until fourth level same as in first case, but we looked for frequent method calls between covered classes. In other words, we looked for methods which are communicating more frequently between some covered classes and we would consider the covered classes with stronger communication value. However, we could not find frequent communication between some covered classes, rather the method calls to different classes are mostly once.

From the above two scenarios, we have chosen the first method to determine the threshold for level, as the second method did not provide us appealing results during our pragmatic experimentation to use it. We have considered to summarize covered classes until fourth level and used the parsing technique to parse through the covered lines of covered classes at each level and determine each method call of a class using a regular expression. The class and method being called from the parsed statement of a covered class is stored as a result of covered class of next level. Then we again use the list of covered classes by the previous level to detect the next level of covered classes and so on. This results in all the covered classes until level four for each test class. The table 3.3 provides the list of test classes (with highest number of covered classes) along with their coverage at level 2, 3, and 4, and level 1 value is always one as it is main class under test.

3.4.2 Filtering Unwanted covered classes

As discussed in Section 3.4.1 in the first case, we found classes which are commonly covered and there are lower-level classes at level three and four. While parsing these levels, we also found that there are few inner classes and their methods which are being collected

Test Class Name	No. of Covered classes at		
	Level 2	**Level 3**	**Level 4**
MaxStarterTest	5	9	15
ParameterizedTestTest	3	7	10
Junit38SortingTest	4	8	12
Junit38ClassRunnerTest	5	7	9
AnnotationTest	2	9	12
SingleMethodTest	3	7	10
ForwardCompatibilityTest	3	6	10
SuiteTest	4	9	10
TestRuleTest	3	7	9
CategoryTest	2	6	8

Table 3.3: Number of covered classes at each level by the test classes

as covered classes and methods, and there are few abstract methods in the called methods. We hypothesized that eliminating the below listed classes will provide a better summary without unnecessary information.

1. Removing *commonly covered classes*: As said previously, when we did the experiment over 10 test classes, we detected that few classes are commonly covered by them. These covered common classes also indicate that they provide some basic functionality which other classes depend on, meaning they are more a kind of lower-level classes. So we have collected all the commonly covered classes by the test classes in the JUnit project. Then, remove these classes from the covered classes until level four of each test class, which can now used in the next step.

2. Removing *inner classes*: Once we have a list of covered classes without common classes, we now refine this list by discarding inner classes. We discarded inner classes from being part of a summary because an inner class is more specific to the class which it is present in and will have no use elsewhere. This means, the parent class can provide a higher-level explanation of what is happening in it and it is not necessary to include the inner classes in the summary. To collect the inner classes, we simply parse through all the classes in the source project and detect inner classes in them to form a list of inner classes. We now remove the inner classes from the list of covered classes from the previous step resulting in a refined list without inner and common covered classes.

3. Removing *abstract methods* from covered methods: There were few abstract methods that are encountered during the parsing of covered classes. We refine these methods from the list of covered methods for a class since abstract methods do not perform any kind of functionality.

Test Class Name	**No. of Covered classes at**		
	Level 2	**Level 3**	**Level 4**
MaxStarterTest	2	4	7
ParameterizedTestTest	2	3	7
Junit38SortingTest	3	5	6
Junit38ClassRunnerTest	3	4	5
AnnotationTest	1	6	8
SingleMethodTest	2	3	7
ForwardCompatibilityTest	1	4	7
SuiteTest	2	5	6
TestRuleTest	1	4	6
CategoryTest	1	3	5

Table 3.4: Number of covered classes at each level by the test classes after filtering

After filtering the commonly covered classes, inner classes, and abstract methods, the table 3.3 will now have the values as in the table 3.4.

3.4.3 How many classes to summarize

After the filtration on each level of covered classes, we have more than 10 classes to summarize for each test case until level 4. So we need to prune some classes, when we reach the list (of covered classes) size equal to or greater than 10 at some level, we stop parsing the next level and use this list for summarization. The reason we stop at number 10 is because if we summarize more than 10 classes, the summary would be lengthy which likely cause lack of interest to read it. For instance, from first and second level we have 7 covered classes and from third level we have 3 classes making total 10 classes, we will not parse through the fourth level. However, if in third level we have 4 classes making the total as 11, we will not consider the classes from third level too as using some classes out of all from a level will be improper way to collect classes from a particular level.

This provides a list of covered classes which may not contain unnecessary information, not making the summary lengthy and these classes can be summarized in the next step. To learn if the summary contains any unnecessary information, we conducted an experiment and the results are discussed in Chapter 5.

After collecting the required number of covered classes, we need to summarize each covered class and it is done implementing the following section.

3.5 Summarization technique

The goal of this step is to provide a higher-level summary explaining the behaviour of the system. So we summarize the list of essential covered classes by a test class from the previous step. First, JavaParser [3] is run on the list of covered classes to collect information from each covered class such as (i) the list of *attributes* and *methods*; (ii) for each invoked method the parser collects all the attributes/variables used, and calls to other methods from the executed statements. To generate the behaviour summary, we have used the Software Word Usage Model (SWUM) proposed by Hill et al. [32] to extract the natural language phrases from the covered classes. We utilised the SWUM tool developed by TestScribe [8] to create the summaries.

To automatically generate the comments or summary about the code, it is necessary to identify the linguistic elements like the *action, theme*, and the *secondary arguments* in the method. SWUM exactly does the same by identifying the linguistic elements. SWUM captures the words in the code along with their linguistic information and structural relationships. It identifies the knowledge expressed through natural language and programming language structure and semantics. It represents the statements in the program as *verbs, nouns, prepositional phrases*. For instance, the *verbs* present in the method names are viewed as *actions* and the *theme* is found in the remaining method name, class name and formal parameters. Consider an example, `list.add(Item i)` which can be phrased as “add item to list” where the *action* is “add”, the *theme* is “item” and the *secondary argument* is “list”. It is also assumed that the method names start with the verbs according to the Java naming conventions. So SWUM can assign the *verb* as an *action* and search for the *theme* in the remaining part of the name, formal parameter and the class. But this will fail when methods like `str.length()` or `obj.toString()` are encountered, then SWUM assigns the *action* as “get” or “convert” for the method. So it can be said that, for text generation, semantics that are captured by *action-theme* relationship combined with natural language to provide phrases that accurately represent the code.

We follow three steps to generate a summary: 1) Pre-processing, 2) Part-of-speech tagging and 3)Summary Generation and Aggregation. Each step is explained in the following sections.

3.5.1 Pre-processing

To identify the linguistic elements in the covered classes, the TestScribe tool used Java camel case convention [8, 30] which splits the names of the identifiers to component terms. The identifiers are split based on capital letters, numbers and underscores. To expand the identifiers and type names, the TestScribe tool used (i) a technique called *contextual-based expansion* [31], which finds the most appropriate expansion for an abbreviation present in the class and method identifiers and (ii) an external English dictionary which contains common short forms for English words [33].

[3] `https://github.com/javaparser/javaparser`

3.5.2 Part-of-speech tagging

After extracting the main terms from the identifier names, the TestScribe tool used a Part-of-speech (POS) tagger called *LanguageTool*[4] which categorizes the terms as *verbs*,*adjectives* and *nouns*. *LanguageTool* is a Java library which provides a lot of semantic tools for many languages such as POS tagger, translator, spell checker, etc. When the Part-of-speech tagging of the terms is finished, then it is determined if the terms should be used as Noun Phrase(NP), Prepositional Phrase(PP), or Verb Phrase(VP) [8]. Depending on the type of phrase, natural language sentences are generated using pre-processed and POS tagged attributes, variables of methods and classes, using a set of heuristics used by *Hill et al.* [8] and *Sridhara et al.* [30].

3.5.3 Summary Generation and Aggregation

As described previously in Section 3.1, we will use the pre-defined BDD scenario template with elements, *contexts*, *events* and *outcomes* as our template and is filled with the output of the SWUM, i.e, the pre-processed tagged code elements from the covered classes. The number of scenarios created for each test class depends on the number of covered classes after filtering from the previous step. The elements in the template are filled in the following way:

- **Context:** The *context* will contain the information about a covered class namely the class name. The reason to consider the class name is because a scenario speaks about what will happen in a particular situation or setting and this what a class mostly deals with, as normally when we read a class name, we can get a brief idea of what a class will do.

  Once we have the class name, pre-processing and POS tagger are performed to identify the verbs, adjectives and noun phrases in it. These linguistic elements are then used to fill as *context* in a scenario. In Figure 3.5, text represented in the blue color is the context in a scenario.

- **Event:** The *event* is described with a method names from a single covered class. When a test class covers a class, we said that the class name can be a context; however a class name can only provide an outline while the covered methods in the class actually provide what kind of functions it is performing. Thus, we believe describing method names as *events* in a scenario is plausible. In Figure 3.5, text represented in the red color are the different events in a scenario.

[4] `https://github.com/languagetool-org/languagetool`

Given maximum core
When build runner, find leaves, stored locally, get malformed test class, sort request,construct leaf request and sorted leaves for test
Then maximum core method name is equal to fast and max core method name is equal to slow

Given junit38 class runner
When filter, make description, create adapting listener, set test, get annotations, get description and get test
Then child test count is equal to 1

Given a test suite
When test the Count, add a test suite, add a test method, check if test method, create a test, get test constructor, warning, add a test, add tests from test case, check if test method is public and run test

Given error reporting runner
When get causes, describe cause, run cause, get class names and get description

Given junit core
When remove listener, run main, default computer, run classes and add listener

Given suite
When get children, run child, describe child and get annotated classes

Given description
When create test description, format display name, get display name, add child, get children, check if test, test count, check if empty, get annotation, get test class, get method name, and method and class name pattern group
Then list of description has succeed, do not succeed and list of description size is equal to 2, description is equal to slow, description is equal to fast

Given request
When class without suite method, classes, runner, filter with description
Then count match of test class is equal to request count

Given result
When get run count, get failure count, get failures
Then run count is equal to 2

Figure 3.5: Automatically generated summary

Since a method implements an operation, it's name typically begins with a verb [8] defining an *action* while the *theme* and *secondary arguments* are determined from the parameters. This information is pre-processed and POS tagged to identify the linguistic elements to fill the *events* in the scenario. For instance, if the method name is `istest` then it will be converted into "checks if it is test". For getters and setters, ad-hoc templates that are different from normal templates for general methods.

- **Outcomes:** The *outcome* of a scenario should represent the result of a particular situation. Each test case in a test class checks if correct results are obtained or not. This check is done with the `assert` statements in a test case. This means that assert statements actually look at the output of a situation, hence we describe assert statements in the *outcome*.

  The assertion statements like assertEquals, assertFalse, notEquals etc determine the type of test, while their parameters determine the expected and actual behaviour. Therefore, an assertion statement template will depend on its name and these are pre-processed and POS tagged to fill the template. The assert statements are filled as *outcome* for the corresponding class (scenario's context), which is used in the statement. But we found that some classes used in the assert statements are commonly covered classes which we removed during the filtration in Section 3.4.2. Since assert statements are necessary to know the outcome, we added the information about commonly covered classes in the summary. In Figure 3.5, text represented in the green color are the different outcomes. The scenarios with context "description" and "request" are commonly covered classes but are added to the summary because of the assert statements.

Once we have all the information about *contexts, events*, and *outcomes* for all the covered classes, we combine it to form a BDD scenarios for a single test class.

Chapter 4

Experimental Procedure and Set up

In this chapter, we perform the evaluation of the behaviour summary which is generated by tool which is implemented as described in Chapter 2. We initially discuss about the questions which are used evaluate the summary and how we planned conduct the empirical study. Then, the procedure followed during the experiment is discussed.

4.1 Study Design

The goal of this study is to investigate to what extent the generated behaviour summaries meet the requirements of an actual BDD scenario and how impactful will these summaries in real situations. The *quality* focus is about the understandability of test case behaviour when enriched with a generated behaviour summary. The *effectiveness* of the summary for a test class is evaluated when it is useful to explain the system behaviour to the stakeholder practically.

We perform the study with a questionnaire where a Likert scale is used to determine the answers. We used the Likert scale with 5 scale rating: *Strongly agree, Agree, Neither agree nor disagree, Disagree* and *Strongly disagree*. The study is conducted in the form of an interview instead of an online survey as we would like to know the reason why they rate a specific scale and stress for the question varies depending on the participant being interviewed, if the participant is a consultant or a manager or of a similar designation then more stress is given on client or stakeholder interaction and if the participant is student or developer then stress is also given on implementing a scenario along with the stakeholder point of view. We designed the following research questions that are answered during the empirical study:

Assessing Contextual Information: Contextual information about a scenario summarized from a test class is meant to help the stakeholders to understand the behaviour of that particular test in a project. Therefore, we study the following research questions about context:

RQ 1. *Does the automatically generated summary from a test class helpful to the stake-*

holders understand the behaviour of a part in a system and will it be useful? Our main objective is to know, to what level the stakeholders can understand a part of system behaviour from these automatically generated summaries and to know, when these summaries are used in place of documentation during the communication between the stakeholders and developer or testers, will it be useful during the communication.

Assessing Overall Quality: To determine the quality of the automatically generated summary and in what areas the quality of the summary can be mostly improved, we have the following research questions:

RQ 2. *How well does the automatically generated summary approach a BDD scenario in terms of preciseness, in having unnecessary information and in types of missing information?* The idea is to know if the automatically generated summary is too long or wordy, or contains overlapping content, to know if the automatically generated summary contains unnecessary information, and to know if the automatically generated summary is missing any type of information when compared to actual BDD scenarios context, which will help in improving the understandability of the summary.

As said previously, the above questions have a scale to measure opinion of the participant. Since, we are interviewing the participant some questions are open-ended, which does not have a scale but the participant can answer the question openly. Therefore, we prepared the below questions to know more about the automatically generated summary:

Q 1: *What improvements can be made on these summaries to attain better understandability about the system behaviour ?* We want to know if there is any possibility to increase the understandability about the behaviour by adding extra information about a system.

Q 2: *At what phase in the development cycle would these be useful, and do they wish to use these automatically generated summaries in future?* We believe that these summaries could help stakeholder better understand the behaviour of a system, but we want to know at which point in a development cycle will these be more useful and to know if the participant wishes to use this tool in real time and what benefits will he get using this tool.

4.2 Study Context

The *context* of this study contains of (i) *object* i.e., a Java test class extracted from Java open-source project and (ii) *participants* who will test the specified object. The object system is obtained from JUnit framework, which is mainly used as an example throughout our research. We selected a Junit test class `MaxStarterTest` as it is covering the highest number of classes in the entire JUnit project. The participants involved in this study would be software employees and students from Delft University of Technology.

The participants recruited for this study are our contacts from industry as well as the

Working Experience	Number of participants
0	2
1 - 2	1
3 - 5	2
6 - 10	5

Table 4.1: Participants working experience

students from the Department of Computer Science at Delft University of Technology, are consulted via email to know their interest to participate in the study. We sent an email to 10 participants asking their interest to participate in our survey. The information about their work or education was collected during the interview. Of them, 8 were either developers, consultants or managers from industry and 2 were students from the Computer Science Department. Out of 8 employees, 5 participants have 6 to 10 years experience, 2 participants have 3 to 5 years experience, 1 participant has less than two years of experience and remaining 2 participants are Master's Students. Each participant had atleast 2 years of experience in programming. Table 4.1 shows the participants with their work experience.

4.3 Experimental Procedure

The experiment was organised by conducting an face-to-face interview via Skype with the questionnaire. An example of the survey can be found in the appendix. The actual survey document each participant received has 4 parts: (i) introduction about survey and instructions to perform, (ii) a brief introduction about BDD along with an example and an external link for more information, (iii) questionnaire about participants work or education and (iv) a task containing our automatically generated summary along with questionnaire. As said above, we initially sent an invitation to all the participants to know their interest in participating in the interview and if the participants are interested then an email with the survey document with only first 2 pages in pdf format was sent as a response. Before the survey, we explained to participants what we expected them to do before and during the survey: they were asked to read the sent document before-hand and to have a brief knowledge about BDD if they are unfamiliar with it, and then during the survey they were asked to perform the survey by first reading our automatically generated summary to answer the questionnaire. We have sent the survey document atleast one day before taking the interview so as to give time to the participant to get a brief knowledge about BDD, as they need to evaluate our summary comparing it with BDD scenarios. Each participant received one task: to read the automatically generated summary from the test class `MaxStarterTest` and then answer the questions which followed it. It was explicitly mentioned that the entire summary was consists of scenarios driven from the test class. The survey had 10 questions about the automatically generated summary, are described in table 4.2. The questions 2,3 belong to RQ1, questions 4 to 6 belong to RQ2 and remaining are open-ended questions.

When the survey was started via Skype, it was simultaneously recorded. Before starting the task, it was confirmed if they have read the sent document so as that duration of interview is not extended for understanding BDD. Once they confirm it, each participant was asked a pre-study questionnaire about their work experience, designation and programming experience. After this questionnaire, they could start the task by reading the generated summary. Then, we started asking question by question. For each question, we first asked them to rate the summary depending on the question and then asked the participant to explain the reason behind their rating. We stressed the questions 4,5,6 and 7 from two different perspectives: these questions can be answered either imagining themselves as a stakeholder or as a tester/developer and hence these questions were asked twice with two different ratings. Remaining questions were asked as direct questions.

The total duration of the interview was between 30-45 minutes on average and the audio was recorded. But since we have provided the document, we asked them to gain a brief knowledge on BDD for their own understanding, and later when asked at the end of the interview on how much time they spent on it, they spent one hour on average. There were 3 participants who are working on BDD as employees, so they did not spend that extra hour on BDD as other participants. So the total time duration of the entire survey depends on the participant.

Question number	**Survey Question**
Q1	Do you have an experience with the project that the above mentioned class belong to?
Q2	Does the automatically generated summary of a single test class help the stakeholders understand the behaviour about a specific part of a system such as a BDD scenario?
Q3	Does the automatically generated summary be impactful during the communication between the stakeholders and testers or developers when compared to documentation?
Q4	Is any kind of information missing in the automatically generated summary when compared to a BDD scenario?
Q5	Is this automatically generated summary precise when compared to the BDD scenarios?
Q6	Is this automatically generated summary containing unnecessary information when compared to the BDD scenarios?
Q7	What improvements can be made on the automatically generated summary to attain better understandability about the system behaviour?
Q8	At what phase in the development cycle would the automatically generated summary be useful?
Q9	Looking at the automatically generated summary, explain how a system behaviour would be?
Q10	Will you use this tool to automatically generate summaries that are similar to BDD scenarios in the future and why?

Table 4.2: Survey Questions

Chapter 5

Results and Discussions

In this chapter, we report the results of our survey, and answer the research questions formulated in Chapter 4.

5.1 Results

5.1.1 RQ1: Usefulness of the summary to Stakeholders

This research question is answered using two sub questions Q2, Q3 described in table 4.2 . Figure 5.1 depicts the bar graph with the number of opinion counts given by the participants, divided into (i) understanding the behaviour of part of the system and (ii) impact on stakeholder communication. The first impression we get when we look at the results is that, for both questions the number of participants agreeing that the summary is explaining the behaviour and will be impactful when stakeholders use it, is high. There are few participants who do not completely agree or disagree and one who disagrees with the given statement.

For Q1, there is one participant who completely agrees that given summary provides behaviour of a part of a system while 6 accept the statement. Remaining 3 have a neutral opinion about the statement. While for Q2, the results are almost similar with 6 participants agree that having these summaries will be impactful, 3 have neutral opinion and 1 disagree with the statement. If we calculate the mean value of participants who agree for both the statements, it is 0.65 i.e, 65% in total, while the value for disagreement is 0.05 i.e, 5% in total. We then compared the working experience of the participant and the choice they made. The mean value of the participants working and agreed for Q1 is 0.625 and is same for Q2. But for Q2, there was one participant with significant work experience disagreed. The mean of the participants not working and agreed for Q1 is 1 and for Q2 is 0.5. This means that participants irrespective of their work experience agree that the summary will be useful.

Therefore, we conclude that

The automatically generated summary from a test class is helpful to the stakeholders in order to understand the behaviour of a part in a system and will be useful.

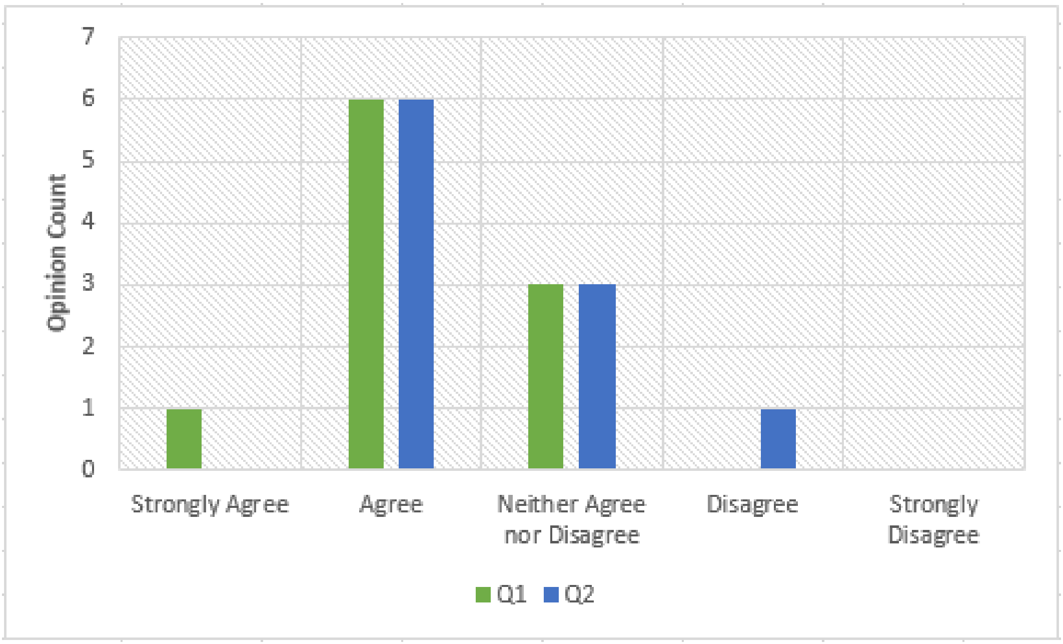


Figure 5.1: RQ1: Usefulness of the summary

5.1.2 RQ2: Closeness of the summary to a BDD scenario

To answer the question RQ2, we need the results of Q4, Q5, and Q6 from table 4.2. The results for those questions are depicted in the Figure 5.2 with a bar graph with the number of opinion counts given by the participants, determined from (i) missing information, (ii) preciseness and (iii) unnecessary information present in the automatically generated summary. A positive response for question Q5 and a negative response for questions Q4, Q6 describes that our generated summary is closer to BDD scenarios.

For question Q4, from the Figure 5.2, we can say that we cannot clearly estimate the result by looking at it. The number of participants who agree and neither agree nor disagree are same with 3 each, and remaining 4 disagreed. This says that, 60% believe that there is some information missing from the automatically generated summary. For Q5, by looking at the graph, we can say that more participant agree with the statement. 6 participants agree and 1 disagree while rest have neutral opinion. This means, 60% of the participants accept that our automatically generated summary is detailed. For Q6, we can find that 6 participants disagree, 2 participants agreed and remaining had neutral opinion. This shows, 60% of the participants say that our automatically generated summary does not contain any unnecessary information. The mean value to agree that RQ2 is 0.53, for disagree is 0.2 and for neutral opinion is 0.26.

If we consider the working experience and their response for this RQ2: for Q4, the mean value of working participants agreeing and disagreeing is 0.375 each; for Q5, the

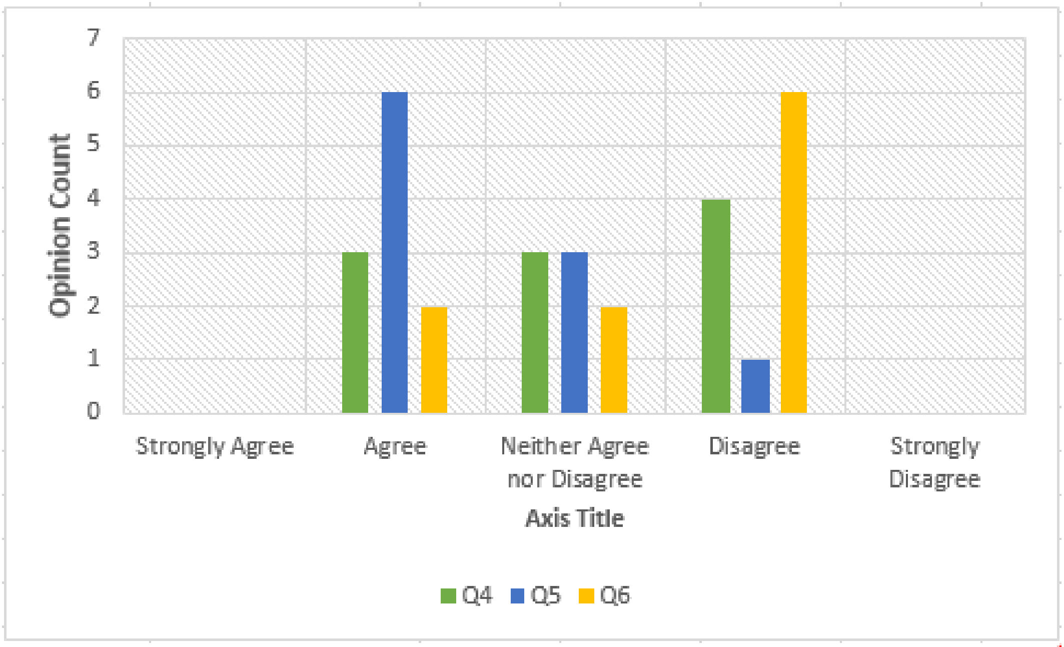


Figure 5.2: RQ2: Closeness of the summary to a BDD scenario

mean value for agreeing is 0.5 and disagreeing is 0.1 and finally for Q6, the mean value to agree is 0.25 and disagree is 0.5. This shows that, participants irrespective of their work experience accept that there is missing information in the summary, while partially agree about the preciseness of the content in the summary and partially agree that the summary does not contain unnecessary information.

Therefore, we conclude that

The automatically generated summary partially approach a BDD scenario in terms of preciseness and in having unnecessary information but misses some information.

5.2 Discussion

In the following, we provide qualitative insights to the quantitative results reported in Section 5.1. At the end of each question in the survey, all the 10 participants were asked the reason for the given Likert scale opinion. The reason for each participant's opinion is discussed here.

Usefulness of the summary: The participants who agree (70%) that the summary provides behaviour of a part of a system had similar reasons and it can be formulated as "the summary gives the sense of requirements needed for a system". While the response of the participants who had a neutral (30%) opinion can be said as "the summary should more elaborated with clear outcome along with an example; depends on the type of stakeholder reading the summary".

The reasons for the summary to be imapctful (60% agree) when compared to a document was because "the summary gives clear picture of what exactly is happening in the system which reduces communication gap; provides more information in a compact form about the system". The reasons for neutral (30%) opinion on impactfulness said "the summary has more technical details which are not required for the stakeholders; keywords in the summary should be clearly explained; sometimes stakeholders are not clear about their requirements then summary will not be helpful". "Detailed explanation about the system is not necessary for the stakeholder" was the reason to disagree by a participant.

Hence, we can say that the summary will be useful for the stakeholders in understanding the system behaviour.

Closeness of the summary: This research question was answered from 2 different perspectives. As said in the Section 4.1, we asked the questions from stakeholder and developer point of view. Below we discuss these two point of views for each statement.

The total percentage of participant who agree that the summary was missing some information is 60%. In that 30% of participants reasoned as: from stakeholders view they think that "clear explanation of the outcome; and benefit of the action" were missing and the summary has enough information for a developer to implement the scenario. The remaining 30% from above 60%, had neutral opinion but it can be considered that they think that summary has missing information. Out of 30%, 20% disagree from the stakeholder point of view, that there is some missing information, but for developers they agree as "input and output specification; range of input" were missing. However, other 10% disagree that developers need more information to implement, while agree that stakeholders need "clear information about the output of a system". While the 40% who say that summary is not missing important information for both stakeholders and developers as it contains "input and output values; all outcomes are present where necessary; and explanation is well enough for both of them".

The participants who agree(60%) that the summary is precise from both stakeholder's and developer's perspective as the summary has: "brief overview of what is happening in a system is present in an understandable way; and presented in a simple and compact way with necessary information". 30% had a neutral opinin that the summary is clear, and in that 20% think that from stakeholder perspective, the summary is clear but from the developers perspective they said that "when compared to an Unified Modeling Language (UML) diagram with input and output, clear understanding of the summary is not achieved". While other 10% claimed that, stakeholders cannot understand the summary clearly as the summary contains lot of technical information which will be useful for developers.

There were 60% of the participants who agree that the summary does not contain unnecessary information from both the perspectives as the summary "is containing useful information to understand the system; has necessary conditions along with output". 20% who agree from stakeholder point of view say that "this summary is complex with technical details" and from developer point of view the summary is "big with lot of data, making it difficult for the developer to find which data is necessary and not, to implement". Remaining 20% had neutral opinion saying the stakeholder needs "only functionality and not how the system functions; less technical details" but agree that technical details are necessary for a developer.

From above discussed points, we can say that most of the participants feel that summary is more favourable in terms of accuracy and information present in it than for the stakeholders. But since a BDD scenario should be useful to both stakeholders and developers, hence we conclude that our automatically generated summary is closer to a BDD scenario but still need improvements to make it better.

5.2.1 Open Questions

As said in the Chapter 4, we have prepared few open-ended questions which are answered after the research questions.

Question Q1 from Section 4.1 is questioned as Q7 in the survey and was about improvements that can be done to the summary, and all the participants have suggested the following:

1. *Sequence of the scenarios in the summary*: Few participants were concerned about the order in which each scenario has to be listed in a summary and, as a solution, they suggested to have each scenario according to their order of execution.

2. *An example of the summary*: Most of the participants said they can understand the summary but they felt that adding an example about the summary at the beginning along with input and output parameters for all the scenarios would help them understand better.

3. *The number of technical terms*: All the participants said that the summary had some technical words and suggested that, for a stakeholder the lesser the technical details the better the understandability and for a developer presence of technical details is acceptable.

4. *Elaborate and descriptive summary*: All the participants expected more elaborate explanation for each scenario in natural language.

5. *Trim the scenario*: Participants with work experience on BDD recommended to trim down large scenarios into smaller scenarios with single *When*, *Then* statements along with the *context*.

6. *Name the scenario*: Each scenario with a heading will help the reader understand what is happening in the scenario at a glance.

7 out of 10 participants suggested point 2 i.e., to use examples for the summary, as this provides clear understanding of the summary. And all participants wanted more descriptive scenarios.

Question Q2 from Section 4.1 is divided into questions Q8, Q10. Question Q8 was about at which phase in the software development cycle will our automatically generated summary be useful, 5 out of 10 participants said it can be used in "Testing Phase" as after finishing the testing and this summary is generated then, cross check with the requirements can be done to verify the fulfilment of the requirements and to know if the system is behaving correctly. While 8 out of 10 said it can be also used in the "Requirements Phase" as it gives the stakeholder better insight about what is happening in the system. Therefore, we can say that the automatically generated summaries will be useful in Requirements phase and Testing phase.

Question Q10 was to know if the participants are willing to use our summaries in the future. 8 out of 10 participants said that they will use it in future if the suggested improvements are added to it. While 2 participants said that they will not use it as the summary needs to have more information about the benefits, input/output in a system along with very clear explanation.

Question Q9 was about explaining the given summary in their own words so as to know to which level the participants were able to understand the system. All the participants were able to give average type of explanation about the summary by saying what type of events were happening in the systems along with their outcomes. But everyone felt that they could have given better explanation when they know about the system which the summarized test class belong to. This is valid point because if we show the summary to a stakeholder, he will definitely have an idea about the system and thus can understand the summary better than a dummy.

Question Q1 was added because if the participant knows about the summarized test class, it will have an effect on the remaining questions as the participant can better understand the summary compared to others and it will effect the results. However, none of the participant had knowledge about the used test class.

5.3 Threats to Validity

This section describes the possible threats to validity of our study and how we solved them.

Construct Validity: Threats to construct validity mainly concern on how we set up the study. Due to the fact that all the participants involved in our study need to have a prior knowledge about BDD as our solution should be compared with a BDD scenario, we could not say if they know BDD. Lack of knowledge about BDD will produce incorrect results. To handle this, we initially sent a pdf document containing a brief overview on BDD that would be sufficient to do our survey. Then before starting the survey, we ask the participant if they read about BDD to continue the survey.

Internal Validity: To find the coverage information about the covered classes by a test class, we used Cobertura. However, Cobertura fails to determine the covered classes by the inner classes. This does not effect our output, since we filter the inner classes as these are unwanted classes for us as an inner class is more specific to the class which it is present in and will have no use elsewhere. Our proposed tool will only work on Java projects with JUnit test cases.

External Validity: Threats to external validity concern the generalization of our results. It is important to point that the object i.e., test class which is summarized could influence the results of our survey. The evaluation here is limited to the summary of a single test class only. Another threat is the size of the participants used for this study, as larger set of participants would increase the confidence about the survey results.

Chapter 6

Related Work

In this Chapter, we discuss the related literature on distinguishing integration tests from unit tests and source code summarization.

6.1 Distinguishing integration tests from unit tests

Weijers dissertation [14] is the earlier work which proposes a tool, *JUnitCategorizer* to distinguish integration tests from unit tests. They distinguish them by following three steps: (i) determining all objects called from a test method, (ii) determining the class under test and (iii) identifying mock objects. Even though we had same steps, the techniques used by them in those steps are some what different from ours. To determine the called objects, they used instrumentation on-the-fly and had better filtering lists as blacklist, suppressor list and whitelist. To determine the class under test, they used a heuristic to score potential classes under test and the class with highest score is most likely the class under test. Other study [11] on detecting class under test compared test code and code under test in four different approaches. The first approach is to look for the classes with the same name as the test class but without `Test`. The next approach finds which classes are called in the statement before the last assertion statement. The third compares the textual similarity between classes under test and test class and the last approach assumes that the test code is co-evolved along with class under test.

6.2 Code Summarization

Sridhara *et al.* [39] proposed a way to generate natural language text phrases using pre-defined templates that are to be filled with linguistic elements like verbs, nouns etc obtained from methods. Using same strategy, there are other studies which proposed techniques to summarize java methods [57, 34, 30], java classes [42, 35], or parameters [37]. There are few studies which aimed to provide high-level summaries by mining textual information from emails [50], bug reports [50] and question and answer sites [48, 56]. The studies to summarize test code aimed to improve the understandability in case of test failures [52], and unexpected exceptions [43]. However, to summarize those tests, a test needs to fail or throw exceptions. A recent study [8] aimed to provide summary of a test code at different levels (explaining main responsibilities of a test class, description of each statement in the test case and description about each branch statement) helping developers fix more bugs.

Our approach is aimed to automatically generate summary which explains the behaviour of a test class of a system rather than explaining test code.

Chapter 7

Conclusions and Future work

Literature suggests few techniques [56, 50, 48] which automatically summarize the source code providing high-level summaries. Instead, these summaries speak about a single class, contain technical details and are typically useful to developers. However, through our project, we proposed an approach to automatically generate a behaviour summary of a test class of a system which are useful to the stakeholders.

We generated a behaviour summary of the system by determining the test classes which represent the behaviour of the system. We derive this information from the integration tests in the system as they show the major parts of a system that work together, providing paths between different parts of the module. All the integration tests are collected by distinguishing them from unit tests in a system. We stated that a test is a unit test when it uses *mock objects* to replace the real objects like network, file system adapters, and real database and a test is an integration test when the mock object is replaced by the real implementation of that object. To know if a test is unit or integration test, our proposed tool determines the class under test, all the objects called from a test and the mock objects. In our implementation, we used a heuristic method to determine the class under test, which consists of five cases. The fulfilment of these five cases by a test determines it as a unit test or not. Our tool determined the objects called from a test by parsing through the test code and find method calls through regular expressions. The list of called classes is filtered using a blacklist and whitelist. Any test class which uses other classes along with single class under test is marked as integration test.

To determine the information that needed to be extracted from an integration test, we used Cobertura to find number of covered classes for a test class. We pragmatically experimented to determine number of covered classes to be summarized. We use the names of the covered classes as *context*, method names of the called methods in the covered classes as *events* and assert statements in the integration test class as *outcome* and are filled in the BDD template.

We conducted the survey by interviewing the participants and using a Likert scale to answer the survey questions. The survey was conducted to determine: if the automatically generated summaries help the stakeholders understand the system behaviour; and if these summaries approach a BDD scenario in terms of preciseness, having unnecessary and missing information.

From the results, 65% in total have agreed that the summary explains the behaviour of the system. So, it can be that the automatically generated summary from a test class is helpful to the stakeholders in order to understand the behaviour of a part of a system and will be useful.

From the results, 60% agree that the summary is precise and does not contain unnecessary information. While only 40% agree that the summary is not missing any information. Thus, we can say that the automatically generated summary partially approaches a BDD scenario.

In conclusion, it can be said that the automatically generated summary is precise without unnecessary information and is helpful to the stakeholders in order to understand the behaviour of a part of a system but only partially approach a BDD scenario.

7.1 Future Work

Future work can be directed towards different directions and further improve summary by:

1. Enhancing our approach with the technique in JUnitCategorizer [14] to distinguish between unit tests and integration tests using on-the-fly instrumentation of Java class files to determine which classes are called in a test method and determine the class under test by a heuristic that scores potential classes under test. This results in a higher accuracy rate in determining integration tests than ours.

2. Considering the improvements suggested in the feedback.

   For stakeholders:

   - *Listing the scenarios in a sequence* is useful as this provides an insight in which order the scenarios are executed to attain certain behaviour. The sequence of the scenarios can be achieved by finding the order of classes being called during the execution.
   - *The summary can have an example* in the front will give better understanding, but achieving this improvement is not possible with the test code only.
   - *Naming every scenario* with overview of what is happening in the scenario will always be helpful, and it can be obtained by re-summarizing the scenario using text summarization tools by finding key words in it.

   For developers or testers:

- *Providing an elaborate and descriptive summary* will give a more clear picture of what is happening in the system and help the developers and testers to easily implement it, but this make summary longer.
- *Trimming the larger scenarios* with multiple events and outcomes into smaller scenarios with single event and outcome will give even more clear information about a scenario.

3. Replicating the survey with more participants and different examples can provide us better insights into the summary.

Out of all the suggested future work, the enhancement of our approach with JUnit-Categorizer and improvements proposed for stakeholders are more appropriate suggestions that can be achieved in the future, as our main aim is to generate a summary that provides behaviour of the system useful to the stakeholders.

Bibliography

[1] Behaviour-Driven Development. `http://behaviourdriven.org/`

[2] R. Binder. Testing object-oriented systems: models, patterns, and tools, Addison-Wesley, 1999.

[3] M.C. Feathers. Working effectively with legacy code, Prentice Hall PTR, 2005.

[4] Solis, Carlos; Wang, Xiaofeng. A Study of the Characteristics of Behaviour Driven Development. In *Software Engineering and Advanced Applications (SEAA)*, 37th EUROMICRO Conference on:pp. 383-387, 2011.

[5] D. Janzen and D. H. Saiedian. Does Test-Driven Development Really Improve Software Design Quality, IEEE Software. vol. 25, no. 2, 2008.

[6] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. Extreme programming examined, pages 287?301, 2001.

[7] H. Aslak . Aslak's view of BDD.
`https://cucumber.io/blog/2015/03/27/aslaks-view-of-bdd`

[8] 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. PeerJ PrePrints, 2015.

[9] L. Koskela. Test Driven: TDD and Acceptance TDD for Java Developers, Manning Publications, 2007.

[10] D. North. Introducing BDD, 2006. `http://dannorth.net/introducing-bdd`

[11] B. Van Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In Software Maintenance and Reengineering, 2009. 13th European Conference on, pages 209?218. IEEE, 2009.

[12] Haring, Ronald. Behaviour Driven development: Better than Test Driven Development, Java Magazine, 2011.

[13] D. Janzen, D. H. Saiedian. Test Driven Development: concepts, taxonomy, and future directions, Computer, vol.38, no. 9, pp. 43-50, Sept, 2005.

[14] Joep Weijers. Extending Project Lombok to improve JUnit tests, Master?s Thesis, 2012

[15] E. Evans. Domain -Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.

[16] D. North. BDD with intent. http://dannorth.net/2006/02/09/bdd-with-intent/

[17] D. North. There's more to BDD than evolving TDD.
http://dannorth.net/2006/06/04/theres-more-to-bdd-than-evolving-tdd/

[18] D. Astels. A new look at test driven development.
http://techblog.daveastels.com/files/BDD_Intro.pdf/

[19] I. Lazăr, I., S. Motogna, and B. Pârv. Behaviour-Driven Development of Foundational UML Components. Electronic Notes in Theoretical Computer Science 264, no. 1 (August): 91-105, 2010.

[20] R. M. Ferreira. Why BDD Can Save Agile.
http://www.infoq.com/news/2015/03/bdd-save-agile

[21] Cucumber JVM. https://cucumber.io/docs/reference/jvm#running-cucumber

[22] Gherkin. https://github.com/cucumber/cucumber/wiki/Gherkin

[23] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, and D. North. The RSpec book: Behaviour Driven Development with RSpec, cucumber and friends, Pragmatic Bookshelf, 2010.

[24] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans*. Software Eng., 2014.

[25] CodePro. https://developers.google.com/java-dev-tools/codepro/doc/

[26] JTest. http://www.parasoft.com/jsp/products/jtest.jsp

[27] EvoSuite. Automatic Test Suite Generation for Java
http://www.evosuite.org/

[28] M. Kamimura and G. Murphy. Towards generating human-oriented summaries of unit test cases. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 215-218. IEEE, May 2013.

[29] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans*. Software Eng., 39(2):276-291, 2013

[30] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 43-52. ACM, 2010.

[31] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 79-88. ACM, 2008.

[32] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 232-242. IEEE, 2009.

[33] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 499-510.

[34] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 279-290. ACM, 2014.

[35] L. Moreno, A. Marcus, L. Pollock, and K. VijayShanker. Jsummarizer: An automatic generator of natural language summaries for java classes. In *Program Comprehension (ICPC)*, 2013 IEEE 21st International Conference on, pages 230-232, May 2013.

[36] M. A. Ogush, D. Coleman, D. Beringer. A Template for Documenting Software and Firmware Architectures. *Hewlett-Packard Product Generation Solutions*

[37] C. Pacheco and M. D. Ernst. Randoop: Feedbackdirected random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 815-816. ACM, 2007.

[38] Cobertura. `http://cobertura.github.io/cobertura/`

[39] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*,pages 101-110. IEEE, 2011

[40] E. Hill. Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration. PhD thesis, Newark, DE, USA, 2010. AAI3423409.

[41] A. Gatt and E. Reiter. Simplenlg: a realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, ENLG '09, pages 90-93, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[42] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 23-32. IEEE, May 2013.

[43] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 273-282. ACM, 2008.

[44] N. Dragan, M. Collard, and J. Maletic. Automatic Identification of Class Stereotypes. In *26th IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1-10.

[45] N. Dragan, M. L. Collard, and J. I. Maletic. Reverse Engineering Method Stereotypes. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, 2006, pp. 24-34.

[46] N. Dragan, M. L. Collard, and J. I. Maletic. Using method stereotype distribution as a signature descriptor for software systems. In *25th IEEE International Conference on Software Maintenance*, 2009, pp. 567-570.

[47] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 71-80. IEEE, 2011.

[48] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 106-109. ACM, 2014.

[49] T. Ball and J. R. Larus. Branch Prediction for Free. In *Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[50] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Proceedings of the International Conference on Program Comprehension, ICPC,* pages 63-72. IEEE, 2012.

[51] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *16th Working Conference on Reverse Engineering, WCRE* 2009, 13-16 October 2009, Lille, France. IEEE Computer Society, 2009, pp. 205-214.

[52] S. Zhang, C. Zhang, and M. Ernst. Automated documentation inference to explain failed tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 63-72. IEEE, 2011.

[53] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. ACM, 2010, pp. 375-384.

[54] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting source code from e-mails. In *18th IEEE International Conference on Program Comprehension, ICPC* 2010, Braga, Minho, Portugal, June 30-July 2, 2010. IEEE Computer Society, 2010, pp. 24-33.

[55] English Standford Parser.`http://www-nlp.stanford.edu`

[56] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 562-567. IEEE, 2013.

[57] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the Inter- national Working Conference on Reverse Engineering (WCRE)*, pages 35-44. IEEE, 2010.

[58] T. Ball and J. R. Larus. Branch Prediction for Free. *Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[59] E. Reiter and R. Dale. Building Natural Language Generation Systems. Cambridge Univ. Press, 2000.

[60] Stanford CoreNLP. `http://nlp.stanford.edu/software/corenlp.shtml`

[61] StackOverflow. `http://stackoverflow.com/`.

Appendix A

Survey Document

The survey document used for our study can be found on the next page.

A Survey on Behaviour Driven Summary

1. Introduction
Our research group at the Delft University of Technology, is performing a study aimed at observing (and/or measuring) how the summary generated from the test code approach a Behaviour Driven Development's scenario. We want to investigate to what extent the generated behaviour summaries by our meet the requirements of an actual BDD scenario and how impactful will these summaries in real situations.

We would be grateful if you could perform the survey by answering few questions face-to-face about the summary. You will initially be given a brief description about Behaviour Driven Development (BDD) and its scenario with an example. Later you will be given our automatically generated summary of a Junit test class (MaxStarterTest.java) belonging to Junit framework (https://sourceforge.net/projects/junit/) respectively.

There are 2 parts in the questionnaire: 1) You should fill your personal details and 2) you should rate the summary depending on the questions and answer why you have given a specific rating.

The entire task will take about 20 to 30 minutes.

Thank you very much for your effort,
Tejaswini Dandi

A Survey on Behaviour Driven Summary

2. Description on Behaviour Driven Development
Behaviour Driven Development is about expressing a requirement in the form of expected behaviour. BDD is a process of exploring, discovering, defining, and then finding out the desired behaviour of a software system. This process is done with the help of conversations, concrete examples to understand the problem that has to be solved for the stakeholders. Then, the examples are refined into automated tests, to describe the desired behaviour of the solution. The conversations in BDD process will involve the stakeholders, developers and testers. During the meeting, the stakeholders come up with a problem, which becomes a “user story”, and developers and testers ask for concrete examples about the problem to find out the constraints and requirements to form “scenarios” which will result in a clear understanding of how the system should behave. Each user story can contain more than one scenario.

More information about BDD can be found here:
https://en.wikipedia.org/wiki/Behavior-driven_development

Look at a simple example to know how BDD is performed using user stories and scenarios

Let the ‘User story’ be:
As a Student
I request a process to square the number
To gain a faster calculation

And the ‘Scenario’ be:
Given a variable x with some value
When I multiply x by x
Then x square should be equal to square of x

Once we have the scenario with the steps, we need to define step definitions to test the scenario steps. If the step definitions are written in Java, and will look as below:

```
public class NumberSquaringSteps {
int x;

@Given(“a variable x with value $value")
public void givenXValue(int value) {
 x=value;
}

@When(“I multiply x by $value”)
public void whenImultiplyXBy(int value){
 x = x * value;
}
```

```
 @Then(“x should equal $value”)
 public void thenXshouldBe(int value) {
 if (value != x)
 throw new RuntimeException(“x is” + x + “, but
should be ” + value);
}
}
```

A Survey on Behaviour Driven Summary

3. Personal Information
All of the information that you provide will be treated as confidential and will only be used for research purposes. Some personal information may be collected about you if you choose to participate in the survey. In particular, your responses to survey questions and your personal details will be collected for the simply reason of linking your responses with the versions of the Java projects. We will not disclose your personal information to third parties.

1.Your Name: *

2.You are a: *

○ Master's student ○ Ph.D. student ○ Software Employee

3.If employee, please specify your designation:

4.Your working experience since: *

○ <1 year
○ 1-2 years
○ 3-5years
○ 6-10 years
○ >10 years

A Survey on Behaviour Driven Summary

4. Task
We have automatically generated a summary from the test class MaxStarterTest.java.
The summary contains a combination of different scenarios for the test class and is as follows:

Given maximum core
When build runner, find leaves, stored locally, get malformed test class, sort request, construct leaf request and sorted leaves for test
Then maximum core method name is equal to fast and max core method name is equal to slow

Given junit38 class runner
When filter, make description, create adapting listener, set test, get annotations, get description and get test
Then child test count is equal to 1

Given a test suite
When test the Count, add a test suite, add a test method, check if test method, create a test, get test constructor, warning, add a test, add tests from test case, check if test method is public and run test

Given error reporting runner
When get causes, describe cause, run cause, get class names and get description

Given junit core
When remove listener, run main, default computer, run classes and add listener

Given suite
When get children, run child, describe child and get annotated classes

Given description
When create test description, format display name, get display name, add child, get children, check if test, test count, check if empty, get annotation, get test class, get method name, and method and class name pattern group
Then list of description has succeed, do not succeed and list of description size is equal to 2, description is equal to slow, description is equal to fast

Given request
When class without suite method, classes, runner, filter with description
Then count match of test class is equal to request count

Given result
When get run count, get failure count, get failures
Then run count is equal to 2

Following are the questions you need to answer about this summary:

1.Do have experience with the project from the above mentioned class belong to? *

○ Yes	○ No

2.Does the automatically generated summary of a single test class help the stakeholders understand the behaviour about a specific part of a system such as a BDD scenario? *

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

3.Does the automatically generated summary be useful during the communication between the stakeholders and testers or developers when compared to documentation? *

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

4.Is any kind of information missing in the automatically generated summary when compared to a BDD scenario? *

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

5.Is this automatically generated summary precise when compared to the BDD scenarios? *

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

6.Is this automatically generated summary containing unnecessary information when compared to the BDD scenarios? *

○ Neither

○ Strongly disagree	○ Disagree	agree nor disagree	○ Agree	○ Strongly agree

7.What improvements can be made on the automatically generated summary to attain better understandability about the system behaviour ? *

8.At what phase in the development cycle would the automatically generated summary be useful? *

9.Looking at the summary, explain how a system behaviour would be? *

10.Will you use this tool to automatically generate summaries that are similar to BDD scenarios in the future and why? *