The Effect of Generated Comments for Integration Tests on Code Comprehension

Master's Thesis

Eveline van der Schrier

The Effect of Generated Comments for Integration Tests on Code Comprehension

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Eveline van der Schrier
born in Leiden, the Netherlands

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

The Effect of Generated Comments for Integration Tests on Code Comprehension

Author: Eveline van der Schrier
Student id: 4154371
Email: E.D.vanderSchrier@student.tudelft.nl

Abstract

Comments play a vital role in the comprehension of source code. To aid software developers in code comprehension, tools have been designed for automatically generating comments. However, developers lack a tool that creates comments for integration tests. To further improve the process of code comprehension, a novel tool is proposed in this thesis based on *TestDescriber* for automatically generating comments for integration tests. For the design, the main stages in the process of generating comments are defined. An experiment is conducted to evaluate the tool with Java developers. Two programming tasks had to be performed, in which a test had to be rewritten. The results show that the generated comments from the tool can improve code comprehension.

Thesis Committee:

Chair:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bacchelli, Faculty EEMCS, TU Delft
Committee Member:	Dr. T. Abeel, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Problem Statement

Reading plays an important role in understanding source code [12]. Since comments contain the main intent behind design decisions and implementation details [5], they can help increase program comprehension [26, 32, 36] and make it possible for developers to improve the readability of code [32, 36]. This is not only convenient for developers, since they spend almost half of their time understanding code [8], but it also reduces maintenance costs [7, 36].

Understanding source code also includes understanding tests. Up to 50 percent of code in a project can be test code [39] and developers spend about 25 percent of their time on writing tests [2]. While unit tests can be easy to understand, since they test only one methods, this could be different for integration tests. Integration tests use multiple parts of a software system i.e. multiple methods, which could make it more difficult to understand. Although tools that aid in code comprehension have been created that generate comments for source code and unit tests, there has never been one for integration tests. That is why in this thesis, a novel tool is proposed that generates comments for integration tests. By providing comments, it might be possible that developers find understanding integration tests less difficult and thus would the comments aid in code comprehension.

In [27] Panichella et al. proposed a method called *TestDescriber*. It creates summaries for automatically generated unit tests. This was done because in general, generated test code is difficult to understand and maintain. It was found that *TestDescriber* helps improving understandability of test code and developers found twice as many bugs as usual during evaluation. That is why the new tool that will be proposed here is based on *TestDescriber*.

1.2 Goals

The main goal of this thesis is to find out what the effect of generated comments for integrations test is on code comprehension. For this a tool has to be developed that generates a description of an integration test. This will be done based on the *TestDescriber* tool and the comment generation process of other existing methods. Then an experiment has to be con-

ducted where developers work with integration tests that contain the generated comments. Here questions can be posed to find out what developers think about the comments and whether or not they aid in understanding the code.

1.3 Research Questions

Based on the goals given in the previous section, research questions can be posed. The first question concerns the design and implementation of a tool generating comments for integration tests. The second question concerns the main goal of this thesis.

RQ1: *What are the main stages in the process of comment generation?*
Before a comment generation tool can be designed and implemented, it needs to be known which steps are taken in summarisation techniques. This can be done by reviewing existing techniques and identifying the main stages in the process of code summarisation.

RQ2: *What is the effect of generated comments for integration tests on code comprehension?*
This question can be answered by conducting an experiment with the developed tool. Here developers work with source code that contains comments generated by the tool. Then questions can be asked concerning the comments and comprehension of source code.

1.4 Thesis Structure

The thesis is structured as follows. In Chapter 2 existing methods for automatically generating comments in source code are described to find out how a summarisation method should be designed. Then in Chapter 3 a description of TestDescriber is given, the tool that creates comments for automatically generated unit tests. Also the adaptation of this tool is shown, which was done to make it possible to generate comments for integration tests. In Chapter 4 the experiment and its results are shown. The experiment was conducted to find out what the effect of the generated comment of integration tests would be on code comprehension. Then in Chapter 5 the results from the experiment are discussed and possible threats to validity are given. In Chapter 6 related work is shown. Finally in Chapter 7 conclusions are made and possible future work is proposed.

Chapter 2

Generating Comments from Source Code

The goal of this chapter is to find out whether main stages can be found in the process of generating comments from source code. This will be done by discussing several existing methods. The methods are arranged according to what is summarised (class, method, various source code entities) and they are put in a chronological order. For each method all techniques involved in the summarisation process are given. Then an overview of the techniques will be given and a comparison is made to extract the main stages in the process of code summarisation. Finally, some additional methods are shown to verify that more methods exist than there will be given in paragraph 2.1 to 2.3.

2.1 Method(s) for Summarising Various Granularity Levels

2.1.1 Extractive Summaries from Source Code Entities

Haiduc et al. [14, 15] propose a method for automatic generation of extractive summaries for source code entities. For summarisation, extractive summaries use (the most important) information that is already present in a code entity. In Figure 2.1 it can be seen that the approach is based on lexical and structural information of the source code.

Lexical information is extracted from the entity, where common English and programming language keywords are removed, identifiers are split in meaningful separate words, and stemming is used. Then the source code is transformed into a text corpus, where each document corresponds to an entity. Text retrieval techniques are used to find out what the n most important terms are of an entity. Together with added structural information, i.e. the role of each term in the source code (class name, method name, etc.), the summary is formed.

In [14] the Latent Semantic Indexing (LSI) text retrieval technique was used in a case study, where in [15] two variants were used, namely LSI and the Vector Space Model (VSM). Besides text retrieval techniques, lead summarisation and random summarisation were tested. Based on the results discussed in the next paragraph, another case study was done combining lead summaries with VSM summaries. Lead summarisation considers the

position of words in a document, where the first appearing terms are the most relevant. In random summarisation *n* terms are randomly selected and then included in the summary. For both LSI and VSM the first step in summarisation is to represent terms and documents in a matrix, where a row represents a term and a column represents a document. Inside the matrix the weight of a term in relation to a document is given. This weight consist of a combination between a local (document) and global (corpus) weight. It was found that the combinations *log*, *tf-idf* and *binary-entropy* [31] work best for both LSI and VSM, so these combinations were used in the case studies. Then, for VSM terms in a document are ordered according to the given weight and the top *n* terms are included in the summary. For LSI however, not the original matrix is used but it is projected to a smaller approximated one, corresponding to the highest eigenvalues in the matrix. This means that terms and documents could be represented in the same coordinates and similarities between terms and documents can be computed. Cosine similarities are computed between vectors of terms in the corpus and vectors of the document to be summarised. Then terms are ordered based on this similarity and the top *n* terms are selected. Because LSI computes a different matrix, it is possible that a summary can contain terms that do not appear in the particular summarised code element, but appear somewhere else in the text corpus. This is not possible using VSM.

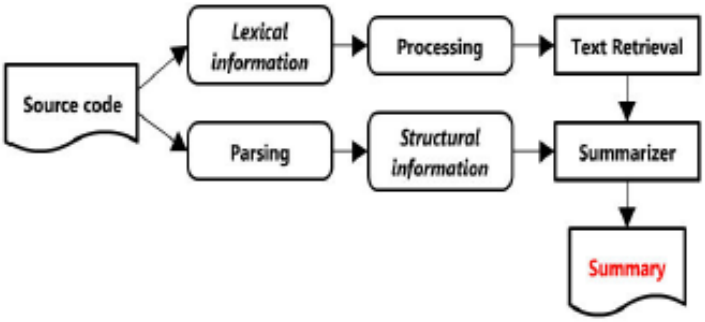


Figure 2.1: Process of automatic source code summarisation [14].

2.1.2 Topic Modelling Approach to Source Code Summarisation

Based on the summarisation method in [14] Eddy et al. [6] proposed a new topic modelling based approach to source code summarisation. It expands the former method by considering more subjects and new summaries.

The techniques used in this new method are almost the same as the original approach, except for the text retrieval technique. In addition to lead, random and the combination of lead and VSM, the hierarchical Panchinko Allocation Model (hPAM) and a combination of lead and hPAM summarisation were used in a case study. hPAM is a generative hierarchical topic model. Terms and topics are represented as a four-level directed acyclic graph, where the nodes represent topics and the leaves represent terms. Each node is associated with a distribution over the vocabulary. With this graph hPAM produces the term-subtopic proba-

bility distribution and the subtopic-document probability distribution. In the case study this is used to identify the most likely subtopic for a document and selecting the top n terms of that subtopic for including in the summary. Just as with using LSI for generating summaries, with hPAM it is also possible to get terms that were not present in the document itself.

2.1.3 Code Summarisation using an Eye-Tracking Study

Rodeghero et al. [30] also take the summarisation method of Haiduc et al. [15] into account and create a new approach. Instead of using the original VSM *tf/idf* as text retrieval technique for extracting keywords from source code, a new method has been designed in which the VSM is improved by incorporating the results of the performed eye-tracking study.

In the eye-tracking study eye movements and gaze fixations of programmers have been analysed while reviewing and summarising code. In this way it could be discovered which sentences and identifiers developers focus on and could be used in a summary of that code. These results were then compared with the results obtained using VSM. It turned out that the VSM approximates about five out of ten most read keywords by programmers during summarisation. In addition method signatures were prioritised above invocation keywords by programmers, and invocation keywords above control flow keywords.

The newly designed approach implemented VSM but was improved by incorporating the results of the eye-tracking study. This meant that terms in method signatures, invocations and control flow were treated differently by giving them different weights. In Table 2.1 the different configurations are given.

Code Area	VSM	Eye_A	Eye_B	Eye_C
Method Signature	1.0	1.8	2.6	4.2
Method Invocation	1.0	1.1	1.2	1.4
Control Flow	1.0	0.9	0.8	0.6
All Other Areas	1.0	1.0	1.0	1.0

Table 2.1: Weights given to terms in specific code areas [30].

2.1.4 High-Level Functionality in Source Code Summaries

McBurney et al. [23] designed a new method for topic model source code summarisation. In this approach the topics are set into a hierarchy in such a way that the highest-level functionality is near the top of the hierarchy list.

The first step is to convert a software project into a call graph. This is done by creating a directed graph with nodes representing the code methods and edges representing the method calls. In addition a phantom node is added as root and is connected to every other node. This is done to make sure that the graph is fully connected. Using this graph the Hierarchical Document Topic Model (HDTM) is used to create a topic model. For this it is necessary to first define each node in the graph as a document in the topic model, which consist of a list of keywords. HDTM creates a hierarchical tree structure, in which each node contains an

ordered list of topics selected from the code method and descendants of that method. This means that keywords can appear in the summary which are not present in that particular code method.

2.2 Method(s) for Summarising Code Classes

2.2.1 Summarising Java Classes

In [24] Moreno et al. propose an approach for generating summaries of Java classes. These summaries are made of the content and main responsibilities of a class. The resulting summary consists of four parts: a general description, a stereotype description, a behaviour description and (if applicable) an inner class enumeration.

Before deciding on which content should be used in a summary, class elements are selected that help describe the intent of a class. These elements are divided into two sets, one set containing elements that are included in the summary directly (names of interfaces, super classes and inner classes), and one containing elements that need further analysis (class attributes and methods). The analysis is done based on the stereotype of a class and its methods. For this JStereoCode [25] was designed, using predefined stereotypes such as *accessors* and *mutators* for methods, and *Entity* and *Controller* for classes. After stereotypes have been identified, methods are selected for summarisation using the designed stereotype-based filter and access-level filter. The stereotype-based filter removes methods whose stereotypes are irrelevant to the class stereotype. The access-level filter removes private, package-protected and protected methods, so that only public methods remain. Finally the text of the summary needs to be generated. This was done by creating templates for each part of the summary, based on the techniques designed by Sridhara et al. [33] described in section 2.3.2. For the general description, the names of super classes and interfaces are used in one of the 22 predefined templates. The stereotype description is made by using the class stereotype and if possible some domain information, such as classes used within the class or certain type of methods. For this description 40 different templates were designed. For the behaviour description, relevant methods are used that were selected after applying the stereotype-based and access-level filter. The template consists of three parts, describing the accessor, mutator and remaining methods. Phrase generation tools from [33] were used to obtain natural-language fragments. Finally the optional inner class enumeration description enumerates the declared inner classes by using their names in a predefined template. The final summary is made by concatenating all four descriptions.

2.3 Method(s) for Summarising Code Methods

2.3.1 Summarising Java Methods

Sridhara et al. [33] proposed an approach for creating descriptive comments for Java methods. This is achieved by extracting the important code statements of a method and converting them into natural language, as shown in Figure 2.2.

Before it can be determined which statements need to be included in a summary, identifiers need to be split into component words. Here camel case splitting is used, which splits words based on capital letters, underscores and numbers. Also abbreviations are detected and expanded. Then the selection process takes place, which consists of three phases. First, statements are selected that call a void method or a method with the same action as the name of the method they are in, and statements that lie at the control exit of a method. In order to identify these statements, the method's Abstract Syntax Tree (AST), the Software Word Usage Model (SWUM) [16] and the Control Flow Graph (CFG) are used respectively. In the CFG, the predecessors of the CFG's exit node are the statements that lie at the control exit of a method. Then, for each selected statement additional statements are chosen that assign values to variables used in the selected statements. Finally, when a selected statement is inside an if/while/for/switch statement, the statement containing one of these keywords gets selected. In addition, during each selection step a filter is used to remove statements containing unnecessary information, such as exception handling and logging. Lastly, all selected statements are ordered ascending based on the line number in a method. Then summaries are generated using the selected statements in predefined templates. Different templates have been designed for single method calls, return statements, nested and composed method calls, assignments, conditional expressions and loop expressions.

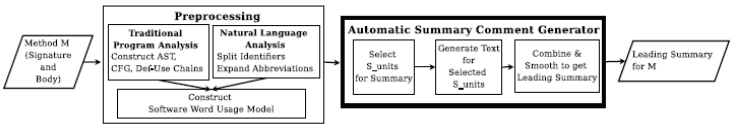


Figure 2.2: Process of generating a methods summary [33].

2.3.2 Summarising High Level Actions within Java Methods

Based on the method summarisation described in the previous paragraph, Sridhara et al. developed a new approach for summarising high level abstractions of actions in methods [34]. This is done by separately analysing code fragments present in a method, of which the process can be seen in Figure 2.3. The different kinds of code fragments analysed are sequences of statements, conditional statement blocks and loops.

The techniques used for splitting and extracting words from identifiers and expanding abbreviations are the same as used in the former approach. Also the AST, CFG and SWUM are used. The statement selection differs from the previous approach, since now this process is based on the three different kinds of code fragments. For sequences, a newly designed algorithm is used to identify statements with similar actions which contain one or more method calls. The algorithm takes into account whether statements contain the same verb, noun and class attribute. For conditional statement blocks, statements in the *then* clause are compared with statements in the *else* clause. The same strategy is used, except that instead of a verb, a predicate is taken into account. In addition not only statements with method calls, but also return statements and assignments to variables are taken into account. For loops, rules have been defined to identify five high level actions implemented by loops.

Templates have been designed so that these actions can be identified. Once this is done, a summary can be generated. Templates as previously defined in [33] are used to generate the summarising sentences for each code fragment, using the template that corresponds to the specific type of fragment.

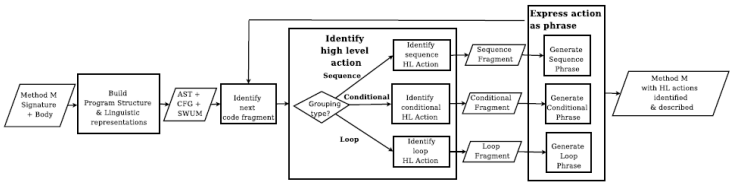


Figure 2.3: Process of identifying and describing high level actions in a method [34].

2.3.3 Generating Descriptive Comments for Java Method Parameters

Another variation on the method described in section 2.3.1 has been made by Sridhara et al. in [35]. Here an approach is given for generating parameter comments, which are added to existing method summaries as text or as a separate *@param* comment.

The same techniques as in the former approach are used for preprocessing statements in methods. Unnecessary statements are removed by combining techniques from [1] and [34]. Then for each statement the closeness of the usage of a parameter to the computational intent of the method is computed. This is done using predefined heuristics, ordered from closest to farthest to computational intent: parameter already appears in the summary text, parameter is used in an already selected statement, a ubiquitous method, link to summary via variable in summary phrase, link to summary via variable in summary statement but not in phrase, link to summary via intermediate variables, and use in a conditional expression not part of the summary. Then phrases are generated using templates from [33]. Additional templates were made to transform phrases so that the emphasis would lay on the parameter rather than the verb. Finally the generated phrases can be added to existing method comments or a *@param* comment can be created.

2.4 Overview

Now that summarisation methods have been described, an overview of all techniques can be given. When grouping all steps taken in the summarisation process, the division as shown in Table 2.2 can be made. All methods contain techniques for lexical preprocessing and text retrieval, so these can be named as the first two stages in the process of comment generation. In addition, the comment itself has to be generated. Most methods use predefined templates as a basis and put the text obtained from the lexical preprocessing and text retrieval into the templates. These steps can be named as the last two stages of the summarisation process. This means that there are four main stages in the process of comment generation: lexical preprocessing, text retrieval, summary generation and summary aggregation. This information can be used as a basis for the design of a comment generation tool for integration tests,

Summarisation Technique	Lexical Preprocessing	Text Retrieval
Haiduc et al. [14]	stop-word filtering, spitting, case normalisation, stemming & creation of text corpus	LSI
Haiduc et al. [15]	stop-word filtering, splitting, case normalisation, stemming & creation of text corpus	log, tf-idf, binary-entropy for LSI & VSM
Eddy et al. [6]	stop-word filtering, splitting, case normalisation, stemming & creation of text corpus	tf-idf for hPAM & lead + hPAM
Rodeghero et al. [30]	stop-word filtering, splitting, case normalisation, stemming & creation of text-corpus	VSM + weights eye-tracking study
McBurney et al. [23]	project conversion to directed graph, addition of root phantom node & conversion of nodes to documents	HDTM
Moreno et al. [24]	grouping class elements into 'include directly' and 'analyse further'	JStereoCode, stereotype-based & access-level filter
Sridhara et al. [33]	splitting & expanding abbreviations	AST, SWUM & CFG
Sridhara et al. [34]	splitting, expanding abbreviations & variable lexicalisation	AST, SWUM & CFG
Sridhara et al. [35]	splitting & expanding abbreviations	SWUM, selecting statement parameters, pruning, closeness & linking context identification

Table 2.2: Division of all techniques used in summarisation methods.

which will be shown in Chapter 3.

RQ1 *There are four main stages in the process of comment generation: lexical preprocessing, text retrieval, summary generation & summary aggregation*

2.5 Additional Methods

More methods exist for summarising source code than shown in the previous paragraphs. After reviewing the nine methods above, a pattern of stages in the process of summarising code could be found. That is why it has been chosen not to discuss additional methods in the same level of detail. To give an idea on what other methods exist, a brief explanation for several more is given here.

An example is the method created by Rastkar et al. [29]. Here a summary is generated from source code to provide information for cross-cutting source code concerns. This has been done to make it easier for developers to perform change tasks. There are also methods that generate a different kind of summary. Examples are summarisation by folding less informative code [9] and generating pseudocode as a summary [11]. Other methods do not use source code itself to create a summary. Question and answer websites are used by Wong et al. [37] to generate summaries for open source projects. This is done by first summarising code fragments and then linking these to similar code in the open source projects. Finally, there are also methods that summarise fragments and use these summaries for the fragments themselves. Pozanelli et al. [28] summarise fragments on Stack Overflow using information of the discussion around a fragment. Guerrouj et al. [13] also use the context for summarising fragments on Stack Overflow, but they focus on library identifiers only. Fragments demonstrating the use of an API are summarised by Ying et al. [38] by creating a fragment smaller than the original one, using only the most informative lines of the original fragment.

Chapter 3

Generating Comments for Test Code

In this chapter a look will be taken at *TestDescriber* and the new tool that generates comments for integration tests. The difference between these two tools will be made clear, looking at the comment itself and implementation techniques. The techniques will also be shown in the pattern of the main stages in the process of comment generation, as discovered in the previous chapter.

3.1 TestDescriber

Comments for automatically generated *JUnit* unit tests are generated by *TestDescriber*. In this section the techniques used for generating test code and comments are given. Then results of the case study are shown. After that, the techniques are filled in the main stages of comment generation to give an overview.

3.1.1 Implementation Technique

Four steps are taken in the *TestDescriber* approach: test case generation, test coverage analysis, summary generation and summary aggregation.

Test case generation is done by using *EvoSuite* [10]. This tool generates *JUnit* test cases for Java code. It is said that using *EvoSuite* is not mandatory, but it has been chosen because it reaches high structural coverage with minimal generated test cases that have the minimal set of test assertions.

In the second step *Cobertura* [3] is used to collect information by looking at the branches and statements under test in each test case. In addition a parser was developed based on *JavaParser* [18], which extracts keywords from identifier names for building the main textual corpus. For each test case the following information is collected: the list of attributes and methods (in)directly invoked by the test case, for each invoked method the statements executed, attributes/variables used and calls to other methods, and lastly the boolean values in if-statements to see which conditions are verified when covering a true/false branch.

During summary generation three steps are undertaken. First camel case splitting and expanding abbreviations is done as preprocessing. Then part-of-speech tagging is done by

using *LanguageTool* [21], where terms are classified as verbs, nouns and adjectives. Heuristics are used to generate natural language sentences. These are similar to ones defined in [17] and the approach by Sridhara et al. described in 2.3.1. Finally the summary generation is done using predefined templates. A summary consists of three different parts. First a general description of the class under test is given, then a brief summary of the structural code coverage of each *JUnit* test case, and lastly a fine grained description of statements inside each test case and if-statements traversed in the executed path, to describe the flow of operations performed during testing. The class level summarisation uses an approach based on the one by Moreno et al. [24] described in 2.2.1. Here however, only the interface and the attributes of a class are used as heuristics, since the behaviour and methods of the class are described in other parts of the summary. The second part consists of a general description of the statement coverage scores obtained by each test case. The coverage information from *Cobertura* is used in a predefined template. The final part uses code elements with SWUM [16] to generate fine-grained statement summaries. Based on the type of statement (constructor, method call or assertion statement) different code elements are used in customised templates. If applicable, a method call description is extended with a summary describing the boolean expression of an if-condition in the called method.

The final step is the summary aggregation, where all the generated information is added to the test class. In Figure 3.1 an example of a summary is given. The general description of the class under test is added as a block comment before the declaration of the test class (3.a). The brief summaries of the statement coverage scores obtained by each test case is added as block comment before the corresponding test method (3.b). Finally, the fine-grained descriptions are inserted inside each test case as inline comments to their corresponding statements (3.c & 3.d).

```
1| /** The main class under test is Option. It describes
2|  * a single option and maintains information regarding:
3|  *  - the option;
4|  *  - the long option;
5|  *  - the argument name;
6|  *  - the description of the option;  3.a
7|  *  - whether it has required;
8|  *  - whether it has optional argument;
9|  *  - the number of arguments;
10|  *  - the type, the values and the separator of the option;**/
11| public class TestOption {
12| /** OVERVIEW: The test case "test0" covers around 3.0% 3.b
13|  * (low percentage) of statements in "Option" **/
14| @Test
15| public void test0() throws Throwable {
16| // The test case instantiates an "Option" with option 3.c
17| // equal to "", and description equal to "1W|^".
18| Option option0 = new Option("", "1W|^");
19| // Then, it tests:
20| // 1)  whether the description of option0 is equal to 3.c
21| // "1W|^";
22|  assertEquals("1W|^", option0.getDescription());
23| // 2) whether the key of option0 is equal to ""; 3.c
24| // The execution of the method call used in the assertion 3.d
25| // implicitly covers the following 1 conditions:
26| // - the condition "option equal to null" is FALSE;
27|  assertEquals("", option0.getKey());
28|  }
29| }
```

Figure 3.1: Example of a comment generated by TestDescriber [27].

3.1.2 Case Study

A case study was performed with a focus on the impact of test case summaries on the bug fixing performance of developers. It was found that developers discovered twice as many bugs when generated comments were provided. The test case summaries also significantly improved the comprehensibility of test cases, which was found useful by developers. For evaluation the quality of the summaries, three questions were given. These were similar to the ones proposed in [24] and [33]. The first question concerned whether the summary was missing important information about a class under test. The second question was about the summary containing unnecessary information or not. In the last question it was asked if the comments were readable and understandable. The results were that 50 percent found it was not missing any information, 37 percent found some information was missing, and 13 percent found very important information was missing. Then 38 percent said no unnecessary information was present, 52 percent said some and 10 percent said a lot of unnecessary information was present. Finally, 70 percent found that the summary was easy to read and understand, and 30 percent found it was somewhat readable and understandable.

3.1.3 Overview

The type of summarisation steps in this approach are similar to the ones in the methods described in Chapter 2. All four steps are present in the summarisation process. In Table 3.1 the techniques are given in the main stages.

Stage	**Technique(s)**
Lexical Preprocessing	camel case splitting, expansion of abbreviations, *LanguageTool*
Text Retrieval	SWUM, *Cobertura*, Java parser
Summary Generation	predefined templates, class level summarisation based on [24], natural language generation heuristics based on [17] and [33]
Summary Aggregation	filling template with retrieved text, adding comments to test code

Table 3.1: *TestDescriber*'s techniques in the main stages of comment generation.

3.2 Generating Comments for Integration Tests

Based on *TestDescriber* a new method has been implemented that generates comments for *JUnit* integration tests. This has been done to try to improve code comprehension. In this section, first the template of the comment is shown, to give an idea of what a comment will look like. Then the implementation techniques are described, including an overview where the techniques are shown in the main stages of comment generation. In addition an example of a generated comment is given.

3.2.1 Comment Template

The generated comment consist of several parts. First, the covered classes and methods in a test case are listed. This is done to give the developer a quick overview of what is tested. Then, for all methods it is given to which class they belong and what the coverage percentage of these methods is. In this way, when more background information needs to be read, it is known where this can be found. The coverage information could be an indication of how big a part the method is of an integration test. In addition the *Javadoc* that belongs to the methods is given as a description. By using *Javadoc*, the main intent behind a method can be found and quick understanding of a method is made possible. Finally, a description of the assert(s) is given. This helps a developer understand what is tested in the integration test. This results in the following template:

Test case ⟨*name of test case*⟩ **covers class(es) [**⟨*name of class(es)*⟩**] and method(s) [**⟨*name of method(s)*⟩**]**

Method ⟨*name of class*⟩**.**⟨*name of method*⟩ **[**⟨*coverage percentage*⟩**]:**⟨*Javadoc of method*⟩

It tests:
⟨*description of assert(s)*⟩

The comment only shows classes and methods that belong to the project itself, so for instance no Java or external libraries. This is done to make sure that a good overview is given of which parts of the project are tested. It also omits a description for get and set methods. These methods might be less difficult to understand compared to other methods. So to avoid lengthy comments and possible redundant information, these methods are left out from the description. When there are no methods and/or asserts in a test case, these parts are not included in the comment to avoid adding empty templates to the comment.

3.2.2 Implementation Techniques

To be able to fill in the template, first text needs to be retrieved. If necessary, lexical preprocessing will be performed. Each part of the comment has information that is gathered in a different way from multiple sources:

Classes Names of classes are obtained by using a regular expression on the body of a test case. The body is retrieved by parsing the test's code file.
Methods Names of methods are obtained by using *TestDescriber*'s functionality of parsing a test class. Here a test case is saved as an object and in that object all called methods are stored.
Classes of methods The class to which a method belongs is found by parsing the files of covered classes. After this, it is checked whether a method is present in a class or not.
Test coverage percentage The coverage percentage of methods is obtained by using the *EclEmma* tool for Eclipse. This tool generates an *XML*-file with all the coverage informa-

tion. The file is parsed and using the method name, the coverage percentage can be found.
***Javadoc* of methods** The *Javadoc* belonging to covered methods is obtained by parsing the class file of the method. When the method is found, the *Javadoc* is extracted, including only the description of the method. This means that for instance @*return* and @*param* statements are left out. This has been done to avoid lengthy comments, since the description itself could already be quite big. Also, the description should contain the main idea behind a method, which is what is aimed for here.
Description of asserts The assert descriptions are obtained by using *TestDescriber*'s functionality of generating comments for asserts.

After all information is gathered, this is put into the predefined template. Then the comment is complete and is added to the test case as a *Javadoc* comment. An example of a generated comment is given in Figure 3.2.

In the original project of *TestDescriber* the *Cobertura* tool was used for obtaining coverage information of test cases. Here however, *EclEmma* is used. *Cobertura* was found to be unstable when testing *TestDescriber* on different projects. That is why *EclEmma* was chosen instead, to make sure that the coverage information could be retrieved for multiple projects.

```
/**
 * Test case testNoShortestPath tests class(es) [Board, Square, Direction, Navigation, Unit]
 * and method(s) [parseMap, shortestPath, getBoard, squareAt].
 *
 * Method MapParser.parseMap [62%]: Parses the text representation of the board into an actual level.
 * Supported characters: ' ' (space) an empty square. '#' (bracket) a wall.'.' (period) a square with a pellet.
 * 'P' (capital P) a starting square for players. 'G' (capital G) a square with a ghost.
 * Method Navigation.shortestPath [100%]: Calculates the shortest path. This is done by BFS.
 * This search ensures the traveller is allowed to occupy the squares on the way,
 * or returns the shortest path to the square regardless of terrain if no traveller is specified.
 * Method Board.squareAt [45%]: Returns the square at the given <code>x,y</code> position.
 *
 * It tests:
 * 1) whether "path" is null.
 */
@Test
public void testNoShortestPath() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# # #", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(3, 1);
    List<Direction> path = Navigation.shortestPath(s1, s2, mock(Unit.class));
    assertNull(path);
}
```

Figure 3.2: Example of a comment generated for an integration test.

3.2.3 Overview

In Table 3.2 it can be seen that the new tool is designed in the same way as the other existing comment generation methods. The techniques are given in the main stages of comment generation. All techniques used here are not so different from the ones used in *TestDescriber*, except using *EclEmma* instead of *Cobertura*. The most important difference is what kind of information sources are used. The biggest difference is the usage of existing *Javadoc* belonging to the code covered by an integration test, which is a novel approach compared to existing methods.

Stage	**Technique(s)**
Lexical Preprocessing	camel case splitting, expansion of abbreviations, *LanguageTool*
Text Retrieval	SWUM, *EclEmma*, Java parser
Summary Generation	predefined template
Summary Aggregation	filling template with retrieved text, adding comment to test code as *Javadoc*

Table 3.2: Techniques of the new tool in the main stages of comment generation.

Chapter 4

Experiment

In this chapter the experiment conducted for evaluating the new tool is described. First the main goals of the experiment are given. Then the design of the experiment is shown. After that the results are given.

4.1 Main Goals

Based on **RQ2**, the most important goal of the experiment is finding out what the effect of a generated comment is on the comprehension of source code. Other goals include retrieving information on the opinion of developers on the comment itself. A look can be taken at the quality, considering precision, conciseness and readability. Also the comment's usability for performing the experiment can be looked into.

4.2 Design

Online surveys have been created for conducting the experiment. A survey consisted of a short introduction and three main parts. The first two parts both consisted of a programming task and a post-test questionnaire. In the third part another questionnaire was given. In the programming task a test case needed to be changed. These test cases were based on the *JPacman* framework [19]. This is a Pacman-like game used for teaching software testing and is written in Java. The code fragments used in the surveys can be seen in Appendix A and the solutions to the exercises in Appendix D.

Two different surveys were available, version A and version B, which can be found in Appendix B and Appendix C respectively. The test cases used in the code fragments and posed questions were the same in both surveys. The only difference was for which code fragment generated comments would be included. The test cases used in both surveys were called *testLeftArrowButton* and *testNearestFood*. For version A, only the fragment containing *testNearestFood* included comments, where for version B comments were only given for the fragment containing *testLeftArrowButton*. For both surveys, in the first part a task had to be performed on the code fragment not including any comments. In the second part, a task where the code fragment included comments had to be done. This setup is

called one group post-test: all participants first have to perform a task and afterwards have to answer questions.

The reason for using a one group post-test experiment setup with two different versions was to avoid a possible learning effect. The participants had to perform their tasks on two different test cases that tested various parts of the *JPacman* framework. The first task without comments, the second one including comments. Another reason was that by performing a programming task on a test case with and without generated comments, the difference in how well a task is performed and how questions are answered can be seen, based on whether or not a comment is provided. In addition, the two versions were assigned to participants in such a way that there was an equal number of participants for both versions.

In the introduction a brief explanation of the survey was given. Also a screenshot of the *JPacman* game had been included to give an idea about what the game looks like.

In part I the first programming task without a generated comment had to be performed. A few sentences with a bit of context were given at the beginning of the exercise. Participants had to write down at what time they started and finished the task. Since the survey's styling was not appropriate for source code, participants had to copy the code to the *CompileJava* website [4] and perform the task there. When they were done, they had to paste their solution in the survey. Then two statements were given concerning the difficulty of the task and whether the full source code of *JPacman* was missed for understanding the code fragment. The participants had to indicate if they agreed or disagreed with these statements. A five-level Likert scale [22] was used with the statements, ranging from strongly agree to strongly disagree. Finally, a question was asked where the steps taken to complete the task had to be written down. This was done to find out if a similarity in approach could be found between participants and if so, whether a certain approach would have influence on how the tasks were performed or how the comments would be evaluated.

In part II the second programming task with a generated comment had to be performed. First the structure of the comment was explained, pointing out what kind of information was included and what was left out. Then, similar as in part I, a few sentences of context were provided and the time of starting and ending the task had to be filled in. Then Likert statements were given, including the same as in part I, and additionally statements concerning: whether the comment made it easier to understand the code, whether the task in part I would have been easier if a similar comment was provided there, and whether this functionality of automatically generating comments for tests would be used in projects participants work on by themselves and/or with multiple people. After that questions were asked, again the one where taken steps had to be given, and additionally: whether the comment was used and if so, which parts of the comments were used, whether the comment was missing important information, whether it contained redundant information and whether it was readable and understandable. In the final question it could be written down how they would change the comment if possible, and why.

In part III three background questions were asked. The first question concerned the study or profession of a participant. The second one was about how many years of Java programming experience someone had, and the final one about the years of testing experience. Finally, any comments that participants wished to share could be given.

4.3 Results

4.3.1 Participants

In total 18 participants took part in the experiment, of which 16 students, 1 software developer and 1 software engineer. Most of the students were MSc Computer Science (7). Then there also were BSc Computer Science students (3) and MSc Human Computer Interaction Design students (2). The remaining 4 students were MSc Embedded Systems, MSc Information Technology, MSc Cyber Security and Msc Security & Privacy. Most of the participants had 3-6 years of Java programming experience (67%), followed by 1-2 years (28%) and less than 1 year (5%). Experience in testing was less, since most of the students had less than 1 year of experience (55%), followed by 1-2 years (28%) and 3-6 years (17%). An overview of experience per participant is given in the figure below.

Participant	Experience in Java	Experience in Testing
P1	1 - 2	1 - 2
P2	3 - 6	1 - 2
P3	1 - 2	3 - 6
P4	3 - 6	< 1
P5	3 - 6	1 - 2
P6	3 - 6	< 1
P7	3 - 6	3 - 6
P8	3 - 6	1 - 2
P9	< 1	< 1

Participant	Experience in Java	Experience in Testing
P10	1 - 2	< 1
P11	1 - 2	< 1
P12	3 - 6	1 - 2
P13	3 - 6	< 1
P14	1 - 2	< 1
P15	3 - 6	< 1
P16	3 - 6	3 - 6
P17	3 - 6	< 1
P18	3 - 6	< 1

Figure 4.1: Java programming and testing experience of participants in version A (left) and version B (right).

For each programming task, participants had to write down which steps were taken to complete a task. The majority of the participants only wrote down what they changed to get the solution. They did not write every step they took for solving the task, e.g. reading all the code first, reading the assert statement first, or looking at specific parts of the comments.

4.3.2 Execution of Programming Tasks

In total 17 out of the 36 performed tasks were solved correctly. When an answer to a task was only partially correct, it was considered incorrect. The tasks without comments got 8 and the tasks including comments got 9 correct solutions. When looking at the test cases used in tasks, for *testLeftArrowButton* 12 correct solutions were given. For *testNearestFood* only 5 correct solutions could be given. In the tables in Figure 4.2 below, for each participant it is shown whether a task was solved correctly or not, indicated by the colour.

The total number of minutes needed to complete a task are also given in these tables. Here it can be seen that for tasks which included a comment, more time was spent when comparing it to the same task without a comment. For instance task 1 in version A (not including comments) and task 2 in version B (including comments). For task 1, 40 minutes were needed, where 45 minutes were needed for task 2. When looking at the difference between time spent on tasks containing containing *testNearestFood* or *testLeftArrowButton*, it can be seen that for *testNearestFood* a lot more time was spent.

Participant	**Task 1 - no comments**	**Task 2 - comments**
P1	1	7
P2	7	5
P3	1	3
P4	4	11
P5	3	8
P6	10	13
P7	2	2
P8	4	12
P9	8	12
Total	**40**	**73**

Participant	**Task 1 - no comments**	**Task 2 - comments**
P10	6	2
P11	6	5
P12	6	5
P13	10	8
P14	6	10
P15	11	6
P16	1	3
P17	18	4
P18	3	2
Total	**67**	**45**

Figure 4.2: Minutes needed for performing a task & correct (green) or incorrect (red) solution given by participants in version A (left) and version B (right).

In version A, 8 participants used the comment to perform task 2. In version B, 6 used the comment. This means that on average, 78.8% of the participants used the comment to solve the exercise. In Figure 4.3, for each part of the comment the number of participants is given that used this part to solve the exercise. The description of methods was used the most, followed by the description of what is tested. The list of classes and coverage percentage were used the least.

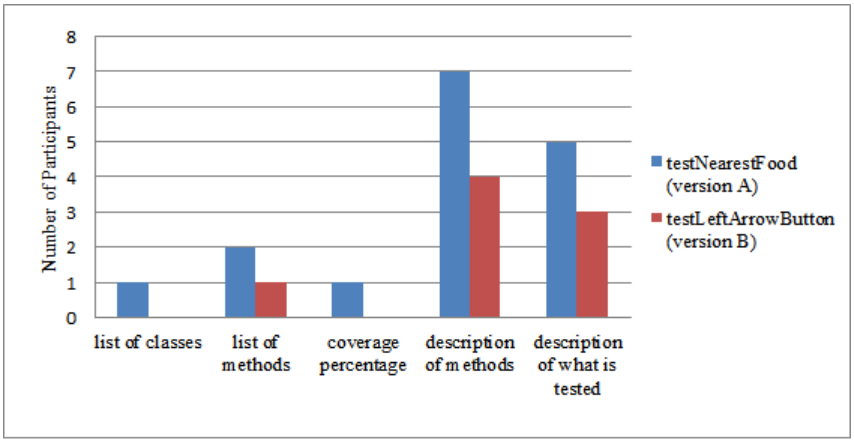


Figure 4.3: Parts of a comment used by participants to perform task 2.

4.3.3 Answers to Questions

After each programming exercise, first participants had to indicate for given statements whether they (strongly) agreed, (strongly) disagreed, or neither. In Figure 4.4 and 4.5 the perception of a task's difficulty is given. For *testLeftArrowButton*, only 1 participant found the task difficult, while for *testNearestFood* 3 found it difficult and 5 neither difficult nor easy.

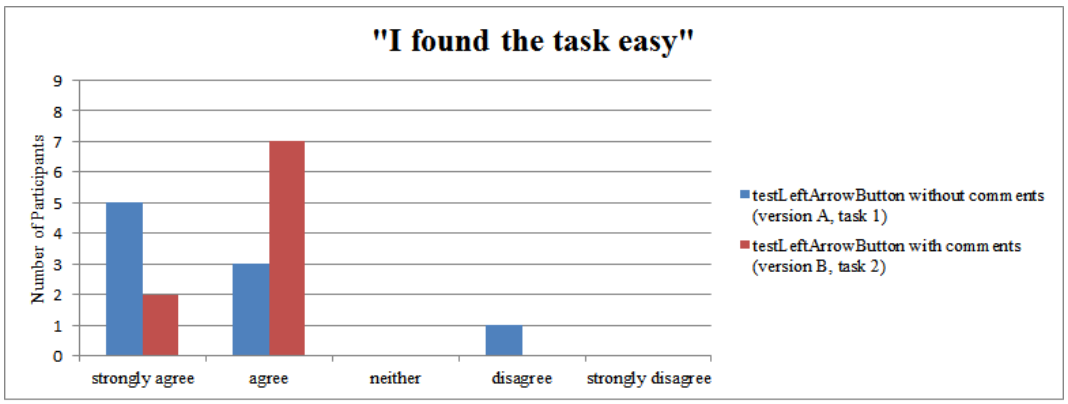


Figure 4.4: Difficulty of *testLeftArrowButton* with and without comments.

Figure 4.5: Difficulty of *testNearestFood* with and without comments.

Whether or not participants missed the full *JPacman* framework is given in Figure 4.6 and 4.7. For *testLeftArrowButton*, the framework was never missed, only 1 case neither missed nor unmissed. For *testNearestFood* it was more frequent. Here the framework was missed in 3 cases and in 1 case neither.

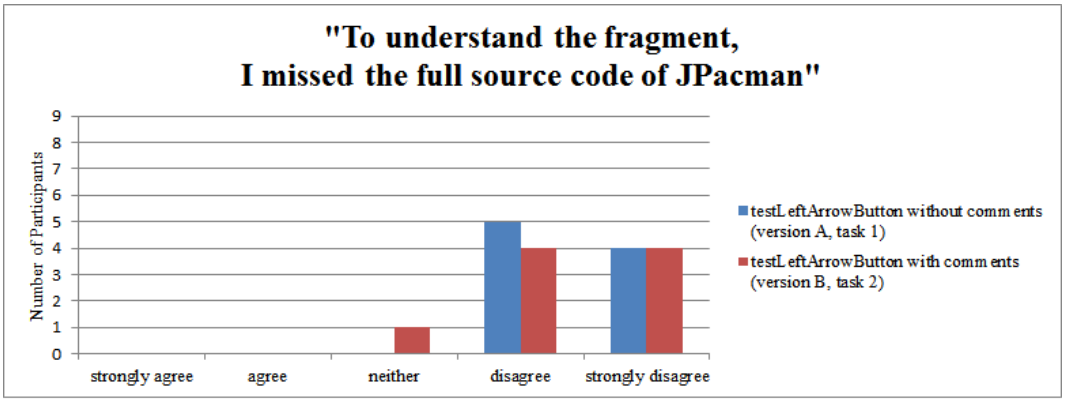


Figure 4.6: Missing full *JPacman* framework for *testLeftArrowButton* with and without comments.

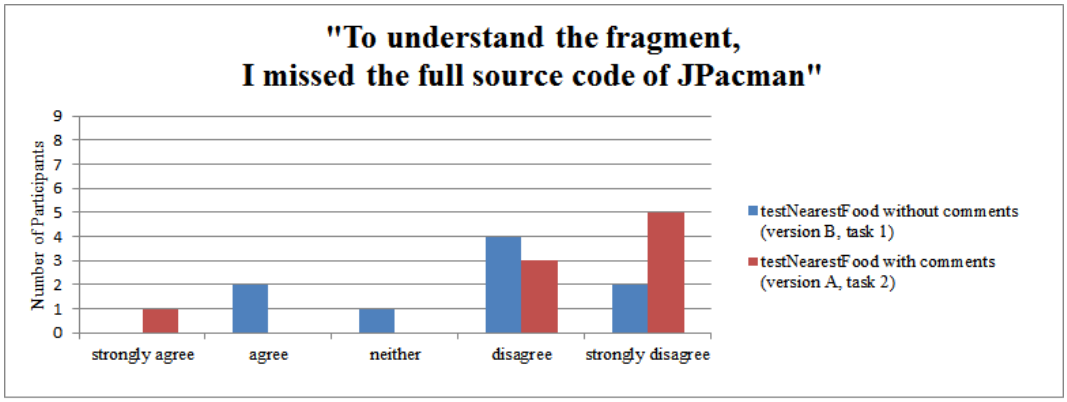


Figure 4.7: Missing full *JPacman* framework for *testNearestFood* with and without comments.

The most important question in the survey is whether or not the comment helps understanding a code fragment. The answer to this are given in Figure 4.8. For *testNearestFood* the majority of the participants found that the comment made it easier to understand the code. For *testLeftArrowButton* however, only one third thought that the comment helped them understanding the code. On average, 44.4% found that the comment helped for understanding code, 33.3% found it did not help and 22.2% found neither.

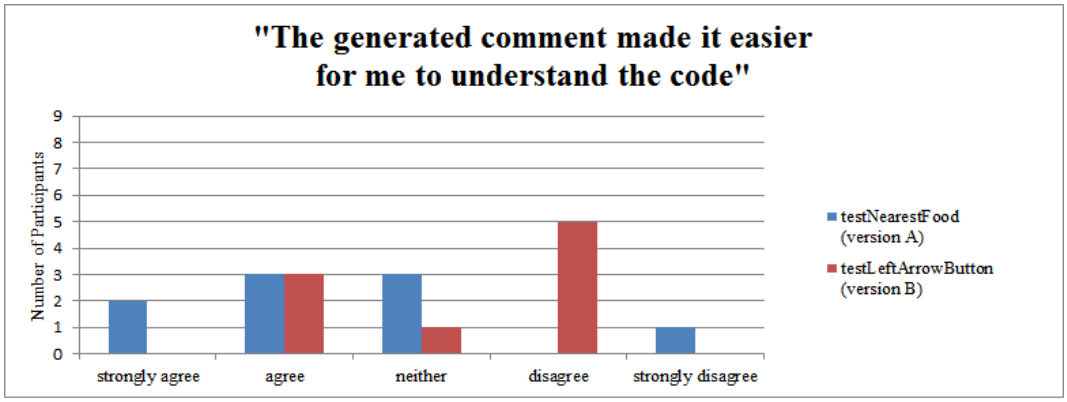


Figure 4.8: Easier to understand code with generated comments.

Figure 4.9 gives information on whether the task without comments would have been easier if generated comments would have been provided. Version A has *testLeftArrowButton* in task 1, version B has *testNearestFood* in task 1. For version A, the majority of the participants found that it would have been easier. In version B the opposite was found by the majority, namely that the task would not have been easier by providing generated comments.

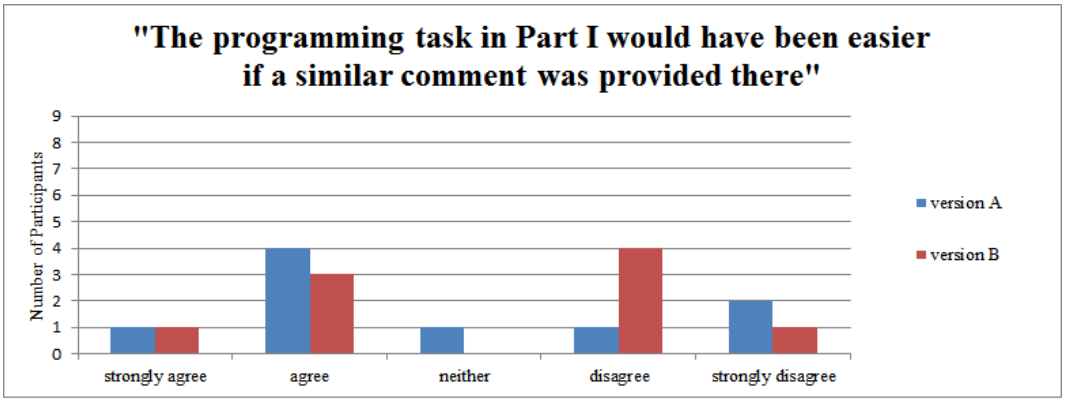


Figure 4.9: First programming task would have been easier with generated comments.

Whether or not the functionality of generating comments for tests would be used in participant's private project and/or shared projects is given in Figures 4.10 and 4.11 respectively. For private projects, most of the participants in version A say that they would use it, while in version B most indicate that they would not. When looking at shared projects, again in version A the majority would use the functionality. This time also for version B most of the participants would use it. On average, 50% would use the functionality on private projects and 44.4% would not. For shared projects, 50% would use it and 27.8% would not.

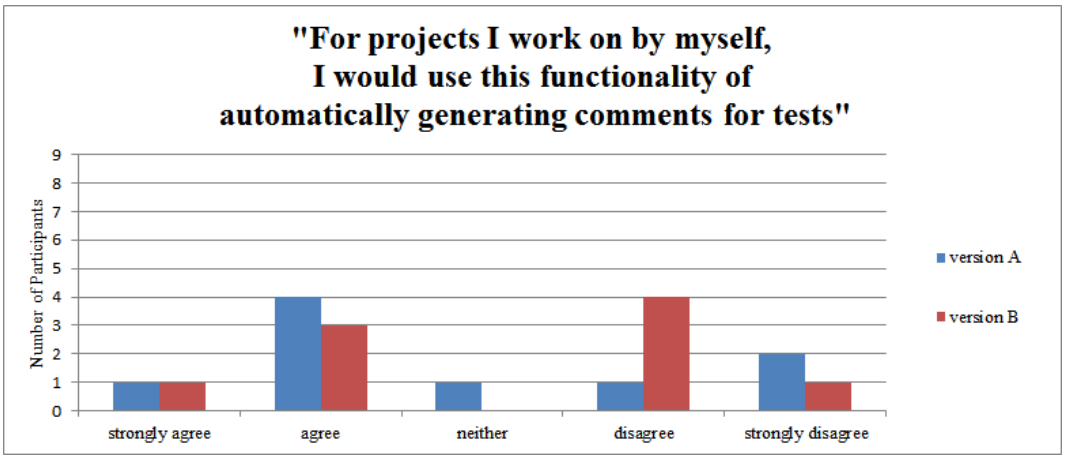


Figure 4.10: Using comment generation functionality in own projects.

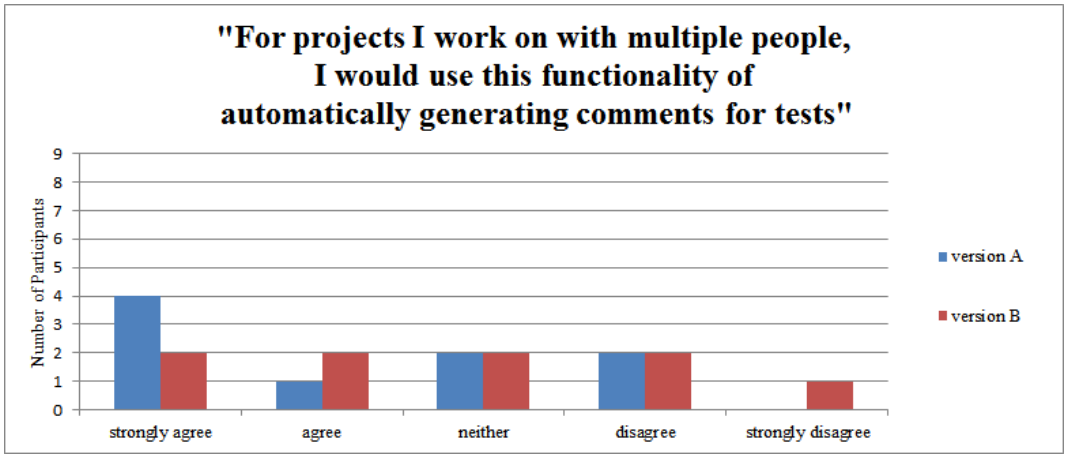


Figure 4.11: Using comment generation functionality in shared projects.

The quality of the comment has been tested on precision, conciseness and readability. The evaluation can be found in Figure 4.12, 4.13 and 4.14 respectively. All three aspects of quality were higher rated in version A than in version B. The majority of participants in version A found that the comment was not missing information and was easy to read and understand. For conciseness, the number of participants found that it had no unnecessary information was the same as those that found it has some unnecessary information. On average, half of the participants found that the comment was precise and readable, while for conciseness most found that some redundancy was present.

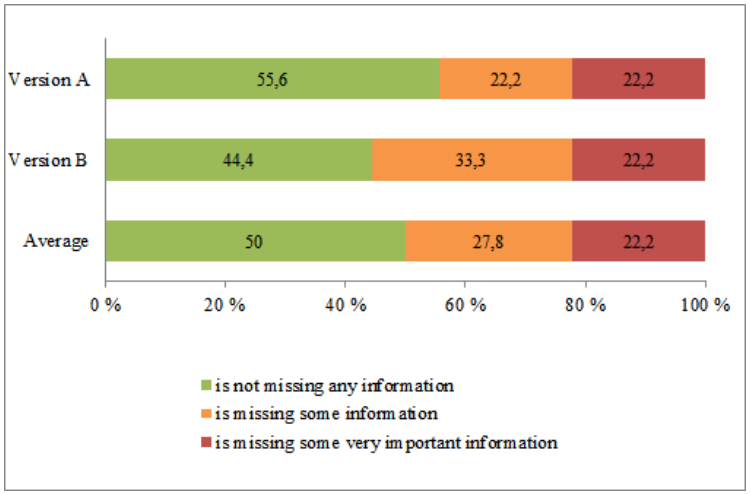


Figure 4.12: Precision of the generated comment.

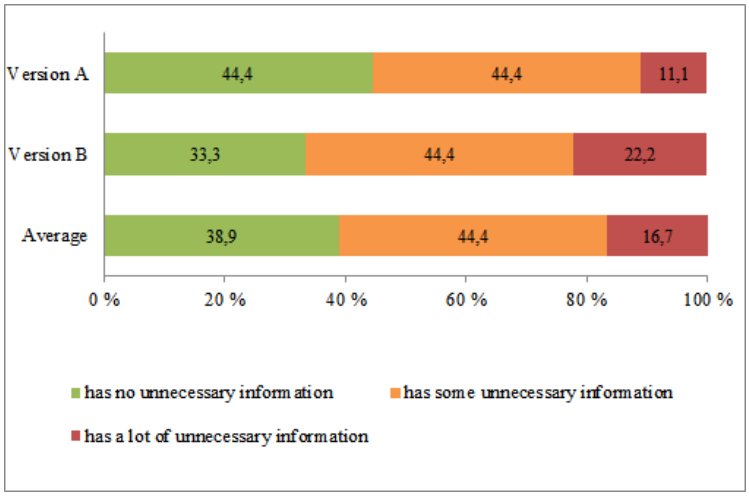


Figure 4.13: Conciseness of the generated comment.

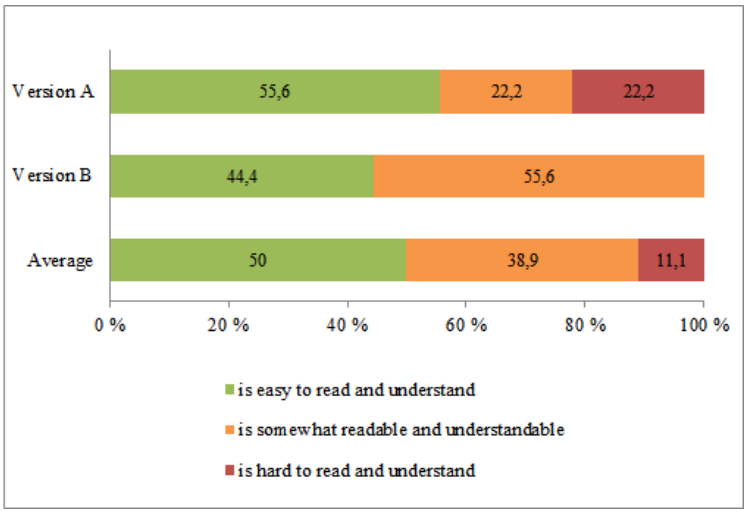


Figure 4.14: Readability of the generated comment.

Suggestion could be made on how participants would change the comment if they could. For 27.8% of the participants, the comment does not have to be changed. The coverage information would be removed by 22.2%, mainly because they did not know what it stood for. Also a few participants indicate that the list of classes and/or methods was not useful for them. Although it was chosen specifically not to include get-methods so that lengthy comments and redundant information could be avoided, one participant wanted to include these methods. Also a few participants did not want a block comment at the beginning of a test case. Instead, they wanted the comment to be divided among the code lines.

Finally, two participants gave a remark on the functionality of generating comments for integration tests: ”It can definitely help understanding the context of the project” and ”I

think this would really be super useful with a bit of tweaking". Here, adding comments in the code lines was suggested as possible tweaking.

There were a lot more participants with less than one year of testing experience compared to ones with 1-2 or 3-6 years of experience. In the following figures results are shown based on the testing experience of participants. The results belong to the statements about generated comments: whether the task in part I would have been easier with comments, whether the comments made it easier to understand code, whether the functionality of generating comments would be used in private and/or shared projects.

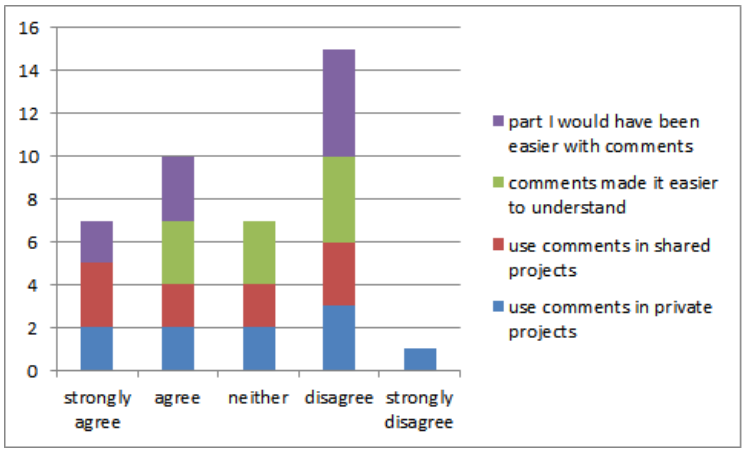


Figure 4.15: Answers by participants with less than one year of testing experience.

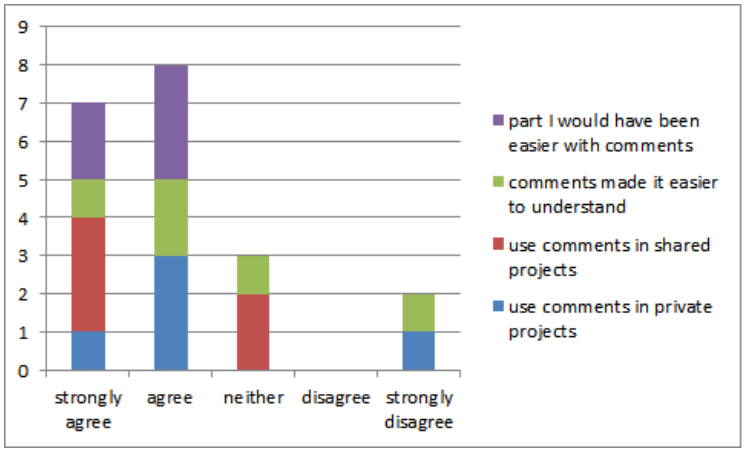


Figure 4.16: Answers by participants with 1-2 years of testing experience.

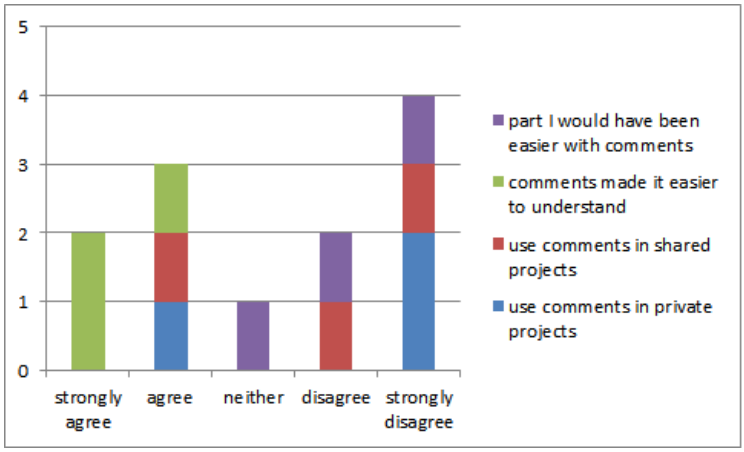


Figure 4.17: Answers by participants with 3-6 years of testing experience.

Chapter 5

Discussion

First an interpretation of the results found in the experiment are discussed. Then threats to the validity of this thesis are given in the framework of internal, external, and construct validation, plus limitations.

5.1 Interpretation of Results

Correct solutions to programming tasks In total 17 out of 36 tasks were solved correctly. For the task including *testLeftArrowButton*, there was no difference in the number of correct answers based on whether or not comments were available. In both cases 6 correct answers were given. The task with *testNearestFood* got 2 correct answers without comments and 3 with comments. This means that based on only the number of correct solutions, no significant result can be found for comments helping participants solving programming tasks.
Time taken for programming tasks When comparing tasks with and without comments, a difference in the amount of time spent can be seen. In total, it took participants 40 minutes without comments and 45 minutes with comments for the task including *testLeftArrowButton*. For *testNearestFood* it took them 67 and 73 minutes respectively. This means that participants needed more time for tasks including a comment, compared to the same task without a comment. This could be due to the fact that when participants used the comment for solving the task, more text had to be read and understood.
Difficulty level of programming tasks Several results indicate that the programming task with *testNearestFood* is more difficult than the one with *testLeftArrowButton*. First, for *testNearestFood* 5 correct solutions were given compared to 12 for *testLeftArrowButton*. Second, much more time was spent on tasks with *testNearestFood*: 67.5% more without comments and 62.2% more with comments. Finally, for *testLeftArrowButton* only one participant found the task difficult and all others found it easy. The task with *testNearestFood* was found difficult by 3 people and 5 found it neither difficult nor easy.

Based on the difficulty of a task, now it can be considered whether there is a difference in the perception of generated comments. In survey version A the more difficult test case is provided with comments, where in version B the more easy one is given comments. When comparing the results regarding generated comments, the following can be seen. In version

B the comments were found not to make it easier to understand code, where in version A it was found easier. In version B it was also thought that the task without comments would not have been easier if they would have been provided, where in version A the opposite was found. Also when looking at participants wanting to use the tool for their own public and/or private projects, people in version B were less willing to use the tool compared to version A. In addition were the generated comments used by 6 participants in version B compared to 8 in version A. This could mean that the generated comments are less useful when the test code is easier. To confirm this, more experiments have to be done including code fragment with different levels of difficulty.

Experience in Java programming & testing It is possible that testing experience has some influence on the ability of solving a programming task. When participants are grouped on their years of testing experience, and the number of correctly solved tasks are compared with the ones that were not solved correctly, the following can be seen. For the group with 3 to 6 years of experience, 83.3% of the tasks were solved correctly. The group with 1 to 2 years of experience solved 50% of the tasks correctly. For the group with less than 1 years of experience, only 35% was solved correctly. This is an indicator that the less testing experience a participant has, the smaller the chance is of solving a task correctly. Survey version B might have been negatively influenced by this, since there were more participants with less than one year of experience compared to the participants in version A.

It is also possible that less testing experience could be in favour of the generated comments. With less experience it is more likely that comments are needed for understanding code. Figure 4.15, 4.16 and 4.17 show the results related to questions about comments, given by participants with less than one, 1-2 and 3-6 years of experience respectively. For all statements, the answers would be in favour of the generated comments when participants agree. Considering this, it can be seen that actually not the participants with less than one year experience value comments the most, but the participants with 1-2 year of experience. This indicates that less experience in testing does not automatically have to mean that generated comments will be more appreciated. Since the number of participants per group of testing experience is not divided equally, this should be verified again with where there are similar numbers per group of experience.

Usability of the commenting tool For project participants work on by themselves, there was only a slightly higher preference for wanting to use the functionality of generating comments for tests: 50% compared to 44.4%. When looking at projects on which multiple people are working, however, the difference is much bigger: 50% against 27.8%. When people work on a project by themselves, they have written all the code and test on their own. Then they know better how everything is working and what is tested. For shared projects, other people could have written test cases or code belonging to a test case. This means that it is more likely for people not being able to understand the code. Here it could be useful to have automatically generated comments.

Quality of the generated comments The quality was tested on precision, conciseness and readability. Overall, 77.8% of the participants found that the comments were not missing anything or only some information. None of some unnecessary information seemed present for 83.3%. Finally, 88.9% found that the comments were (somewhat) easy to read and understand. This means that the generated comments are perceived as good quality by most

participants.
Content of the generated comments During the performance of programming tasks, the descriptions of methods were used the most by participants. After these descriptions, the description of what is tested was used most. This means that for understanding and solving programming tasks, these parts of the comments can be considered the most useful. On the other hand, suggestions have been made towards changing the content of comments. From all the parts that could be removed, most participants thought this should be the coverage information, mainly because they did not know what it meant. This was followed by the list of classes and methods. Since the participants were only provided with a code fragment and not the full framework, it is understandable that the coverage information and list of classes/methods has not much use to them. This might be different when the full project would have been provided. Another suggestion was changing the position of the comments, namely divided among the code lines in the test case. It could be useful to place the description of a method above the line where it is called. Then a developer does not have to look up and down switching between the description and method call. However, this could make the test case look cluttered, also because sometimes *Javadoc* can be quite long. It could be especially inconvenient for developers that already know how a test case works, then the comments only make it more difficult to read a test case.
Comprehension of source code Most of the participants found that the comment made it easier for them to understand the code. Also, more participants thought that the task without comments would have been easier if comments were provided. This means that **RQ2** can be answered as follows:

RQ2	*Generated comments for integration tests can help making it easier to understand code.*

5.2 Threats to Validity

Internal validity concerns with threats of causal conclusions based on the study performed. There might have been a difference in difficulty of test cases used in the programming exercises. For *testNearestFood* the task took much longer, was perceived as more difficult and there were less correct solutions given. It would have been better if first a test was done to find out what the average perception of difficulty was on multiple programming tasks. Then two tasks could have been selected with a similar difficulty level.
External validity involves the generalisability of the results found in this study. When a possible relation between testing experience and the approval of generated comments was looked into, participants were divided in three groups based on their experience. There were much more people in the group of least experienced compared to the other two groups. Although the groups were used as if they were balanced. This means that the indication obtained from this is not 100% correct. As said before, this test should be performed again with equals groups. Then it can be seen whether the indication that less testing experience is not automatically in favour of generated comments is surely true.

Also in the experiment only test cases have been used based on the *JPacman* framework. More experiments should be conducted where test cases from other project are chosen.

Construct validity includes the setup of the experiment. A threat to validity is that the groups used for survey version A and B did not have an equal number of participants with the same level of testing experience. To make sure that this could have less influence on the results, the participants should have been divided equally over the two versions based on their experience.

Concerning the format of the survey, the plan was to use the following format of a Likert scale item: (strongly) agree, (strongly) disagree and neither agree nor disagree. However, the format of the survey only allowed the assignment of the labels strongly agree and strongly disagree. All labels in between could not be shown. This can for instance be seen in Appendix B: Part I question 4. The questions would have been more clear if all labels could have been provided.

Limitations. The newly developed tool cannot be used together with *TestDescriber*. This is because the tool generates comments for integration tests, while *TestDescriber* does this for unit tests. *TestDescriber* is designed in such a way that information can only be obtained from the main class under test. For unit tests this is only one class. For integration tests however, multiple classes are under test, so information needs to be collected from multiple classes. Another reason why it is not integrated in *TestDescriber*'s project, is that *TestDescriber* uses *Cobertura* to retrieve test coverage information, which was found to be unstable when testing *TestDescriber* on several projects. If the new tool and *TestDescriber* would be run together, this could result in a failure.

Another limitation is that *Javadoc* needs to be present for covered code in order to get a complete comment for an integration test. When *Javadoc* is not available, an explanation of a method cannot be given. This means that a comment will be less useful when little or no *Javadoc* is available.

Chapter 6

Related Work

As said earlier, no methods have been created before for summarising integration tests. Related work can be found in methods summarising unit tests or source code.

Summarising test code Kamimura and Murphy [20] proposed a method for generating human-oriented summaries of test cases. Their goal was to help developers improve understanding unit test cases. The summaries were created by performing a static analysis on the code belonging to test cases. The result was a method that could be seen as a starting point for generating test case summaries.

Based on this method, Panichella et al. [27] proposed a new approach for summarising generated unit tests: *TestDescriber*. This was done to find out what the effect of test case summaries would be on bug fixing performance of developers. It was found that developers found twice as many bugs as usual. In addition the summaries also aided in test code comprehension. Because of these positive results, the tool proposed in this thesis was designed based on *TestDescriber*. The main difference is that this tool generates comments for human-written integration tests instead of generated unit tests. What is also new, is that existing *Javadoc* belonging to an integration test is used in the generated comments.

Summarising source code In Chapter 2 several methods have been discussed that generate summaries for source code: [6, 9, 11, 13, 14, 15, 23, 24, 25, 28, 29, 30, 33, 34, 35, 37, 38]. These can all be considered as related work. The main difference between these methods and the developed tool is that all these methods have been made to generate comments for regular source code. Also none of the methods use *Javadoc* as a resource of information.

Chapter 7

Conclusion

Comments play a vital role in the comprehension of source code. However, software developers lack a tool that creates comments for integration tests. This thesis proposes a solution. It presents a tool that automatically generates comments for integration tests. To make this possible, the main stages in the process of comment generation had to be defined. The first research question was:

RQ1: *What are the main stages in the process of comment generation?*

There are four main stages in the process of comment generation: lexical preprocessing, text retrieval, summary generation & summary aggregation. These stages were used in the design of the generation tool. *TestDescriber* was used as a basis, because it generates comment for unit tests. It also received good evaluation results in the case study. The novelty of the developed tool is that unlike existing tools, it uses *Javadoc* as a resource. A comment generated by the tool consists of a list of covered classes and methods, the coverage percentage, a description of covered methods and a description of what is tested.

An experiment was conducted to evaluate the tool. The second research question was:

RQ2: *What is the effect of generated comments for integration tests on code comprehension?*

In the experiment Java developers performed two programming tasks in which a test had to be rewritten. Results show that the comments generated by the tool can aid the code comprehension of developers. In their tasks, developers preferred to have the comments provided, and they found it easier to understand the code with them. The comments were mostly found to be of good quality: developers thought they were precise, readable and quite concise. Developers also indicated that they would use the tool more in collaborative projects, involving other developers, as opposed to individual projects.

In future work, the content of the comments can be shortened: developers suggested that the coverage information and list of classes/methods could be excluded. Further experiments should be conducted to find out if developers actually perceive this as an im-

provement. Also, a variation of the experiment could be to use a whole framework, not only code fragments. It can be interesting to compare these results with the results from this thesis. Then, since the majority of the experiment's participants had less than one year of testing experience, the experiment should be repeated with a greater variety of experience. The participants also need to be equally divided over different surveys based on their testing experience. Then the groups would be truly randomised and it can be seen if there is a difference in the experiment's outcome. Finally, it has been found that the usefulness of generated comments could be dependent on the difficulty of the code it belongs to. To confirm this, more experiments have to be done which include code fragment with different levels of difficulty.

Bibliography

[1] Thomas Ball and James R Larus. *Branch prediction for free*, volume 28. ACM, 1993.

[2] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[3] Cobertura. `http://cobertura.github.io/cobertura/`, 2015.

[4] CompileJava. `https://www.compilejava.net/`, 2016.

[5] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. On the coherence between comments and implementations in source code. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 76–83. IEEE, 2015.

[6] Brian P Eddy, Joshua A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 13–22. IEEE, 2013.

[7] James L Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8):512–521, 1982.

[8] Richard K Fjeldstad and William T Hamlen. Application program maintenance study: Report to our respondents. *Proceedings Guide*, 48, 1983.

[9] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. Tassal: Autofolding for source code summarization.

[10] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276–291, 2013.

[11] Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Pseudogen: A tool to automatically generate pseudo-code from source code. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 824–829. IEEE, 2015.

[12] Adele Goldberg. Programmer as reader. *IEEE Software*, 4(5):62, 1987.

[13] Latifa Guerrouj, David Bourque, and Peter C Rigby. Leveraging informal documentation to summarize classes and methods in context. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 639–642. IEEE, 2015.

[14] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

[15] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 35–44. IEEE, 2010.

[16] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration*. PhD thesis, University of Delaware, 2010.

[17] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242. IEEE Computer Society, 2009.

[18] JavaParser. `https://github.com/javaparser/javaparser`, 2015.

[19] JPacman. `https://github.com/SERG-Delft/jpacman-framework`, 2016.

[20] Manabu Kamimura and Gail C Murphy. Towards generating human-oriented summaries of unit test cases. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE, 2013.

[21] LanguageTool. `https://github.com/languagetool-org/languagetool/`, 2015.

[22] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[23] Paul W McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 291–294. ACM, 2014.

[24] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32. IEEE, 2013.

[25] Laura Moreno and Andrian Marcus. Jstereocode: automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 358–361. ACM, 2012.

[26] Eriko Nurvitadhi, Wing Wah Leung, and Curtis Cook. Do class comments aid java program understanding? In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, volume 1, pages T3C–13. IEEE, 2003.

[27] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. Technical report, PeerJ PrePrints, 2015.

[28] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Summarizing complex development artifacts by mining heterogeneous data. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 401–405. IEEE Press, 2015.

[29] Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 103–112. IEEE, 2011.

[30] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401. ACM, 2014.

[31] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[32] Diomidis Spinellis. *Code quality: the open source perspective*. Adobe Press, 2006.

[33] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[34] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 101–110. IEEE, 2011.

[35] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 71–80. IEEE, 2011.

[36] Ted Tenny. Program readability: Procedures versus comments. *Software Engineering, IEEE Transactions on*, 14(9):1271–1279, 1988.

[37] Elaine Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567. IEEE, 2013.

[38] Annie TT Ying and Martin P Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658. ACM, 2013.

[39] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

Appendix A

Test Cases Used in Experiment

In this appendix the test cases used in the experiment are given. In survey version A, the code fragment in Figure A.3 was used in task 1 and the code in Figure A.2 in task 2. In version B, the fragment in Figure A.1 was used in task 1 and the code in Figure A.4 in task 2.

```
private Mapparser parser;

@Before
public void setUp() {
    PacManSprites sprites = new  PacManSprites();
    parser = new MapParser(new LevelFactory(sprites, new GhostFactory(sprites)),
                                            new BoardFactory(sprites));
}

@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# ..#", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(2, 1);
    Square result = Navigation.findNearestFood(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}

```

Figure A.1: Test case *testNearestFood*.

```
private Mapparser parser;

/**
 * Test case setUp tests class(es) [PacManSprites, MapParser, LevelFactory, GhostFactory, BoardFactory].
 */
@Before
public void setUp() {
    PacManSprites sprites = new  PacManSprites();
    parser = new MapParser(new LevelFactory(sprites, new GhostFactory(sprites)),
                                            new BoardFactory(sprites));
}

/**
 * Test case testNearestFood tests class(es) [Board, Square, Navigation, Pellet]
 * and method(s) [getSquare, findNearest, parseMap, getBoard, squareAt].
 *
 * Method Navigation.findNearest [100%]: Finds the nearest unit of the given type and returns its location. This
   method will perform a breadth first search starting from the given square.
 * Method MapParser.parseMap [62%]: Parses the text representation of the board into an actual level.
   Supported characters: ' ' (space) an empty square. '#' (bracket) a wall. '.' (period) a
   square with a pellet. 'P' (capital P) a starting square for players. 'G' (capital G) a square with a
   ghost.
 * Method Board.squareAt [45%]: Returns the square at the given x,y position.
 *
 * It tests:
 * 1) whether "result" is equal to "s2".
 */
@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# ..#", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(2, 1);
    Square result = Navigation.findNearest(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}

```

Figure A.2: Test case *testNearestFood* with comments.

```java
private PacmanUI pacmanUI;
private FrameFixture window;

@Before
public void setUp() {
    JFrame frame = GuiActionRunner.execute(new GuiQuery<JFrame>() {
        protected JFrame executeInEDT() throws FactoryException {
            pacmanUI = new PacmanUI();
        pacmanUI.initialize();
        return pacmanUI;
        }
    });
    window = new FrameFixture(frame);
    window.show();
}

@Test
public void testLeftArrowButton() {
    pacmanUI.eventHandler().start();

    Tile position = pacmanUI.getGame().getPlayer().getTile();
    int expectedX = position.getX() - 1;

    int keyCodeLeftArrow = 37;
    window.pressAndReleaseKeys(keyCodeLeft);
    int newX = pacmanUI.getGame().getPlayer().getTile().getX();

    assertEquals(expectedX, newX);
}

```

Figure A.3: Test case *testLeftArrowButton*.

```java
private PacmanUI pacmanUI;
private FrameFixture window;

/**
 * Test case setUp tests class(es) [PacmanUI]
 * and method(s) [initialize].
 *
 * Method PacmanUI.initialize [100%]: Create the controllers of the UI.
 */
@Before
public void setUp() {
    JFrame frame = GuiActionRunner.execute(new GuiQuery<JFrame>() {
        protected JFrame executeInEDT() throws FactoryException {
            pacmanUI = new PacmanUI();
        pacmanUI.initialize();
        return pacmanUI;
        }
    });
    window = new FrameFixture(frame);
    window.show();
}

/**
 * Test case testLeftArrowButton tests class(es) [PacmanUI, Tile]
 * and method(s) [getGame, getPlayer, getTile, getX, eventHandler, start].
 *
 * Method PacmanUI.eventHandler [100%]: Returns the mapping between keyboard events and model events.
 * Method PacmanInteraction.start [42%]: Actually start the the controllers, and show the UI.
 *
 * It tests:
 * 1) whether "newX" is equal to "expectedX".
 */
@Test
public void testLeftArrowButton() {
    pacmanUI.eventHandler().start();

    Tile position = pacmanUI.getGame().getPlayer().getTile();
    int expectedX = position.getX() - 1;

    int keyCodeLeftArrow = 37;
    window.pressAndReleaseKeys(keyCodeLeft);
    int newX = pacmanUI.getGame().getPlayer().getTile().getX();

    assertEquals(expectedX, newX);
}

```

Figure A.4: Test case *testLeftArrowButton* with comments.

Appendix B

Survey - Version A

In this appendix version A of the survey is shown. The survey starts on the next page.

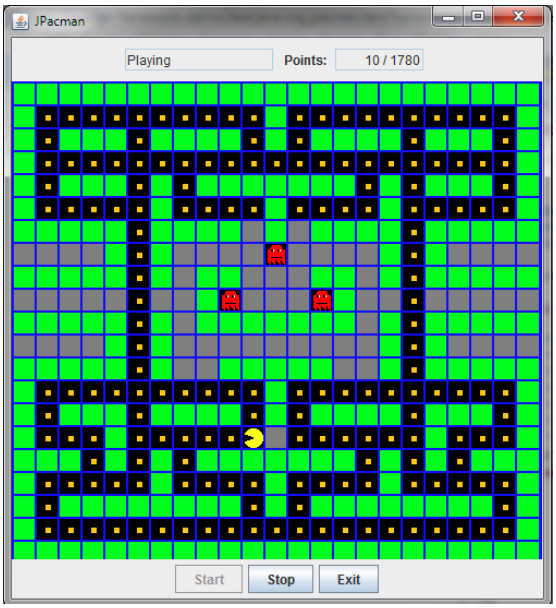
Survey on Understanding Tests

In this survey code fragments are used based on the JPacman Framework. This is a Pacman-like game used for teaching software testing and is written in Java. The goal of this survey is to find out what the effect is of generated comments on understanding integration tests.

This survey consists of three parts. In the first two parts a small programming exercise needs to be performed and some questions will be posed regarding this exercise. In the final part a few background questions will be asked.

Thank you for participating!

*Required



Part I

You are asked to make small changes to a code fragment. Since the styling of this survey is not appropriate for source code, please copy paste the code fragment to https://www.compilejava.net/ and perform the programming task there. When you are finished, paste your solution in the survey. If you want, you can also use a program like Notepad instead of the website.

1. **Please enter the current time before you start:** *

   Example: 8.30 a.m.

Programming Task

In the game, the player can press the arrow button on a keyboard to move Pacman. It is possible to test whether the position of Pacman is updated after pressing these buttons.

Change the test "testLeftArrowButton" so that the following is tested:

"The position of the player changes correctly when the left arrow button is pressed twice."

Please note that you do not have to compile the code.

Code Fragment

```
private PacmanUI pacmanUI;
private FrameFixture window;

@Before
public void setUp() {
    JFrame frame = GuiActionRunner.execute(new GuiQuery<JFrame>() {
        protected JFrame executeInEDT() throws FactoryException {
            pacmanUI = new PacmanUI();
            pacmanUI.initialize();
            return pacmanUI;
        }
    });
    window = new FrameFixture(frame);
    window.show();
}

@Test
public void testLeftArrowButton() {
    pacmanUI.eventHandler().start();

    Tile position = pacmanUI.getGame().getPlayer().getTile();
    int expectedX = position.getX() - 1;

    int keyCodeLeftArrow = 37;
    window.pressAndReleaseKeys(keyCodeLeft);
    int newX = pacmanUI.getGame().getPlayer().getTile().getX();

    assertEquals(expectedX, newX);
}
```

Solution

2. **Paste your solution here:** *
   You don't have to worry about the styling

3. **Please enter the current time again when you finished the task:** *
   Example: 8.30 a.m.

Statements

Please enter whether you agree or disagree with the written statements.

4. **"I found the task easy"** *
   Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

5. **"To understand the fragment, I missed the full source code of JPacman"** *
   Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

Question

6. **Which steps did you take to complete the task?** *

.....

.....

.....

.....

Part II

You are asked to make small changes to a code fragment. This time an automatically generated comment is included in the code fragment. Please use https://www.compilejava.net/ again to perform the programming task (also copy the comment). When you are finished, paste your solution in the survey. Again, you can also use a program like Notepad instead of the website.

The generated comment is structured as follows:
First the classes and methods that are tested are listed. Then, for all methods it is given to which class they belong and what the coverage percentage of these methods is. In addition the Javadoc that belongs to the methods is given as a description. Finally, it is described what is tested by giving a description of the assert(s).

The comment only shows classes and methods that belong to the project itself, so for instance no Java or external libraries. It also omits a description for get and set methods, since these should be clear from the method name. When there are no methods and/or asserts in a test case, these parts are not included in the comment.

7. **Please enter the current time before you start:** *

   .....
   Example: 8.30 a.m.

Programming Task

In the game, Pacman can eat food. Food is visible on the board as a dot: ".". It is possible to test what the nearest food unit is to a certain square on the board.

Change the test "testNearestFood" so that the following is tested:

"The nearest food to s1 is 2 squares away."

Please note that you do not have to compile the code.

Code Fragment

```
private Mapparser parser;

/**
 * Test case setUp tests class(es) [PacManSprites, MapParser, LevelFactory, GhostFactory,
BoardFactory].
 */
@Before
public void setUp() {
```

```
    PacManSprites sprites = new  PacManSprites();
    parser = new MapParser(new LevelFactory(sprites, new GhostFactory(sprites)),
                                        new BoardFactory(sprites));
}

/**
 * Test case testNearestFood tests class(es) [Board, Square, Navigation, Pellet]
 * and method(s) [getSquare, findNearest, parseMap, getBoard, squareAt].
 *
 * Method Navigation.findNearest [100%]: Finds the nearest unit of the given type and returns its location.
This
   method will perform a breadth first search starting from the given square.
 * Method MapParser.parseMap [62%]: Parses the text representation of the board into an actual level.
   Supported characters: ' ' (space) an empty square. '#' (bracket) a wall. '.' (period) a
   square with a pellet. 'P' (capital P) a starting square for players. 'G' (capital G) a square with a
   ghost.
 * Method Board.squareAt [45%]: Returns the square at the given x,y position.
 *
 * It tests:
 * 1) whether "result" is equal to "s2".
 */
@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# ..#", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(2, 1);
    Square result = Navigation.findNearest(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}
```

Solution

8. **Paste your solution here:** *
   You don't have to worry about the styling

9. **Please enter the current time again when you finished the task:** *
   Example: 8.30 a.m.

Statements

Please enter whether you agree or disagree with the written statements.

10. **"I found the task easy"** *
    Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

11. **"The generated comment made it easier for me to understand the code"**
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

12. **"To understand the fragment, I missed the full source code of JPacman"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

13. **"The programming task in Part I would have been easier if a similar comment was provided there"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

14. **"For projects I work on by myself, I would use this functionality of automatically generating comments for tests"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

15. **"For projects I work on with multiple people, I would use this functionality of automatically generating comments for tests"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

Questions

16. **Which steps did you take to complete the task?** *

.....

.....

.....

.....

17. **Did you use the comment for performing the task?** *
If not, skip next question
Mark only one oval.

◯ Yes
◯ No

18. **Which part(s) did you use?**
Check all that apply
Tick all that apply.

- [ ] List of classes
- [ ] List of methods
- [ ] Coverage percentage
- [ ] Description of methods
- [ ] Description of what is tested

19. **Is the comment missing important information?** *
The comment ...
Mark only one oval.

- ◯ is not missing any information
- ◯ is missing some information
- ◯ is missing some very important information

20. **Is the comment containing redundant information?** *
The comment ...
Mark only one oval.

- ◯ has no unnecessary information
- ◯ has some unnecessary information
- ◯ has a lot of unnecessary information

21. **Is the comment readable and understandable?** *
The comment ...
Mark only one oval.

- ◯ is easy to read and understand
- ◯ is somewhat readable and understandable
- ◯ is hard to read and understand

22. **If you could change the comment, what would you change and why?** *

.....

.....

.....

.....

Part III

23. **What are you studying / What is your profession?**
If you are a student, please also mention whether your study is BSc/MSc/PhD

24. **How many years of experience do you have in Java programming?** *
Mark only one oval.

- ◯ < 1
- ◯ 1 - 2
- ◯ 3 - 6
- ◯ 7 - 10
- ◯ > 10

25. **How many years of experience do you have in Java testing?** *
Mark only one oval.

- ◯ < 1
- ◯ 1 - 2
- ◯ 3 - 6
- ◯ 7 - 10
- ◯ > 10

26. **If you have any comments you would like to share, you can write them down here**

.....

.....

.....

.....

Appendix C

Survey - Version B

In this appendix version B of the survey is shown. The survey starts on the next page.

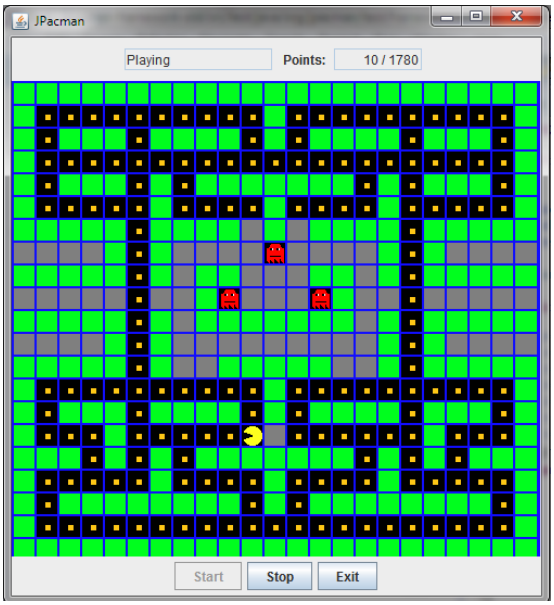
Survey on Understanding Tests

In this survey code fragments are used based on the JPacman Framework. This is a Pacman-like game used for teaching software testing and is written in Java. The goal of this survey is to find out what the effect is of generated comments on understanding integration tests.

This survey consists of three parts. In the first two parts a small programming exercise needs to be performed and some questions will be posed regarding this exercise. In the final part a few background questions will be asked.

Thank you for participating!

*Required



Part I

You are asked to make small changes to a code fragment. Since the styling of this survey is not appropriate for source code, please copy paste the code fragment to https://www.compilejava.net/ and perform the programming task there. When you are finished, paste your solution in the survey. If you want, you can also use a program like Notepad instead of the website.

1. **Please enter the current time before you start:** *

   Example: 8.30 a.m.

Programming Task

In the game, Pacman can eat food. Food is visible on the board as a dot: ".". It is possible to test what the nearest food unit is to a certain square on the board.

Change the test "testNearestFood" so that the following is tested:

"The nearest food to s1 is 2 squares away."

Please note that you do not have to compile the code.

Code Fragment

```
private Mapparser parser;

@Before
public void setUp() {
    PacManSprites sprites = new  PacManSprites();
    parser = new MapParser(new LevelFactory(sprites, new GhostFactory(sprites)),
                                        new BoardFactory(sprites));
}

@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# ..#", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(2, 1);
    Square result = Navigation.findNearestFood(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}
```

Solution

2. **Paste your solution here:** *
   You don't have to worry about the styling

3. **Please enter the current time again when you finished the task:** *
   Example: 8.30 a.m.

Statements

Please enter whether you agree or disagree with the written statements.

4. **"I found the task easy"** *
   Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

5. **"To understand the fragment, I missed the full source code of JPacman"** *
   Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

Question

6. **Which steps did you take to complete the task?** *

.....

.....

.....

.....

Part II

You are asked to make small changes to a code fragment. This time an automatically generated comment is included in the code fragment. Please use https://www.compilejava.net/ again to perform the programming task (also copy the comment). When you are finished, paste your solution in the survey. Again, you can also use a program like Notepad instead of the website.

The generated comment is structured as follows:
First the classes and methods that are tested are listed. Then, for all methods it is given to which class they belong and what the coverage percentage of these methods is. In addition the Javadoc that belongs to the methods is given as a description. Finally, it is described what is tested by giving a description of the assert(s).

The comment only shows classes and methods that belong to the project itself, so for instance no Java or external libraries. It also omits a description for get and set methods, since these should be clear from the method name. When there are no methods and/or asserts in a test case, these parts are not included in the comment.

7. **Please enter the current time before you start:** *

Example: 8.30 a.m.

Programming Task

n the game, the player can press the arrow button on a keyboard to move Pacman. It is possible to test whether the position of Pacman is updated after pressing these buttons.

Change the test "testLeftArrowButton" so that the following is tested:

"The position of the player changes correctly when the left arrow button is pressed twice."

Please note that you do not have to compile the code.

Code Fragment

```
private PacmanUI pacmanUI;
private FrameFixture window;

/**
 * Test case setUp tests class(es) [PacmanUI]
 * and method(s) [initialize].
 *
 * Method PacmanUI.initialize [100%]: Create the controllers of the UI.
 */
@Before
public void setUp() {
    JFrame frame = GuiActionRunner.execute(new GuiQuery<JFrame>() {
        protected JFrame executeInEDT() throws FactoryException {
            pacmanUI = new PacmanUI();
            pacmanUI.initialize();
            return pacmanUI;
        }
    });
    window = new FrameFixture(frame);
    window.show();
}
```

```
/**
 * Test case testLeftArrowButton tests class(es) [PacmanUI, Tile]
 * and method(s) [getGame, getPlayer, getTile, getX, eventHandler, start].
 *
 * Method PacmanUI.eventHandler [100%]: Returns the mapping between keyboard events and model
events.
 * Method PacmanInteraction.start [42%]: Actually start the the controllers, and show the UI.
 *
 * It tests:
 * 1) whether "newX" is equal to "expectedX".
 */
@Test
public void testLeftArrowButton() {
    pacmanUI.eventHandler().start();

    Tile position = pacmanUI.getGame().getPlayer().getTile();
    int expectedX = position.getX() - 1;

    int keyCodeLeftArrow = 37;
    window.pressAndReleaseKeys(keyCodeLeft);
    int newX = pacmanUI.getGame().getPlayer().getTile().getX();

    assertEquals(expectedX, newX);
}
```

Solution

8. **Paste your solution here:** *
   You don't have to worry about the styling

   .....

   .....

   .....

9. **Please enter the current time again when you finished the task:** *

   .....

   Example: 8.30 a.m.

Statements

Please enter whether you agree or disagree with the written statements.

10. **"I found the task easy"** *
    Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

11. **"The generated comment made it easier for me to understand the code"**
    Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

12. **"To understand the fragment, I missed the full source code of JPacman"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

13. **"The programming task in Part I would have been easier if a similar comment was provided there"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

14. **"For projects I work on by myself, I would use this functionality of automatically generating comments for tests"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

15. **"For projects I work on with multiple people, I would use this functionality of automatically generating comments for tests"** *
Mark only one oval.

	1	2	3	4	5	
Strongly agree	◯	◯	◯	◯	◯	Strongly disagree

Questions

16. **Which steps did you take to complete the task?** *

.....

.....

.....

.....

17. **Did you use the comment for performing the task?** *
If not, skip next question
Mark only one oval.

◯ Yes
◯ No

18. **Which part(s) did you use?**
Check all that apply
Tick all that apply.

- [ ] List of classes
- [ ] List of methods
- [ ] Coverage percentage
- [ ] Description of methods
- [ ] Description of what is tested

19. **Is the comment missing important information?** *
The comment ...
Mark only one oval.

- ◯ is not missing any information
- ◯ is missing some information
- ◯ is missing some very important information

20. **Is the comment containing redundant information?** *
The comment ...
Mark only one oval.

- ◯ has no unnecessary information
- ◯ has some unnecessary information
- ◯ has a lot of unnecessary information

21. **Is the comment readable and understandable?** *
The comment ...
Mark only one oval.

- ◯ is easy to read and understand
- ◯ is somewhat readable and understandable
- ◯ is hard to read and understand

22. **If you could change the comment, what would you change and why?** *

.....

.....

.....

.....

Part III

23. **What are you studying / What is your profession?**
If you are a student, please also mention whether your study is BSc/MSc/PhD

24. **How many years of experience do you have in Java programming?** *
Mark only one oval.

- ◯ < 1
- ◯ 1 - 2
- ◯ 3 - 6
- ◯ 7 - 10
- ◯ > 10

25. **How many years of experience do you have in Java testing?** *
Mark only one oval.

- ◯ < 1
- ◯ 1 - 2
- ◯ 3 - 6
- ◯ 7 - 10
- ◯ > 10

26. **If you have any comments you would like to share, you can write them down here**

.....

.....

.....

.....

Appendix D

Solutions to Programming Tasks

For the programming task including *testNearestFood* multiple correct solutions could be found. Two examples that were submitted by participants as solutions are given in Figure D.1 and D.2.

```
@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "#  .#", "#####")).getBoard();
    Square s1 = b.squareAt(1, 1);
    Square s2 = b.squareAt(3, 1);
    Square result = Navigation.findNearestFood(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}
```

Figure D.1: Example of a comment generated for an integration test.

```
@Test
public void testNearestFood() {
    Board b = parser.parseMap(Lists.newArrayList("#####", "# ..#", "#####")).getBoard();
    Square s1 = b.squareAt(0, 1);
    Square s2 = b.squareAt(2, 1);
    Square result = Navigation.findNearestFood(Pellet.class, s1).getSquare();
    assertEquals(s2, result);
}
```

Figure D.2: Example of a comment generated for an integration test.

For the programming task including *testLeftArrowButton*, only one possible solution could be given. The solution is shown in Figure D.3.

```
@Test
public void testLeftArrowButton() {
    pacmanUI.eventHandler().start();

    Tile position = pacmanUI.getGame().getPlayer().getTile();
    int expectedX = position.getX() - 2;

    int keyCodeLeftArrow = 37;
    window.pressAndReleaseKeys(keyCodeLeft);
    window.pressAndReleaseKeys(keyCodeLeft);
    int newX = pacmanUI.getGame().getPlayer().getTile().getX();

    assertEquals(expectedX, newX);
}

```

Figure D.3: Example of a comment generated for an integration test.