Code-Comment Inconsistency Detection in Java Projects with Focus on Ranges

Michael Cuénez

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 26, 2017

Michael Cuénez

Contents

Abstract

Documenting source code is a routine that occurs almost on a daily basis for any programmer, regardless of the programming field. However, due to time limits, too little motivation and possibly other factors, often times only little effort is made to properly document any implemented feature. Especially, if the functionality of the code was altered. This can lead to bugs in source code, written by a programmer who relies on third party libraries and, therefore, their documentation.

Automatically analyzing such documentation – usually written in form of comments – may help detect code-comment inconsistencies (i.e., the documentation implies different behavior than the source code). By doing so, the inconsistencies can be fixed and any programmer using a third party library can rely on the documentation. Therefore, less bugs will be introduced because of *inadequate* comments. For this reason, a project was implemented and evaluated, in order to determine the feasibility of code-comment inconsistency detection. The implemented approach is based on an existing project and has been extended to focus on ranges, rather than null-related information (which is the focus of the original approach).

The evaluation was conducted manually, with the help of automatically created reports. Ultimately, this lead to an *overall* accuracy of approximately 98% for the inference algorithm. However, as expected, due to rather low metrics, the code-comment inconsistency detection is only about 56%. Additionally, the results of the evaluation are compared to other projects and discussed extensively.

Kurzfassung

Jeder Programmierer – unabhängig vom Bereich der Spezialisierung – ist zwangsläufig (zumindest teilweise) damit beschäftigt, Programmcode zu kommentieren und somit zu dokumentieren. Allerdings wird diese Aufgabe oft nur unvollständig umgesetzt. Dies begründet sich durch viele verschiedene Faktoren – unter anderem, durch zu wenig Zeit und Motivation. Besonders dann, wenn sich die Funktionalität des Codes ändert, wird das zugehörige Kommentar oft nicht auf den neuesten Stand gebracht. Programmierer, die andere Projekte verwenden, um das Rad nicht neu zu erfinden, sind allerdings auf die Dokumentation ebendieser Projekte angewiesen. Ist diese Dokumentation nicht korrekt, kann dies zu Fehlern im Code führen. Es ist daher wünschenswert, die vorhandene Dokumentation (in Form von Kommentaren) automatisch zu analysieren, um an Hand dieser Analyse festzustellen, ob sie mit der Funktionalität des Codes übereinstimmt.

Sollten solche sogenannten Inkonsistenzen gefunden werden, können die verantwortlichen Programmierer die Dokumentation (bzw. das Kommentar) berichtigen. Dies führt dazu, dass sich andere Programmierer (welche externe Projekte verwenden) auf die Dokumentation verlassen können. Somit wird Fehlern im Programmcode vorgebeugt. Aus diesem Grund wurde ein Projekt umgesetzt, welches Kommentare analysiert und entsprechende Metriken und Statistiken aufstellt. Basierend auf einem Ansatz für nullbezogene Inkonsistenzen, wurde das Projekt erweitert, um Wertebereiche analysieren zu können.

Die Evaluierung des Projekts erfolgte manuell, durch das Analysieren von automatisch generierten Berichten. Der Algorithmus für das Ableiten von Spezifikationen erreichte dabei eine *durchschnittliche* Genauigkeit von ungefähr 98%. Im Vergleich dazu wurde für das automatische Erkennen von Inkonsistenzen eine eher geringe Genauigkeit von 56% festgestellt. Die resultierenden Werte wurden ausführlich besprochen und zusätzlich mit Arbeiten im selben Bereich verglichen.

Chapter 1

Introduction

This chapter focuses on a concise introduction, by explaining the relevance of the topic, the problem which originated in insufficient documentation and the structure of this Master's thesis.

1.1 Problem Statement

Any software producing company has to ensure code quality to some degree. Of course, this degree is supposed to be as high as possible. Only by keeping high quality the production costs can be hold at a minimum – as already cited by [9]:

> Elshoff and Marcotty states that the comments, as well as the structure of the source code are good aids for understanding programs and therefore reduce maintenance costs [...].

However, this can be a struggle-some task to fulfill without any tools at hand. Even with the current tools in use, such as JUnit and the likes, it is a – at least – time-consuming task. Furthermore, developers tend to become discouraged after hours of searching for an error – where the search-time sometimes would stretch over days. This can introduce errors based on sloppiness, due to unmotivated programmers.

When programming, exceptions will eventually arise (if exceptions are supported by the programming language in use). While exceptions are useful, it is hard to handle them correctly if they are not documented. Imagine a programmer who is relying on a third-party API (such as Apache Common Collections[1]). In order for the programmer to know how to *properly* use any method within the API, she will have to read the documentation at least once. If the documentation is non-existent, outdated (i.e., incorrect, due to a change of code) or simply inaccurate the programmer will have

[1] https://commons.apache.org/proper/commons-collections/

a hard time finding any error that causes an undocumented exception. Of course, *checked exceptions*[2] will cause a compile-time error. Thus, using an IDE should help mark such exceptions as error before the developer could execute the code. By doing so, this types of exceptions will be handled in any case. Additionally, this type of exception is documented in the top-level code (`throws` clause) and does not necessarily have to be documented in the Javadoc – though it should be, for the sake of completeness. On the contrary, *unchecked exceptions* (i.e., `RuntimeException`, `Error`, and their subclasses) do not cause compile-time errors. Thus, the code can be executed without handling the exceptions. Oracle explains that the reason for this is, because the user of an API does not necessarily know how to handle the exception correctly, however, the developer of the API does [18].

Without proper documentation, the user might encounter a bug, due to an unexpected exception. In order to avoid bugs originated in inadequate comments (see section 2.3), the developer would need to take the time to check every Javadoc comment. This is a time consuming task. Furthermore, due to laziness, deadlines, obliviousness and other factors, inadequate Javadoc comments might increase rapidly during the life-cycle of an API. Leading, again, to bugs in the user's code.

1.2 Relevance

Detecting code-comment inconsistencies can prevent bugs – by showing outdated or wrongly stated comments. These detected comments can then be removed or updated. Once this is done, other programmers can rely on the information found within this documentation. The importance of updating comments was also concluded by Malik *et al.* [9].

If the detected comment is not an inadequate one (see section 2.3) then there is a chance that a bug was detected (i.e., a logical error within the code itself). Also, as other work already suggests (see chapter 3), code-comment inconsistency detection does help to improve code. It is noteworthy that these approaches often found bugs rather than an inadequate comment. Thus, having a tool which provides a (semi-)automated code-comment inconsistency detection is useful, considering the problem stated in section 1.1. This is also discussed in chapter 6.

1.3 Structure

After this chapter's brief introduction, chapter 2 continues with context-dependent explanations of relevant terms and introduces the libraries used or currently in use. This is necessary, in order to comprehend the chap-

[2] https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

ters following afterwards. Next, to be able to compare this work with existing work, some state-of-the-art approaches are explained in chapter 3. Also, related work is listed – which is thought to be not as closely related as state-of-the-art work. Afterwards, chapter 4 highlights the difference of the basis implementation compared to the original approach, which was required to be re-implemented in order to improve it. This improvement is described in the very same chapter by giving an overview of the approach in use and utilizing examples to clarify the intention of this thesis' project. Furthermore, to provide metrics and a proof of concept, an evaluation was concluded and is elucidated in detail in chapter 5. Not only does it describe the used measures, but it also shows the final results and their comparison with existing work. Last but not least, the final chapter (see chapter 6) concludes the insights and results of this thesis and provides ideas for future work. Additionally, an extensive discussion of the relevance of the project and its fields of application can be found within this chapter. Last but not least, a few appendixes are provided in order to complement this Master's thesis content.

Chapter 2

Concepts

This chapter discusses the terms used in this thesis and explains the concrete meaning in this context.

2.1 Javadoc

A Javadoc comment is a *header comment* as defined by Jiang and Hassan in their study of PostgreSQL[1] comments [5]. Additionally, Javadoc comments consist of natural language and special keywords, namely *tags*. Where each tag is related to a certain part of the method the Javadoc comment describes. If a tag is missing, even though the corresponding code element is available then this comment is most likely an *inadequate comment* (see section 2.3).

2.2 Range

As the focus of the analysis lies on ranges, it is important to explicitly define how a range is denoted. A range can hold any number n_{r}, where $n_{\mathrm{r}} \in \mathbb{R}$. For example, a range for an index is usually defined as $0, \ldots, l$ – where l is the length of the object that accesses any element accordingly. However, there are multiple ways of denoting a range, when it comes to informal information (i.e., natural language – see section 2.6). Examples of how such a sentence might be written – explaining why an exception is thrown – are shown in table 2.1.

2.3 Inadequate Comment

An inadequate comment in this context is defined as either *outdated*, *inaccurate* or *incomplete*. Where outdated comments are usually created through code refactoring. When code is updated (as such that the functionality

[1] https://www.postgresql.org/

Table 2.1: Examples of extracted ranges corresponding to their textual counterpart. The variable `len` refers to the length of the object accessing any element accordingly. These kinds of text, usually, appear after an `@throws` tag.

Informal Information	*Range*
if the index is out of bounds	`index < 0 && index >= len`
if the index is negative	`index < 0`

changes), but the comment is not then an outdated comment is created. An inaccurate comment, however, holds text without proper specifications.

For example, a comment defined as `@param index the index` does only give a *hint* about the specification of the index (i.e., its permitted range). This is, because absolutely no information about the range is given within the parameter's description. Only when a proper `@throws` tag is available, the range extraction might still be possible (refer to table 2.1 for some examples).

Furthermore, an incomplete comment is lacking information. Using program 4.1 as an example for a proper comment, imagine one of the tags listed in the Javadoc was missing. In this case the comment would be incomplete. Finally, auto-generated comments (i.e., comments generated by using a shortcut of the IDE in use) that are not updated hold no information at all and should be avoided.

2.4 Code-Comment Inconsistency

A code-comment inconsistency is an unwanted difference between a comment's description and the functionality of the code it describes. Introducing such inconsistencies to the user might cause a misinterpretation of the code's functionality, thus, leading to possible errors or bugs [13]. In order to detect such code-comment inconsistencies, the functionality of the code has to be inferred (using the Javadoc comments) and compared to the actual behavior of the code. Any unexpected behavior is treated as an inconsistency. However, if the comment appears to be correct even though a code-comment inconsistency was detected, an implementation-wise bug was probably found. While this would be the best case scenario, it is also possible that a *false positive* was detected.

2.5 Formal Information

Formal information is, basically, a rule-set used to generate *languages* as Harrison explains [3]:

> Formal theory concerns itself with sets of strings called ”languages” and different mechanisms for generating and recognizing them. Certain finitary processes for generating these sets are called ”grammars”.

Harrison also explains how formal information is defined. Interpreting his explanation, an example for the English language can be conducted. An *alphabet* Σ is a non-empty, finite set of elements. Let $\Sigma = \{a, b, c, \ldots, z\}$. Letters, words and sentences can be created (so called Σ-sequences), when using Σ in combination with *grammars*.

2.6 Informal Information

Based on the definition of formal information in section 2.5, natural language (used as input in the project) is categorized as informal information. This informal information will be converted into code (an example is given in table 2.1), in order to check the code’s functionality.

2.7 Defensive Programming

Defensive programming is the type of programming where every parameter is always checked, even if the method call before validated its integrity already. This ensures that any given, invalid parameter is handled correctly in any existing method. However, it clutters the code tremendously, which is why another technique called *design by contract* (see section 2.8) should be preferred [4].

2.8 Design by Contract

The purpose of design by contract, basically, is to ensure that fundamental specifications are stated closely to the code [4]. Furthermore, Meyer states:

> One of the principles of design by contract, [...], is that any software element that has such a fundamental constraint should state it explicitly, as part of a mechanism present in the language.

For Java, this means a Javadoc comment is probably the best place to describe any specifications. Especially, since a user of any API will most likely read such comments and infer the proper use of the methods, based on the comments. Of course, it is possible to use assertions within the code (i.e., `assert condition : "Error message";`), in order to both have a run-time check (if assertions are enabled) and a proper documentation of specifications. However, the end-user of an API might use the documentation created from the Javadoc comments instead of the source code directly – at least in the beginning. Thus, documenting specifications in Javadoc comments is essential, especially if the source code is not available.

2.8.1 Precondition

When further reading the work of [4], it get's clear that preconditions are usually obligations *for the callee* (i.e., the user of a service). If, and only if, these conditions are fulfilled, the callee can expect the *benefits* denoted in the contract. When programming, such a precondition is the use of a *correct* value, since using an invalid value will result in a compile- or runtime error.

For example, if the callee uses a method provided by the service to retrieve an object within a list (e.g., `getObject(index);`) then the precondition could be that only positive integer values are valid. Usually, a proper range is – implicitly or explicitly – given, such as `index > 0 && index < list.size`. If this precondition is fulfilled by the callee, then the result will be as expected, according to the contract (e.g., an object will be returned).

2.8.2 Postcondition

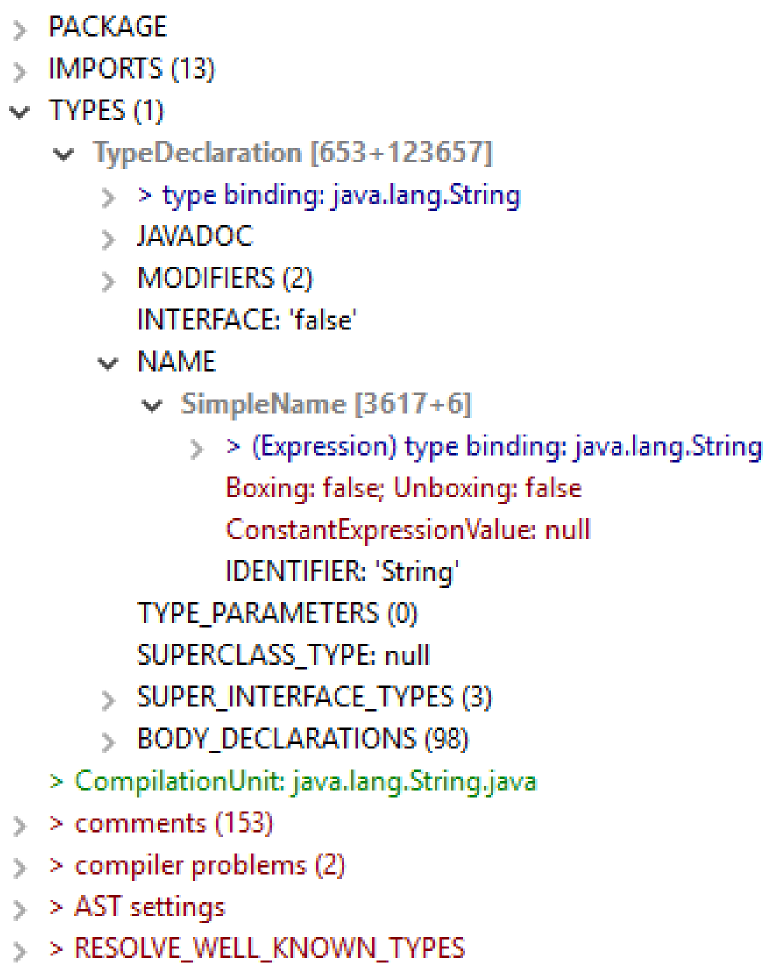
On the contrary to preconditions, postconditions are the obligations *for a service* (used by the callee). These obligations have to be fulfilled if, and only if, all preconditions are satisfied.

For example, calling the `getObject(index)` method (see section 2.8.1) with an invalid value (e.g., `index < 0 || index >= list.size`) will lead to an error – usually some kind of `IndexOutOfBoundsException`. However, if the given value is valid and the service fails to fulfill its obligations denoted in the contract then the service has to improve its algorithm [4].

2.9 Wrapper Class

A wrapper class is merely a class that *wraps* around another class. Depending on the use case, this may have different advantages.

For example, Java uses wrapper classes for its primitive types (e.g., the integer primitive `int`) [19]. However, in the project of this Master's thesis the main advantage is a significant reduction of unnecessary test creation – due to only relevant methods of the API's classes being wrapped. Therefore, Randoop (see section 4.1.2) will not check irrelevant methods (which would happen without wrapper classes).



```
> PACKAGE
> IMPORTS (13)
v TYPES (1)
    v TypeDeclaration [653+123657]
        > > type binding: java.lang.String
        > JAVADOC
        > MODIFIERS (2)
          INTERFACE: 'false'
        v NAME
            v SimpleName [3617+6]
                > > (Expression) type binding: java.lang.String
                  Boxing: false; Unboxing: false
                  ConstantExpressionValue: null
                  IDENTIFIER: 'String'
          TYPE_PARAMETERS (0)
          SUPERCLASS_TYPE: null
        > SUPER_INTERFACE_TYPES (3)
        > BODY_DECLARATIONS (98)
  > CompilationUnit: java.lang.String.java
> > comments (153)
> > compiler problems (2)
> > AST settings
> > RESOLVE_WELL_KNOWN_TYPES
```

Figure 2.1: The AST View for the `java.lang.String` class. Note that it is not fully extended, as it would simply take too much space.

2.10 Abstract Syntax Tree

An *Abstract Syntax Tree* (AST) is a model created from source code. It gives access to code and comment information, in such that certain data structures are used to represent any part of the source code (e.g. a method declaration, Javadoc comments, . . .). For this thesis' project the *JDT AST*[2] is used in combination with Eclipse's *AST View*[3] (which visualizes the AST model of a Java file). An example of the AST View's visualization is shown in figure 2.1.

[2]http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/AST.html

[3]http://www.eclipse.org/jdt/ui/astview/

Chapter 3

State of the Art

This chapter details with state-of-the-art work and their approaches of code-comment inconsistency detection.

3.1 Code-Comment Analysis

According to [12], Tan *et al.* (not to be confused with Tan *et al.* in [13]) are the pioneers in the field of code-comment analysis, with the tool *iComment*. It is emphasized that no solution existed beforehand [12]:

> This paper takes the *first* step in automatically analyzing comments written in natural language to extract implicit program rules [...].

Furthermore, *iComment* uses different techniques (e.g., *Natural Language Processing*) and focuses on another domain (i.e., C/C++).

3.1.1 Approach

As the domain varies and does not have Javadoc comments (see section 2.1), it is necessary to detect *rule containing comments* [12]. This is done using Natural Language Processing (NLP) and other techniques. Figure 3.1 shows the complete process of *iComment* (including optional steps).

Every step within the dotted rectangle is optional and might be desired by the user. This is, because the *in-house training* (see figure 3.1) was already done once and may or may not suffice. In any case, once the optional phase is completed, the *topic keywords*, used as part of the input for the *TR-Comment Extractor*, are given by the user directly. Based on the model used as input for the *Rule Generator*, certain rules are conducted, which will then be used to check for code-comment inconsistencies. The results are error reports and the generated rules themselves. Further details can be found in [12].

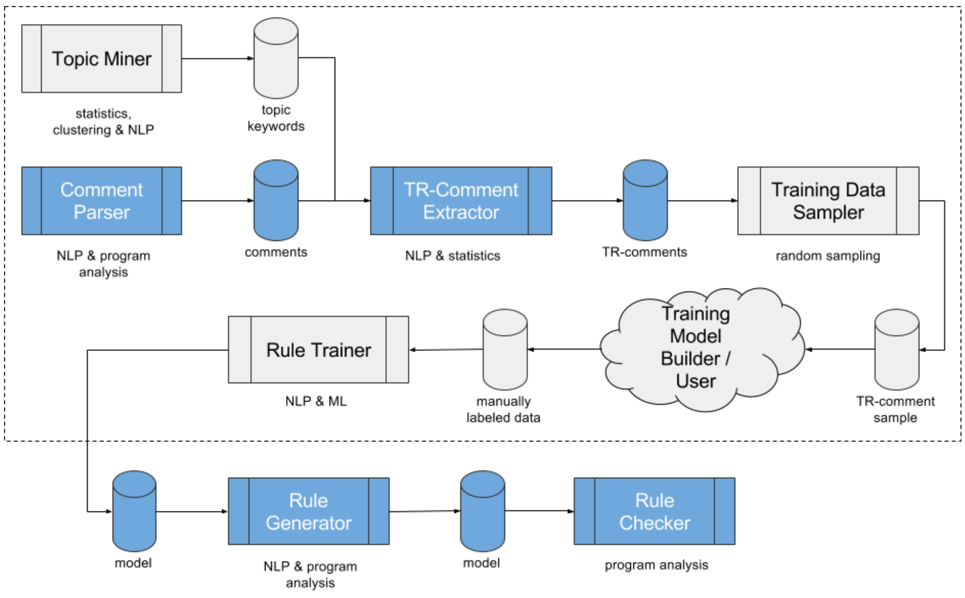


Figure 3.1: Every element within the dotted rectangle is an optional step, referred to as *In-House Training / Program-Specific Training*. However, every colored element is a fixed part of the process that occurs after all optional steps (referred to as *Comment Analysis & Checking*). Note that this overview is taken from [12].

3.2 Interrupt Related Annotations

In addition to *iComment* (see section 3.1), Tan *et al.* further developed *aComment* [11]. One of the differences is that the new approach also leverages code, instead of comments only. Furthermore, annotations are used in order to detect *interrupt related bugs* [11]:

> As we are concerned with the OS synchronization in the special interrupt context, we design annotations in the following format: @IRQ(Precondition, Postcondition), where Precondition and Postcondition can have one of the 4 values, i.e., 0, 1, X and P.

Examples of how a concrete annotation may look like within source code and the meaning of these four values are available in [11].

3.2.1 Approach

The approach of *aComment* utilizes the *Comment Parser* of *iComment* (see figure 3.1). However, it is altered in order to properly extract comments that contain interrupt-related preconditions. As figure 3.2 indicates, heuristics are used to determine whether the comment is relevant. After this step is

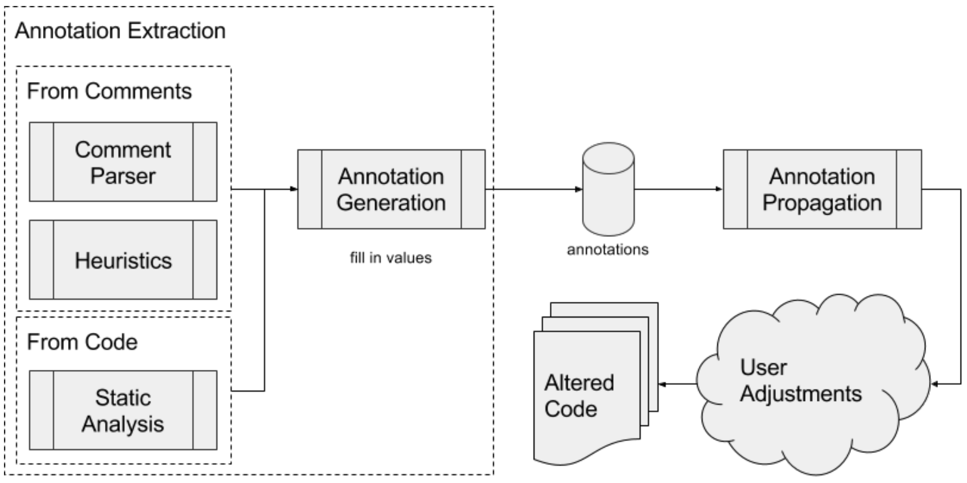


Figure 3.2: An overview of the approach used in the *aComment* tool.

completed – or maybe simultaneously (depending on the implementation) – the code is statically analyzed in order to find further preconditions. All preconditions are then used to generate abstract annotations (i.e., the values of the annotations are not known yet). In order to propagate these values to the annotations, the code is traversed in a *bottom-up fashion.* Once this is done, a report is created – with the found bugs ranked by various *confidence values* (e.g., *seed annotation confidence* and *violation confidence* [11]). This report can then be adjusted by the user – in a sense that the user may change proposed annotation values, for example. After the user's adjustments are completed, the code is altered to contain these annotations (as additional comment, for backwards compatibility).

3.3 Exception Testing

The work of Tan *et al.* regarding exception testing is extensively explained in [13]. Even though it was a first step in the very direction of Javadoc comment analysis, it leaves a lot of room for enhancement[1]. Which is why a re-implemented version is used as the basis of this work.

3.3.1 Approach

Two different tools are used in Tan's approach, namely *Doclet* and *Randoop* (see subsections 4.1.1 and 4.1.2, respectively). The complete approach is shown in figure 3.3 (for a detailed explanation see [13]). While Doclet is used to parse the Javadoc comments, Randoop is extended (namely, *@Ran-*

[1]As of October 2016 the tool *@tComment* is publicly available on GitHub [24]

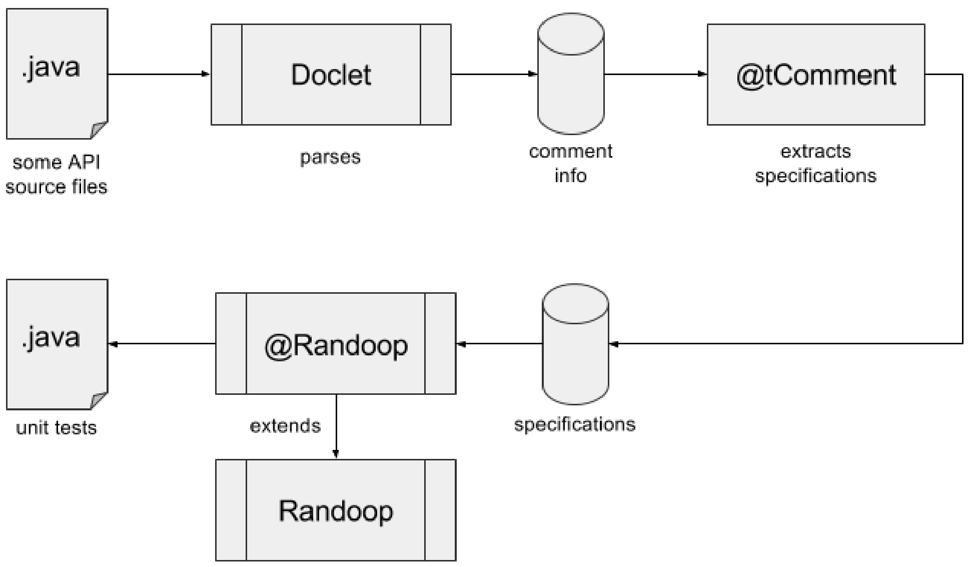


Figure 3.3: An overview of the architecture of the approach by [13].

Program 3.1: This example is taken from [13] (part of *Figure 2*). It shows the method under test, while program 3.2 shows the resulting JUnit test.

```
/**
 * ...
 * @param map the map to synchronize, must not be null
 * @return a synchronized map backed by the given map
 * @throws IllegalArgumentException if the map is null
 */
static Map synchronizedMap(Map map){
  // Omitted...
}
```

doop) to gain fine-grained control over Randoop's test generation. Basically, the source code of a Java file is used as input. It is then parsed using Doclet, in order to retrieve Javadoc comments. Next, the specifications are extracted (i.e., informal information is converted to code – see subsection 4.3.1). These specifications are then fed into the Randoop extension (as described in [13]), which then generates JUnit tests automatically, if any. An example of a method under investigation (and its corresponding Javadoc comment) is shown in program 3.1. The corresponding JUnit test can be found in program 3.2.

Program 3.2: This example is taken from [13] (part of *Figure 2*). It shows the resulting JUnit test, while program 3.1 shows the method under test.

```
void test2() throws Throwable{
  java.util.Map var0 = null;
  try{
    java.util.Map var1 = org.apache.commons.collections.MapUtils.
    synchronizedMap(var0);
    fail("Expected IllegalArgumentException, " + "got
    NullPointerException");
  }catch(IllegalArgumentException expected){}
}
```

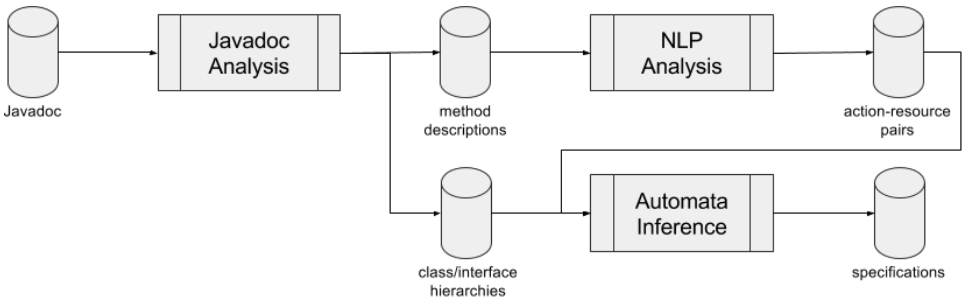


Figure 3.4: Doc2Spec's approach. This overview is taken from [14].

3.4 Resource Specification

Similar to *iComment*, a tool to infer specifications from documentation, namely *Doc2Spec*, is available. However, the approaches and their focus differ significantly. *Doc2Spec* focuses mainly on resource-related specifications and utilizes the online documentation of Java APIs instead of Javadoc comments (as stated in [14]).

3.4.1 Approach

Figure 3.4 shows the overview of the approach, as described in [14]. The *Javadoc* element, however, describes an actual webpage of any documentation under inspection. This differs from other approaches (see sections 3.1, 3.2, 3.3, 3.5), as these use the comments found within the source code. Nonetheless, the information available does not differ – only the way of retrieving it does. Furthermore, the use of *Natural Language Processing* and *Machine Learning* allows for the extraction of *action-resource pairs*, which are then used in combination with the *class/interface hierarchies* in order to infer specifications (see [14] for an explanation of the various terms).

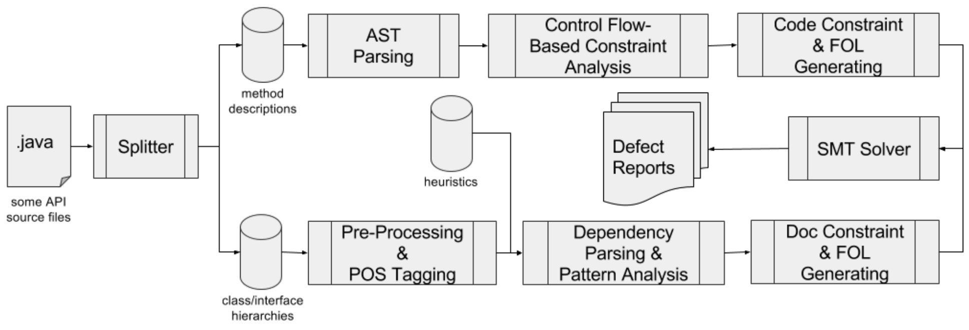

.java
some API source files
Splitter
method descriptions
AST Parsing
Control Flow-Based Constraint Analysis
Code Constraint & FOL Generating
heuristics
Defect Reports
SMT Solver
class/interface hierarchies
Pre-Processing & POS Tagging
Dependency Parsing & Pattern Analysis
Doc Constraint & FOL Generating

Figure 3.5: This approaches overview is taken from [15].

3.5 Directive Defects

Similar to this thesis' work, the work by Zhou *et al.* [15] focuses on inconsistencies (called *defects*). The main differences are the additional step of *code analysis*, the *use of NLP and other techniques* and the check for *type restriction*. It was published in 2017 and is, thus, the most recent work.

3.5.1 Approach

The source code is split into code and documentation – this creates two branches, as seen in figure 3.5. Considering the upper branch first, an *Abstract Syntax Tree* is constructed (see section 2.10), then any information about exceptions is extracted, calibrated and classified. To complete the branch, constraints are generated. As for the lower branch, the comments are pre-processed (i.e., tags and other irrelevant characters – such as HTML tags – are removed). Further Natural Language Processing techniques (e.g., *POS-Tagging*, *Dependency Parsing*) are leveraged. Afterwards, heuristics are used to analyze any occurring patterns in order to generate constraints. Last but not least, a *satisfiability modulo theories solver* (SMT Solver) is used to detect defects. A detailed explanation is available in [15].

3.6 Related Work

This section contains related work that might be of interest due to either some part of or their complete approach.

3.6.1 Static Analysis

FindBugs™[2] is a tool dedicated to bug detection in Java code through static analysis [23]. It is specifically closely related to *@tComment*, as it also checks for `NullPointerException` errors (and others) – however, the approach is completely different (compare [23] with [13]).

3.6.2 Automatic Quality Assessment

JavadocMiner is a tool that automatically assesses the quality of source code comments. While the tool itself was not to be found, the paper describing it was. *JavadocMiner* differs from other approaches, as it uses heuristics – based on heavy use of Natural Language Processing – to assess the quality of comments. These heuristics are split into two categories [7]:

> The heuristics are grouped into two categories, *(i)* internal (NL quality only), *(ii)* code/comment inconsistency.

Obviously, the second category establishes a relation to this Master's thesis (as well as other work – see chapter 3).

3.6.3 Action-Oriented Graph

The approach by Fry *et al.* analyses both, code and comments in order to find *verb-DO pairs* and leverage them to create an action-oriented graph [2]. The comment analysis is based on Natural Language Processing (e.g., POS-Tagging, Chunking, etc.).

According to [2], existing software maintenance tools do help comprehending, navigating, testing, debugging and refactoring and therefore answer a lot of questions asked by developers. However, some questions seem to be unanswered:

> [...] often this information is not sufficient to assist the user in answering high-level questions that software maintainers want answered (e.g. Which class represents concept X? or Where is there any code involved in the implementation of behaviour Y?)[...].

While the result is different to other work and used to enhance software maintenance tools, it is related to this work, due to its comment analysis.

3.6.4 Custom Javadoc Tags

Another interesting approach is the one used by *iContract* [8]. It uses custom Javadoc tags, such as `@pre` and `@post` (and others), in order to detect pre-

[2]The latest version of the project can be found on GitHub [22] with the last update being on March, 15th 2017 at the time of writing.

Program 3.3: An example class for the *jContractor* tool. This example is taken from [6].

```java
class Dictionary_CONTRACT extends Dictionary {

  protected boolean put_PostCondition(Object x, String key){
    return ((has (x)) && (item (key) == x) && (count == OLD. count + 1))
  }

  protected boolean Dictionary_ClassInvariant() {
    return (count >= 0);
  }
}
```

and postconditions, respectively. These pre- and postconditions are then converted into assertions and inserted into the code. If the assertions fail, corresponding exceptions are thrown.

3.6.5 Custom Class Loader

The *jContractor* tool allows a programmer to declare special methods and classes, which define pre- and postconditions. An example of such a class is shown in program 3.3. With this class defined, the custom class loader (used by *jContractor*) does not only load the `Dictionary` class, but also tries to load the `Dictionary_CONTRACT` class [6]. Since it is available, the put method will now be checked for its class invariant and postcondition (during runtime). If a check fails (i.e., returns `false`) then a corresponding exception is thrown.

Chapter 4

Implementation

This chapter contours the basis implementation of the project, which is based on Tan's approach, *@tComment* (see [13]). Furthermore, the extension of the approach – considering ranges – is explained.

4.1 Relevant Libraries

This section briefly explains the libraries used or currently in use – which is necessary to fully comprehend the implementation's approach.

4.1.1 Doclet

As in Oracle's official definition, a doclet is an extension to define the format of the output of the Javadoc tool[1] (which is used to generate documentation based on Javadoc comments) [17].

> Doclets are programs written in the Java™ programming language that use the doclet API to specify the content and format of the output of the Javadoc tool. By default, the Javadoc tool uses the "standard" doclet provided by Sun™ to generate API documentation in HTML form. However, you can supply your own doclets to customize the output of Javadoc as you like. You can write the doclets from scratch using the doclet API, or you can start with the standard doclet and modify it to suit your needs.

When referring to *Doclet* the implementation of the *@tComment* approach is meant (see [13] and chapter 3).

[1] http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html

4.1.2 Randoop

Randoop is a tool used to automatically generate JUnit tests [10]. These tests are *randomly* generated, in order to provide various test cases of which a programmer might not think. While there are various settings to configure Randoop, to fit the needs of its current purpose, the most relevant setting is the *time used to generate the tests per class* (referred to as *time*). For example, using a time of 15 seconds will result in less results than a time of 60 seconds. Obviously, the trade-off is coverage versus computation time.

Additionally, the *null-ratio* can be set. It defines the probability of `null` being used as parameter. This is used in the original approach [13], however, since ranges are denoted as numbers, this ratio is set to default. So, as to cover both the re-implementation of the first part and the extension.

4.1.3 JUnit

Using JUnit, either *regression* or *error tests* are generated. Where regression tests fail only if the behavior of the method changed after it was altered. This ensures that the programmer does not accidentally introduce a bug. Error tests, however, fail every time, since an error was detected (i.e., the programmer defines preconditions and assumes postconditions, but they are not fulfilled). Therefore, JUnit is a simple framework, providing functionality to create repeatable tests [16]. JUnit is, thus, used to execute the tests created with Randoop (see section 4.1.2). It will notify the programmer with any error message thrown while testing for code-comment inconsistencies.

4.1.4 Rythm Engine

RythmEngine (Rythm) is a tool to create templates. Its main purpose is to create HTML templates, either statically or dynamically (using Java). However, the ability to compute plain text allows for a utilization of Rythm, in order to create Java code. For the project of this work, the template is constructed as so it creates a wrapper class (see section 2.9). This is achieved by leveraging the possibility of Java code within the template in order to use dynamic templates.

4.2 Same But Different

While the new approach utilizes the original approaches idea (see section 3.3), the tools in use differ significantly. Furthermore, the implementation varies completely. Figure 4.1 shows the architecture of the new implementation.

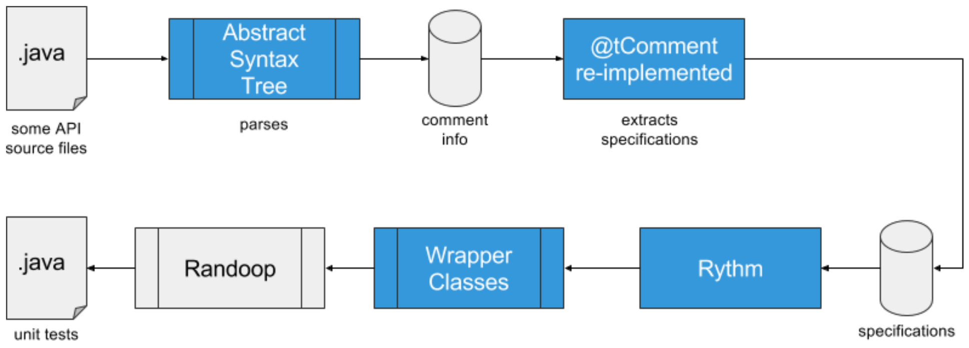


Figure 4.1: An overview of the new approaches architecture. All differences to the approach in [13] are highlighted.

4.2.1 Approach

Instead of Doclet an *Abstract Syntax Tree* is in use. This allows for a more fine-grained control over the parsed source files, as the AST provides additional information (besides comments – see section 2.10). Since the Javadoc comments are the main focus, it is arguable whether to use an AST at this state of the project. However, future extensions (see section 6.3) might depend on it. Especially, if the focus changes or extends to consider method declarations as well, for example.

The *@tComment* re-implementation leverages the idea described in [13], but differs significantly from the original implementation. This also explains any variations in the results of the evaluation (see chapter 5 for evaluation details). Next, the extracted specifications are fed into Rythm templates (see section 4.1.4), with which wrapper classes are created (see section 2.9). This allows to avoid unnecessary tests being created by Randoop (see section 4.1.2). For example, if the original class is used to create JUnit tests (see section 4.1.3) via Randoop then all of the methods are checked, regardless of the information available in the Javadoc. Randoop uses all methods of the wrapper class as well, but since the wrapper class contains only methods relevant for testing, more useful JUnit tests will be generated. Additionally, the flexibility of the new approach is tremendously increased, since there is no dependency on Randoop – the @Randoop extension is not required. This means that any other tool for automatically creating JUnit tests can be used.

4.3 Range Extraction

As an extension for the approach described in section 4.2 and therefore an enhancement of the *@tComment* tool by [13], a precondition extraction based on ranges was implemented (and evaluated – see chapter 5).

4.3.1 Converting Informal Information to Code

In order to convert informal information (see sections 2.5 and 2.6) to code, various alphabets need to be defined:

$$\mathcal{A} = \{a, b, c, \dots, z\}, \tag{4.1}$$
$$\mathcal{R} = \{\texttt{<}, \texttt{>}, \texttt{=}\}, \tag{4.2}$$
$$\mathcal{N} = \{0, 1, \dots, 9\}. \tag{4.3}$$

Firstly, $\mathcal{A}$ is used to create a sequence p, which contains any arbitrary parameter name. Furthermore, $\mathcal{R}$ is used to create a combination of operators r (e.g., $r = (\texttt{<}, \texttt{=})$ or $r = (\texttt{=}, \texttt{=})$). Finally, $\mathcal{N}$ is used to create any real number n, which may or may not occur within a range notation in a Javadoc comment. These sequences build the required basis for the definition of a proper *grammar*

$$G = \langle p, r, n \rangle. \tag{4.4}$$

Note that p and n can be exchanged with each other, to result in any combination. For example, exchanging n with p will result in the grammer $G = \langle p, r, p \rangle$. Applying this operation will lead to the desired output, namely *conditions*. These conditions can then be used within the wrapper classes (see section 2.9) to check for code-comment inconsistencies.

4.3.2 Approach

The approach compared to the one described in section 4.2 only differs in two steps. Addtionally to the *@tComment* extension, the range extraction logic is available (see figure 4.2). Obviously, any following steps have been altered too be able to work with the new input (i.e., the extracted ranges). Furthermore, *reports* (see section 5.3.1) are now part of the final result. They aid in reasoning about why a specification was or was not extracted.

Considering the example given in program 4.1, manually extracting the conditions for the range defined in this Javadoc comment – incorporating the *alphabets* and *grammar* defined in section 4.3.1 – the result is as follows. The parameters `srcBegin`, `srcEnd`, `dst` and `dstBegin` are recognized, but do not contain ranges themselves (i.e., within their corresponding, descriptive text). Next, the `IndexOutOfBoundsException` is recognized and checked against the parameters. The list (i.e., `<ul>...</ul>`) contains all the conditions in this case. Converting them into code yields the following results:

- `srcBegin < 0`,
- `srcBegin > srcEnd`,
- `srcEnd > this.delegate.length()`,
- `dstBegin < 0`,
- `dstBegin + (srcEnd - srcBegin) > dst.length`.

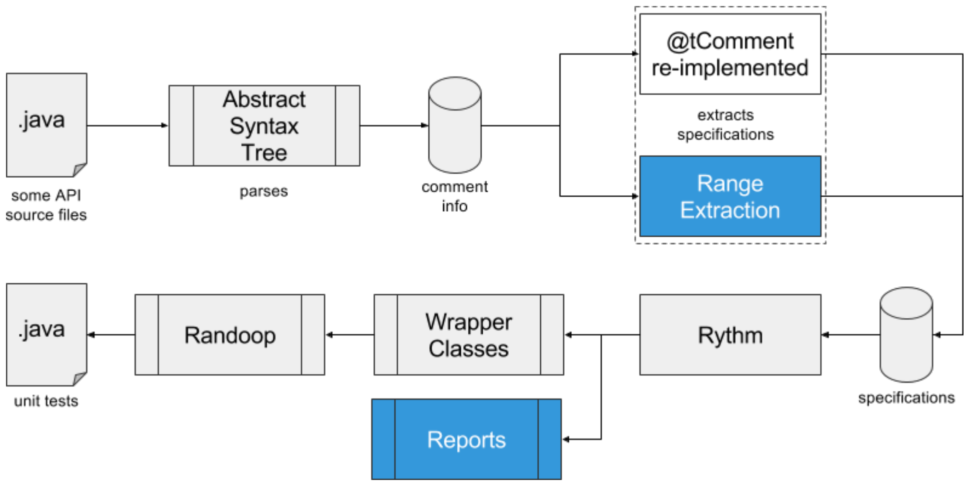


Figure 4.2: An overview of the new approaches extension. All differences to the approach in section 4.2 are highlighted.

Logically combining them with *or* allows for checking against any violation in a later step (wrapper classes). Note that `this.delegate.length()` is used due to wrapper classes using the *delegation pattern* [2].

[2] http://www.oracle.com/technetwork/java/businessdelegate-137562.html

Program 4.1: An example of a real-world use of a Javadoc comment. The code was taken from the `java.lang.String` class.

```
/**
 * Copies characters from this string into the destination character
 * array.
 * <p>
 * The first character to be copied is at index {@code srcBegin};
 * the last character to be copied is at index {@code srcEnd-1}
 * (thus the total number of characters to be copied is
 * {@code srcEnd-srcBegin}). The characters are copied into the
 * subarray of {@code dst} starting at index {@code dstBegin}
 * and ending at index:
 * <blockquote><pre>
 *     dstBegin + (srcEnd-srcBegin) - 1
 * </pre></blockquote>
 *
 * @param      srcBegin   index of the first character in the string
 *                        to copy.
 * @param      srcEnd     index after the last character in the string
 *                        to copy.
 * @param      dst        the destination array.
 * @param      dstBegin   the start offset in the destination array.
 * @exception IndexOutOfBoundsException If any of the following
 *            is true:
 *            <ul><li>{@code srcBegin} is negative.
 *            <li>{@code srcBegin} is greater than {@code srcEnd}
 *            <li>{@code srcEnd} is greater than the length of this
 *                string
 *            <li>{@code dstBegin} is negative
 *            <li>{@code dstBegin+(srcEnd-srcBegin)} is larger than
 *                {@code dst.length}</ul>
 */
```

Program 4.2: The method signature corresponding to the Javadoc comment shown in program 4.1. The code was taken from the `java.lang.String` class – only the formatting was changed for the sake of readability.

```
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)
{
  // Omitted...
}
```

Chapter 5

Evaluation

This chapter examines the implementation of this Master's thesis' project. It contains all *relevant* metrics for the conducted evaluation and their corresponding meanings. However, some metrics are not described in this chapter, as they are not relevant for the evaluation. Refer to appendix C for a complete overview of all metrics. Additionally, commonly used measures were conducted in order to compare the implemented approach to state-of-the-art work (see section 5.2).

5.1 Relevant Metrics

This section provides an overview of all *relevant* metrics, in order to understand the results displayed in the sections 5.4 and 5.5.

5.1.1 General Metrics

The general metrics (see table 5.1) show the extent of the size of the API under investigation. They also show the amount of accessible classes (i.e., classes that can be used within wrappers), and give an overview of some general Javadoc statistics.

5.1.2 Tag Metrics

All metrics regarding tags are shown in table 5.2. The most interesting tags, however, are most likely the `@param`, `@throws`, `@exception` and *faulty* tags count (t_p, t_t, t_e and t_f, respectively). Other metrics in this table, such as *All Tags* and *Unsupported Tags* (i.e., t_{all} and t_u, respectively), are merely an overview of additional statistics, or hold interesting but not too relevant information (e.g., the amount of deprecated tags, t_d). The conducted metrics also list every unsupported tag by name. These tags can be found in table C.5.

Table 5.1: General metrics. Note that c_l and c_{an} may show only a part of the non-accessible classes. For example, `private` classes are not accessible as well. However, they are not counted.

Metric	*Identifier*	*Description*
All Files	f_{all}	All Java files of the API.
All Classes	c_{all}	All classes found in f_{all}.
Local Classes	c_l	All local classes in f_{all}. These are not accessible.
Anonymous Classes	c_{an}	All anonymous classes in f_{all}. These are not accessible.
Accessible Classes	c_a	All classes accessible for the wrappers.
Useful Classes	c_u	All classes (in c_a) containing possible specifications.
Non-Useful Classes	c_{nu}	c_a - c_u
Created Wrappers	w_{all}	All created wrappers.
All Javadocs	j_{all}	All Javadoc comments related to a method.
Useful Javadocs	j_u	All Javadoc comments in j_{all} containing tags.
Non-Useful Javadocs	j_{nu}	j_{all} - j_u

Table 5.2: Tag-Related metrics.

Metric	*Identifier*	*Description*
All Tags	t_{all}	All tags in j_{all}.
`@param` Tags	t_p	All `@param` tags.
`@throws` Tags	t_t	All `@throws` tags.
`@exception` Tags	t_e	All `@exception` tags.
`@deprecated` Tags	t_d	All `@deprecated` tags.
Faulty Tags	t_f	Tags that are not properly formatted.
Unsupported Tags	t_u	All unsupported tags.

Table 5.3: All metrics regarding the specifications that were extracted from the Javadocs.

Metric	*Identifier*
Null Unknown Specifications	s_{nu}
Null Normal Specifications	s_{nn}
Null Any Exception Specifications	s_{na}
Null Specific Exception Specifications	s_{ns}
Range Unknown Specifications	s_{ru}
Range Normal Specifications	s_{rn}
Range Any Exception Specifications	s_{ra}
Range Specific Exception Specifications	s_{rs}
All Unknown Specifications	s_u
All Normal Specifications	s_n
All Any Exception Specifications	s_a
All Specific Exception Specifications	s_s
All Null Specifications	s_{null}
All Range Specifications	s_{range}
All Specifications	s_{all}

5.1.3 Specification Metrics

All metrics related to specifications are shown in table 5.3. However, at this point, the meanings of the specification metrics might not be clear. Thus, they are further explained – for the sake of comprehension.

Null Specifications

Specifications related to `null` are named accordingly (e.g., *Null Unknown Specification*). The meaning of each null-related specification is defined in [13]. Concretely, *Null Unknown* (`NullUnknown`) means that no information about `null` was found (within the current tag description). Every other null-related specification defines the expected behavior of the corresponding method. For every *Null Normal* (`NullNormal`) specification it is expected that the code will execute normally if `null` is passed as value. An error is expected for *Null Any Exception* (`NullAny`) and *Null Specific Exception* (`NullSpecific`) specifications. Where `NullAny` expects *any* error, regardless of type – unlike `NullSpecific`, which is expecting a certain type of error.

Range Specifications

Similar to null-related specifications, range-related specifications are constructed. The only difference is the relation to a range, rather than `null`. They will be referred to as `RangeUnknown`, `RangeNormal`, `RangeAny` and `RangeSpecific`, with respect to the naming convention of the null-related specifications.

Note that `RangeAny` is not yet supported in the implementation, as this case can only be inferred from `@param` tags (i.e., their description), but it was examined that ranges, denoted in `@param` tags, are always of type `RangeNormal`. At least, for every manually inspected `@param` tag (this inspection was executed before the implementation).

5.2 Statistics

In addition to the metrics described in section 5.1, various statistics were constructed. Namely, true and false positives and negatives were counted (where applicable), in order to calculate commonly used measures. In this evaluation, the measures are *dichotomous* (i.e., either *1* or *0* – *true* or *false*, respectively). They are usually displayed in a contingency table (also known as *confusion matrix*) [1] – see section 5.2.1.

5.2.1 Confusion Matrix

A confusion matrix (CM) holds the values for *true positives* (*TP*), *false positives* (*FP*), *true negatives* (*TN*) and *false negatives* (*FN*). An example of such a CM can be found in [1]. From this CM, certain (commonly used) measures can be calculated. Namely, *accuracy A*, *error ratio E*, *precision P* and *recall R* (note that the chosen identifiers may not coincide with the ones chosen in other work). These values are defined in [1] as:

$$A = \frac{TP + TN}{TP + FP + TN + FN}, \tag{5.1}$$

$$E = \frac{FP + FN}{TP + FP + TN + FN}, \tag{5.2}$$

$$P = \frac{TP}{TP + FP}, \tag{5.3}$$

$$R = \frac{TP}{TP + FN}. \tag{5.4}$$

Using these values, it is possible to reason about the reliability of the inference algorithm. Furthermore, *TP* and *FP* can be used to approximate the accuracy of the wrapper classes (when executed with JUnit). Where *A*, *P* and R are at their best when they are equal to 1. With the accuracy denoting the *correctness* of the approach (i.e., how often is the true or false positive or negative correctly inferred). The precision describes the rate of how often the inference correctly inferred a true positive, considering all (true and false) positives. The recall, however, denotes the rate of all true positives inferred, considering all actual true positives (since an FN is a *non-inferred* true positive). On the contrary, the error rate E is at its best when it is 0. It can also be defined as $E = 1 - A$.

5.3 Methodology

Realizing an evaluation for an API is a rather troublesome task, as it needs a lot of manual effort. Especially, for huge libraries like Java (which has more than 7689 files in which over 10500 classes were detected). To significantly decrease this manual effort (i.e., step-by-step debugging and recording by hand) reports were created for every file and examined afterwards (see section 5.3.1). Additionally, the wrapper classes (created by the project) were audited manually, by executing JUnit for all error tests (see section 5.3.2). Once an error and, thus, a code-comment inconsistency was detected, it was examined whether the error showed a true inconsistency (i.e., the code's functionality really varied from the comment). The reports and error tests were marked accordingly (see tables 5.4 and 5.5 for an overview of the used markers), which enabled the software to count some of the measures discussed in section 5.2.

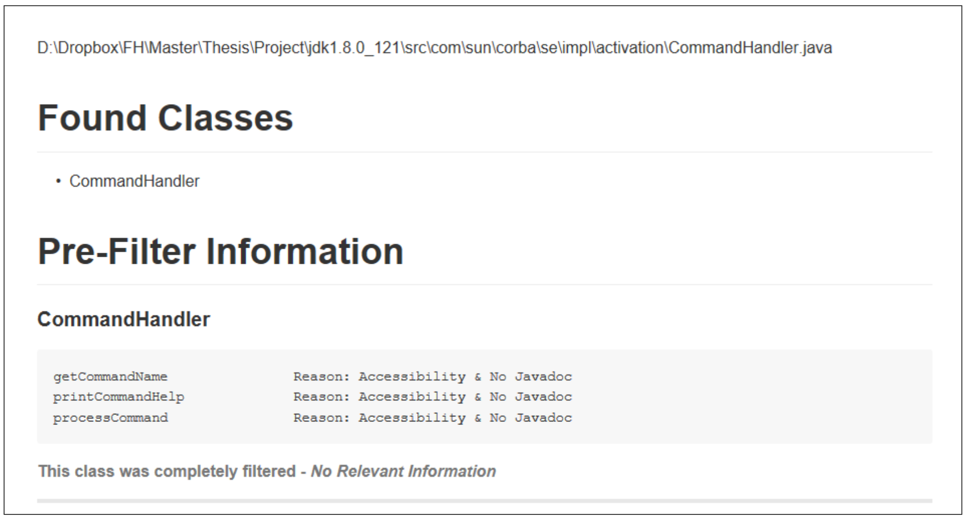
5.3.1 Reports

The reports, which are a part of the final result of the implemented software, give further insight into the text created from the AST parser, how the implementation parsed this text, what conditions were computed and which specifications were extracted. These reports were checked extensively, to see whether they are reliable (i.e., the filtering processes are and the information contained in the reports is correct).

For example, figure 5.1 shows the `CommandHandler` report. It is completely filtered, as there is no relevant information (which is, in fact, correct). Since the reports were inspected and are correctly constructed, they are the only required resource for the evaluation of the specification inference. This evaluation will allow for a compensation of and reasoning about the metrics inaccuracy.

Markers in Reports

When inspecting the reports, the *possible conditions* (i.e., the conditions created before they are sanitized) attract the most attention. This is, because


D:\Dropbox\FH\Master\Thesis\Project\jdk1.8.0_121\src\com\sun\corba\se\impl\activation\CommandHandler.java

Found Classes

- CommandHandler

Pre-Filter Information

CommandHandler

```
getCommandName            Reason: Accessibility & No Javadoc
printCommandHelp          Reason: Accessibility & No Javadoc
processCommand            Reason: Accessibility & No Javadoc
```

This class was completely filtered - *No Relevant Information*

Figure 5.1: An example of a created report. Note that this report contains only one class, which was pre-filtered, as it contains no information about specifications. This example was chosen for the sake of readability. Additionally, the report was checked manually and found to be correct.

they allow to check whether the inference algorithm works as expected. However, the reports are missing an additional parsing step (due to readability). This step is also considered in the evaluation, which may lead to a result that is not obvious at first (i.e., a marker, different from the one firstly expected, is used). See table 5.4, for all markers.

When considering false positives, for example, the parameter `seedLen` of the class `DSAGenParameterSpec` was described as `(omitted...), shall be equal to or greater than subprimeQLen`. From this description, it is expected that the possible condition `seedLen >= subprimeQLen` is extracted. However, instead of one condition, two are extracted. Concretely, `seedLen == seedLen` and `seedLen > subprimeQLen`. Obviously, both conditions are false positives. However, in order to safely determine that the first one is an *FP*, it is necessary to understand why this condition was extracted. Now, it is clear that the first condition is incorrect, since it was stated that the condition results from parsing the description of the `seedLen` parameter. However, when examining the reports, other tags are available as well. Thus, the description – from which the condition resulted – has to be detected first. This means, depending on the comment, a lot of text may have to be read before actually determining the correct marker for only those two conditions.

Another example clarifies the use of a *TN*. For a method `setModel`, the parameter `dataModel` and its corresponding description was found. The description itself does not give any hint about `null` or ranges: `the new data`

Table 5.4: The markers used to alter the reports and error tests. Note that two underscores ("___") are used as both prefix and appendix for the markers.

Marker	*Description*
RTP	Range-Related true positive.
RTN	Range-Related true negative.
RFP	Range-Related false positive.
RFN	Range-Related false negative.
NTP	Null-Related true positive.
NTN	Null-Related true negative.
NFP	Null-Related false positive.
NFN	Null-Related false negative.

`source for this table`. However, the second tag (`@throws`) describes an `IllegalArgumentException` with the description being `if newModel is null`. Apparently, the Javadoc comment was altered at some point of the code's life cycle. But the parameter name was not updated at all places necessary, for the comment to be adequate. Since `newModel` is not the same as `dataModel` the algorithm infers no condition, which is, in fact, correct and therefore a true negative.

As for a *TP*, the class `SizeRequirements` is a fitting example. Various parameters are described, of which all have range-related information. Picking the first parameter, namely `min`, the description is `the minimum size >= 0`. This will result in the possible condition `min >= 0`, as expected.

Last but not least, an example for an *FN* was found in Java's class `TextLayout`. The `hitToPoint` method describes the parameters `hit` and `point`, as well as a `NullPointerException`. The exception's description is `if hit or point is null`. While the expected possible conditions are `hit == null` and `point == null`, only the latter was inferred. Thus, with the latter being a *TP*, the former is an *FN*.

5.3.2 Wrapper Analysis

Similar to the reports, the wrapper classes are analyzed. This allows to validate the functionality of the project's main purpose. Using JUnit, the created error tests are executed, which will, in return, call the corresponding methods of the wrapper classes. If an error is thrown then the user will get notified accordingly. The corresponding method is then examined (code and comment) in order to determine whether the error is a true or false code-

Table 5.5: The markers used to alter the error tests. Note that two underscores ("___") are used as both prefix and appendix for the markers.

Marker	*Description*
TP	A true inconsistency (i.e., an error was thrown and the Javadoc comment *wrongly* describes the code's functionality).
FP	A false inconsistency (i.e., an error was thrown but the Javaodc *correctly* describes the code's functionality).
DC	Dismissed due to a constructor that was required for the instantiation, but did not hold any specifications in the Javadoc comment, thus, always denoting an *FP*.
DS	Dismissed due to an unsupported part of the comment (i.e., an exception was thrown, but the Javadoc comment has an unsupported descriptive text, which couldn't be parsed).
PB	Possible bug found; not related to the wrappers. For example, if the `equals` method was not properly implemented.

comment inconsistency (*TP* and *FP*, respectively).

Note that *syntactical* errors within the wrapper classes are always true positives. These errors should be fixed (by either updating the comment or the code) and then re-analyzed. An example for such a case is shown in program 5.1.

Markers in Wrappers

While *TP* and *FP* are the main measures, some additional markers were introduced during the evaluation (see table 5.5), as other errors were detected as well. All of these errors are, in the end, false positives. However, since they cannot be avoided with the chosen approach (specifically, due to using Randoop), they are counted separately.

For example, a false positive denoted as *DS* (as defined in table 5.5) was found for the `ArrayListIterator` class in the *Apache Commons Collections* library. Program 5.2 shows the code within the corresponding wrapper class. The Javadoc comment of the classes constructor defines two exceptions, of which the `IllegalArgumentException` – with the descriptive text being `if array is not an array` – is the cause of the false positive. Now, Randoop

Program 5.1: A syntactical error in the wrapper classes, due to an invalid comment. The exception `NAMESPACE_ERR` does not exist.

```
/**
* Omitted...
*
* @throws NAMESPACE_ERR
*       : Raised if the qualifiedName has a prefix that is "xml" and
*       the namespaceURI is neither null nor an empty string nor
*       "http://www.w3.org/XML/1998/namespace", or if the
*       qualifiedName has a prefix that is "xmlns" but the
*       namespaceURI is neither null nor an empty string, or if if
*       the qualifiedName has a prefix different from "xml" and
*       "xmlns" and the namespaceURI is null or an empty string.
* @since WD-DOM-Level-2-19990923
*/
public void setAttributeNS(String namespaceURI, String qualifiedName,
    String value) throws Throwable {
  try {
    this.delegate0.setAttributeNS(namespaceURI, qualifiedName, value);
    assert !((namespaceURI == null)) : "Expected NAMESPACE_ERR, but none
     was thrown.";
  }catch (NAMESPACE_ERR specific) {
    if ((namespaceURI == null)) {
      throw specific;
    }
    assert false : "No exception expected, but NAMESPACE_ERR was thrown.
    ";
  }catch (Exception any) {
    assert !((namespaceURI == null)) : "Expected NAMESPACE_ERR, but " +
    any.getClass().getCanonicalName() + " was thrown.";
    assert false : "No exception expected, but " + any.getClass().
    getCanonicalName() + " was thrown.";
  }
}
```

created an error test, passing a *non-array* value as parameter, causing the test to fail with an `IllegalArgumentException`. Thus, the software detects a code-comment inconsistency, even though the comment is correct. Therefore, an *FP* was detected, however, since the text cannot be parsed (as this is not supported), the marker used was *DS*, rather than *FP*.

Every time the marker *DC* is used, a false positive was caused by a constructor without any relevant Javadoc comment. These constructors are not filtered, as they are needed for instantiation of the class under investigation.

When using *PB* to mark an error test, the issue does not lie within the wrapper classes, but rather the original files. Thus, a *possible* bug was detected. The emphasis is on *possible*, because it was not feasible to debug the analyzed libraries. It is possible that this behavior is desired, however, a

Program 5.2: An example of a false positive, marked as *DS* (due to the `IllegalArgumentException`).

```
/**
* Constructs an ArrayListIterator that will iterate over the values in
* the specified array.
*
* @param array
*           the array to iterate over
* @throws IllegalArgumentException
*            if <code>array</code> is not an array.
* @throws NullPointerException
*            if <code>array</code> is <code>null</code>
*/
public ArrayListIteratorWrapper(Object array) throws Throwable {
  try {
    this.delegate0 = new ArrayListIterator(array);
    assert !((array == null)) : "Expected NullPointerException, but none
     was thrown.";
  }catch (NullPointerException specific) {
    if ((array == null)) {
      throw specific;
    }
    assert false : "No exception expected, but NullPointerException was
    thrown.";
  }catch (Exception any) {
    assert !((array == null)) : "Expected NullPointerException, but " +
    any.getClass().getCanonicalName() + " was thrown.";
    assert false : "No exception expected, but " + any.getClass().
    getCanonicalName() + " was thrown.";
  }
}
```

PB was always detected in correlation with the `equals` method of the classes. This means that it is highly likely that all *PB*s are bugs. Randoop will directly comment the issue for the according line of code. Such a comment may be defined as `// This assertion (symmetry of equals) fails`.

5.4 Results

This chapter deals with the results of the evaluation. It shows concrete statistics and metrics for the corresponding libraries. Furthermore, the results are interpreted immediately afterwards.

5.4.1 Apache Commons Collections

The *Apache Commons Collections* library, version 3.2.1, was analyzed, in order to compare the results to the original approach (see chapter 3).

Table 5.6: The results of the analysis for the *Apache Commons Collections* library – version 3.2.1.

Identifier	*Count*	*Identifier*	*Count*
f_{all}	273	c_{nu}	276
c_{all}	412	w_{all}	131
c_l	0	j_{all}	2487
c_{an}	0	j_u	2059
c_a	407	j_{nu}	428
c_u	131	–	–
t_{all}	5648	t_p	2348
t_t	1135	t_e	18
t_d	3	t_f	0
t_u	2144	–	–
s_{nu}	1852	s_{na}	439
s_{nn}	123	s_{ns}	339
s_{ru}	1852	s_{ra}	**0**
s_{rn}	**1**	s_{rs}	**3**
s_u	3704	s_a	439
s_n	124	s_s	342
s_{null}	2753	s_{range}	1856
s_{all}	4609	–	–

Overview

The concrete values for the conducted metrics are shown in table 5.6. Refer to section 5.5, to see the results of the comparison. Furthermore, the concrete statistics (as defined in section 5.2) can be found in table 5.7. Additionally, the result of the analysis of the wrapper classes is shown in table 5.8. Finally, an example of detected code-comment inconsistencies for both a true and a false positive error test can be found in programs 5.3 and 5.4, respectively.

Interpretation

The results of the analysis for the *Apache Commons Collections* (see table 5.6) are rather unsatisfying. Those oddly low results lead to an additional inspection of the inference algorithm, which then shed light on the main

Table 5.7: The results of the analysis of the inference algorithm – *Apache Commons Collections* library (version 3.2.1).

Identifier	*Value*	*Identifier*	*Value*
NTP	859	NFN	85
NFP	62	NTN	1939
A	**95%**	E	**4.99%**
P	**93.26%**	R	**90.99%**
RTP	55	RFN	8
RFP	5	RTN	2877
A	**99.55%**	E	**0.44%**
P	**91.66%**	R	**87.30%**

Table 5.8: The results of the analysis of the wrapper classes – *Apache Commons Collections* library (version *3.2.1*). Note that P_{all} uses $TP + DC + PB + FP$ as divisor, rather than $TP + FP$.

Identifier	*Value*	*Identifier*	*Value*
TP	13	FP	10
DC	15	DS	40
PB	29	–	–
P	**56.52%**	P_{all}	**12.14%**

issue. While the results of the inference are more than satisfying (reaching 95% accuracy for null-related and 99.55% accuracy for range-related specifications – see table 5.7), the counts for the range-related metrics s_{ra}, s_{rn} and s_{rs} are way too low. Especially, when considering the inference algorithm's accuracy. Closely inspecting the reports (see section 5.3.1), it quickly became clear that the implemented sanitation for conditions causes the numbers to drop to (almost) zero. This is, because possible conditions (such as `index < 0`) are inferred correctly, but when trying to sanitize the conditions (in order to filter other, syntactically incorrect conditions), the legit condition is falsely removed. This will cause the software to discard the specification, therefore, it is not counted. Furthermore, the condition is also not available within the wrapper, causing a lot of false positives, or (if all possible conditions get removed) a method not being added to its wrapper class (which corresponds to a false negative). Fixing this issue will result in a higher count for the metrics s_{ra}, s_{rn} and s_{rs}. It was also found that this happens

Program 5.3: Example of a true code-comment inconsistency – *correctly* detected. Randoop used `null` as value for the `predicate` parameter, which is not permitted (according to the comment), but no exception was thrown.

```
/**
* Constructor that performs no validation. Use <code>getInstance</code>
* if you want that.
*
* @param predicate
*           predicate to switch on, not null
* @param trueClosure
*           closure used if true, not null
* @param falseClosure
*           closure used if false, not null
*/
public IfClosureWrapper(Predicate predicate, Closure trueClosure,
    Closure falseClosure) throws Throwable {
  // Omitted...
}
```

mostly for range-related conditions, which is reflected in the results of the evaluation (again, refer to table 5.6) as the corresponding null-related values s_{na}, s_{nn} and s_{ns} are significantly higher. The range-related accuracy (of the inference algorithm), however, is surprisingly high – even though this was hoped for. During the evaluation of the reports, it was found that the inference is highly accurate. However, it is noteworthy that the supported phrases are quite few, compared to the variety of phrases available. Thus, every time a syntactically wrong condition was inferred from unsupported phrases, the condition was considered a true negative.

Since the main focus of this thesis is the detection of code-comment inconsistencies, the wrapper classes were also inspected (see table 5.8). Unfortunately, the values are unsatisfying, due to the conditions being filtered (as mentioned before). Out of 107 created error tests, only 13 true positives were detected. This leads to an overall precision of approximately 12%. While DC, DS and PB are considered to be false positives, it really depends on the definition. For example, a PB is a possible bug – found by Randoop – due to an error in the implementation of the original source file. The end user of the software will get notified about this error. However, considering only supported *and* wrapper-related error tests will result in a much higher accuracy of approximately 56% (due to DC, DS and PB being ignored). Obviously, by doing so, it would be better to inspect more than the remaining 23 error tests.

Program 5.4: Example of a true code-comment inconsistency – *falsely* detected. The range-check was correctly inferred by the algorithm, but the conditions got filtered. Randoop used an index equal to the size, but the comment denotes that this should be okay (note the missing *equals* sign after `&gt`).

```
/**
* Set the Iterator at the given index
*
* @param index
*            index of the Iterator to replace
* @param iterator
*            Iterator to place at the given index
* @throws IndexOutOfBoundsException
*             if index &lt; 0 or index &gt; size()
* @throws IllegalStateException
*             if I've already started iterating
* @throws NullPointerException
*             if the iterator is null
*/
public void setIterator(int index, Iterator iterator) throws Throwable {
  // Omitted...
}
```

5.4.2 Java

Java itself (version 1.8.0_121) was analyzed, to show the effectiveness of this Master's thesis' project in a large-scaled API. Furthermore, for the sake of comparing this work with the approach by [15], a stripped-off Java version was used (i.e., only the packages stated in [15].

Overview

The conducted metrics are shown in table 5.9. The metrics for the analysis of the smaller version are available in table 5.10. As it is not feasible to manually analyze a library as huge as Java [15], the statistics were conducted for the smaller version only – which can be found in table 5.11. See section 5.2 for an explanation of the statistics.

Interpretation

The rather disappointing results in section 5.4.1 (regarding the wrapper classes) resulted in the decision not to analyze the wrapper classes for *Java*, for it is not worth the effort (since respectively low results are expected anyway). Instead, the complete Java library was analyzed, as well as a stripped-off version. Where the results are surprisingly high (see tables 5.9 and 5.10), compared to *Apache Commons Collections*. Even though range-related con-

Table 5.9: The results of the analysis for the *Java* library – version *1.8.0_121*.

Identifier	*Count*	*Identifier*	*Count*
f_{all}	7689	c_{nu}	9234
c_{all}	10578	w_{all}	850
c_l	117	j_{all}	59544
c_{an}	43	j_u	43974
c_a	10084	j_{nu}	15570
c_u	850	–	–
t_{all}	167319	t_p	48946
t_t	13095	t_e	9408
t_d	433	t_f	1
t_u	95437	–	–
s_{nu}	29033	s_{na}	372
s_{nn}	1595	s_{ns}	1127
s_{ru}	29033	s_{ra}	**0**
s_{rn}	**60**	s_{rs}	**126**
s_u	58066	s_a	372
s_n	1655	s_s	1253
s_{null}	32127	s_{range}	29219
s_{all}	61346	–	–

ditions are falsely removed, some of them pass the sanitation step. Usually, these conditions compare two parameters or a parameter with a method call. For example, the condition `index >= size()` may be inferred from a Javadoc comment. In this case, the sanitation will consider the condition as syntactically correct (which it is, indeed), and, thus, not discard the condition. Of course, there are some more cases like this, which explains the higher counts for range-related values (compare s_{ra}, s_{rn} and s_{rs} in table 5.6 with the values in the tables 5.9 and 5.10). However, while the counts are higher than for the *Apache Commons Collections*, they are still extremely low. Interestingly, the range-related accuracy and precision are only a little lower than they are for the *Apache Commons Collections* library. Even better, the null-related accuracy and precision actually increased. Since these libraries are completely different, the high values for accuracy and precision indicate that the inference algorithm works rather well for differently sized libraries.

Table 5.10: The results of the analysis for the *stripped Java* library – version *1.8.0_121*. These results are used for the comparison with the results by [15].

Identifier	*Count*	*Identifier*	*Count*
f_{all}	2514	c_{nu}	3658
c_{all}	4430	w_{all}	465
c_l	106	j_{all}	29613
c_{an}	36	j_u	22938
c_a	4123	j_{nu}	6675
c_u	465	–	–
t_{all}	99243	t_p	26403
t_t	6788	t_e	4298
t_d	300	t_f	0
t_u	61454	–	–
s_{nu}	15749	s_{na}	102
s_{nn}	849	s_{ns}	754
s_{ru}	15749	s_{ra}	**0**
s_{rn}	**48**	s_{rs}	**83**
s_u	31498	s_a	102
s_n	897	s_s	837
s_{null}	17454	s_{range}	15880
s_{all}	33334	–	–

Most likely, similar values can be expected for other libraries with a similarly well-documented source code.

Note that the values in table 5.11 are conducted for the stripped-off version *only*, as it is not feasible to completely, manually evaluate a library with over 10500 classes [15].

5.5 Comparison

In order to comprehend the impact of the elucidated metrics and statistics, it is necessary to compare them to previous work [13, 15] – where applicable.

Table 5.11: The results of the analysis of the inference algorithm – stripped-off *Java* library (JDK 1.8.0_121). Note that the sum of $TP+FP+TN+FN$ for null-related inferences differs by a value of 2, compared to range-related specifications. This indicates an error during the analysis. Since the error is so little, it is considered negligible.

Identifier	*Value*	*Identifier*	*Value*
NTP	1624	NFN	176
NFP	106	NTN	18306
A	98.6%	E	1.39%
P	93.87%	R	90.22%
RTP	661	RFN	163
RFP	130	RTN	19260
A	**98.55%**	E	**1.44%**
P	**83.56%**	R	**80.21%**

5.5.1 Original Approach

The results of the evaluation of the original approach by Tan *et al.* are shown in [13]. However, for the sake of the comparison, the most important values are shown in table 5.12 – these values are mapped to the metrics of this thesis (i.e., their name may have changed). Interestingly, the metrics vary completely (compare table 5.6 with table 5.12). Figures 5.2 and 5.3 show the differences more clearly, as they directly show the values of interest compared to each other as chart. According to these charts, the original approach finds a lot more `@param` tags, but also significantly less of both `@throws` tags and any specification type (see s_{nu}, s_{nn}, s_{na} and s_{ns} in figure 5.3). As for the accuracy, the original approach states 99% (as seen in table 5.12), which correspond to the inference algorithm, rather than code-comment inconsistencies (see [13]). Knowing this, the value to compare this accuracy with, is the one found in table 5.7 (upper left section), denoted as A – which is 95% and, thus, less.

Interpretation

Considering the charts shown in figures 5.2 and 5.3, the conducted metrics seem quite off. Not only is the `@param` tags count (t_p) of the original approach significantly higher, but also the `@throws` tag count (t_t) considerably lower than the values of this thesis. Even more confusing is that the thesis approach finds more `NullNormal` specifications (s_{nn}). According to the difference in the count for t_p, this should be vice-versa. Furthermore, the

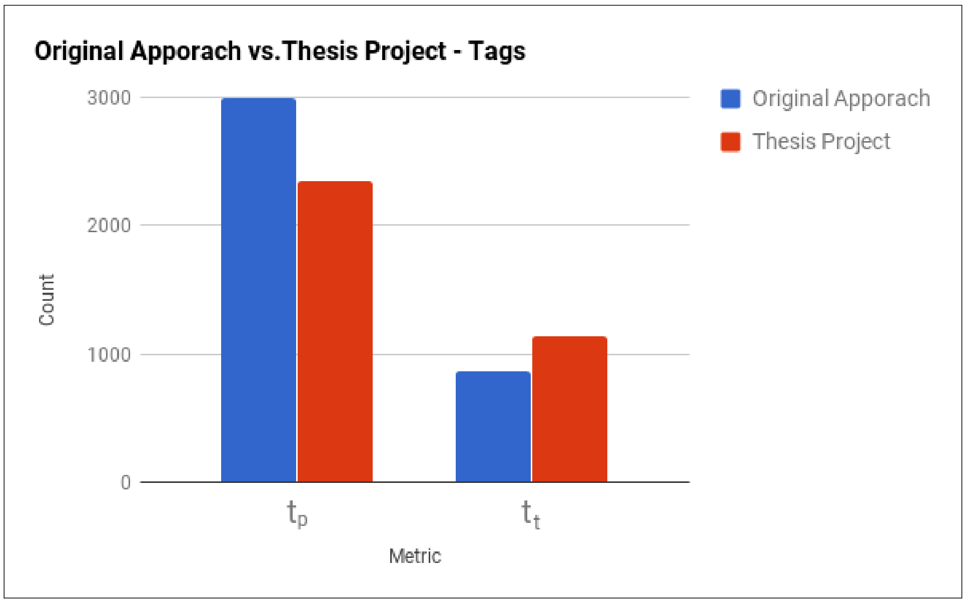


Figure 5.2: The comparison of this work's project with the original approach (see [13]). The metrics refer to the counts of all `@param` (t_p) and `@throws` (t_e) tags.

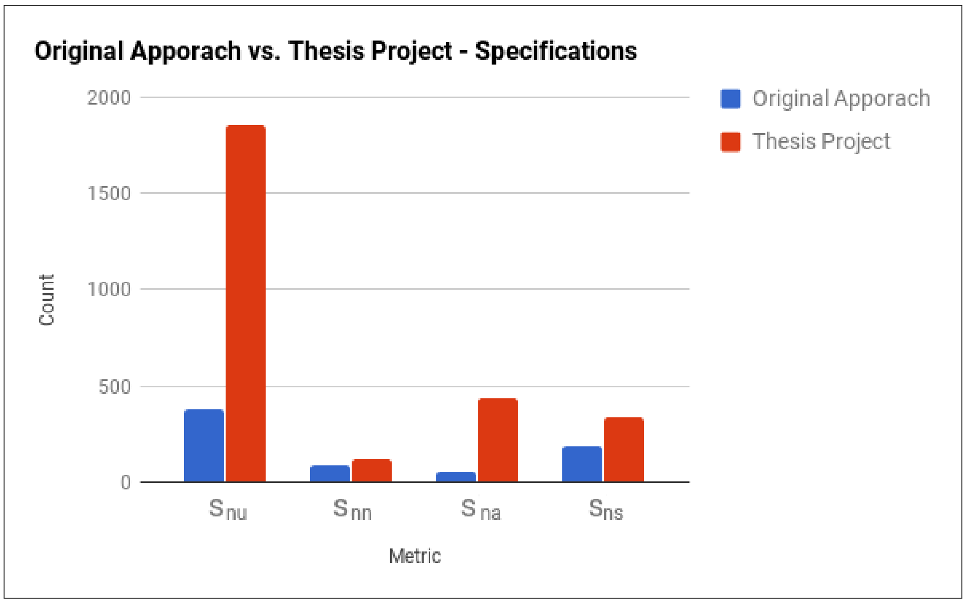


Figure 5.3: The comparison of this work's project with the original approach (see [13]). The metrics refer to the null-related specifications `NullUnknown` (s_{nu}), `NullNormal` (s_{nn}), `NullAny` (s_{na}) and `NullSpecific` (s_{ns}).

Table 5.12: The results of the analysis conducted by [13]. Note that these values are mapped to the metrics of this thesis (i.e., their name may have changed).

Identifier	*Count*	*Identifier*	*Count*
t_p	2996	t_t	862
s_{nu}	375	s_{na}	54
s_{nn}	83	s_{ns}	190
A	99%	–	–

extreme difference for `NullUknown` specifications (s_{nu}) is beyond odd.

In order to properly investigate the differences, the *@tComment* project was set up and executed. The resulting counts differed from the ones in the work by Tan *et al.*, which was rather unexpected. Since the values shown in [13] were not reproducible, an approximation of the count was conducted, using the Linux command `grep` in combination with a word count `wc`. The count displayed by the terminal was 2349, which is only 1 off (compared to the value of this work). This indicates that the values of the original approach are faulty. Knowing this, an utterly high accuracy of 99% for the original approach seems questionable. While the accuracy of this thesis is 95% – and, thus, less – it is not feasible to directly compare the values, as long as they cannot be reproduced. As last resort, one of the authors was contacted, namely *Shin Hwei Tan*. While discussing the issue, she came to the following conclusion:

> It could be that some files are not loaded so that their comments are not parsed.

This means that a further examination of the *@tComment* is necessary, in order to determine whether Tan is correct. However, this is not feasible within the scope of this thesis.

5.5.2 Directive Defects

Similar to the comparison shown in section 5.5.1, the results of the evaluation conducted by [15] are mapped to fit this work's naming conventions.

Overview

The results can be found in table 5.13. Figure 5.4 shows the tag-related comparison. While the approach by [15] also varies, regarding the detected tags, the difference is not as significant as in the comparison with the original approach (see section 5.5.1). When comparing the statistics, however,

Table 5.13: The results of the analysis conducted by [15]. Note that these values are mapped to the metrics of this thesis (i.e., their names may have changed). Additionally, the results were summed up, in order to compare the overall values.

Identifier	*Value*	*Identifier*	*Value*
t_p	26692	TP	1629
t_e	4303	FP	758
t_t	6810	FN	276
P	69.71%	R	82.19%

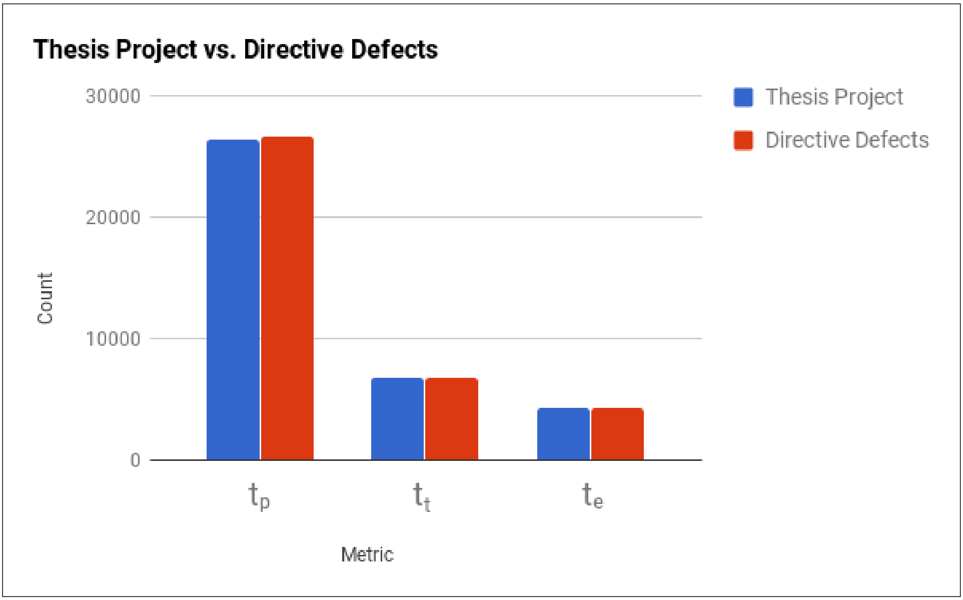


Figure 5.4: The comparison of the Master's thesis' project with the directive defects approach (see [15]). The contrast between the detected tags is shown.

the difference clearly rises (see figure 5.5). Not only does the approach by [15] find less true positives, but also more false positives. False negatives, however, are slightly less than in this Master's thesis' approach. Oddly, the accuracy is never calculated in [15], only the precision (69.71%) and the recall (82.19%) are calculated. The overall precision and recall values for this work are 88.71% and 85.21%, respectively. Comparing these values with the ones shown in table 5.13 emphasizes that the implemented approach achieves slightly higher values than the approach by [15].

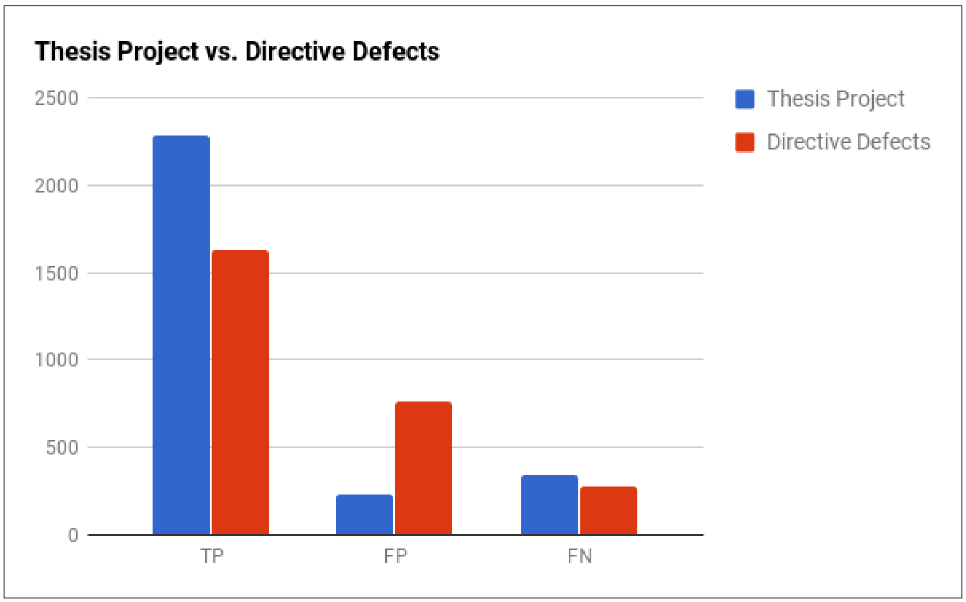


Figure 5.5: The comparison of this work's project with the directive defects approach (see [15]). The contrast between *TP*, *FP* and *FN* is shown.

Interpretation

Since the approach by [15] is the newest one, analyzing a stripped-off version of *Java*, it is quite interesting to see differences as shown in figures 5.4 and 5.5). Figure 5.4 seems to show almost no difference at all. While the difference is not as huge as for the original approach, the scale (which resulted from the high count of t_p) dissolves the differences for t_t and t_e. They *are* different, however. This difference results from the fact that Zhou *et al.* uses additional 10% of the *Java* library for the evaluation, which is not the case to this Master's thesis evaluation. Curiously, even without these additional 10%, the inference seems to be considerably better, as shown in figure 5.5. This was unexpected, due to Zhou *et al.* using rather advanced techniques, compared to this thesis. However, only little support for different phrases in the approach implemented for this work is available. It is highly likely that the high false positives conducted by [15] result from a higher support of different phrases. Thus, supporting more, different phrases than the current approach does, is desirable. Finally, the precision and recall of this thesis are higher than the ones of Zhou *et al.*, which is expected (considering the differences shown in figure 5.5).

Chapter 6

Conclusion

During the project's implementation (see chapter 4) it quickly became clear that analyzing Javadoc comments is a highly complex task. While this was expected – due to statements in other work (see [13]) – the real extent of the issue reaches way beyond the expected complexity.

Probably, one of the biggest issues during the analysis and its implementation was ambiguity, as it is simply not feasible to cover all possible phrases. However, working with third party libraries can also have its disadvantages (e.g., bad documentation makes it hard to properly use these tools – ironically, this was also mentioned in section 1.1).

Finally, a project of such a huge scale is rather hard to develop single-handedly. While it was clear that the project is of greater extent, the overall magnitude was rather unexpected.

6.1 Summary

This thesis discusses the examination of Javadoc comments and any occurring tags, in order to detect code-comment inconsistencies. This detection shall help to improve documentation, which, ultimately, may lead to less bugs in software code.

For the examination to take place, a project was implemented, as discussed in chapter 4. After the implementation was finished, a further extension was implemented to create reports and specify metrics. Where the reports (see section 5.3.1) give detailed insight into (almost) every step of the implementation. This helped finding bugs and reason about why some specifications were or were not extracted. Which allowed to justify some unexpected results during the evaluation (see chapter 5). The metrics, on the other hand, are supposed to be the absolute values for the evaluation. However, as there are still bugs in the implementation, they are not (yet) reliable.

6.2 Discussion

When it comes to Javadoc comment analysis, various approaches have already been introduced (see chapter 3). The most relevant approaches are described in section 3.3, 3.4 and 3.5. The approach by [13] (which is the original approach) investigates null-related values. Due to the remarkable results, the relevance of analyzing Javadoc comments is emphasized. Not only were many code-comment inconsistencies found, but also actual bugs in the analyzed libraries [13]. When re-implementing this approach, however, the results varied significantly. While the evaluation showed a similarly high accuracy for the project of this thesis, the differences in the conducted metrics indicate that such an analysis is highly complex and error-prone.

Comparatively, the work by [14] leverages more advanced techniques, such as Natural Language Processing (NLP) and Automata Inference. Furthermore, the tool analyzes the online documentation of the libraries under investigation. While using NLP seems to be an advantage (refer to appendix A), analyzing the online documentation might be a huge disadvantage. This is because, besides the *class/interface hierarchies* and *method descriptions* (see section 3.4), almost no other information is available. This means that an extension of the approach is not possible – in a sense that techniques, such as static code analysis, cannot be leveraged (since no information about the code is available).

On the contrary to [13, 14], the approach by [15] is rather advanced. While supporting range- and null-related analysis, Zhou *et al.* also check for *type restriction* (see section 3.5). Furthermore, the source code is statically analyzed and Natural Language Processing is in use, as well. This allows for an aided parsing of the Javadoc comment (through NLP) and to additionally access further information. As this approach also uses heuristics for its inference algorithm, it heavily depends on the amount of the *manually* defined heuristics. The higher the amount, the better will the approach detect specifications.

When comparing the advantages and disadvantages of the approaches with the ones of this project's approach, it quickly becomes clear that a combination of some of the techniques of the different approaches should be used. Ultimately, the AST parser in combination with Natural Language Processing can help inferring specifications. Furthermore, removing the Randoop, Rythm and JUnit dependencies (by leveraging other techniques, such as the one used by *iContract* [8]) might further improve the software – due to fewer dependencies.

6.3 Future Work

With the current state of the software, various enhancements are possible (besides some bug-fixes), which will be stated in this section.

6.3.1 Considering Additional Tags

Besides the currently parsed `@param` and `@throws` tags there are many more, some of which might contain relevant information.

For example, the `@return` tag could be parsed and the extracted information compared to the actual return value, in order to find code-comment inconsistencies (i.e., extract postconditions – see section 2.8.2). Furthermore, it was found that often times the specifications for parameters are defined in the `@return` tag rather than the `@param` or `@throws` tag.

6.3.2 Natural Language Processing

As other work suggests already (see chapter 3), NLP can be leveraged for a project like the one of this Master's thesis. Therefore, the parsing of the Javaodoc comments could be altered to use NLP, as well. Further information can be found in appendix A.

6.3.3 Using a Different Approach

Chapter 3 states many different approaches, of which the approach described in section 3.6.4 is quite promising. Leveraging the tool *iContract* and its ability to pre-compile code based on custom Javadoc tags, may help simplifying the current approach.

For example, instead of creating wrapper classes, the extracted specifications could be converted into the format used by [8]. After this conversion the result could be added to the corresponding Javadoc comment. After the Javadoc comment was altered, *iContract* could be executed in order to find contract violations (see section 2.8). By doing so, Randoop becomes obsolete, which also means that no JUnit tests will be generated anymore. Additionally, this will significantly decrease the amount of files created and, therefore, the required disc space.

Appendix A

Natural Language Processing Approach

As other tools already leveraged *Natural Language Processing* (NLP), and working with *natural language* clearly suggests, using NLP techniques can be of advantage. Due to constraints (i.e., deadlines, team size, etc.), however, NLP is not (yet) in use.

In any case, NLP techniques were tested in various approaches (see [15], for example) and seem to be the way to go. However, while testing NLP, it quickly became clear that it is not to be used as only-technique. Instead, other – more advanced – techniques, such as *Machine Learning* should be used in combination with NLP (as shown by some state-of-the-art work). Therefore, this section focuses on explaining some available techniques in the field of NLP. Especially, the technique called *dependency graph* seems to be promising (see section A.1.5).

A.1 Techniques

This section briefly discusses common techniques in the field of NLP, in order to give a basic overview of the possibilities, which may be leveraged for an extension of the project.

A.1.1 Tokenization

Tokenization is the task of creating tokens out of some character sequence, as the *StanfordNLP Group* describes [21]:

> Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation.

Table A.1: The resulting tokens after tokenization took place. Note that not all are shown – for the sake of readability. Read from left to right, top to bottom.

Tokens			
If	any	of	the
following	is	true	:
srcBegin	is	negative	.

For example, consider the `IndexOutOfBoundsException` in the comment in program 4.1. Using the *StanfordNLP* library, the tokenization yields the tokens shown in table A.1. Obviously, tokenization, as is, does not improve anything, as this can be achieved with a simple split as well. Therefore, it is only the basis for further techniques.

A.1.2 Stemming

In order to *stem* words, various algorithms are available (such as *Porter* or *Lovins* stemming algorithm). Depending on the chosen algorithm the results may differ. However, stemming could help improving the resolution of *code references* in comments. For example, the comment `if the index is greater than this list's size` has to be converted into `index > this.size()`, if the word `list's` refers to the class. If not, it might refer to a parameter within the method declaration. In order to find the reference, stemming is necessary. This means, after the tokens (see section A.1.1) were computed, the word `list's` is stemmed into `list`. Using this information to check for the *same* stem word in the classes name, a reference can be detected. Thus, if a reference was found `this.size()` can be used (of course, the brackets should only be appended if there is a method available). However, if no reference was found then the parameters within the method declaration could be checked. If there is still no reference found, there might be a code-comment inconsistency.

A.1.3 Part-Of-Speech Tagging

Part-Of-Speech Tagging (POS-Tagging) is a technique that allows to identify the type of a word. For example, the words *or*, *and* and others are called *coordinating conjunction* (CC). Detecting these CC's is essential in order to resolve logical combinations. Using POS-Tagging to tag any word, might yield CC's within a sentence (see *StackOverflow*[1] for a complete list of the *Peen Tree Bank Project*). These tags could be used (instead of the words

[1]https://stackoverflow.com/a/1833718/6656268

Table A.2: The POS-Tagging result, using the tokens from subsection A.1.1. Note only the POS-Tags for table A.1 are shown – for the sake of readability.

Token	*POS-Tag*
If	IN
any	DT
of	IN
the	DT
following	VBG
is	VBZ
true	JJ
:	:
srcBegin	NN
is	VBZ
negative	JJ
.	.

themselves) to allow for higher abstraction, which is, thus, desirable. See table A.2 for an example. With this information, certain tokens can easily be searched for, however, sometimes it is necessary to have information about the dependency between the words (see subsection A.1.5).

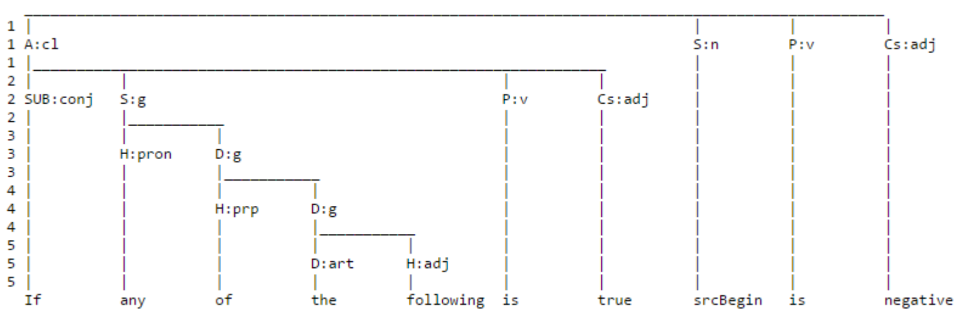
A.1.4 Parsing

Parsing is the technique used to create *parse trees* (PT). For every sentence, a PT can be created, in order to provide the structure of a sentence [20]:

> A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as ”phrases”) and which words are the subject or object of a verb.

Using the same sentence as in table A.1, the result is as shown in figure A.1 (created with a tool provided by *VISL*[2]). Not only does the created PT contain the tags created via POS-Tagging (see subsection A.1.3), but also shows the constitution for each sentence. While this is great, it is also an example of an NLP technique that is not suited to aid the Javadoc analysis.

[2] http://visl.sdu.dk/visl/en/parsing/automatic/trees.php



```
  __________________________________________________________________________________________
1 |                                                                     |         |         |
1 A:cl                                                                  S:n       P:v       Cs:adj
1 |___________________________________________________________          |         |         |
2 |         |                             |         |         |         |         |
2 SUB:conj  S:g                           P:v       Cs:adj    |         |         |
2 |         |_________                    |         |         |         |         |
3 |         |         |                   |         |         |         |         |
3 |         H:pron    D:g                 |         |         |         |         |
3 |         |         |_________          |         |         |         |         |
4 |         |         |         |         |         |         |         |         |
4 |         |         H:prp     D:g       |         |         |         |         |
4 |         |         |         |_________|         |         |         |         |
5 |         |         |         |         |         |         |         |         |
5 |         |         |         D:art     H:adj     |         |         |         |
5 |         |         |         |         |         |         |         |         |
  If        any       of        the       following is        true      srcBegin  is        negative
```

Figure A.1: The parse tree of a *part* of the sentence used as example in section A.1.1. The parse tree was create with a tool provided by *VISL*.

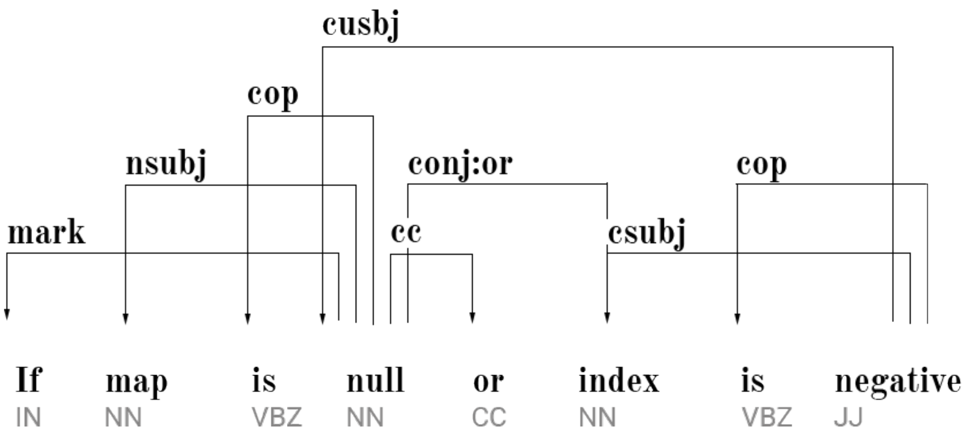


Figure A.2: The dependency graph for the sentence `If map is null or index is negative`.

A.1.5 Dependency Graph

A dependency graph is quite similar to a parse tree. The main difference is that information about the dependency between the words of a sentence is available, rather than information about how a sentence is constituted.

An example is shown in figure A.2. The dependencies for `index is negative` and `map is null` can be detected. Ultimately, these are also the conditions of interest. A corresponding algorithm for detecting these conditions, allows for a rather simple conversion to the corresponding code representations (i.e., `map == null` and `index < 0`, respectively). Obviously, this algorithm also needs to be implemented, as such that it is able to parse arbitrary sentences.

Program A.1: Example code, using the *StanfordNLP* API, to create a *parse tree*.

```java
import edu.stanford.nlp.simple.Document;
import edu.stanford.nlp.simple.Sentence;

public class Main {
  public static void main(String[] args) {
    String comment = "If any of the following is true: srcBegin is
    negative. ";
    comment += "srcBegin is greater than srcEnd srcEnd is greater than
    the length ";
    comment += "of this string dstBegin is negative dstBegin+(srcEnd-
    srcBegin) is larger than dst.length";

    // Create a document. No computation is done yet.
    Document document = new Document(comment);
    for (Sentence sentence: document.sentences()) {
      // When we ask for the parse, it will load and run the parser
      System.out.println(sentence.parse());
    }
  }
}
```

A.2 Code Example

The *StanfordNLP* API is a comprehensive library. Thus, the code required to create a PT (see subsection A.1.4) is rather simple, as shown in program A.1).

Appendix B

Wrapper Generation

In order to automatically create wrappers (see section 2.9), Rythm is used (refer to section 4.1.4). This allows for dynamic creation of Java code (even though Rythm is usually used to create HTML templates). However, to fully comprehend how such a wrapper is created, this chapter will explain the developed algorithm and give a concrete example (starting with the comment and ending with the resulting code).

B.1 Algorithm

The algorithm for creating a wrapper is shown as pseudo code. For better comprehension, the pseudo code is split up in three different parts:

- algorithm B.1 for generating code inside the `try`-block.
- algorithm B.2 for generating the `catch`-block for *specific exceptions* (i.e., the exceptions are explicitly stated).
- algorithm B.3 for generating code inside the *general* `catch`-block (i.e., every exception not declared in a specific `catch`-block is caught here).

Algorithm B.1 is rather simple, as it only is supposed to print the code required for the invocation of the method being analyzed, as well as the assertions for every *expected* exception. Note that the methods, starting with *remove*, are supposed to filter certain, negligible specifications. If the specification is *unknown* then no assertion is possible. If the specification is *normal* then the method should be executed without errors (regardless of the value of the parameter), therefore, no assertion is required.

Compared to algorithm B.1, algorithm B.2 is way more complex. Only extracted specifications that throw a specific exception are relevant. For every *different* exception, a separate `catch`-block has to be created. Within the `catch`-block, the correct *checks* and *assertions* have to be printed. In order to do so, all conditions for the same exception are logically merged (*or*-wise). While this will avoid false positives, the end-user is left to some manual effort, as it might not be immediately clear by which parameter the exception was caused. Similarly, all checks for the exception of the type *any* (i.e., `NullAny` and `RangeAny` – see chapter 5) are printed (afterwards). Next, the remaining specifications of type *specific* (i.e., `NullSpecific` and `RangeSpecific` – see chapter 5) have to be printed. This ensures inconsistency detection, in case of a specific exception being document but not actually thrown. Last but not least, a *failing assertion* (i.e., always throwing an error) is printed, in order to mark an inconsistency. This is necessary, because at this point no exception was expected at all.

Finally, algorithm B.3 handles all other types of exceptions that are not explicitly declared. First of all, every specific exception has to be asserted (as discussed before). This check helps to detect bugs within algorithm B.2 (i.e., whether a specific exception did not create a proper `catch`-block), as well as inconsistencies. Next, the specifications, expecting *any* exception, have to be asserted. These specifications are merged again, to create one single assertion. If this assertion passes then no exception was expected at all. Therefore, a failing assertion has to be printed.

Algorithm B.1: Pseudo code for the *content* of the `try`-block.

```
1:  CreateTryBlockContent(method)
2:      specifications ← getSpecifications(method)
3:      specifications ← removeUnknownSpecifications(specifications)
4:      specifications ← removeNormalSpecifications(specifications)

5:      printInvocation(method)

6:      for each spec ∈ specifications do
7:          condition ← getCondition(spec)
8:          printAssertion(condition)
9:      end for
10: end
```

B.2 Example

Imagine a method with four parameters, namely `index`, `map`, `coll`, `obj`, as seen in program B.1. When analyzing its Javadoc comment, specifications and conditions will be extracted, as shown in table B.1. Based on the algorithms explained in section B.1, the wrapper as seen in program B.2 will be created. While this example might not be applicable in a real program, it clarifies the case when different specifications were inferred during the analysis. Every other case is also handled, as the only difference is less code. This is, because – depending on the specifications – the wrapper is

Algorithm B.2: Pseudo code for the `catch`-blocks with *specific exceptions*.

```
 1: CreateSpecificCatchBlocks($method$)
 2:     $specifications \leftarrow getSpecifications(method)$
 3:     $specifics \leftarrow getSpecifics(specifications)$
 4:     $anys \leftarrow getAnys(specifications)$

 5:     for each $specific \in specifics$ do
 6:         $exception \leftarrow getException(specific)$

 7:         if not $alreadyCreated(exception)$ then
 8:             $printCatchBlockOpening(exception)$

 9:             $printCurrentSpecificCheck(specific)$
10:             $printAnyChecks(anys)$
11:             $printRemainingSpecificAssertions(specifics)$
12:             $printFailingAssertion()$

13:             $printCatchBlockClosing()$
14:         end if
15:     end for
16: end
```

Algorithm B.3: Pseudo code for the *content of the general* `catch`-block.

```
1: CreateAnyBlockContent($method$)
2:     $specifications \leftarrow getSpecifications(method)$
3:     $specifics \leftarrow getSpecifics(specifications)$
4:     $anys \leftarrow getAnys(specifications)$

5:     $printAllSpecificAssertions(specifics)$
6:     $printAnyChecks(anys)$
7:     $printFailingAssertion()$
8: end
```

build in the same way, but some lines may not be created. For example, if there is no specific exception inferred then the `catch`-blocks handling the `IndexOutOfBoundsException` and the `NullPointerException` in program B.2 will not be created (same applies for any assertions or `if`-statements).

The generated wrapper for the method, shown in program B.1, will be used by Randoop (see section 4.1.2), in order to create JUnit tests (see section 4.1.3). These JUnit tests can then be executed, which will lead to any found inconsistency being detected, as the end user will get notified by Eclipse (if Eclipse is in use).

Program B.1: An example for explaining the wrapper generation algorithm. This is the input for the analysis, which extracts specifications. These specifications are used to generate wrappers. Note that such a method is highly unlikely to be written in actual source code. However, it helps clarifying the wrapper creation for different specification types.

```
/**
 * Omitted...
 *
 * @param index no range declared here
 * @param map may not be null
 * @param coll may be null
 * @param obj no specification here
 *
 * @throws IndexOutOfBoundsException if the index is negative
 * @throws NullPointerException if obj is null
 */
public void method(int index, Map<?, ?> map, List<?> coll, Object obj){
  // Omitted...
}
```

Table B.1: Extracted specifications and their corresponding conditions, according to the Javadoc comment, shown in program B.1. Note that lines without information are omitted.

Comment Description	*Specification*	*Condition*
`may not be null`	`NullAny`	`map != null`
`coll may be null`	`NullNormal`	`coll == null`
`if the index is negative`	`RangeSpecific`	`index < 0`
`if obj is null`	`NullSpecific`	`obj == null`

Program B.2: The wrapper created from the Javadoc comment shown in program B.1.

```
public void method(int index, Map<?, ?> map, List<?> coll, Object obj){
  try{
    this.delegate.method(index, map, coll, obj);

    assert !(index < 0) : "IndexOutOfBoundsException expected, but none was thrown.";
    assert (map != null) : "Any exception expected, but none was thrown.";
    assert !(obj == null) : "NullPointerException expected, but none was thrown.";
  }catch(IndexOutOfBoundsException specific){
    if((index < 0)){
      throw specific;
    }
    if(!(map != null)){
      throw specific;
    }
    assert !(obj == null) : "NullPointerException expected, but IndexOutOfBoundsException was thrown.";

    assert false : "No exception expected.";
  }catch(NullpointerException specific){
    if((obj == null)){
      throw specific;
    }
    if(!(map != null)){
      throw specific;
    }
    assert !(index < 0) : "IndexOutOfBoundsException expected, but NullPointerException was thrown.";

    assert false : "No exception expected.";
  }catch(Exception any){
    assert !(index < 0) : "IndexOutOfBoundsException expected, but " + any.getClass().getCanonicalName() + "was thrown.";
    assert !(obj == null) : "NullPointerException expected, but " + any.getClass().getCanonicalName() + "was thrown.";

      if(!(map != null)){
        throw any;
      }
    assert false : "No exception expected.";
  }
}
```

Appendix C

Metrics

This section lists the metrics conducted by the project of this work. It merely contains tables, listing *all* metrics and briefly describing their meanings, their names used in this thesis (if used), and their names used in the project's output.

Table C.1: General metrics.

Metric	*Identifier*	*Description*
All Files	f_{all}	All Java files of the API.
All Classes	c_{all}	All classes found in f_{all}.
Local Classes	c_l	All local classes in f_{all}. These are not accessible.
Anonymous Classes	c_{an}	All anonymous classes in f_{all}. These are not accessible.
Accessible Classes	c_a	All classes accessible for the wrappers.
Useful Classes	c_u	All classes containing possible specifications.
Non-Useful Classes	c_{nu}	c_a - c_u
Created Wrappers	w_{all}	All created wrappers.
All Javadocs	j_{all}	All Javadoc comments related to a method.
Useful Javadocs	j_u	All Javadoc comments in j_{all} containing tags.
Non-Useful Javadocs	j_{nu}	j_{all} - j_u

Table C.2: All metrics regarding the specifications that were extracted from the Javadocs. The *Metric* column reflects the description, which is, thus, omitted.

Metric	*Identifier*
Null Unknown Specifications	s_{nu}
Null Normal Specifications	s_{nn}
Null Any Exception Specifications	s_{na}
Null Specific Exception Specifications	s_{ns}
Range Unknown Specifications	s_{ru}
Range Normal Specifications	s_{rn}
Range Any Exception Specifications	s_{ra}
Range Specific Exception Specifications	s_{rs}
All Unknown Specifications	s_u
All Normal Specifications	s_n
All Any Exception Specifications	s_a
All Specific Exception Specifications	s_s
All Null Specifications	s_{null}
All Range Specifications	s_{range}
All Specifications	s_{all}

Table C.3: Tag-Related metrics. Note that other tables (e.g., table C.5) might contain tag-related metrics as well.

Metric	*Identifier*	*Description*
All Tags	t_{all}	All tags in j_{all}.
`@param` Tags	t_p	All `@param` tags.
`@throws` Tags	t_t	All `@throws` tags.
`@exception` Tags	t_e	All `@exception` tags.
`@deprecated` Tags	t_d	All `@deprecated` tags.
Faulty Tags	t_f	Tags that are not properly formatted.
Unsupported Tags	t_u	All unsupported tags.

Table C.4: Filter information.

Metric	*Description*
Filtered Tags	All filtered tags.
`@param`	All filtered `@param` tags.
`@exception`	All filtered `@exception` tags.
`@see`	All filtered `@see` tags.
`@return`	All filtered `@return` tags.
`@deprecated`	All filtered `@deprecated` tags.
`@inheritDoc`	All filtered `@inheritDoc` tags.
`@throws`	All filtered `@throws` tags.
Accessibility	Tags filtered due to accessibility.
Accessibility & No Javadoc	Tags filtered due to accessibility and no available Javadoc.
No Javadoc	Tags filtered due to no available Javadoc.
Accessibility & Deprecation	Tags filtered due to accessibility and deprecation.
Accessibility & Deprecation & No Javadoc	Tags filtered due to accessibility, deprecation and no available Javadoc.
Deprecation & No Javadoc	Tags filtered due to deprecation and no available Javadoc.
Deprecation	Tags filtered due to deprecation.

Table C.5: All unsupported tags that were found during the analysis. Note that some tags have typos (e.g., `@pararm`) and are, thus, not the detected by the software.

Tag	
`@spec`	`@long`
`@result`	`@since`
`@link`	`@implNote`
`@literal`	`@returns`
`@docRoot`	`@serialData`
`@see`	`@code`
`@implSpec`	`@xsl.usage`
`@out`	`@pararm`
`@value`	`@serial`
`@raises`	`@linkplain`
`@ocde`	`@params`
`@return`	`@apiNote`
`@throw`	`@beaninfo`
`@jls`	`@revised`
`@author`	`@baseID`
`@linke`	`@gsee`
`@inheritDoc`	`@comp`

Table C.6: All metrics related to parsing errors. The lower the value, the better the algorithm is able to infer specifications.

Metric	*Identifier*	*Description*
Parse Exceptions	e_p	All exceptions that occurred while parsing the input.
Null Parse Exceptions	e_{np}	Null-Related parsing errors.
Range Parse Exceptions	e_{rp}	Range-Related parsing errors.

Table C.7: All metrics regarding the created conditions. Every time such a type of a condition is detected, the condition is filtered.

Metric	*Identifier*	*Description*
Null-Checks Against Primitives	c_{np}	A parameter of primitive type cannot be checked against null.
Primitive-Checks Against Non-Primitives	c_{pnp}	A parameter of primitive type cannot be checked against a parameter of non-primitive type (i.e., an object). Unless the object is the wrapper class of the primitive parameter.
Parameter-Check Against Itself	c_{pp}	A condition, where a parameter is checked against itself, does not make sense.

Appendix D

Library Setup

Not only is the setup of a library necessary to fetch Javadoc comments for the analysis, but also to resolve any dependency issues that might otherwise occur (i.e., when using the source code only). Thus, this chapter focuses on explanations of how to set up different libraries, of which *all within this chapter* were used to evaluate the project of this work.

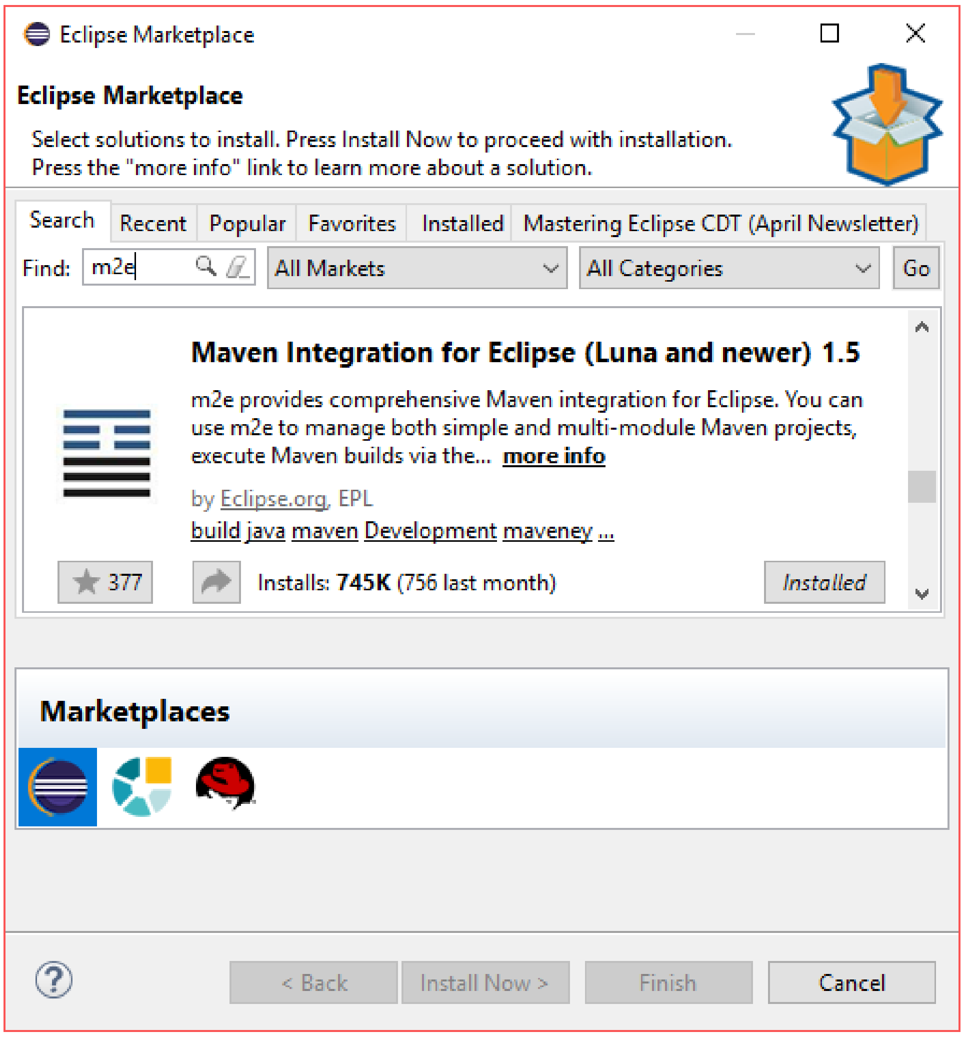
Note that even with a library set up, some dependencies may not be resolved. This is, because the import might not be explicitly stated in the original file, which leads to a missing import within the created wrappers. These missing imports need to be resolved manually, as Eclipse might not be able to infer the correct import (due to multiple classes with the same name in different packages).

D.1 Java

Setting up Java is rather complex, compared to other projects, such as *Apache Commons Collections* (see section D.2). In order to avoid or resolve any errors that might occur, the following step-by-step instruction may aid:

1. Open Eclipse.
2. Create a new project and name it properly.
    - For example, *jdk1.8.0_121* if that is the used JDK version.
3. Locate the file *src.zip*.
    - This file is within the Java installation folder.
    - For Windows[1]: C:\Program Files\Java\jdk1.8.0_121.
4. Extract the files from *src.zip* to the project's *src* folder.
5. Refresh the project within Eclipse (if necessary).

[1]No other OS was in use.


Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Find: m2e | All Markets | All Categories

Maven Integration for Eclipse (Luna and newer) 1.5

m2e provides comprehensive Maven integration for Eclipse. You can use m2e to manage both simple and multi-module Maven projects, execute Maven builds via the... more info

by Eclipse.org, EPL

build java maven Development maveney ...

377 | Installs: **745K** (756 last month) | Installed

Marketplaces

Figure D.1: A screen shot of the modal window that appears when navigating to Help > Eclipse Marketplace... (in Eclipse's menu). Search for *Maven*, find the version compatible with Eclipses version and simply click Install.

At this point, there is a chance (due to previous project setups or other reasons) that the library is all set up. If not, see the following steps on how to resolve these issues[2]:

1. Right-Click on the project (in Eclipse).
2. Use the context menu to navigate to Properties and click it.
3. Within the appearing modal window, find Java Compiler > Errors/Warnings > Deprecated and restricted API and change Forbidden reference (access rules) to Warning.

[2]Not all issues might be covered.

Program D.1: This code is semantic-wise equal to the original code. However, it resolves the issue as Eclipse is then able to resolve the type properly. Credits for this fix go to *Mario Winterer*. Note that the formatting was changed – for the sake of readability.

```java
@Override
@SuppressWarnings("unchecked")
public final <A> A[] toArray(IntFunction<A[]> generator){
  // Since A has no relation to U (not possible to declare that A is an
  // upper bound of U)
  // there will be no static type checking.
  // Therefore use a raw type and assume A == U rather than propagating
  // the separation of A and U
  // throughout the code-base.
  // The runtime type of U is never checked for equality with the
    component
  // type of the runtime type of A[].
  // Runtime checking will be performed when an element is stored in A
    [], thus
  // if A is not a
  // super type of U an ArrayStpreException will be thrown.
  @SuppressWarnings("rawtypes")
  IntFunction rawGenerator = (IntFunction) generator;
  Node<A> evaluateToArrayNode = evaluateToArrayNode(rawGenerator);
  return (A[]) Nodes.flatten(evaluateToArrayNode, generator).asArray(
    rawGenerator);
}
```

Once this is done, most of the errors should be gone (if this configuration was not set before). However, there might be more errors. Resolve those errors like this:

1. Find the package `com.sun.java.swing.plaf.gtk` and delete it.
   - The dependencies for this package do not come with the Java installation for Windows, as they are only *UNIX* related.
   - If deleting is not desired, the project's configuration could be used to exclude the folder from being read. However, this will not resolve the dependency errors.

Last but not least, another error – found during the setup for the project's evaluation – lies within the `java.util.stream` package of Java. This issue, however, can be fixed by re-writing the code in question as seen in program D.1.

D.2 Apache Commons Collections

This library is quite straightforward to set up. Since this project uses *Maven* it has to be installed before the actual setup can take place. Refer to figure D.1, to see how Eclipse's integrated marketplace can be used to install Maven. Once this is done, the actual installation can take place:

1. Create a new project within Eclipse.
2. Convert the project into a *Maven project*.
   - Right-Click on the project, navigate to Configure > Convert to Maven Project.
3. Download[3] the Apache Common Collections library.
   - Either clone the files directly into your project folder,
   - or download them manually and extract them accordingly.
4. Use the context menu to navigate to Maven > Update Project....

Note that Eclipse may detect some issues within the *pom.xml* file. Simply use another IDE to update the project, if required. *IntelliJ IDEA*[4] should be able to achieve the desired result and, therefore, finish the setup.

[3]https://github.com/apache/commons-collections
[4]https://www.jetbrains.com/idea/

Appendix E

CD-ROM/DVD Content

Format: CD-ROM, Single Layer, ISO9660-Format

E.1 PDF-Dateien

Path: /

_Thesis.pdf	Master's thesis regarding code-comment inconsistencies

E.2 Results

Path: /project/*

*	All relevant files of the project

Path: /reports/*

*.md	Evaluated reports

E.3 Online Sources

Path: /online-sources

*.jpg, *.png	Screenshots of online sources

E.4 Miscellaneous

Path: /images

*.pdf	Logo of the Upper Austria University of Applied Sciences
*.jpg, *.png	Original images

References

Literature

[1] Tom Fawcett. “An Introduction to ROC Analysis”. *Pattern Recognition Letters* 27.8 (June 2006), pp. 861–874 (cit. on p. 26).

[2] Z. P. Fry et al. “Analysing source code: looking for useful verb-direct object pairs in all the right places”. *IET Software* 2.1 (Feb. 2008), pp. 27–36 (cit. on p. 15).

[3] M. A. Harrison. *Introduction to Formal Language Theory*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978 (cit. on p. 6).

[4] J. M. Jazequel and B. Meyer. “Design by contract: the lessons of Ariane”. *Computer* 30.1 (Jan. 1997), pp. 129–130 (cit. on pp. 6, 7).

[5] Zhen Ming Jiang and Ahmed E. Hassan. “Examining the Evolution of Code Comments in PostgreSQL”. In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. Shanghai, China: ACM, 2006, pp. 179–180 (cit. on p. 4).

[6] Murat Karaorman, Urs Holzle, and John Bruno. *jContractor: A Reflective Java Library to Support Design by Contract*. Tech. rep. Santa Barbara, CA, USA, 1999. University of California at Santa Barbara (cit. on p. 16).

[7] Ninus Khamis, René Witte, and Juergen Rilling. “Automatic Quality Assessment of Source Code Comments: The JavadocMiner”. In: *Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems*. NLDB’10. Cardiff, UK: Springer-Verlag, 2010, pp. 68–79 (cit. on p. 15).

[8] R. Kramer. “iContract - The Java(Tm) Design by Contract(Tm) Tool”. In: *Proceedings of the Technology of Object-Oriented Languages and Systems*. TOOLS ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 295– (cit. on pp. 15, 45, 46).

[9] H. Malik et al. “Understanding the rationale for updating a function’s comment”. In: *Proceedings of the 2008 IEEE International Conference on Software Maintenance*. Sept. 2008, pp. 167–176 (cit. on pp. 1, 2).

[10] Carlos Pacheco and Michael D. Ernst. “Randoop: Feedback-directed Random Testing for Java”. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 815–816 (cit. on p. 18).

[11] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. “aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 11–20 (cit. on pp. 10, 11).

[12] Lin Tan et al. “/*Icomment: Bugs or Bad Comments?*/”. *SIGOPS Operating Systems Review* 41.6 (Oct. 2007), pp. 145–158 (cit. on pp. 9, 10).

[13] Shin Hwei Tan et al. “@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies”. In: *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. ICST ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 260–269 (cit. on pp. 5, 9, 11–13, 15, 17–19, 25, 38–41, 44, 45).

[14] Hao Zhong et al. “Inferring Specifications for Resources from Natural Language API Documentation”. *Automated Software Engineering* 18.3-4 (Dec. 2011), pp. 227–261 (cit. on pp. 13, 45).

[15] Yu Zhou et al. “Analyzing APIs Documentation and Code to Detect Directive Defects”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 27–37 (cit. on pp. 14, 36, 38, 41–43, 45, 47).

Online sources

[16] JUnit. *JUnit*. URL: http://junit.org/junit4/ (cit. on p. 18).

[17] Oracle. *Doclet Overview*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/doclet/overview.html (cit. on p. 17).

[18] Oracle. *Lesson: Exceptions*. URL: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html (cit. on p. 2).

[19] Oracle. *The Numbers Classes*. URL: https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html (cit. on p. 7).

[20] Cambridge University Press. *The Stanford Parser: A statistical parser*. URL: https://nlp.stanford.edu/software/lex-parser.shtml (cit. on p. 49).

[21] Cambridge University Press. *Tokenization*. URL: https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html (cit. on p. 47).

[22] Bill Pugh and Andrey Loskutov. *FindBugs*. URL: https://github.com/findbugsproject/findbugs (cit. on p. 15).

[23] Bill Pugh and Andrey Loskutov. *FindBugs^{TM} - Find Bugs in Java Programs*. URL: http://findbugs.sourceforge.net/ (cit. on p. 15).

[24] Lin Tan. *atComment*. URL: https://github.com/stan6/atComment (cit. on p. 11).