DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Nils-Jakob Kunze

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Maschinelles Lernen von Analyseregeln zur Erkennung von Fehlermustern

Author:	Nils-Jakob Kunze
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens, Dr. Benjamin Hummel, Dr. Lars Heinemann
Submission Date:	May 15, 2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, May 15, 2018 Nils-Jakob Kunze

Acknowledgments

Abstract

shorten! + mention software reuse?

Large software systems rely on frameworks and libraries to do the core work. The application programming interface (API) they provide should be designed to be as easy to use as possible. However, because of the great complexity and functionality provided by current frameworks, their APIs cannot be trivial. This and the fact that their documentation is often not kept up to date leads to errors in invoking the APIs.

In the context of object-oriented programming, one of these errors related to API usages are missing method calls. They occur when a developer correctly instantiates some object but forgets to call one or more of the necessary methods. We study a new method for detecting missing method calls in Java applications which was proposed by Monperrus et al. [7]. The main goal is to understand how good this method is at detecting code smells and true bugs when applied to a large software system.

more concrete research question

To investigate this, we analyze a large body of open source android applications. Besides manual evaluation of the findings in X randomly selected apps, we also apply an automatic benchmark which relies on artificially degrading existing code. We compare the originally proposed technique to some slight variations of it and find [...]. The results are kind of mixed, but point in the direction of this method being marginally useful.

describe results + concrete numbers

Contents

1 Introduction

insgesamt zu lang somewhere explain that “framworks” (libraries / plattforms) is all the same in this context and we will use THIS for now / use them all interchangeably because basically developer using some classes which he might not be familiar with this can also apply to classes nto from external but simply from within a big project where the dev doesn't know how to use it correctly zu viel repetition + remove the long example (Alice) kondensieren -> jemand der wissen will worums geht bekommt direkt und präzise einen überblick! max 2-3 seiten

add some personal anecdote somewhere?

-> reuse instead of outsource Larger software systems often outsource a significant amount of work to existing libraries or software frameworks, which expose their functionality through an application programming interface (API). Even if the designers of those APIs focus on making the interface as easy to use as possible, there is always a trade-off between usability and flexibility. Especially more complex libraries cannot provide a trivial API if they want to enable the programmer to facilitate it in the way most appropriate for their specific use case. Thus, often a lot of knowledge is required to invoke the API correctly, even more so in the case of large frameworks or powerful libraries. This can be because of constraints or requirements which are not clear from the outset, but which have to be heeded to avoid serious bugs or complications, or because of an involved interplay between different parts of the library. In the worst case, not even the documentation might contain this knowledge.

The fact that some APIs are not trivial and might have complex requirements makes it inevitable that there will be erroneous invocations of these APIs. [developer will make mistakes, . . .] Such errors can relate to parameter choice, method order, or many other factors (e.g., some methods must be invoked in an extra thread, some specific precondition has to be satisfied, or some setup work must be performed). An example, which comes to mind in Java, is an iterator on which the programmer calls `next()` without first checking with `hasNext()` if it even contains another object. Another one might be a class which overrides `equals()` without ensuring that an invocation of `hashCode()` always produces the same output for two equal objects.

include frameworks as well in some better way

Despite the fact that there might be numerous correct ways to use an API, often there are underlying patterns which the correct invocations have in common. These

patterns can take a lot of different forms and shapes, for example "call method `foo` before calling method `bar`", "if an object of type `Foobar` is used as a parameter to method `baz` condition X has to be fulfilled", "never call method `qux` in the GUI thread", etc. For the Java examples mentioned above they could take the form of "an object which implements `equals()` must also implement `hashCode()`", "if object A equals object B, their `hashCode()` must also be equal" or "a call to `next()` on an iterator should be preceded by a call to `hasNext()` to ensure that it actually contains another object".

Patterns like these are also called API usage patterns [12] and they can aid in detecting potential defects. If some code in a software project deviates (too much) from the usual patterns when using an API, this hints at a bug or problem or is at least a code smell which should probably be corrected. This makes it interesting to detect these unusual instances.

1.1 Motivation

As already mentioned above, there are many subtle mistakes a developer can make when invoking an API. In this work, we focus on one specific type of API usage problem, namely missing method calls, which can occur in the context of Object Oriented Programming (OOP). In OOP software an object of a specific type is usually used by invoking some of its methods. Some types will then have underlying patterns such as: "when methods `A` and `B` of type T_1 are invoked, then method `C` is called as well" or "methods `X` and `Y` of type T_2 are always used together". Given an object of this type T_1 on which only methods `A` and `B` are called, we can say that a call to `C` is missing, respectively if we have an object of type T_2 where only `X` or only `Y` is invoked, we can say that a call to the other is missing.

As an example consider a Button class in a GUI system. This button can either appear as a TextButton or as an ImageButton, where the first displays a word or short text regarding the button's functionality, while the second one only displays an image. Then one could imagine that the function `setText()` is usually called together with `setFont()`, whereas the function `setImage()` is called together with `setToolTipText()`. If a programmer writes some new code in which she creates a button and assigns it some text to display by calling `setText()`, but forgets to call `setFont()` as well, this would be a missing method call and a bug in the code. Imagine all buttons in the application using a unique and beautiful font, but this one button displaying the standard Comic Sans!

absatz sehr lang...

1.1.1 Examples from the Real World

Unclear APIs or frameworks and the resulting missing method calls are a real problem developers struggle with. Consider for instance this example from Monperrus et al. [7]: The developer Alice wants to create a dialog page for Eclipse. After some searching, she finds the corresponding class `DialogPage` in the API reference. She creates a new class using the Eclipse helper and ends up with the following boiler-plate code:

```
public class MyPage extends DialogPage {
    @Override
    public void createControl(Composite parent) {
        // TODO Auto-generated method stub
    }
}
```

Since nothing special was mentioned in the documentation of `DialogPage`, Alice simply creates the control by instantiating a `Composite` which contains all the widgets of `MyPage`. She knows she has to instantiate it with the parent as a constructor parameter:

```
public void createControl(Composite parent) {
    Composite mycomp = new Composite(parent);
    ....
}
```

However, in the first test run she gets the following error message along with an empty error log:

```
An error has occurred. See error log for more details.
org.eclipse.core.runtime.AssertionFailedException
null argument:
```

This is a typical case of implicit contracts which are not mentioned in the API documentation. Here, the Eclipse JFace user-interface framework expects that any class overriding `createControl` also ensures that the created control can be accessed later by calling `setControl`. Unfortunately, the documentation of `DialogPage` does not mention this and Alice assumed that registering the new composite is enough.[1] In this specific scenario, additionally, the resulting error message is not helpful at all, which makes debugging the problem more difficult and time-consuming.

[1] Actually in the current version[2] of the documentation this is mentioned, albeit not in the section directly related to `DialogPage`.

[2] `http://help.eclipse.org/oxygen/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/dialogs/IDialogPage.html#createControl-org.eclipse.swt.widgets.Composite-`

Because of this, Alice had to ask a question in the Eclipse mailing list to discover that this problem is related to a missing call to `setControl`. Only after receiving help she understands that it is necessary to call `this.Control(mycomp)` at the end of her `createControl` method. While her code finally works, she spent several hours of her valuable time on debugging a relatively simple problem related to just one method call which was missing. According to Monperrus et al. the described scenario regularly happened in the Eclipse newsgroup, thus showing that it is quite easy to make an error relating to missed method calls.

However, developers not only spend time during development on bugs related to missing method calls, but this kind of bugs also survive development and get checked into the code repository where they cause problems in the future. As an example, consider this[3] bug report on Apache Torque, an object-relation mapper for Java, which is intended to facilitate the access and manipulation of data stored in relational databases. This is the diff for the patch which was issued to fix the problem:

```
@@ -2307,7 +2307,7 @@
      */
     public Criteria andDate(String column, int year, int month, int date)
     {
- and(column, new GregorianCalendar(year, month, date));
+ and(column, new GregorianCalendar(year, month, date).getTime());
         return this;
     }

@@ -2332,7 +2332,7 @@
     public Criteria andDate(String column, int year, int month, int date,
             SqlEnum comparison)
     {
- and(column, new GregorianCalendar(year, month, date), comparison);
+ and(column, new GregorianCalendar(year, month, date).getTime(), comparison);
         return this;
     }
```

This is a very simple function, but the developer still forgot to call the `getTime` method call on the newly created `GregorianCalendar` object. Before the patch, invoking the `andDate` method would lead to an `SQLException`, because it constructs the SQL query in the wrong manner.

The example above is hardly the only bug report related to missing method calls in established software projects. In an informal review, Monperrus et al. found bug reports[4]

[3] `https://issues.apache.org/jira/browse/TORQUE-42`
[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=222305`

and problems[5] related to missing method calls in many newsgroups, bug trackers, and forums. The issues range from runtime exceptions to problems in some limit cases, but generally reveal at least a code smell if not worse. In addition to the informal review, Monperrus et al. [8] also did an extensive analysis of the Eclipse bug repository. First, they searched for syntactic patterns which they deemed related to missing method calls, such as: "should call", "does not call", "is not called" or "should be called". Manual inspection then confirmed 117 of the 211 (55%) thus obtained bug reports as indeed related to a missing method call.

This shows that even mature code bases can contain many bugs related to missing method calls, especially considering that this number is a lower bound on the total number of related bugs in the repository. After all, they might have missed some syntactic patterns or bugs which might not even have been discovered yet. Together, this makes it highly desirable to be able to automatically detect missing method calls in production code, not only to save expensive developer time but also to make maintenance cheaper and more manageable.

1.1.2 Detection - but how?

A simple and straightforward approach for this would be to build a set of hard-coded rules regarding method calls, such as:

- "always call `setControl()` after instantiating a `TextView`"
- "in Method `onCreate()` of classes extending `AppCompatActivity` always call `setContentView()`"
- "when calling `foo()` also call `bar()`"
- "when calling `next()` on an Iterator always call `hasNext()` (before)"

Well-crafted and thought-out rules along those lines could facilitate a very high precision in detecting missing method calls and contribute to better, more bug-free code. However, creating and maintaining a list of rules like this would require a tremendous investment of time and money, especially in a world where software is continually changing and improving. While this might be justified for large and important libraries, the necessary effort would also grow with the size of the library until it becomes completely infeasible.

To circumvent this problem, we would like to automatically detect locations in a code base where a method call is potentially missing without needing any further input besides the code itself. Such an approach would adapt to changing libraries without requiring additional work from a developer and could also be applied to proprietary code which is

[5] `https://www.thecodingforums.com/threads/customvalidator-for-checkboxes.111943/`

not open to the public. While the discovered locations will probably not be as accurate as those discovered by a hand-crafted list of rules, they could then be examined by an expert who would determine the severity of the finding and issue a fix if necessary.

additional advantages to automatic detection: continuous integration, adaptability, can be used on closed software -> express some more much better than fixed preprogrammed rules, can adapt to changing system, be specific for own not open library, etc

1.2 Contribution

In this thesis, we present a thorough reevaluation of the type usage characterization first introduced by Monperrus et al. [7] and further refined in a follow-up publication [8]. A type usage is the list of method calls which are invoked on an object of some type and occur in the body of some method (the context). The general idea behind the technique by Monperrus et al. is to check for outliers among the type usages by using the majority rule: If a type is used in one particular way many, many times (that is, in the majority of cases) and differently only one (or a few) times, this probably indicates a bug.

We evaluate this concept by applying it to a data set of more than 600 open source android applications and performing a manual evaluation of the results. We further experiment with small changes to their initial idea and put them under the scrutiny of an automated benchmark. Additionally, we compare the results of the manual evaluation against those of the automated one and consider what this means for future research.

mention some results!

In Chapter 2 we present previous work which goes in a similar direction.

FINALLY: Summary of the other chapters of this thesis

2 Related Work

new ideas: mention code smells and their origin?, less focus on "bugs" per se (see aDoctor paper) -> while the thesis title is related to bugs, I think I can / want to use smell as well Reference die belegt, dass es eine Korrelation zwischen bugs und smells gibt look in fontana2016comparing, has references to this in introduction

generally in this section: think about the tense I wanna use, present / past and if it works well! + stick to it!

The process of realizing that there exists a bug in a software system, identifying its cause and understanding the steps necessary to remove it, is difficult and time consuming. Especially for larger software systems automatically detecting low quality code and improving it can contribute significantly to the maintainability of the code and prevent bugs in the future. Thus, there has been a lot of interest in approaches for automatically detecting bugs or even just smells in code.

In this chapter we will present a number of different approaches for smell and bug detection. We start with some "conventional" methods which mostly rely on hard coded rules and pattern matching(?). We then give a quick overview of smell detection specifically related to android applications, because we are applying the method studied in this thesis to a number of android apps in the evaluation section. Finally, we look into techniques which attempt to learn rules and properties from the code they are analyzing instead of relying on rules which were given by the developer a-priori. Such techniques have the advantage of needing fewer manual oversight and changes when the system which is to be tested evolves.

2.1 Finding Code Smells with Static Analysis

Findbugs[1] is a static analysis tool that finds bugs and smells in Java code. It reports around 400 bug patterns and can be extended using a plugin architecture. Each bug pattern has its own specific detector, which uses some special, sometimes quite complex detection mechanism. These detectors are often built starting from a real bugs, first attempting to find the bug in question and then all similar ones automatically as well.

[1] `http://findbugs.sourceforge.net/`, in the meantime there has been a successor: `https://spotbugs.github.io/`

In their work Ayewah et al. [1] apply it to several large open source applications (Sun's JDK and Glassfish J2EE server) and portions of Google's Java codebase. Their premise is, that static analysis often finds true but trivial bugs, in the sense of these bugs not really causing a defect in the software. This can be because they are deliberate errors, occur in situations which cannot happen anyways or situations from which recovery is not possible. In their analysis Findbugs finds 379 medium and high priority warnings which they classify as follows:

links!

- 5 are due to erroneous analysis by Findbugs
- 160 are impossible or have little to no functional impact
- 176 can potentially have some impact
- 38 are true defects which have substantial impact, i.e. the real behavior is clearly not as intended

The takeaway is, that this kind of static analysis can find a lot of true bugs, however there will also be a lot of false positives among them and it can be difficult to distinguish them.

To alleviate the problem of a high false positive rate among the findings reported by Findbugs, Shen et al. [13] propose a ranking method which attempts to rank true bugs before less important warnings. It is based on a principle they call "defect likelihood". With the findings of a large project (in this case the JDK) as a basis, they manually flag each finding as a true or false positive, before using this data to calculate the probability that a finding is a true finding. This likelihodd can not only be calculated for one specific bug pattern but also across categories and types with variance as a tiebreaker. The resulting ranking can further be refined with user feedback, when it is applied to a specific project. In their evaluation on three open source applications (Tomcat, AspectJ and Axis) they compare their ranking against the default severity ranking of Findbugs, which is basically a hardcoded value for each bug type. Using cutoffs at 10%, 20%, 30%, ... of the total findings, they achieve precision and recall systematically better than the default ranking. This especially holds true for cutoff values around 50%.

2.2 Android specific Smell Detection

In Chapter 5 we are analyzing a large number of open source android applications. Thus, as a small overview of android related code smell detection consider the following papers.

Hecht et al. [6] presented an approach they call PAPRIKA which operates on compiled Android Apps, but tries to infer smells on the source code level. From the byte-code analysis they build a graph model of the application and store it in a graph database.

The nodes of the graph are entities such as the app itself, individual classes, methods and even attributes and variables. They are then further annotated with specific metrics relevant to the current abstraction level, e.g. "Number of Classes" for the app, "Depth of Inheritance" for a class or "Number of Parameters" for a method. Finally, they extract smells using hand-crafted rules written in the Cypher query language[2]. This enables them to recognize 4 Android specific smells and 4 general, OOP related smells.

To evaluate this approach they developed a witness application, which contains 62 known smells. Using the right kind of metrics, they where able to obtain precision and recall of 1 on this witness application. Further, they apply their tool to a number of free Android apps in the playstore and make some assertions about the occurrence of antipatterns in publicly available applications. One finding to emphasize is that some antipatterns, especially those related to memory leaks appear in up to 39% of applications, even "big" ones like Facebook or Skype.

After Reimann et al. [11] presented a catalogue of 30 Android specific code smells, Palomba et al. [10] developed a tool to detect 15 of them. It works on the abstract syntax tree of the source code and uses specifically coded rules to detect each of the smells. For the evaluation of their approach they examine 18 Android applications by comparing the results of their tool against a manually built oracle. The oracle was created by having two Masters students study the code of the applications in depth and manually flag offending locations. While they reach an average precision and recall of 98%, there are a number of cases in which their tool fails or yields false positives.

One especially interesting case is the smell of missing compression. When making external requests it is advisable to compress the data to save bandwidth. To detect places in the code where a request is made, but compression is missing, their hard-coded rule tests if one of the two popular compression libraries is used. However, recently there has been a new competitor compression library gaining in popularity. Their rule does not include it and, thus, falsely flags the usage of this library as the "missing compression" smell. This is a great example of a case where hard-coded rules fail, because the are not kept up to date (new library becomes popular, interface changes, ...), do not consider a special case or otherwise hanging criteria.

2.3 Inferring Properties from the Code at Hand

As the results of the previous paper showed, it can be difficult and costly to keep hard coded detection rules up to date and relevant. Because of this, there has been a lot of interest in approaches which adapt automatically to changing requirements by learning rules from the source code and looking for violations of those rules.

[2] `https://neo4j.com/developer/cypher-query-language/`

[Why is this potentially better than other approaches (actually learning an api vs static rules comes to mind), also mention some stuff about recommender systems in general (it is the official thesis topic after all...)]

Mention [3] as probably the first paper which proposed the general idea behind DMMC -> learning from the code at hand, instead of using static predefined rules

Mention MUBench?! -> useful classification of api misuses Dataset of bugs related to API misuses ...

General: read the summary paper [12] again and check which approaches are there and which could / should be mentioned + check to read stuff... + for more information: check this summary paper / the 2017 one? / both?

Remember the general machine learning approaches which were not super successful, but at least learned a LOT especially like general code smells for example

2.4 Previous work on detecting missing method calls / object usages

There have been a number of works concerned with finding patterns in object usages and using those to find potential bugs. Most relevant for this work are two papers by Monperrus et al.[7][8] which introduced the notion of almost similarity and are the primary inspiration for this work. Their method considers the invocation of methods on an object of a particular type in a particular context a type usage. Here, the context is nothing more than the name of the method in which the type usage occurs together with its signature (the types of the method parameters). After mining a list of all the type usages present in a code base, they relate the number of exactly equal type usages to the number of almost equal ones. Exactly equal means that context, type and method list are exactly identical, while almost equal means the same, only that the method list can contain one additional method. This technique is explained in detail in Chapter 3.

results of monperrus et al.:

Before this, Wasylkowski et al. [14] introduced a method to locate anomalies in the order of methodcalls. First, they extract usage models from Java code by building a finite state automata for each method. The automata can be imagined similarly to the control flow graph of the method with instructions as transitions in the graph. From these they mine temporal properties, which describe if a method A can appear before another method B. One can imagine this process as determining if there exists a path through the automata on which A appears before B, which in turn implies that a call to A can happen before one to B. Finally, they are using frequent itemset mining [5] to combine the temporal properties into patterns.

In this work an anomaly also occurs when many methods respect a pattern and only a few (a single one) break it. In their experiments they find 790 violations when analyzing an open source program and find 790 violations. Manual evaluation classifies those into 2 real defect, 5 smells and 84 “hints” (readability or maintainability could be improved). This adds up to a false positive rate of 87.8%, but with an additional ranking method they were able to obtain the 2 defects and 3 out of 5 smells within the top 10 results.

In a related work Nguyen et al [9] use a graph-based representation of object usages to detect temporal dependencies. This method stands out because it enables detecting dependencies between multiple objects and not just one. The object usages are represented as a labeled directed graph where the nodes are field accesses, constructor or method calls and branching is represented by control structures. The edges of the graph represent the temporal usage order of methods and the dependencies between them. Patterns are then mined using a frequent induced subgraph detection algorithm which builds larger patterns from small patterns from the ground up, similar to the way merge sort operates. Here an anomaly is also classified as a “rare” violation of a pattern, i.e. it does not appear often in the dataset in relation to its size. In an evaluation case study this work finds 64 defects in 9 open source software systems which the authors classifies to 5 true defects, 8 smells and 11 hints, which equals a false positive rate of 62.5%. Using a ranking method the top 10 results contain 3 defects, 2 smells and 1 hint.

mention more related work from detecting mmc paper?

2.5 Other Ideas

change title

[15] already mentioned problem of incorrect API documentation this often happens when a developer adapts or improves the code but forgets to change the documentation as well they propose an automated approach to detecting “defects” in the api documentation defect = two cases: either api is incomplete in describing some constraints or description exists, but does not match the situation in the code assumption: code is correct (has been extensively tested)

too detailed?! technique: static analysis of the code, then pattern based natural language processing -> first oder logic formula same for the api document: part of speech tagger (find specific parts which usually indicate some specific situation) + some heuristics -> restrictions and constraints -> FOL the resulting first-order logic formula is fed into an SMT (satifiablity modulo theories) solver [2] to detect inconsistencies (if fol of code <> fol of doc) is able to detect four classes of documentation defects

results: a prototype implementation applied to java.awt and java.swing finds 1419 defects of which 81.6% are true positives (check by manual evaluation of some masters

students) + applied to some other packages but with the heuristics extractd from the previous two, find 1188 defect of which 659 are TP (precision 55.5%). -> lots of defects in a supposedly well documented JDK

[4] different approach to code smells: machine learning -> use manually evaluated code + tagged at appropiated places to train machine learning system this work supports 4 basic smells (data class, large class, feature envy, long method) compares 16 different approaches, all have a good performance, J48 and random forest are best, support vector machines are worst [long enough? i think yes?]

questionable: how useful is this? these smells are also detectable using "normal" techniques?

3 Detecting Missing Method Calls

analogy: remove with waiter irgendwo erläutern PART 1 ist background (Foundation: Majority rule) 3.2 = mein teil

better intro...

As mapped out in Section 1 bugs related to missing method calls occur in real software systems. These bugs are often related to complicated APIs and frameworks, and we would like to detect them. The idea is to detect patterns in correct usages which will enable us to single out the faulty ones.

In this chapter we will first give a proper explanation of the method proposed by Monperrus et al. [7][8] based on the majority rule. This includes a proper qualification of the notions of type usages, as well as their equality and almost equality, which we already mentioned in the previous chapter. We will then explain how the strangeness score, which describes the degree to which a particular type usage is different from its relevant neighbors, is calculated, before going into detail how the potentially missing calls are determined.

In the second section of this chapter, we explain some minor tweaks to the type usage notion, whose ability to detect missing method calls we explore further in Chapter 5.

more details? -> when proper contents are decided + do not mention chapter 5

3.1 Foundation: The Majority Rule

make sure that chapter intro + this section intro work well together

The intuition behind the majority rule is that a piece of code is likely to contain a defect if there are none or very few equal instances, but a lot of slightly different instances. It then seems likely that the code which appears a selected few times only should be adapted in a way to match the majority of slightly different instances.

-> remove?! / use with methods! An analogy which explains the idea well goes as follows. Imagine being the waiter preparing the tables in a restaurant. There are 100 seats, and each of them is equipped with a plate, a knife on the right and a fork on the left. However, of these seats only 99 have a spoon above the plate, and there is one seat s which is missing the spoon. Then it is highly likely, that this one exception to the rule

"each seat should have a plate, a knife, a fork and a spoon" is a mistake, and you should add the spoon to the single seat s where it is missing.

The remainder of this section extends this idea to object-oriented software and formalizes the different notions required to implement a system detecting such places. In this context each piece of cutlery (knife, spoon, fork) can be imagined as a method call, i.e., we extend the concept of the majority rule to types and method calls.

3.1.1 Type Usages

make paragraphs instead of subsections?

Generally speaking, type usages are an abstraction over the analyzed code. They ignore the order of method calls and are only concerned with the list of method calls invoked on some object. The critical information which is associated with a type usage is thus as follows: the type of the object, the list of methods invoked on it and the context in which the object is used. We define the context as the name of the method in whose body the type usage appears together with the type of its parameters.

Formalized, given some code the type usages are extracted as follows: For each variable x, note its type $T(x)$ and the context $C(x)$ in which it occurs. Additionally, save the list of methods which are invoked on x within $C(x)$: $M(x) = \{m_1, m_2, \ldots, m_n\}$. If there are two variables of the same type, this results in two type usages being extracted (unless they refer to the same object, more on this in Section 4.1).

```
class A extends Page {
    Button b;

    Button createButton() {
        b = new Button();
        b.setText("hello");
        b.setColor(GREEN);
        ... (other code)
        Text t = new Text();
        return b;
    }
}
```

```
T(b) = 'Button'
C(b) = 'Page.createButton()'
M(b) = {<init>, setText, setColor}

T(t) = 'Text'
C(t) = 'Page.createButton()'
M(t) = {<init>}
```

Figure 3.1: Example illustrating the Extraction of Type Usages (taken from [8])

As an example, consider the code in Figure 3.1. It contains two type usages, one of type `Button` and another one of type `Text`. The context for both is `createButton()`. The methods invoked on the `Button` object are initialization (`new`), `setText` and `setColor`.

The `Text` object, on the other hand, is only initialized. Given the two variables b and t as input, the corresponding values of $T(x)$, $C(x)$ and $M(x)$ are shown on the right hand side.

3.1.2 Exact and Almost Similarity

todo: explain better what a SIMILAR method is (if so mention what it is) + SAME method

To detect type usages which are anomalous by the majority rule, we now need a notion what it means for two type usages to be exactly similar and only almost similar. Informally, we say that two type usages are exactly similar, iff they have the same type, the type usage appears in a similar method (i.e., the context is identical) and if their list of method calls is identical. We call these type usages "similar" instead of "equal" because several things are indeed not the same: variable names, surrounding or interspersed code, method order and the concrete location (class) are all disregarded and might indeed differ.

some better explanation WHY we need this / add it in section intro?

The notion of almost similarity is also not far away. We say that a type usage y is almost similar to a given type usage x, if they share the same type and context and the list of method calls of y is the same as the method calls of x plus one additional call.

Consider the example code snippets in Figure 3.2. The type usages of `Button` in classes `A` (top left) and `B` (top right) are exactly similar, because they not only appear in the similar method `createButton`, but also invoke the same methods `setText` and `setColor`. Observe that variable names and method parameter play no role in this. Additionally, neither the order of methods nor any additional code (which does not call a method on the `Button` objects) does have any influence on it. The type usage in class `C` (bottom left) is almost similar to the ones in `A` and `B`, because besides the two methods `setText` and `setColor` it also invokes one additional method: `setLink`. Finally, the type usage in class `D` (bottom right) invokes **two** additional methods in comparison to the ones in `A` or `B`, meaning that it is not almost similar to them. However, it is in fact almost similar to the type usage in class `C`, since it only invokes one additional method: `setTooltipText`.

To formalize these notions, we define two binary relationships over type usages. The relationship for exact similarity is called E' and we say that two type usages x and y are exactly similar if and only if:

$$\begin{aligned} xE'y \iff & T(x) = T(y) \ \wedge \\ & C(x) = C(y) \ \wedge \\ & M(x) = M(y) \end{aligned}$$

Given this, the set of exactly similar type usages in relation to x is defined as:

$$E(x) = \{y \mid xE'y\}$$

```
class A extends Page {
    Button createButton() {
        Button b = new Button();
        ... (interlaced code)
        b.setText("hello");
        ... (interlaced code)
        b.setColor(BLUE);
        return b;
    }
}
```

```
class B extends Page {
    Button createButton() {
        ... (code before)
        Button aBut = new Button();
        ... (interlaced code)
        aBut.setColor(RED);
        aBut.setText("goodbye");
        return b;
    }
}
```

```
class C extends Page {
    Button myBut;
    Button createButton() {
        Button myBut = new Button();
        myBut.setColor(ORANGE);
        myBut.setText("Great Company!");
        myBut.setLink("https://www.cqse.eu");
        ... (code after)
        return b;
    }
}
```

```
class D extends Page {
    Button createButton() {
        Button tmp = new Button();
        tmp.setLink("https://slashdot.org/");
        ... (interlaced code)
        tmp.setText("All the news");
        tmp.setColor(GREEN);
        tmp.setTooltipText("Click me!");
        return b;
    }
}
```

Figure 3.2: Examples of similar and almost similar Type Usages (also inspired by [8])

Observe that this relation holds true for the identity, i.e. $xE'x$ is always true and $\forall x : x \in E(x)$ (and thus $|E(x)| \geq 1$).

For almost similarity, we define the relation A' which holds if two type usages are almost similar. A type usage y is almost similar to a type usage x iff:

$$\begin{aligned} xA'y \iff & T(x) = T(y) \ \wedge \\ & C(x) = C(y) \ \wedge \\ & M(x) \subset M(y) \wedge \\ & |M(y)| = |M(x)| + 1 \end{aligned}$$

Similarly to $E(x)$ we define $A(x)$ as the set of type usages which are almost similar to x:

$$A(x) = \{y \mid xA'y\}$$

In contrast to $E(x)$, $A(x)$ can be empty and $|A(x)| \geq 0$.

mention it is NOT symmetric!

It is possible to further refine the definition of almost similarity. Instead of looking for type usages which have the same method calls plus one additional one, it is also possible to

consider at those type usages which feature the same method calls plus k additional ones. Then, in the definition of A the last line changes as follows: $|M(y)| = |M(x)| + k, k \geq 1$

mention ignoring the context here?!

Note that given a codebase with n type usages the sets $E(x)$ and $A(x)$ for one given type usage x can be computed in linear time $O(n)$ by iterating once through all type usages and checking for similarity or almost similarity.

3.1.3 The Strangeness Score

Recall the assumption behind the majority rule: A type usage is abnormal if a small number of type usages is exactly similar, but a large number are almost similar. Informally, this means a few places use this type in exactly the same way, but a significant majority use it in a way which is slightly different (by exactly one method call). Assuming the majority is correct, the type usage under scrutiny is deviant and potentially erroneous.

To capture concretely how anomalous an object is, we need a measure of strangeness. This will be the strangeness score. It allows to order type usages by how much of an outlier they are and thus to identify the "strangest" type usages, which are most interesting for evaluation by a human expert.

Formally, we define the S(trangeness)-Score as:

$$\text{S-score}(x) = 1 - \frac{|E(x)|}{|E(x)| + |A(x)|}$$

To see that this definition makes sense, consider the following extreme cases: Given one type usage a without any additional exactly similar or almost similar type usages, it will have $|E(a)| = 1$ and $|A(a)| = 0$. Then, we can calculate the strangeness score: $\text{S-score}(a) = 1 - \frac{1}{1} = 0$. So a unique type usage without any "neighbors" to consider is completely normal and not strange. On the other hand, given a type usage b with 99 almost similar and no other exactly similar type usages, we have $|E(b)| = 1$ and $|A(b)| = 99$. Which results in a strangeness score of $\text{S-score}(b) = 1 - \frac{1}{1+99} = 0.99$. Following intuition, such a type usage is very strange, and indeed, the S-score supports this.

3.1.4 Which Calls are missing?

Only detecting the type usages which are outliers is not enough. We would also like to present the developer with some candidate suggestions, what might be the method call that is missing.

The intuition for this is to look at all the almost similar type usages and the additional method call that they invoke. Then take the frequency of each unique method call among them and take this as the suggestion likelihood.

do not like the formulation, improve!

Formally, the calls we can recommend for a type usage x are the calls that are present in the type usages in $A(x)$ but not in $M(x)$. The set of these methods shall be called $R(x)$ and is defined as follows:

$$R(x) = \bigcup_{z \in A(x)} M(z) \setminus M(x)$$

Now for each of these methods m in $R(x)$, we can calculate their likelihood as follows:

$$\phi(m, x) = \frac{|\{z \mid z \in A(x) \wedge m \in M(z)\}|}{|A(x)|}$$

This is identical to the share of type usages which use this method among all the type usages which are almost similar to x. Observe that this definition also works when $M(x)$ is empty or x is `this`.

$T(x) =$ `Button`
$M(x) = \{$<`init`>$\}$
$A(x) = \{a, b, c, d, e\}$
$M(a) = \{$<`init`>, `setText`$\}$
$M(b) = \{$<`init`>, `setText`$\}$
$M(c) = \{$<`init`>, `setText`$\}$
$M(d) = \{$<`init`>, `setText`$\}$
$M(e) = \{$<`init`>, `setFont`$\}$

$R(x) = \{$`setText`, `setFont`$\}$
$\phi(setText, x) = \frac{4}{5} = 0.8$
$\phi(setFont, x) = \frac{1}{5} = 0.2$

Figure 3.3: Example for computing the Likelihood of missing Calls

again reference source of this example!!!

To illustrate these definitions, consider the example in Figure 3.3. The type usage in question is denoted with x, operates on a `Button` object and the only method that is invoked is the initialization. There are five almost similar usages denoted as a, b, c, d and e. Four of them (a through d) have a call to `setText` after the initialization. The last one, e, has a call to `setFont` instead. Thus, we can recommend the methods `setText` and `setFont` with a likelihood of $4/5 = 80\%$ and $1/5 = 20\%$ respectively.

3.1.5 Ignoring the context

the idea behind using the context in the definition of similarity and almost similarity is that types might be used in different ways, depending on the method one is in an

example would be FileInputStream which is rarely opened and closed in the same method Monperrus et al find that not using the context adds a lot of noise (not relevant items in E and A) and using it improves the precision of the precision of their experiments

the context drastically reduces the number of type usages which are considered in the almost / similarity this can facilitate that "patterns" appear, which in truth are only artefacts of randomness

one might also question the general assumption behind using the context ARE types actually used in different ways depending on the name of the function they are used in? this assumption might hold for some types and some systems, but is it generally true? We will shed some more light on this in Section ...

Formally this method shall be called $sub_noContext$ definitions of E and A are changed as follows [include or not neccessary and just describe that the context equality definition is removed from them (i think this)] s score, phi stay the same, only that they now use the new relations $sub_noContext$

3.2 Extensions

The previous section summarizes the ideas of Monperrus et al. In this section we will explore some possible modifications to their method (my work) in the evaluation we will explore if any of them performs better than their method

theoretical explanation of the different approaches

3.2.1 Class-based merge

kind of following the ideas behind not using the context: does it make sense to look at type usages with a method granularity (observe: even if ignoring the context, the type usages will still be extrated on a per method basis! -> here or above or both?) who says that it doesn't make a lot more sense to look at CLASS granularity type usages often this already includes the whole lifetime of an object, but even if not, includes a way bigger window into the objects utilization if a method is missing from there, it is even more likely that it's completely missing and thus reveals an error

mine the type usages at method level as before but before calculating the strangeness score, merge all type usages which are of the same type and belong to one class note that this will merge them even if the original typeusages originate from different objects (following the object lifetime through the whole class would be too expensive) with the new tus, calculate strangeness score etc as before

let $Cls(x)$ be the class from which a type usage was extracted. Then the new set of type usages is made up as follows: $x_0, \dots x_n = alltuswhereT(x_0) = T(x_1) = \dots =$

$T(x_n) and Cls(x_0) = Cls(x_1) = \cdots = Cls(x_n)$ then the new resulting tu will be: $x' - > T(x') = T(x_0), C(x') = Union(C(x_i)) and M(x') = Union(M(x_i))$

for almost / exact similarity we will choose the version which ignores the context, so C(x') is not actually that important mention some effects (such as much smaller dataset, blabla?)

3.2.2 Investigate different Anomalies

While a missing method call might be the error type taht comes first to mind and probably the most frequent one this general technique (majority rule!) can also easily adapted to look for two different anomalies, related to method calls

namely, it can investigate if there is a superfluous method call somewhere (would expect this not too work too well...) or simple the WRONG method call, that is instead of calling method X the developer is erroneously calling method Y

Superfluous Method

basically the reverse of the method proposed by monperrus et al check if something is calle dwhich should not be called

unclear, why would we even wanna do this? I feel like it will mostly find outliers which are outliers INTENTIONAlly maybe best rationalization: it is there before trying it we don't actually know if its bad, could be that this is actually a type of error which has some impact

(somewhere example?)

Adapting the system to do this is rather simple only need to adapt the definition of A(x)

new equation:

$$A_{superfluous}(x) = \{x | y A' x\}$$

recall that the almost similar relation is not symmetric observe that here x and y are flipped in the application of A'

this definition of almost similarity considers all those type usages which call one method LESS than the input type usage to be almost similar the definition of the S-score can stay the same, as can the definition of phi only difference is that now it indicates which method is most likely to be TOO much

Wrong Method

Further extending the idea in the previous section, and thinking consequently Developers might MISS a method call, the might ADD a useless one, but of course, they could also simply choose the WRONG one

example? maybe some sort of conversion bug?

especially if the developer is not all too familiar with a given framework bad method-names / documentation -> simply pick the wrong method for the job

of course can be that this results in noticable bugs / errors much faster then missing method calls, but again we don't know (do i even wanna mention this?)

again need to change the definition of almost similarity as follows: xA'y <> T(x) = T(y) and C(x) = C(y) and |M(x)| = |M(y)| and exists a set Sub(x, y), such that |Sub(x,y)| = |M(x)| - 1, and Sub(x,y) subset of M(x) and M(y) -> can I define this a bit better or not? + what kind of description? Type and context again are the same require that all but one methods of M(x) and M(y) are identical

3.2.3 Working with the Inheritance Hierarchy

I guess keep this until very last (somehow doubt I'll have the time to look into this)

4 Implementation

details about implementation and pitfalls that had to be overcome

operates by statically analyzing a piece of software (not on the source code, but rather on the compiled byte code -> why: easier)

4.1 Bytecode Analysis

maybe rename section?

soot as analysis framework -> what is soot why bytecode over sourcecode ...

inaccuracy about "object" vs "variable" of a particular type? -> there is the option for some analysis but it is super expensive + seems leaky -> not doing it but it will fix this (more or less)

4.2 Procedure

1. extract tus from software 2. for every tu: a) search for tus which are EXACTLY similar b) search for TUs which are almost similar c) compute the strangeness score d) extract list of potentially missing calls 3. output a list of tus, sorted by S score, the the ones with a score above XX are considered to be an anomaly + their missing calls

4.3 Improvements

using a database + flexible python benchmarking / analysis -> proper overview of resulting system: java analysis -> db -> python -> somewhere a proper overview of the concrete steps that are taken + explanation of them?! (maybe in extra chapter BEFORE this one?) - similar to monperrus2010 p7

explaining all the work i did first reading their code, coming across a couple of discrepancies between code and paper, later check if everthing still works (evaluation) refactoring everything + saving stuff to database building python infrastructure for analysis

4.4 Dead Ends

why some solutions where discarded (eg pure database / could be revisited if it turns out to be the best anyways - performance) Better anomaly detection (is the anomality rule actually a GOOD measure for this kind of anomalies, or should we use something totally different?)) clustering detector try (+ hypersphere idea?)

static functions evaluation! something to fix the dotchaining problems

4.5 Benchmark

building benchmarking infrastructure downloading + automatically analyzing android apps (in evaluation section?)

some changes that had to be made to the analysis framework for android analysis

5 Evaluation

roter faden: results of manual analysis vs benchmark (ie. is context better than no context) take their method + analyze it on a qualitative + quantitative basis on a big android dataset

does this order make sense or rather split by research question?

5.1 Methodology

5.1.1 Qualitative Evaluation / Android Case Study

android apps, blabla manual review of top 50 findings for each "method" -> which are the different methods?

5.1.2 Automated Benchmark

degrading type usages + checking if they will be detected

sometimes in even further degraded settings (degrade more than one)

general idea behind evaluation method and what we are trying to detect actually it's a sort of simulation (see recommender system book p. 301f) - micro vs macro evaluation, ...imitation of the real system of software development using a much simpler system, namely dropping method calls obvious question: how similar are the such created mmcs to mmcs in the wild?

Metrics explanation - Precision, Recall additional metrics from recommender systems book p.245f (robustness, learning rate - p. 261) performance -> training time + performance when working

5.2 Results

Which questions are we trying to answer with the evaluation? quick overview + method to do this

qualitative evaluation: are the findings useful in practice? quantitative evaluation: given some assumptions, how useful can we expect this technique to be? robustness?

Qualitative vs automatic same results? make sure I will be able to a) answer those questions and b) the answers are "interesting"

große frage: bringt das Verfahren etwas?

for each question: explanation of question, then results + interprettation

5.2.1 Dataset Overview

rausziehen (extra section)

what kind of dataset do we have? (how many apps, how big, how many TUs, how are the TUs split between the apps -> size of applications) what are typeusages in Android apps like? (length of method list, percentage on Android framework, percentage on other stuff, etc)

general data about avg tus related to different settings etc -> what does our dataset even consist of?

look into and understand the type of data i have -> how often small inputs, how often big, ... -> paint a bunch of graphs of the pkl results! (histogram of tu list sizes, etc) (correctly bin histogram!) tu method list sizes - also check their paper, q mas?

indea behind using android: do real software systems actually behave in this way ie: is it "necessary" to use classes in the same way probably: most likely present in GUI systems common way of using some classes -> investigate via Android (+ additional advantages: rich open source community, java, blabla)

5.2.2 RQ1: Does the Strangeness Score behave as expected on Android Apps?

-> ist keine forschungsfrage, gehört zur vorherigen section! (the behavior that I'm expecting is basically a mathematical necessity!)

do the general assumptions hold? (most tus have a low score, most apps have few findings, etc) -> answer with graphs related to general score verteilung -> most tus are "normal", a few are "abnormal" what do those assumptions mean? -> they expect some kind of uniformity to the type usages, probably mostly present in GUI etc frameworks verteilung of strangeness scores, histogram of it

Frage: Ist das ganze überhaupt schnell genug? -> JA! (auslesen kosten +)

5.2.3 RQ2: How "useful" is this technique?

wie relevant sind diese obtained results in der praxis (as determined by fp vs tp)

change title...

qualitative evaluation findings: how many true findings in relation to false positives does it find + average number of findings per app (remember that the findings now are for all 626 apps)

pick a couple of high scoring findings and explain the failure modes (eg doesn't take branching into account, etc) + of course the REAL bug + some real smells

and of course regarding the "results" always qualify -> this only for android, open source apps, manual evaluation, blabla, so we never actually know for sure

think if i should totally change the manual eval: take random sample of Apps and there look at all the anomalous (>.9) findings (kind sounds a bit smarter?)

RQ2.1: Is there a noticeable difference between the different variants?

each for the slightly adapted forms (no context, class merged, etc)

RQ2.2: Given a known missing method call bug, can this approach detect it?

only if time!

reverse test: find actual missing call in bug database -> can I find it using the approach and is it among the top findings?

5.2.4 RQ3: How meaningful are the benchmarking results?

"meaningful" not a good word, rather already describe what I'm doing (ie do the bnechmark results align with the results of the manual evaluation oä)

does it make sense to evaluate this method using the benchmarking as proposed by Monperrus et al.? -> comparison between manual evaluation of findings vs automatic benchmark over the different techniques used (context, no context, class merge, ...)

first present benchmark results of different techniques (context, class merge, ...) then relate to qualitative results

are the results better if we leave out the context / merge on a per class basis / FOR DIFFERENT Ks!...

5.2.5 RQ4: How robust is this technique in the face of erroneous input data?

I would add this, even if the results of comparison with manual evaluation are relatively negative / don't say anything worauf is robustness bezogen? -> results quality or smth else?

Robustness explanation wie viel brauche ich um sinnvoll viel zu lernen - hälfte der leute machen fehler -> wie robust brauchst du die daten / ist das verfahren ! what is needed for it to work (lines of code, feature richness, etc) problem: how to evaluate when it is "working" vs not working -> again simulate "missing methods", compare performance?

additional test: for true findings, try to throw away parts of the data -> when can we not find this anymore -> mathematical answer! / instead of throwing away: introducing random errors in the related tus

5.2.6 RQ5: What are the requirements for the input size?

benny didn't like "requirements"... ie: how big of a codebase do we need to be able to "learn". When can we apply this to a project which does not rely on some well known open source framework (Android) where it is easy to gather additional data, but on for example an in-house-closed-source library oä.

extremely hard to answer especially if the findings regarding benchmark validity are more or less negative. however, even if I cannot give experimental answers, I can at least give some lower bounds + thoughts (need at least 10 tus within the same "category" to even reach a score > 0.9, from practical results in qualitative analysis, that is probably not enough, blabla)

5.3 Discussion

5.4 Threats to Validity

explain problems with the automatic evaluation how related are the degraded TUs to missing method calls you can find in the wild? improving the metrics by dropping cases where we know we won't find an answer

also mention runtime! wie sehr sind die resultate aus der Android case study 1. Wahr (subjektive bewertet etc) 2. Übertragbar auf andere Anwendungsfälle(sind open-source programme of vergleichbar mit professionellen, Android eco System mit anderen, etc) ...

6 Conclusion

general takeaway about suitability of this method + reasons WHY I would say so

new method is amazing and way better in case xy but worse in zz robustness has shown this and that future research would be interesting in the direction of ...

6.1 Contribution

-> even have this section?

Datenset generiert, zumindest was über software sturktur lernen, bla !

6.2 Future Research

apply this anomaly detection method to other features (implements, overrides, pairs of methods, ...) Reihenfolge von Methoden - hat gut funktioniert und wäre potentiell nützlich, aber kleines subset + lots of research already done order of method calls? (probably should wait for empirical results - how long are typical method lists (state explosion, ...) / latice solution?) efficient update (only update files touched by change + the scores for affected type usages) - maybe database view for scores will already do this automatically? higher precision by some means? (clustering, etc) -> better anomaly detection algorithm! (but difficult / constrained by the input size (few methods,...) -> refer to dataset analysis section) Performnance (shouldn't be a problem?) . or potentially later look into other features (like implements/override), pairs of methods, etc

Bibliography

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. "Evaluating static analysis defect warnings on production software." In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2007, pp. 1–8.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, et al. "Satisfiability modulo theories." In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[3] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. "Bugs as deviant behavior: A general approach to inferring errors in systems code." In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 57–72.

[4] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. "Comparing and experimenting machine learning techniques for code smell detection." In: *Empirical Software Engineering* 21.3 (2016), pp. 1143–1191.

[5] J. Han and M. Kamber. *Data mining: concepts and techniques*. Second. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[6] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. "Detecting antipatterns in android apps." In: *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. IEEE. 2015, pp. 148–149.

[7] M. Monperrus, M. Bruch, and M. Mezini. "Detecting missing method calls in object-oriented software." In: *European Conference on Object-Oriented Programming*. Springer. 2010, pp. 2–25.

[8] M. Monperrus and M. Mezini. "Detecting missing method calls as violations of the majority rule." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (2013), p. 7.

[9] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. "Graph-based mining of multiple object usage patterns." In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 383–392.

[10] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. "Lightweight detection of Android-specific code smells: The aDoctor project." In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE. 2017, pp. 487–491.

[11] J. Reimann, M. Brylski, and U. Aßmann. "A tool-supported quality smell catalogue for android developers." In: *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*. Vol. 2014. 2014.

[12] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. "Automated API property inference techniques." In: *IEEE Transactions on Software Engineering* 39.5 (2013), pp. 613–637.

[13] H. Shen, J. Fang, and J. Zhao. "Efindbugs: Effective error ranking for findbugs." In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE. 2011, pp. 299–308.

[14] A. Wasylkowski, A. Zeller, and C. Lindig. "Detecting object usage anomalies." In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 35–44.

[15] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. "Analyzing APIs documentation and code to detect directive defects." In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, pp. 27–37.