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

don't forget! mention: torben 4 grammarly the readers maybe thesis workshop (parents?, roommates, ...) supervisors

I would like to thank the preliminary readers of this thesis XXX, YYY and the participants of the thesis writers workshop for their effort and valuable feedback. Without you this thesis would contain a heap of errors and the style would be bleak. I would also like to extend my gratitude to Torben, who helped me out with a useful software subscription, and of course my supervisors, Benjamin and Lars, who helped me stay on track and had useful questions and insights when they were necessary. Finally, I am grateful to my parents for supporting me and to my roommate Caroline for keeping me sane.

Abstract

smth like: developers make mistakes when invoking apis. these mistakes can be because of xYZ (already don't like it again..)

Software reuse is a central pillar of software development, enabling developers to rely on existing frameworks and libraries to do most of the heavy lifting. Frameworks and libraries attempt to be as easy to use as possible, but often they provide so much functionality that their application programming interface (API) cannot remain trivial. This complexity and the fact that the documentation is often not kept up to date can lead to errors when developers invoke these APIs.

In the context of object-oriented programming, one of the 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. [17]. 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.

To investigate this, we analyze more than 600 open source android applications. We manually evaluate the proposed findings of X randomly selected apps and also use an automatic benchmark which relies on artificially degrading existing code. We, also, 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.

still not a fan of this paragraph - is it even necessary? / shorten it?

include any more information on the actual method? aka "relies on the majority rule / the insight that..." oä?

more concrete research question

describe results + concrete numbers

Contents

1 Introduction

Developers often outsource a significant amount of work to existing libraries or software frameworks, which expose their functionality through an application programming interface (API). These APIs would ideally be simple and easy to use, but there is always a trade-off between usability and flexibility. More powerful frameworks in particular struggle to provide a trivial API without limiting developers' ability to take greatest advantage of their functionality. Correctly invoking an API often requires that a developer know it well; not considering specific constraints and requirements can lead to serious bugs or complications.

Sometimes, these problems could have been avoided if the developer had paid better attention to the API's documentation. However, documentation is often vague or incomplete leading to misinterpretations or oversights. Moreover, even if it is of high quality, the complexities of some libraries make it inevitable that there will be erroneous invocations of their APIs. Regardless of their origin, mistakes can relate to parameter choice, method order, or a range of other factors (some methods may need to be invoked in an extra thread, a specific precondition may have to be satisfied, setup work may need to be performed).

Examples, which come to mind in Java, are a programmer calling `next()` on an iterator without first checking with `hasNext()` if it even contains another object, or a class which overrides `equals()` without ensuring that an invocation of `hashCode()` always produces the same output for two equal objects. Despite the fact that there might be numerous appropriate ways to use an API, there are underlying patterns that the correct invocations have in common. For the Java examples, 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".

API usage patterns [25] can aid in detecting these kinds of defects in a software project, by noting where the code deviates too far from the patterns associated with the API it is employing. The challenge here is twofold. First, it is extracting and recognizing the patterns. Second, it is understanding when a deviation from a pattern actually signifies a code smell or an error that should be corrected.

benny stolpert über "outsource": Developers often use existing libraries or software frameworks, which expose their functionality through an application programming interface (API), to take care of the most complex tasks. These APIs would ideally be simple and easy to use, but there is always a trade-off between usability and flexibility.

1.1 Missing Method Calls

Of the many subtle mistakes a developer can make when invoking an API, this work focuses on one specific type: missing method calls, which occur in the context of Object Oriented Programming (OOP). In OOP software, an object of a specific type is almost always used by invoking some of its methods. 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 type T_1 on which only methods `A` and `B` are called, we can say that a call to `C` is missing. Similarly, if we have an object of type T_2 where only `X` is invoked, we can say that a call to `Y` is missing.

Developers often fail to make important method calls when using classes with which they are not deeply familiar. As this can occur with a new library, an extensive framework, or even a whole platform (e.g., Android), we use these terms interchangeably. Lack of familiarity may also arise not just when using code from an external source, but also when working on a big codebase. In large software projects it is common that one developer does not know all of the code, and thus, they will often encounter classes which they do not know how to use correctly.

Studies have confirmed the intuition that missing method calls are common pitfalls across a range of applications and even in mature code bases. In an informal review [17], Monperrus et al. found bug reports[1] and problems[2] related to missing method calls in many newsgroups, bug trackers, and forums. The issues range from runtime exceptions[3] to problems in some limit cases, but generally reveal at least a code smell, if not worse. They also performed an extensive analysis [18] of the Eclipse bug repository, searching for syntactic patterns that they deemed related to missing method calls, such as: "should call", "does not call", "is not called", or "should be called". Manual inspection confirmed that more than half (117 of 211) of the bug reports located in this manner were indeed related to a missing method call.

This number is if anything an underestimation of the total number of missing method call bugs in the code base. After all, the researchers might have missed some syntactic patterns, and the bug repository can only contain those bugs that have already been discovered. Given how common they are, it seems clear that automatically detecting missing method calls in production code would be very helpful, not only saving developer time but also making maintenance cheaper and more manageable.

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=222305`
[2] `https://www.thecodingforums.com/threads/customvalidator-for-checkboxes.111943/`
[3] `https://issues.apache.org/jira/browse/TORQUE-42`

1.2 Detection – but how?

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

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

Well-crafted and -reasoned rules along these lines could facilitate precise detection of missing method calls and contribute to better, less buggy code. However, creating and maintaining a list of rules like this is a manual effort and would require tremendous amounts of time and money, especially in a world where software is changing continually. While the effort might even be justified for large and important libraries, it multiplies with the size of the library until becoming completely infeasible.

Instead, we propose automatically detecting locations in a code base where a method call is potentially missing, using only the code itself as input. Such an approach would adapt to changes without requiring additional work from a developer and could also be applied to proprietary code, which is closed to the public. The locations this method pinpoints might not be as accurate as those discovered by a hand-crated list of rules, bug a second step could address this problem: manual examination by an expert, who would determine the severity of the problem and, if necessary, issue a fix. Whether this proves to save time and money hinges on the accuracy of the approach.

clearly formulate problem statement somewhere (3-5 sentences which describe the problem to be solved + the ansatz) -> i think this is fine with the paragraph above?

1.3 Contribution

To understand and tackle the missing method call problem, Monperrus et al. [17][18] introduced the notion of type usages. 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 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 once (or a few times), this probably indicates a bug.

In this work, we introduce some variants of this idea and examine if they yield any improvements. Additionally, we propose using the majority rule to also detect superfluous

or wrong method calls. Based on the implementation by Monperrus et al. we develop a system for detecting missing method calls. We eliminate some minor inaccuracies from their implementation and add a database backend which enables the system to take more data into consideration. Furthermore, we conduct an evaluation with the goal of understanding how well this approach works in general and which of the variants performs best. For this, we apply the system to a dataset of more than 600 open source Android applications and manually analyze the results. Finally, we compare the results of the manual evaluation against those of an automated benchmark.

needs more content! what am i actually doing and why (hoping to find what?) and what are the results? implementation also part of my work! -> part of contribution!!! mention the results! evaluation -> more detail + what are the QUESTIONS?

1.4 Outline

In Chapter 2 we present previous work which goes in a similar direction. Chapter 3 explains the theoretical background of the method proposed by Monperrus et al. and in Chapter 4 we propose some variations to their method. We explain the inner workings of our implementation in Chapter 5, before summarizing the details and results of the evaluation in Chapter 6. The last Chapter 7 concludes this work and gives an outlook for future research.

2 Related Work

It is difficult and time-consuming to fix bugs in software. The maintainer has to realize that there is a bug, identify its cause and understand the steps necessary to remove it. 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. Because of this, there has been a lot of interest in approaches for automatically detecting bugs, or even just smells in code.

In this chapter, we 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 patterns. Since we are applying the method studied in this thesis to some Android apps in Chapter 6, we then give a quick overview of smell detection related explicitly to Android applications. Finally, we look into techniques which attempt to learn rules and properties from the code they are analyzing instead of relying on rules which their developers defined a priori. These automated techniques need fewer changes when the system under analysis evolves, but they might not be as accurate.

here citation for codesmells are useful? (in fontana2016com or aDocotor i think)

2.1 Finding Code Smells with Static Analysis

title accurate enough or rather smth like "hardcoded rules"?

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 a specific detector, which uses some special, sometimes quite elaborate detection mechanism. These detectors are mostly built starting from a real bug, first attempting to find the bug in question and then all similar ones automatically as well.

In their work Ayewah et al. [3] apply Findbugs to several large open source applications (Sun's JRE[2] and Glassfish J2EE server[3]) and portions of Google's Java codebase. Their premise is that static analysis often finds true but trivial bugs, in the sense that these bugs do not actually cause a defect in the software. This can be because they are deliberate

[1] `http://findbugs.sourceforge.net/`, successor: `https://spotbugs.github.io/`

[2] Version 1.6.0 (different builds) `http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase6-419409.html`

[3] v2 `http://www.oracle.com/technetwork/java/javaee/downloads/java-archive-downloads%2Dglassfish-419424.html`

errors, occur at locations which are unreachable or in situations from which recovery is not possible. In their analysis Findbugs detects 379 medium and high priority warnings which they classify as follows:

- 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

Their main takeaway is that this kind of static analysis can find a lot of true bugs, but 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. [26] propose a ranking method which attempts to rank true bugs before less important warnings. It is based on a principle they call "defect likelihood", which is essentially the probability that a finding is a real finding. They calculate this probability using the findings of a large project (in this case the JDK), each of which they manually flag as either a true or a false positive. The defect likelihood 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 be refined with user feedback when it is applied to a specific project. In their evaluation on three open source applications (Tomcat[4], AspectJ[5] and Axis[6]) they compare their ranking against the default severity ranking of Findbugs, which uses a hard-coded value for each bug type. With 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%.

Other tools for detecting code smells are Decor [16] and iPlasma [14].

some more info for those two? Which other papers?! -> should be some which are NOT about findbugs > present alterantives, maybe work in a slightly different manner, or just some sort of overview paper for static code smell detection oä!

[4]Version 6.0.18 `http://tomcat.apache.org/`
[5]Version 1.6.4 `http://eclipse.org/aspectj/`
[6]Version 1.4 `http://jlint.sourceforge.net/`

2.2 Android-specific Smell Detection

Verloop [27] uses Java refactoring tools (e.g. PMD[7] and JDeodorant[8]) to detect basic code smells like large class or long method in mobile applications. A noticeable finding is that smells can appear at different frequencies in classes inheriting from the Android framework than in other classes, for example they detect the "long method" smell nearly twice as often in classes that inherit from the framework than in those that do not.

Hecht et al. [12] present an approach they call PAPRIKA which operates on compiled Android applications but tries to infer smells on the source code level. From the byte-code analysis, it builds a graph model of the application and stores 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. These nodes 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[9]. This enables them to recognize four Android specific smells and four general, OOP related smells.

To evaluate this approach, the authors use a witness application, which contains 62 known smells. Using the right kind of metrics, they achieve precision and recall of 1 on this 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.

Another paper worth mentioning is the work by Murali et al. [19]. They leverage a Bayesian approach for learning specifications in conjunction with neural networks, to discover API usage errors. They evaluate their method on a large corpus of Android applications and are able to detect multiple subtle bugs.

After Reimann et al. [23] presented a catalog of 30 Android specific code smells, Palomba et al. [21] developed a tool to detect 15 of them. It operates on the abstract syntax tree of the source code and uses specifically constructed rules to detect each of the smells. The authors analyze 18 Android applications and compare the results of their tool against a manually built oracle, an oracle that 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 some cases in which their tool fails or yields false positives.

[7] `https://pmd.github.io/`
[8] `http://www.jdeodorant.org`
[9] `https://neo4j.com/developer/cypher-query-language/`

One especially interesting failure is the smell of 'missing compression'. It advises to compress the data when making external requests in order to save bandwidth. To detect places in the code where a request is made, but compression is missing, their hard-coded rule checks if the developer used one of two popular compression libraries. However, recently a competing compression library has been 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 an excellent example of a case where hard-coded rules fail because they are not kept up to date (new library becomes popular, interface changes, ...), do not consider a special case or conditions change in some other way.

2.3 Learning from the Analyzed Code

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 interest in approaches which adapt automatically to changing requirements by learning rules from the source code and looking for violations of those rules.

Engler et al. [6] were probably the first to propose the general idea of learning from the code at hand instead of using predefined static rules. In their paper they rely on contradiction and common behavior to "find what is incorrect without knowing what is correct". To do this, they use static analysis to infer what a programmer believes about the system state and then check these beliefs for contradictions. They provide several "templates" for beliefs the programmer must or may hold. Each template is catered for a specific type of belief and can concern wide range of things (a pointer being null or not being null, a lock being locked or unlocked, etc). Among the beliefs they extract in this manner they look for common behavior or outright contradictions and use this to flag potential anomalies. In their evaluation on Linux and OpenBSD they find hundreds of bugs related to the different bug types that their templates can discover.

is that true? or just the "general idea behind dmmc"? + kinda sort sentences below better...

Pradel et al. [22] noticed that in statically typed languages the compiler helps the programmer with passing method arguments in the correct order. However, the compiler cannot ensure the correct order of equally typed arguments (such as `setEndPoints(int low, int high)`). Confusing the order of these arguments can cause unforeseen and hard to detect errors. Their idea is to detect mixed up method arguments using only semantic information in the source code, namely variable and parameter names. They extract these and compare the names found at the call sites to the names from the implementation body using string similarity metrics. If reordering the names yields a significantly better similarity score, they report an anomaly. In their evaluation on 12 Java programs from the DaCapo Benchmark suite [5] (more than 1.5 Million lines of code), they reach a precision of 72% and recall of 38% on seeded (i.e. "faked") anomalies.

difference from what I'm doing? -> check monperrus

In real code, they found 29 anomalies of which 22 revealed true problems (76%).

In a related work, Rice et al. [24] also use identifier names to detect argument selection defects, in which the programmer has chosen the wrong argument to a method call. Using a high sensitivity threshold, their method reaches a precision of 85%. Lower thresholds improve the recall but also lower the true positive rates. The last two works are different from our approach as they are looking at a different type of error.

In a more general attempt Fontana et al. [8][7] compare 16 different machine learning techniques for code smell detection. As their training data, they manually evaluate a large body of code and tagged four basic smells (data class, large class, feature envy and long method). All of the different approaches show good performance, but J48 and random forest have the best results, while support vector machines fare the worst. These works differ from ours in that they are looking for very general smells and are using supervised machine-learning techniques to detect them.

As already mentioned, incorrect API documentation can be a big problem. It often occurs when an API developer changes the code but forgets to adapt the documentation accordingly. If a user of the API then relies on the documentation, she can run into unforeseen bugs, and what is worse, the documentation might even hinder her. To alleviate this problem, Zhou et al. [31] propose an automated approach for detecting "defects" in the API documentation. They consider two situations: either the documentation is incomplete in describing the constraints at hand, or the description exists, but it does not match the realities of the code. They make one crucial assumption: the code is correct because, contrary to the documentation, it has been tested extensively.

Their approach analyzes the code statically and uses pattern-based natural language processing to build a first order logic formula of the constraints which are present. For the API documentation, they use heuristics and a part-of-speech tagger to find the restrictions and constraints which are mentioned, before synthesizing them into another first order logic formula. The resulting formulas are fed into an SMT (satisfiability modulo theories) solver [4] to detect inconsistencies between them. They apply their prototype implementation to the `java.awt` and `java.swing` packages and find 1419 defects, of which 81.6% are true positives. Using the heuristics extracted from these packages, they also apply the prototype to other packages and find 1188 defects of which 659 are true positives (a precision of 55.5%). Overall it is worrying (if not surprising), that there are so many inconsistencies in the JDK documentation, which even has a reputation for being of rather high quality. Less well-known APIs will probably have inferior documentation and thus, even further increase the risk of API usage related bugs.

Mention MUBench?! [1] -> useful classification of api misuses Dataset of bugs related to API misuses
not necessary only learning from the code at hand: [11] unsupervised machine learning over bug FIXES?!
stuff related to "usage diversity" of classes [15], again monperrus involved though...
uses n-grams so intersting cause different approach? [28] + check related work section for some stuff that could be important!

General: read the summary paper [25] again and check which approaches are there and which could / should be mentioned + for more information: check this summary paper / the 2017 one? / both? also mention overview paper [2]

2.4 Focus on Method Calls

title okay?

Bad documentation and high complexity often make it difficult to correctly invoke API methods. To circumvent this problem, developers often search for examples of how to use a particular API. To aid with this search, Zhong et al. [30] built a search tool for example code snippets. It extracts API usages which describe how in certain scenarios some API methods are frequently called together and if the usage follows any sequential order. Using clustering and frequent subsequences mining it then groups the extracted usages into usage patterns over which the search operates. While the tool does not look for any anomalies or code smells, it extracts exactly the type of information that is interesting if one wants to detect anomalies.

There have been some works concerned with finding patterns in the way specific types are used and utilizing those patterns to find potential bugs. Most relevant for this work and its primary inspiration are two papers by Monperrus et al. [17][18] that propose a technique for detecting missing method calls. It is based on the intuition that an instance is probably an anomaly if it is alone in a large number of instances that do something similar, but not exactly the same. They call this the "majority rule" and we explain it in detail in Chapter 3.

explain object usages somewhere? + cite some related work from monperrus here (just citation)?

The evaluation they present is split into two parts. The first is based on a simulation which creates defects by degrading real software. It removes singular method calls from the code and check if the majority rule can detect these known missing calls. They perform this simulation on several software packages, among them the Eclipse user interface, Apache Derby[10] and Apache Tomcat[11]. Their system can answer between 54%

[10] https://db.apache.org/derby/
[11] https://tomcat.apache.org/

and 76% of these simulated queries (depending on the package) and of the queries it can answer 73% to 89% contain the actually missing call, albeit not necessarily as the first recommendation. All in all, it reaches precision between 59% and 83% and recall of 41% to 68%. However, it is questionable how relevant these numbers are, because it is not clear in how far real bugs will share characteristics with these artificially created cases.

For this reason, Monperrus et al. also perform an additional, qualitative evaluation. They apply their tool to Eclipse, Derby and Tomcat and manually analyze 30 very anomalous cases reported by it. They find that there are different reasons for an instance to be anomalous, not all of them actually indicate a missing call. Some are just a code smell, i.e., a situation which could be rewritten or refactored for more clarity, or related to software aging, like code using an old version of the API. They reported 17 issues and got 9 patches accepted, but there are also some negative results, where the anomaly is just that: an outlier, which is totally valid, even if a majority of other instances behave differently.

Before this, Wasylkowski et al. [29] introduced a method to locate anomalies in the order of method calls. First, they extract usage models from Java code by building a finite state automaton 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. This process determines 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 [10] 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 several open source programs. Manual evaluation classifies these into 2 real defects, 5 smells and 84 “hints” (readability or maintainability could be improved). This adds up to a false positive rate of 87.8%, but with the help of a ranking method, they can obtain the 2 defects and 3 out of 5 smells within the top 10 results.

Gruska et al. [9] extend the work of Wasylkowsky et al. and use a lightweight parser to extract temporal properties from more than 6,000 open source Linux projects. They manually evaluate the findings of 20 randomly selected projects and find that around 25% of the high-ranked anomalies uncover actual code smells or defects.

In a related work, Nguyen et al. [20] 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. Here the object usages are represented as a labeled directed graph in which the nodes are field accesses, constructor or method calls and branching occurs because of control structures. Thus, the edges of the graph represent the temporal usage order of methods and the dependencies between

them. They mine patterns 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., in relation to its size it does not appear frequently in the dataset. In an evaluation case study the authors find 64 defects in 9 open source software systems which they classify as 5 true defects, 8 smells and 11 hints, equalling a false positive rate of 62.5%. Using a ranking method the top 10 results contain 3 defects, 2 smells and 1 hint.

The last three works differ from our approach in that they are concerned with the order of method calls rather than their presence.

3 Detecting Missing Method Calls

In the introduction, we explored errors related to missing method calls. These occur frequently, particularly when developers are working with unfamiliar or new code. We would like to automatically detect instances of missed method calls to make maintenance easier and cheaper. In this chapter, we explain in detail the method proposed by Monperrus et al. [17][18] which is based on the majority rule.

or: Developers working with complicated frameworks and libraries or generally with resources they are new to are particularly prone to missing an important call.

!

3.1 Foundation: The Majority Rule

The intuition behind the majority rule is that the way something is done most often is probably the correct way. It follows that if everyone or almost everyone is doing something differently than you, you are probably doing it wrong. (This is only relevant when the action being performed is in some way close to what you want to do.) -> smth more... smth about how if it is too far away we don't care anymore at all (spoon / fork analogie...) This gives rise to the definition of an anomaly by the majority rule: instances with few or no "equal" but many "slightly different" neighbors. Such anomalies should probably adapt to behave like their very similar neighbors. What exactly "equal" and "slightly different" mean has to be defined for each application of the majority rule.

To illustrate how this idea might work in the context of object-oriented software and method calls, consider the following example. We are analyzing an Android app which uses a total of 100 instances of the `Button` class. On all 100 the method `setText()` is invoked to describe the functionality of the button, but only 99 of them also call `setFont()`. There is one button that is missing this call and, thus, uses the default font. It seems highly likely that this one button is an exception to the rule "each button which invokes `setText()` should also call `setFont()`" and therefore a mistake. To correct this, it should also make a call to `setFont()`. Imagine all buttons in the application using a unique and beautiful font, but this one button displaying Comic Sans!

3.2 Type Usages

Generally speaking, type usages are an abstraction over the code. They ignore the order of method calls and are only concerned with the list of method calls invoked on a

particular object. The critical pieces of information associated with a type usage are 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.

It is useful here to define formal terms. For each variable x, note its type $T(x)$ and the context $C(x)$ in which it occurs. The list of methods which are invoked on x within $C(x)$ is denoted as $M(x) = \{m_1, m_2, \ldots, m_n\}$. 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 5.2).

```
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 [18])

As an example, consider the code in Figure 3.1. It contains two type usages, one of type `Button` and another 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.

3.3 Exact and Almost Similarity

To detect type usages that are anomalous by the majority rule, we need a notion of what it means for two type usages to be exactly similar or only almost similar. Informally, we say that two type usages are exactly similar if they have the same type, if the type usage appears in a similar method (i.e., the context is identical), and if their list of method calls is identical. Recall that the context is the name of the method in which the type usage occurs, together with the type of its parameters. We call these type usages "similar"

instead of "equal" because several things are not the same: variable names, surrounding or interspersed code, method order or parameters, and the actual location (class).

The notion of almost similarity is analogous. We say that a type usage y is almost similar to a given type usage x if they share the same type and if context and the list of method calls of y is the same as the method calls of x plus one additional call.

```
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 [18])

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`. Notice that variable names and method parameter have no impact on this. Additionally, the comparison is not influenced by the order of methods or any interspersed code that does not call a method on the `Button` objects. The type usage in class `C` (bottom left) is almost similar to those in `A` and `B`, because apart from `setText` and `setColor` it also invokes one additional method – `setLink`. Finally, the type usage in class `D` (bottom right) invokes **two** additional methods compared with the ones in `A` or `B`, meaning that it is not almost similar to them. However, it is 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:

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

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 $|A(x)| \geq 0$. Consider also that A' is not symmetrical, i.e., $xA'y \iff yA'x$ since one of the two type usages must have fewer methods. This also means that $y \in A(x) \implies x \notin A(y)$.

We can further refine the definition of almost similarity. Instead of including type usages which have the same method calls plus one additional one, it is also possible to consider those type usages which have the same method calls plus k additional ones. Then, in the definition of A the last line changes to: $|M(y)| = |M(x)| + k, k \geq 1$. The purpose of this change would be to detect type usages which are missing k calls instead of just one.

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.

3.4 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 significant 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 one method call). Assuming the majority is correct, the type usage under scrutiny is deviant and potentially erroneous, because it is missing a method call.

To capture concretely how anomalous an object is, we need a measure of strangeness. This will be the strangeness score. We can use it to order type usages by how much of an outlier they are and 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 $|A(b)| = 99$. This results in a strangeness score of $\text{S-score}(b) = 1 - \frac{1}{1+99} = 0.99$. Intuitively such a type usage is very strange, and indeed, the S-score supports the intuition.

3.5 Which Calls are missing?

It is not enough to detect the type usages that are outliers. We would also like to present the developer with some candidate suggestions, which method call might be missing. The intuition for this is to look at all the almost similar type usages and collect the additional method calls that each of them invokes. Recall, that an almost similar type usage will make exactly one additional call in comparison to the anomalous one. We then take the frequency of unique calls among the set of additional method calls and use it as the suggestion likelihood.

Formally, the calls we can recommend for a type usage x are the calls that are present in the type usages contained in $A(x)$ but are 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`.

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

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

Figure 3.3: Example for computing the Likelihood of missing Calls ([18])

To illustrate these definitions, consider the example in Figure 3.3. The type usage under analysis is denoted with x, operates on a `Button` object and the only method that it invokes 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.

3.6 Ignoring the Context

We integrate the context into the definition of similarity and almost similarity because a type might be used in different ways depending on the situation it is used in. An example would be the class `FileInputStream` which is rarely opened and closed in the same method. In their evaluation, Monperrus et al. [18] find that not considering the context adds a lot of noise (not relevant items in $E(x)$ and $A(x)$) and using it improves the precision of their experiments. However, one can question the general assumption behind using the context. The idea that types are used in different ways seems reasonable, but is the name of the function that they are used in a good separator between those use cases? Furthermore, using the context might work well for some types and some

systems, but does it make sense everywhere? One problem with using the context is that it drastically reduces the number of type usages which are even considered for (almost) similarity. In smaller datasets, this makes it much more likely that "patterns" appear, which in truth are only artifacts of randomness.

All in all, it is not clear that using the context necessarily improves the performance of the system. Because of this, next to the standard variant $DMMC$ (Detecting Missing Method Calls), we also define a variant called $DMMC_{noContext}$, which does not take the context into account. It uses adapted measures for similarity and almost similarity, $E'_{noContext}$ and $A'_{noContext}$ which are defined in the same manner as E' and A', except that they do not require the context of the type usages to be identical. The definitions of E, A, S-score and ϕ stay the same, besides that they now make use of the newly defined relations.

4 Extensions

In the previous Chapter, we summarized the work of Monperrus et al. [17][18], in this one we propose some possible modifications to their method. For instance, it might be possible to leverage the majority rule not only for detecting missing method calls, but also to identify superfluous or flat-out wrong method invocations. Additionally, there are different ways of organizing the type usage data, which could yield better results. In this section, we give the motivation and theoretical background for these modifications, in Chapter 6 we will explore further how they perform in comparison with the original.

4.1 Class-based merge

The $DMMC_{noContext}$ variant seeks to answer the question if it makes sense to group the type usages based on the method they appear in. Following this line of thought, one might also ask if it is optimal to collect the type usages with method granularity. Even when ignoring the context, the type usages will still be extracted on a per-method basis, that is that is only the methods invoked in one method body will belong to one type usage. Maybe grouping method calls with class granularity yields much better results. Often the methods invoked on one object within a specific class include many, if not all, of the methods invoked on it in its whole lifetime. Thus, a class-based type usage offers a bigger window into the objects utilization and, potentially, this makes it easier to detect if some method is missing.

We will denote this variant as $DMMC_{class}$. To obtain the class-based type usages, we extract the type usages at the method level, exactly as before. However, before calculating the strangeness score, we merge all those type usages which have the same type and originate from the same class. We merge them even if the original type usages originate from different objects because tracing each objects life through the whole class would be too expensive. With these newly produced type usages, we can then calculate the set of similar and almost similar type usages using $E'_{noContext}$ and $A'_{noContext}$, respectively.

To formalize $DMMC_{class}$, we need the new operator $\mathrm{Cls}(x)$ that denotes the class from which a type usage was extracted. We can then partition the set of all type usages

into subsets TC_i which have the same type and class:

$$\begin{gathered} x, y \in TC_i \iff T(x) = T(y) \wedge \mathrm{Cls}(x) = \mathrm{Cls}(y) \\ \forall i \neq j \;.\; TC_i \cap TC_j = \emptyset \\ \bigcup TC_i = \{\text{all type usages}\} \end{gathered}$$

Then we iterate through the partitions and calculate the set of new type usages as follows:

$$\begin{aligned} \{x' \mid\; & T(x') = T(x_0), \\ & C(x') = \bigcup C(x_i), \\ & M(x') = \bigcup M(x_i), \\ & \text{with} \quad x_i \in TC_j\} \end{aligned}$$

In this process, we are merging all type usages that belong to the same partition into one new type usage. Recall that for exact and almost similarity we will use the version which ignores the context, so the context $C(x')$ of the new type usage is not that relevant.

4.2 Investigating different Anomalies

While a *missing* method call might be the first error type that comes to mind and also potentially the most frequent one, the general technique of the majority rule can be adapted to detect two different anomalies related to method calls. First off, it is possible to flip around the notion of almost similarity and investigate if there exist any superfluous method calls. Additionally, one can envision a developer invoking the *wrong* method, rather than forgetting a call or adding one too many. The primary difference of these variants lies with the definition of almost similarity between type usages.

4.2.1 Superfluous Method

The idea behind $DMMC_{superfluous}$ is the inverse of the method proposed by Monperrus et al. Instead of checking if a lot of other type usages are calling an additional method and the current one is thus missing a call, here we check if a lot of other type usages are making *fewer* calls, and the current one is doing something extra which it should not. It is not intuitively clear what kind of errors this procedure will discover or indeed if there will be any. Furthermore, it is possible that it will only find outliers which are intentional outliers, be it to handle a special case, to work around a bug or for any other reason. Nonetheless, it seems reasonable to investigate this idea and only dismiss it if it does not yield any interesting results.

Adapting the normal system to detect this type of outlier is rather simple. We only need to adapt the definition of $A(x)$ to:

$$A_{superfluous}(x) = \{x \mid yA'x\}$$

Recall that the almost similar relation A' is not symmetric and observe that in comparison to $A(x)$, here x and y are flipped in the application of A'. Thus, this definition of almost similarity considers type usages to be almost similar, when they call one method less than the input type usage. With the definition of S-score and ϕ as before, the scores will now indicate which type usage is calling anomalously many methods and which method might be the one that is too much.

4.2.2 Wrong Method

Further extending the idea of the previous section and thinking consequently, developers might *miss* a method call, they might *add* a useless one but, of course, they could also simply choose the *wrong* one. Especially if the developer is not all too familiar with a given framework or there is some ambiguity in the method names or documentation, it seems feasible that he chooses the wrong method for a given task. Of course, it is possible that this type of error results in noticeable bugs much faster than a missing call, but again we do not know this before investigating.

To formalize this new variant $DMMC_{wrong}$ looking for erroneous method invocations, we again need only change the definition of almost similarity. This time we define a new relation A'_{wrong} as follows:

$$\begin{aligned} xA'_{wrong}y \iff & T(x) = T(y) \ \wedge \\ & C(x) = C(y) \ \wedge \\ & M(x) \neq M(y) \ \wedge \\ & |M(x)| = |M(y)| \ \wedge \\ & \exists S \,.\, S \subset M(x) \wedge S \subset M(y) \wedge |S| = |M(x)| - 1 \end{aligned}$$

For this relation to hold, again the type and the context of both type usages have to be the same. Furthermore, it requires that all but one method of $M(x)$ and $M(y)$ be identical, that is, that their sets of method calls differ by exactly one method. With this definition of almost similarity, the majority rule flags those type usages that make an unusual method call as an anomaly.

anything more here?, better description?

5 Implementation

This chapter focusses on the practical side and on the internal workings of the system we developed. It explains the steps from receiving the input program to outputting a list of anomalies which hint at potentially missing method calls. We give details about design decisions taken and the reasoning behind them, pitfalls that had to be overcome and the trade-offs that had to be accepted. Additionally, we explore some of the mistakes made and dead ends that we encountered.

5.1 Overview

Our system is (primarily) a system that detects missing method calls. It works by statically analyzing a given software application, extracting the type usages it contains and finally determining if any of them are anomalous by the majority rule as described in Chapter 3. The end result is a list of locations which are potentially missing a method call. Given a software system which shall be tested for anomalies, the following steps have to be realized:

1. Extract all type usages from the software;
2. For each type usage x, among them:
   a) Search for type usages which are exactly similar to x (i.e. calculate $E(x)$);
   b) Search for type usages which are almost similar to x (that is determine $A(x)$);
   c) Calculate the strangeness score of x;
   d) Extract the list of potentially missing calls and their likelihood ϕ;
3. Output a list of anomalous type usages, sorted by their S-scores, together with the calls they are potentially missing.

However, this simple outline does not represent the actual realities of the system. Instead of a singular process with sequential flow, it is split into three different parts, which are laid out in Figure 5.1. First, a Java application reads the bytecode of the program under analysis and iterates through all methods to extract the type usages

POTENTIALLY add the retrieve sets step to python?

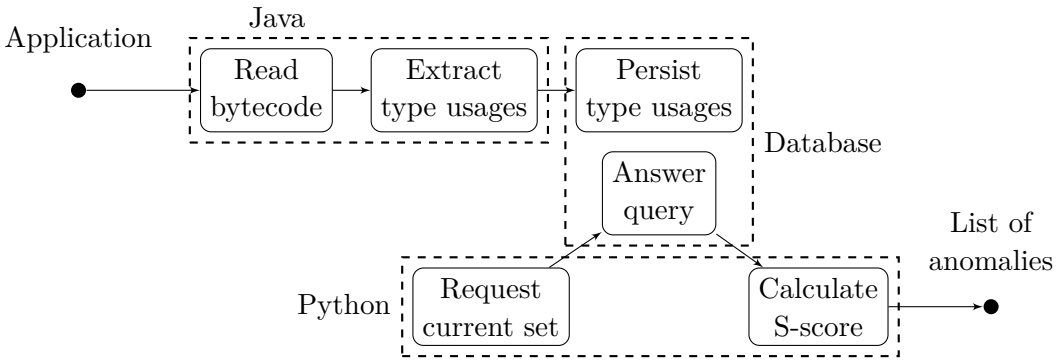

Figure 5.1: System overview

which are present. The extracted type usages are then persisted in a database to ensure flexibility and high performance in the following analysis phase. For the actual anomaly detection, we use a small Python program. It goes through all partitions of type usages (more on the partitions in Section 5.6), requests the type usages in the current set and calculates their strangeness scores. After iterating through all the sets of the application, it finally outputs the results.

The following sections give a more in-depth explanation of the separate steps.

5.2 Bytecode Analysis

We use Soot[1] for the bytecode analysis. Soot was originally a Java optimization framework but has since evolved to support a wide range of use cases. It can analyze, instrument, optimize and visualize Java and Android applications. For this purpose, it provides call graph construction, points-to analysis, def or use chains, inter- and intra-procedural data-flow analysis and taint analysis. We do not need most of its functions, and merely use it to statically extract type usages from the input application.

In theory, it would be possible to extract type usages from the source code, but extracting them from compiled code has some advantages. Both the JVM bytecode and the Dalvik bytecode found in Android applications are very standardized, which facilitates the analysis. While Soot supports source code analysis on paper, it only works up to Java 7 and, furthermore, bytecode analysis is the approach recommended by its authors. Besides that, using compiled applications simplifies our experimental setup, and as a marginally useful side-effect, it also helps with analyzing obfuscated applications. Finally, there are no real disadvantages to this approach; if the program is only available

[1] `https://sable.github.io/soot/`

Option	Parameter	Explanation
`-app`	-	Run in application mode
`-keep-line-number`	-	Keep line number tables
`-output-format`	none	Set output format for Soot
`-allow-phantom-refs`	-	Allow unresolved classes
`-src-prec`	apk-class-jimple	Sets source precedence to format files
`-process-multiple-dex`	-	Process all DEX files found in APK
`-android-jars`	[path]	The path for finding the android.jar file

Figure 5.2: The parameters passed to Soot

as source code, we can compile it before analyzing it.

Soot operates by transforming the given program into intermediate representations, which can then be optimized or analyzed. It provides four intermediate representations with different use cases; of these, we use Jimple, Soot's primary representation. Jimple is a typed 3-address representation which is especially suited for optimization but also suffices for our purpose.

Soot is a very powerful framework, and one can customize it to cater for quite specialized requirements. We analyze mostly Android applications, and the settings we apply to configure it (listed in Figure 5.2) reflect this.

5.3 Extracting Type Usages

The Soot execution is divided into 'phases,' each of which has a specific task, for example the aggregation of local variables. Phases consist of transformations, which can modify the input they receive but are not required to do so. The phases are grouped into 'packs,' and the registered packs are applied successively during the execution. To extract type usages, we register a custom transformation in the Jimple transformation pack, which is applied to every method in the analyzed program.

The transformation receives as input a Jimple representation of the method body it is currently analyzing. It iterates through all the statements in the body and marks those that invoke a method. It then iterates through this list of method calls and groups them into type usages based on the objects that the calls are made on. For each call, it first checks whether a type usage corresponding to the object the call is invoked on already exists. If such a type usage exists, the transformation adds the current call to it and advances to the next method call in the list. If, on the other hand, there is no type usage for the object, it creates a new one. After completing this process for all method calls in the method body, the extraction finishes and returns the list of type usages.

One noteworthy step of this analysis is the `LocalMustAliasAnalysis`, which attempts to determine if two local variables (at potentially different program points) must point to the same object. This is necessary to decide whether the calls made on these locals should be grouped into one type usage or not. The underlying abstraction is based on global variable numbering and follows the ideas presented by Lapowsky et al. [13], with some minor adaptations. The analysis is a Soot feature, and in test runs on big applications, we noticed that it requires a lot of memory and time. Thus, we made it optional; however, we were able to use it for our benchmarks and evaluation.

For performance reasons, we exclude some classes from the extraction. We do not store type usages that occur *inside* those classes; we do, however, still record usages of them. The packages we exclude in this manner are `java.*`, `android.*`, `soot.*` and `javax.*`. These are all framework classes, and we are primarily interested in the type usages occurring in the application itself, not in the framework.

explain a bit better that these classes are kind of exactly what we are most interested in as far as type usages go, but we don't want too step ITNO them + make sure this works good together with coresponding paragraph in the next section

5.4 Storing Type Usages

After the transformer has analyzed the current method body and extracted a list of type usages, we store them in a database. For this, we use HSQLDB[2], an SQL relational database written in Java, which provides a multithreaded and transactional database engine with memory- or disk-based tables. We chose HSQLDB because it is small and offers excellent performance as well as an easy setup. Additional arguments were its permissive license and the large number of features it supports.

In their work, Monperrus et al. save all data in a text-based format. Consequently, they need to parse everything again for the analysis, which can be slow and require a lot of memory for large inputs. Additionally, the data takes up a lot of storage space when persisted to disk. In contrast, using a database has many positive ramifications. First, retrieving data from a database is fast, and we can access the data selectively based on changing criteria. Additionally, we gain a lot of flexibility, and it becomes easier to extend both the stored data and the analysis itself.

We can already expect useful results when analyzing one application in isolation, especially if it is a large application. However, our real interest lies in the framework and library classes that are used in more than one application. By analyzing many applications that use the same framework, we can build a large dataset of type usages on the framework classes and thus, make a more informed decision whether a type usage is

[2] `http://hsqldb.org/`

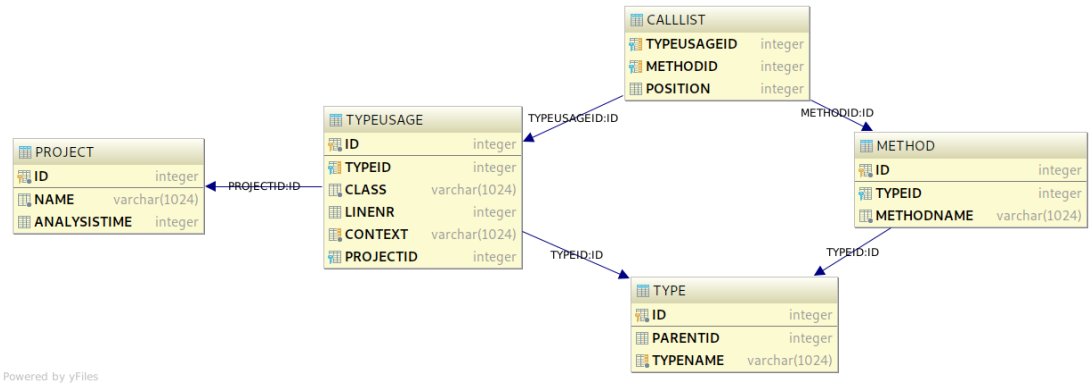

Figure 5.3: Overview of Database Tables

an anomaly or not. Having a database backend facilitates building such a large dataset because it can store large amounts of data and we can later query it selectively.

REMOVE the xfiles watermark

Figure 5.3 depicts the layout of our database. The central piece is the `TYPEUSAGE` table whose rows store the information related to one particular type usage. This includes the class (complete with fully qualified package name) and the context in which the type usage occurred and, if possible, the line of code in which the object was first encountered. Additionally, it references two other tables, `PROJECT` and `TYPE`. In our setup, the `PROJECT` table describes the Android application from which the type usage was extracted together with the time it took to analyze this application.

The `TYPE` reference specifies to which type the type usage belongs. Each type holds a reference to its parent type (if one exists), to enable rebuilding the inheritance hierarchy if so desired. The methods that can be invoked on a type are stored in the `METHOD` table. Finally, the `CALLLIST` table connects type usages and method calls and specifies which of the methods that *could* be invoked are actually called on the type usage in question.

5.5 Improvements and Dead Ends

Monperrus et al. released their code and data on GitHub[3]. We started with their implementation and refactored it extensively[4] to understand it better and to facilitate slight improvements like the database backend connection. During the refactoring process, we discovered small discrepancies between what they describe in the paper and how their

[3] `https://github.com/stg-tud/typeusage`
[4] `https://github.com/ke-kx/typeusage`

code behaves. In the calculation of almost similarity between two type usages x and y, the code only checks that the size of $M(y) \setminus M(x)$ is equal to one; however, it fails to verify that $M(x)$ does not include any other methods. As such, in the situation that $M(x) = \{A, B, C\}$ and $M(y) = \{A, B, Z\}$, the code considers y to be almost similar to x, because y calls the additional method Z. Recall that almost similarity expects one type usage to have one method call more than the other, and this is clearly not the case here with both type usages calling three methods. However, we reran their experiments after correcting this small inaccuracy, and the results did not change significantly.

After we refactored the code and added the database backend, the database stores all of the type usage data. Therefore, we thought it would be convenient to calculate the strangeness scores in the database as well and only query it for the finished results. In hindsight, this was not the best idea. The first query we designed to calculate the strangeness score was quite complicated, with many joins over views containing many more joins. Even on a small test dataset, it was quite slow. We attempted many improvements to the query, including better indices and smart cached subtables, but the query remained slow and memory intensive. We even invested some time into exploring the possibility of using PostgreSQL[5], but the results were equally disappointing.

Careful analysis revealed that the biggest problem was the one-by-one comparison of each type usage with all other type usages to determine their [almost / exact] similarity, resulting in runtime in $O(n^2)$ for n type usages. It seemed that the database was not able to unravel these numerous comparisons with our insufficient instruction. However, if one thinks carefully about the process of determining the exact and almost similar type usages, something stands out: it is not necessary to compare all of the type usages one by one. The only type usages that are potential candidates for exact or almost similarity are those that have the same type and context as the type usage under investigation. This gives rise to natural partitions of the dataset: the sets of type usages which share the same type and context (or just the same type for the variant $DMMC_{noContext}$). Within these partitions, we need to compare each type usage with all of the others, but each partition will only be a fraction of the size of the whole dataset. Fortunately, the database is already perfectly suited for obtaining these subsets. With this realization, we can continue to the final step of the analysis: detecting anomalies.

5.6 Anomaly Detection

not super happy with this paragraph, but how to change it?

Some relatively simple Python scripts handle the anomaly detection. The first step is to obtain the possible partitions of the dataset, depending on the variant, either all

[5] `https://www.postgresql.org/`

types in the dataset or all valid combinations of type and context. Observe that the database makes it very simple to obtain these partitions with just one query. In its outer loop, the script iterates through all partitions and retrieves the type usages belonging to the current partition. Again, querying the database for these type usages is simple and also quite fast because of indices on the relevant columns. In the inner loop, the script iterates through all the type usages in the current partition and compares the methods that they invoke. We compare each type usage with each other type usage only once, similar to filling an upper triangle matrix, thus saving nearly half of the comparisons. After determining the number of exactly and almost similar neighbors for each type usage in this manner, it is easy to calculate the strangeness score for all of them. Finally, the script saves the results for further analysis and evaluation by an expert.

The anomaly detection is the only part that of the implementation that is affected by the different variants presented in Chapter 3. We already mentioned the differences between including or excluding the context. For the class merge variant, we perform some simple preprocessing before starting the analysis. We load the data with type partitioning and merge those type usages which originated in the same class. For the detection of superfluous or wrong method calls, we can use the same setup and only need to adapt the almost equal check to the new requirements.

6 Evaluation

take some stuff from results intro 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 goal of evaluation: große frage: bringt das Verfahren etwas? understand how well this method works in the context of the android ecosystem

In the following we present our evaluation of the method proposed by Monperrus et al. We strive to understand how well the majority rule is suited to detect missing method calls in object oriented software. For this we apply it to a large dataset of Android applications and evaluate the results qualitatively and quantitatively.

We give an overview of the methodology used during the evaluation, explore the dataset we are using and how the type usage notion operates on it. For the main evaluation we perform qualitative and quantitative evaluation on a randomly selected subset of the whole dataset. We present the results in answer to X research questions which are aimed ad shedding some light on the applicability of this method. Finally, we discuss the meaning of these results and how to understand them.

paragraphs don't work well together + might need some other content?

6.1 Methodology

The goal of the method we explore in this thesis is to detect missing method calls. We use a method proposed by Monperrus et al. and some slight variations of it and would like to establish if and how well they work. Our experimental evaluation is twofold: a qualitative evaluation of the findings in 10 randomly selected Android apps and a simulation experiment. For the qualitative evaluation we manually check each anomalous type usage and flag it as a true or false positive. The simulation consists of artificially removing method calls from the analyzed code and checking if we can detect this known missing method call. Since it is not assured that the results of this simulation correspond to the real performance, we also attempt to evaluate the applicability of the simulation experiment.

more?

maybe a bit nicer / more clear / less information?

6.1.1 Qualitative Evaluation / Android Case Study

F-droid[1] is a repository for Free and Open Source software (FOSS) on the Android platform. We used a simple scraper to download the latest version of all available applications on the 6th of March 2018. From these XXX Android applications, we extract all type usages that are present and apply our implementation of the DMMC system to identify any anomalies.

We are using software from the Android ecosystem for the evaluation to understand how this method behaves on real software. Advantages of Android are that it is based on Java and has a rich open source community. We can generate a large dataset because of the wide availability of sample applications and each of them will facilitate the Android API. Using this we can detect any common patterns that emerge when using the rich Android API. Especially because of the emphasis on GUI we consider this one of the environments in which patterns will appear most frequent and most clearly.

needs rewriting, shuffling + making the points more clear... maybe Start with the idea and then say android is good for this do real software systems actually behave in this way ie: is it "necessary" to use classes in the same way

needs way better coherence! mention the android.* filter! change all dmmc to using text? reshuffle a bit

The procedure of the evaluation is as follows: We extract the type usages of all XXX Android applications that we downloaded and persist them into the database. We then use the 3 different variants $DMMC$, $DMMC_{noContext}$ and $DMMC_{class}$ to analyze the type usages and determine any anomalous findings. Monperrus et al. consider a strangeness score bigger than 0.9 to mark an anomaly and this is our cutoff as well. We manually evaluate the anomalous findings and flag each of them as one of the following:

real bug (B) a real defect, change definitely necessary

real smell (S) another way of doing things would be better, but probably not a defect

implementation error (on my side) (IE) dot chaining not recognized correctly (not sure if actualy possible) (-> not sure i should include this)

false positive (FP) the usage is totally fine, there is no causal relation which would imply that the "missing" method also needs to be used (subcategory: it makes sense that the pattern exists ie that many ppl use this together but it simply doesn't indicate an error or smell)

Special case(?) (SC) "weird" implementation with extra comment, for compatibility reasons, something like that

[1] `https://f-droid.org/en/`

not clear (NC) source code not available, ...

make sure that these categories make sense and adapt descriptions to be more serious

the decision is based on reviewing the source code and api documentation

6.1.2 Automated Benchmark

The underlying assumption behind the automated benchmark is that this method for detecting missing method calls should be able to detect artificially removed calls. As such, it is essentially a simulation which imitates the process of a developer forgetting a method call with the much simpler variant of deleting one. !! general idea behind evaluation method and what we are trying to detect !!

building benchmarking infrastructure flexibility with python class setup degrading type usages + checking if they will be detected some more subsections? (different ways for degrading type usages?) sometimes in even further degraded settings (degrade more than one)

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

compare results to the results of manual evaluation between the different methods (ie does one perform much worse in both or just in one?) This type of simulation is at the heart of the evaluation by Monperrus et al. While it seems like a reasonable procedure to perform, it relies on a crucial assumption: the artificially created missing method calls are comparable to missing method call bugs in real software.

a bit less convoluted... and more inforamtive, what is actually being done

this is not inheritly clear to us which is why we actually wanna check this? without this assumption the numbers produced by the procedure are meaninglesss obvious question: how similar are the such created mmcs to mmcs in

6.2 Dataset Overview

general data about avg tus related to different settings etc -> what does our dataset even consist of? mention somewhere: the monperrus2013 p11 highlight? (at least equivalent for my dataset) Does the Strangeness Score behave as expected on Android Apps? (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) -> 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

should answer: 1 What kind of dataset do we have? 2 How are type usages in Android apps like? 3 How does the strangeness score behave? 4 Is it even fast enough? (this here or as separate research question?) -> get general feeling for this method and the dataset

look into and understand the type of data i have -> how often small inputs, how often big, ...

1 Dataset: How many Apps + How many TUs in total + each (avg / median) -> how big are the apps, (how are the tus split between the apps) number of partitions (types + context / types) + maybe verteilung of number of tus per partition (that seems like an interesting / important one!) different number of tus for class merge? -> yes necessarily, but equal to number of partitions! mention how the dataset becomes smaller -> need to see if results become better

2 type usages: length of method list -> histogram? percentage on Android framework, percentage on other stuff

3 strangeness score: verteilung of strangeness score (histogram) graphs related to general score verteilung

4 performance: total time + duration of tu extraction in relation to number tus (proxy for project size) + mention somewhere that analysis itself is relatively cheap (especially for new project coming in)

6.3 Results

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"

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

6.3.1 RQ1: How many true versus false Positives are among the anomalous Type Usages flagged by the Majority Rule?

wie relevant sind diese obtained results in der praxis (as determined by fp vs tp) how many true findings in relation to false positives does it find +

is this something that a developer would use to help with maintenance? are the results useful / helpful

start with avg + median number of findings per app + in relation to their total TUs only the results of manual evaluation of the 10 random apps with the context variant + add the android filter (this will only reduce the number of findings, but maybe improve the quality?!)

RQ1.1: Is there a noticeable difference between the different variants?

do the manual eval each for the slightly adapted forms (no context, class merged) and see if there is any difference + include the android filter

RQ1.2: Given a known missing method call bug, can this approach detect it?

only if time! -> probably not

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

6.3.2 RQ2: Do the Benchmark Results align with the Results of the Manual Evaluation?

how 'meaningful' are the benchmarking results? 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!...

potentially simulate the benchmark only with tus from exactly the same 10 random apps? (so i don't have to do the whole dataset?) simply give metrics (precision, recall) for the different variants and relate them to the manual results (maybe relate the precision etc to the number of tus in the partition -> already have that data)

6.3.3 RQ3: 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

I mean I have the benchmark test, so i might as well do it, but also mention mathematical results!

6.3.4 RQ4: What are the requirements for the input size?

benny didn't like "requirements" -> How much input is needed to produce 'useful' results? How big does the system under analysis need to be? How many type usages are necessary to ...

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)

6.3.5 RQ5: Can the Majority Rule also detect superfluous or wrong Method Calls?

manual eval of results when using superfluous / wrong variants -> same mode of evaluation as RQ1

6.4 Discussion

also mention runtime!

average number of findings per app (remember that the findings now are for all 626 apps) how is the quality of findings (RQ1), how many findings are there, how much work is it to evaluate them

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

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

mention in the end that i tried a 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?) but all in all it didnt work out / not enough time + vermutungen warum es nicht funktioniert -> dataset exploration + size of method and typeusage lists per partition!
mention this when discussing the results! static functions evaluation! something to fix the dotchaining problems -> for both basically I'm just saying that i looked into it and it was too much work / impossible?

6.5 Threats to Validity

needs to be extra section or can be together with discussion?

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

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)

7 Conclusion

we want to detect missing method calls because that would be useful over whole software lifetime, development of new software and maintenance of old mature software take method monperrus and improve the implementation by adding database + more flxibility / extensibility also propose a bunch of variations to their method, with the goal of a) BETTER detection of missing calls (no context, class merge) and b) detecting different kinds of bugs (wrong / superfluous)

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

performed twofold evaluation, qualitatively and quantitatively 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 ...

7.1 Future Research

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) tried it but so far no luck mention the hypersphere thing? specific problem: static functions -> potentially step into them sometimes something else that comes up in analysis? (dot chaining, ...) Performnance (shouldn't be a problem?) but could use cython, etc bigger dataset! use different tu partitions (e.g. inheritance hierarchy?, ...) maybe do some meta evaluation there -> which partition is best able to detect smth here?

apply this anomaly detection method to other features (implements, overrides, pairs of methods, ...) objectübergreifend somehow? what other pairings might there be? or potentially later look into other features (like implements/override), pairs of methods, etc monperrus et al propose looking into traces or conditions as well

of course about temporal properties... 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?

Bibliography

[1] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. "MUBench: A benchmark for api-misuse detectors." In: *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE. 2016, pp. 464–467.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. "A Systematic Evaluation of API-Misuse Detectors." In: *arXiv preprint arXiv:1712.00242* (2017).

[3] 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.

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

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. "The DaCapo benchmarks: Java benchmarking development and analysis." In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 169–190.

[6] 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.

[7] 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.

[8] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla. "Code smell detection: Towards a machine learning-based approach." In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE. 2013, pp. 396–399.

[9] N. Gruska, A. Wasylkowski, and A. Zeller. "Learning from 6,000 projects: lightweight cross-project anomaly detection." In: *Proceedings of the 19th international symposium on Software testing and analysis*. ACM. 2010, pp. 119–130.

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

[11] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. "Discovering bug patterns in javascript." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 144–156.

[12] 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.

[13] C. Lapkowski and L. J. Hendren. "Extended SSA mumbering: Introducing SSA properties to languages with multi-level pointers." In: *International Conference on Compiler Construction*. Springer. 1998, pp. 128–143.

[14] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design." In: *ICSM*. 2005.

[15] D. Mendez, B. Baudry, and M. Monperrus. "Empirical evidence of large-scale diversity in api usage of object-oriented software." In: *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE. 2013, pp. 43–52.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. "Decor: A method for the specification and detection of code and design smells." In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[17] 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.

[18] 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.

[19] V. Murali, S. Chaudhuri, and C. Jermaine. "Bayesian specification learning for finding API usage errors." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 151–162.

[20] 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.

[21] 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.

[22] M. Pradel and T. R. Gross. "Detecting anomalies in the order of equally-typed method arguments." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 232–242.

[23] 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.

[24] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes. "Detecting argument selection defects." In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), p. 104.

[25] 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.

[26] 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.

[27] D. Verloop. "Code smells in the mobile applications domain." PhD thesis. 2013.

[28] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. "Bugram: bug detection with n-gram language models." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 708–719.

[29] 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.

[30] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. "MAPO: Mining and recommending API usage patterns." In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 318–343.

[31] 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.