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:	June 15, 2018

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

Munich, June 15, 2018 Nils-Jakob Kunze

Acknowledgments

I would like to thank the preliminary readers of this thesis Albert, Bene, Pete, Siggy and Fabi for their efforts 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 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

When a developer works with an object-oriented framework that he does not know well it is easy to forget an important method call. We study a new technique for detecting missing method calls in Java applications that was proposed by Monperrus et al. [14]. It does not use hard coded rules or any input besides the source code itself, instead it detects outliers based on the majority rule: If a type is used in one particular way many times and differently only once, this probably indicates a bug.

mention some more here about the variations?, is the results description okay?

Based on the implementation by Monperrus et al., we develop a system that supports several different different techniques for detecting missing method calls. To investigate which one produces the best results, we evaluate them on a dataset of more than 600 open source Android applications. We manually review the anomalies our implementation finds in 10 randomly selected applications and also use an automated benchmark that relies on artificially degrading existing code. In the comparison between the different variations, the original technique by Monperrus et al. comes out ahead. However, even if it manages to uncover one true bug, its total results are mediocre with only 3 true positives among 17 total findings. Additionally, our evaluation points in the direction of the majority rule being sensible to input perturbations and needing a lot of data that is not easy to obtain.

Finally, we propose utilizing the majority rule to also detect superfluous or wrong method calls. We present an implementation of the proposed system and perform a manual review of its findings. The outcome suggests that the majority rule is not suitable for detecting this particular type of error.

Contents

1 Introduction

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. More powerful frameworks in particular struggle to provide a trivial API without limiting developers' ability to take full advantage of their functionality. Correctly using an API often requires a profound understanding of it; 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 a separate 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 [22] 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.

1.1 Missing Method Calls

Of the many subtle mistakes a developer can make when working with an API, this work focuses on one specific type: missing method calls, which occur in the context of Object Oriented Programming (OOP). In OOP, 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 types 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 large codebase. In large software projects it is common that one developer does not know all of the code, and thus, they will often encounter types 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, even in mature code bases. In an informal review [14], 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 corner cases, but generally reveal at least a code smell, if not worse. They also performed an extensive analysis [15] 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. All in all, 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 requires manual effort and likely demands 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 to the point of being 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-crafted list of rules, but 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.

1.3 Contribution

To understand and tackle the missing method call problem, Monperrus et al. [14][15] 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 incorporate more data into its decisions.

To understand how well this approach works, we apply the system to a dataset of more than 600 open source Android applications and manually analyze the results. Besides the manual review, we also take the results of an automated benchmark into consideration. We examine which of the variations performs best and give insight on the prerequisites that this technique has with respect to input size and quality. Finally, we study if the majority rule is well suited for detecting superfluous or wrong method calls.

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 fix 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.

2.1 Finding Code Smells with hard coded 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.

Ayewah et al. [1] 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 errors, occur at locations which are unreachable or in situations from which recovery is

[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`

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. [23] 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 true defect. 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 [13] and iPlasma [12].

2.2 Android-specific Smell Detection

Verloop [24] uses Java refactoring tools (e.g. PMD[7] and JDeodorant[8]) to detect basic code smells like large classes or long methods in mobile applications. A noticeable finding is that smells can appear at different frequencies in classes inheriting from the Android

[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/`
[7]`https://pmd.github.io/`
[8]`http://www.jdeodorant.org`

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. [10] 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 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 well known and widely used ones like Facebook or Skype.

Another paper worth mentioning is the work by Murali et al. [16]. 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. [20] presented a catalog of 30 Android specific code smells, Palomba et al. [18] 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.

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"

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

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. [4] were probably the first to propose the general idea behind the majority rule. 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. Their approach is more general, whereas we are focused on detecting one specific error type. Conversely, our approach is more automated and does not need templates.

Pradel et al. [19] 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 [3] (more than 1.5 Million lines of code), they reach a precision of 72% and recall of 38% on seeded (i.e. "faked") anomalies. In real code, they found 29 anomalies of which 22 revealed true problems (76%).

In a related work, Rice et al. [21] 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. [6][5] 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. [28] 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 [2] 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.

Wang et al. [25] propose a method that leverages n-gram language models to detect bugs. Other approaches do not learn from the code at all, for example Hanam et al. [9] use unsupervised machine learning over bug fixes to extract bug patterns. For more research in the direction of learning API properties, we recommend the survey paper by Robillard et al. [22].

2.4 Detecting Method Call Anomalies

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. [27] 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. [14][15] 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.

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% and 76% of the simulated queries (depending on the package) and of the queries it can answer 73% to 89% actually contain the 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.

Therefore, 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.

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

Before this, Wasylkowski et al. [26] 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 [8] to combine the temporal properties into patterns.

In their 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. [7] 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. [17] 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. [14][15] which is based on the majority rule.

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 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 set 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 [15])

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 [15])

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 possibly missing 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 ([15])

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. [15] 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. Thus, next to the standard variant DMMC (Detecting Missing Method Calls) that we outlined in the previous sections, we also define a variant called $\text{DMMC}_{\text{noContext}}$, which does not take the context into account. It uses adapted measures for similarity and almost similarity, $E'_{\text{noContext}}$ and $A'_{\text{noContext}}$ 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. [14][15], 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. Here, 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 $\text{DMMC}_{\text{noContext}}$ 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 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 $\text{DMMC}_{\text{class}}$. 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 $E'_{\text{noContext}}$ and $A'_{\text{noContext}}$, respectively.

To formalize $\text{DMMC}_{\text{class}}$, we need the new operator $\text{Cls}(x)$ 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 $\mathrm{DMMC}_{\mathrm{superfluous}}$ 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, whether it is 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:

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

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. 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 $\text{DMMC}_{\text{wrong}}$ looking for erroneous method invocations, we again only need to change the definition of almost similarity. This time we define a new relation A'_{wrong} as follows:

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

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.

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 which are present. The extracted type usages are then persisted in a database to ensure

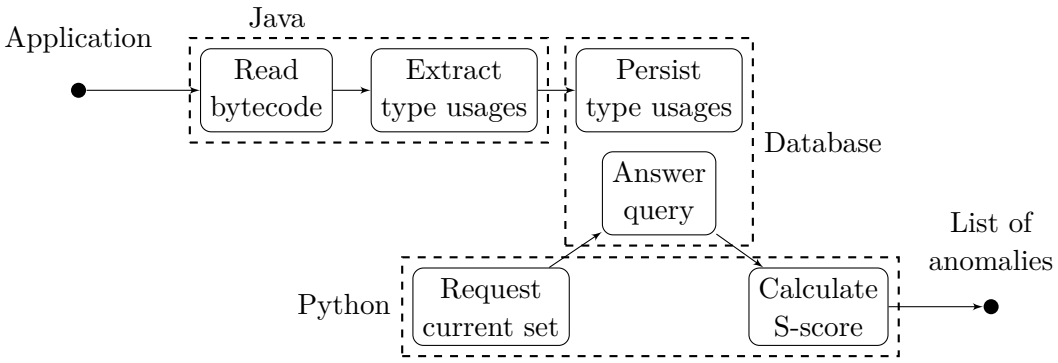


Figure 5.1: System overview

flexibility and high performance in the following analysis phase. For the actual anomaly detection, we use a small Python program. It iterates 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 as source code, we can compile it before analyzing it.

[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

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. More detailed explanation is available in the Soot documentation[2].

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

[2] `https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm`

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. [11], 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 -> right now it's a bit confusing with first talking about excluding them and then how they are important!

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[3], 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.

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

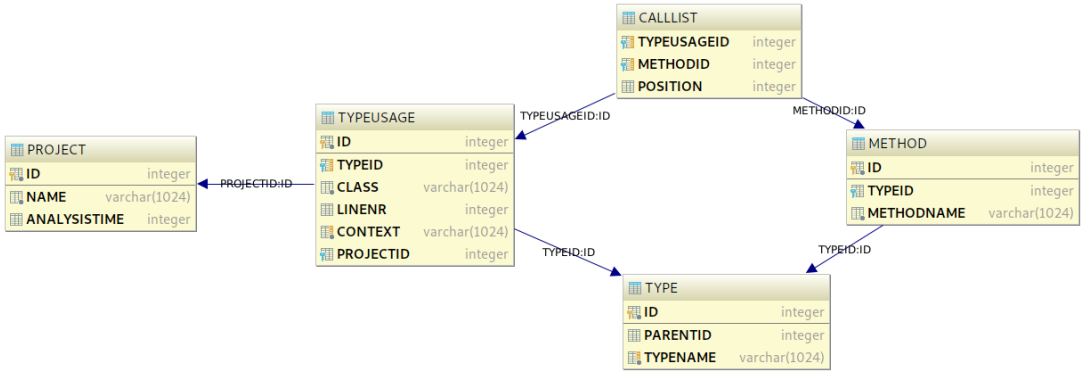


Figure 5.3: Overview of Database Tables

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 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.

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[4]. We started with their implementation and refactored it extensively[5] 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 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 pretty 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[6], 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 $\text{DMMC}_{\text{noContext}}$). 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

[4] `https://github.com/stg-tud/typeusage`
[5] `https://github.com/ke-kx/typeusage`
[6] `https://www.postgresql.org/`

continue to the final step of the analysis: detecting anomalies.

5.6 Anomaly Detection

Several simple Python scripts handle the anomaly detection. The first step is to partition the type usages by type for $\text{DMMC}_{\text{noContext}}$ or a combination of type and context for DMMC. The script obtains these partitions with a simple query to the database. In its outer loop, it 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 fast because of indices on the relevant columns.

In the inner loop, the script iterates through all 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 of the implementation that is affected by the different variants presented in Chapter 4. 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 definitions.

6 Evaluation

Our evaluation of the method proposed by Monperrus et al. is aimed at understanding how well the majority rule is suited to detect missing method calls in object oriented software. We apply our implementation to a large dataset of Android applications and evaluate the results qualitatively and quantitatively. In the following, we first present the research questions and why they are relevant for our understanding of this method. We examine the dataset, present some general data on it and analyze the properties of type usages in Android applications. After giving details on the study design, we present the results of our evaluation in response to our research questions. Finally, we consider the meaning of these results and what might threaten their validity.

6.1 Research Questions

Over the course of this evaluation, we would like to improve our understanding of the nature and quality of findings produced by our implementation of the DMMC system for detecting missing method calls. We answer the following more detailed research questions.

RQ 1: How many true versus false positives are among the anomalous type usages flagged by the majority rule?

It is among the primary goals of our evaluation to understand if the technique we implemented is something that a developer would use to improve the quality of the software he is developing. It seems that the deciding factor for this would be the ability of the majority rule to detect bugs and to detect them with clarity. If the developer needs to sift through thousands of anomalies to detect a single true mistake, the technique is not very useful. If, on the other hand, most of the high ranked anomalies are indeed hints for potential bugs, it would be of tremendous worth.

RQ 1.1: Is there a noticeable difference between the different variants?

In Chapter 4, we suggest $\text{DMMC}_{\text{class}}$ and Monperrus et al. already propose $\text{DMMC}_{\text{noContext}}$. We would like to understand how these variations behave in comparison to the original DMMC system. Are the results they produce of comparable, better or worse quality?

RQ 2: Do the benchmark results align with the results of the manual evaluation?

Monperrus et al. base a significant part of their evaluation on an automated benchmark with the aim of understanding how the majority rules behaves under different circumstances. Because they do not have a large dataset of known missing method calls available that they could use to assess the quality of their method, they create instances of missing calls by sampling a type usage from the dataset and removing one of its calls. They then calculate the strangeness score of this degraded type usage and determine if their implementation can detect the known missing call. Using this procedure, they can test the system on many cases for that they know the correct answer. Knowing the expected answer, they can calculate relevant metrics that capture the systems success and compare the different variants on an objective basis.

While such a benchmark makes sense on the face of it, one can question how meaningful are its results. Monperrus et al. are essentially simulating the situation that a developer forgets the method call they remove and verify if their implementation would be helpful in that case. Here, the critical assumption is that the artificially created missing calls are in some way comparable to real missing method call bugs in software. If they are not, the benchmark only measures how well the implementation performs on the exact problem of detecting randomly removed method calls, which is not very meaningful at all.

RQ 3: How many type usages are necessary to detect anomalies?

Apart from investigating the quality of results, it is also important to understand what prerequisites this technique has. Here, we examine how much data the majority rule needs to produce sensible results. Ideally, we would like to understand how big a codebase needs to be before patterns and their violations emerge. Can we apply this method to a in-house, closed-source library or does it only work on a well known open source framework like Android where it is easy to gather additional data?

RQ 4: How robust is the majority rule in the face of erroneous input data?

It is not only important to understand how much data the majority rule requires, we are also interested how sensitive it is to perturbed inputs. The type usages we extract originate from any input code and it is entirely feasible that there will be some erroneous code among it. In fact, we expect some errors to be present, otherwise we could not hope to detect any. However, the question is, how much correct input is needed to detect the incorrect usages? Imagine, for example, a new library that is not very well designed

causing many developers to use it in incorrect ways. Can we expect the majority rule to be of any help in such a scenario or will it just fail completely?

RQ 5: Can the majority rule also detect superfluous or wrong method calls?

In Chapter 4, we propose to use the method for detecting missing method calls that Monperrus et al. introduced to detect two other anomalies: superfluous and wrong method calls. We would like to understand if the variations $\text{DMMC}_{\text{superfluous}}$ and $\text{DMMC}_{\text{wrong}}$ can successfully detect bugs in software.

6.2 Study Objects

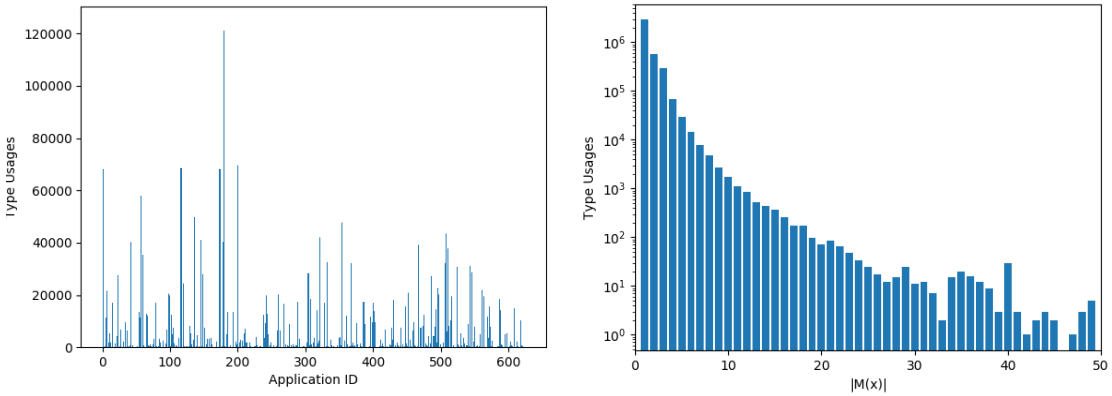
F-droid[1] is a repository for Free and Open Source software (FOSS) on the Android platform. We used a scraper to download the latest version of all available applications on the 6th of March 2018. From the 625 Android applications obtained in this manner, we extract all type usages that are present and apply our implementation of the DMMC system to identify any anomalies.

The Android ecosystem is particularly well suited for this evaluation and enables us to evaluate how type usages and the majority rule behave on real software. The Java code is easy to analyze and there is a rich open source community, placing many sample applications at our disposal. Each of these applications uses the Android API, so that we can generate a large dataset on the classes in the API, and detect any common patterns that emerge when using it.

Our dataset consists of a total of 3,880,556 type usages that the Java part of our implementation extracted from the 625 Android applications downloaded from F-droid. On average there are 6209 type usages per application and the median is 1157. In Figure 6.1a, we have plotted the number of type usages per application. Most applications are small with a few outliers responsible for most of the type usages. Of all the type usages in the dataset, 7.66% operate on types from the Android framework (i.e., their fully qualified package name starts with `android.*`). In Figure 6.1b, we have plotted how many method calls there are per type usage to give a feeling for the amount of information they provide. Observe the logarithmic scale on the y-Axis, indicating that a great majority of type usages only call few (<10) methods.

In the variation DMMC, a partition is a valid combination of type and context and there are 1,410,709 combinations like that in the dataset. In the variant $\text{DMMC}_{\text{noContext}}$, the number of partitions is equal to the number of types and the whole dataset includes

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



(a) Number of Type Usages per Application

(b) Length of Method Lists

Figure 6.1: Dataset Overview I

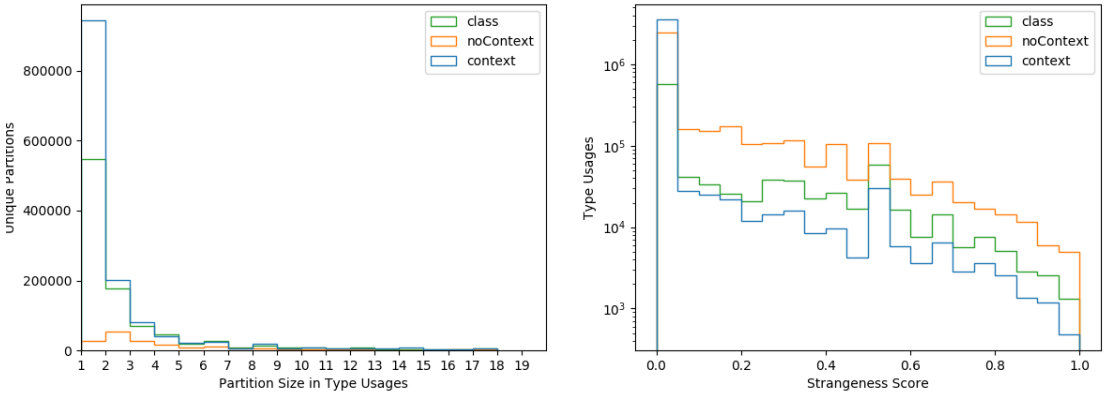
a total of 202,988 different types. Since $\text{DMMC}_{\text{class}}$ relies on $A'_{\text{noContext}}$, the partitions for it are the same as for $\text{DMMC}_{\text{noContext}}$. However, because we merge the type usages before analyzing them, the total number of type usages for this variant decreases to 958,306. In Figure 6.2a, we have plotted the distribution of type usages per partition for the different variants. Observer that especially for the variation that takes the context into account most type usages lie in a small partition.

a bit more than just "small partition?"

We have plotted the distribution of strangeness scores across all type usages in Figure 6.2b. Most type usages are normal and have a low or very low strangeness score. Using DMMC, our dataset contains a total of 1650 anomalies, that is an average of 2.64 anomalies per application. We perform all our experiments on a MacBook Pro with an Intel®Core™ i7-3720QM CPU @ 2.60GHz and 16GB of RAM. Extracting the type usages from all applications took a total of 3 hours and 27 minutes, which is an average of around 20 seconds per application.

6.3 Study Design

For RQ 1: True vs False Positives To determine if an anomaly that was flagged by our implementation is a true finding, we need to manually review it. Unfortunately, our implementation uncovered too many anomalous type usages to review all of them, prompting us to randomly select 10 applications for detailed analysis. We examine each anomalous type usages of these applications in detail and classify them as either true or false positive. This mimics the experience a developer would have when applying our



(a) Distribution of Partition Sizes (b) Strangeness Scores using DMMC

Figure 6.2: Dataset Overview II

tool to his software. To answer the research question, we consider the total number of findings per application and the ratio of true to false positive findings.

For RQ 1.2, whether the variations $\text{DMMC}_{\text{noContext}}$ and $\text{DMMC}_{\text{class}}$ produce different results, we apply them to the same 10 applications that we sampled for RQ 1. Once more, we review all findings and classify them as true or false positives.

For RQ 2: Benchmark Results To decide if the benchmark results coincide with the results of the manual analysis, we first perform the benchmark itself. That is, we sample type usages from the dataset, degrade them by removing one of their calls and check if our implementation can detect these known anomalies. Because we know the correct answer for all degraded type usages, we can use the results to calculate the classical information retrieval metrics:

Precision describes how good our implementation is at suggesting the correct missing call. It is calculated as the weighted confidence the system places on its suggestion.

Recall is the percentage of degraded type usages that were detected as anomalous. It tracks if our system correctly detects outliers.

$\mathbf{F_1}$ score is the harmonic average of precision and recall.

These metrics are already mildly interesting by themselves if we compare them across the three variants DMMC, $\text{DMMC}_{\text{noContext}}$ and $\text{DMMC}_{\text{class}}$. However, as we are not sure about their significance, we also want to evaluate the benchmark itself. To do so, we

compare the benchmark results of each variant to the results of the manual evaluation we performed for RQ 1. We would expect them to align, that is if one variant produces bad results in the manual evaluation it should also yield bad results in the benchmark. If there is a serious mismatch between the manual and the benchmark results, this would question the validity of the benchmark. However, if they agree, it would at least strengthen the idea that the benchmark has some relation to the real world performance of these techniques.

rephrase end of this paragraph - the second part does not actually follow!

For RQ 3: Input Size Even if we just questioned the validity of the benchmark, we still think that it can offer some insight into the working of our implementation. Thus, to understand how the quality of findings depends on the amount of input data that is available, we perform a similar kind of benchmark as for RQ 2. That is, we sample type usages, degrade them by removing a method call and then check if our implementation recognizes them as anomalies. We calculate the same metrics as in the previous experiment, but this time we consider one additional factor: the size of the partition. We aim to understand if the partition size has any influence on the quality of results and if so, what partition sizes are advantageous. In the discussion, we also present some mathematical considerations to support the results.

For RQ 4: Robustness It is difficult to give an accurate answer to the question how sensitive the majority rule is to low-quality input data. To narrow it down, we perform another simulation experiment, similarly to the one we do for the previous question. During the experiment, we again sample a type usage from the dataset and degrade it to create a known missing method call. However, this time, degrading only one type usage is not enough. Instead, we pick a second random type usage from the same partition as the first one and also remove one of its methods calls, before checking if our implementation can detect the known missing call. Thus, we simulate the situation that the type in question is in some way difficult to use and it has been used wrongly in multiple places. We can then calculate precision and recall as before and investigate how this lower quality input data affects the performance of our implementation. To get the full picture, we observe how the metrics change when we degrade more and more type usages at once.

For RQ 5: Other Anomalies To understand if the majority rule can also be used to detect superfluous or wrong method calls, we apply our implementation of $\text{DMMC}_{\text{superfluous}}$ and $\text{DMMC}_{\text{wrong}}$ to the 10 random applications we sampled for RQ 1. We manually review their findings in the same manner as we review those of the original variant and label them as true or false positives.

6.4 Study Procedure

The only parameter that our implementation requires is the limit that defines when a type usage is an anomaly. For this evaluation, we consider type usages with an S-score bigger than 0.9 to be anomalies. This is the value that Monperrus et al. suggest and since it implies that 90% of similar type usages are almost similar it seems like a reasonable cutoff.

6.4.1 Manual Review

We used a random number generator[2] to sample the 10 applications that should be examined in detail. The names of the selected applications can be found in Figure 6.1. We analyze all of their type usages with the three different variants DMMC, $\text{DMMC}_{\text{noContext}}$ and $\text{DMMC}_{\text{class}}$ and determine their respective S-scores. We then manually review all anomalous type usages and carefully consider the source code and the Android API documentation before classifying 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 in the given instance

hint (H) the code is fine, but the pattern is also legitimate; some connection between the methods

false positive (FP) the usage is totally fine; no causal relation implying the "missing" method must be called

We consider bugs and smells to be true positives. We classify type usages as hints when it makes sense that the pattern exists, because many people will use the methods together and there is some causal connection, but, strictly speaking, there is nothing wrong with the code in question. As an example, consider the situation in which a developer calls `next` on an iterator that he just obtained from a list of size one. Here, he knows that the iterator contains one element so he does not need to check for it with `hasNext`, nonetheless, it is useful to detect locations where `next` is invoked without a preceding `hasNext`.

6.4.2 Automated Benchmark

We perform the benchmark using a custom benchmarking infrastructure written in Python. It randomly chooses type usages from the dataset and applies a degradation

[2] Google frontpage on: `https://www.google.com/search?q=random+number`

Nr	Name	Type Usages	Findings	B	S	H	FP
1	ar.rulosoft.mimanganu_78.apk	11241	12	1	1	4	6
2	com.zeapo.pwdstore_94.apk	12659	1			1	
3	net.bitplane.android.microphone_7.apk	80					
4	com.quaap.dodatheexploda_2.apk	87					
5	com.health.openscale_23.apk	3645	3		1	2	
6	se.tube42.kidsmem.android_16.apk	13547	1				1
7	org.billthefarmer.diary_125.apk	880					
8	org.ligi.blexplorer_12.apk	8350					
9	org.kaqui_27.apk	1675					
10	com.afollestad.nocknock_13.apk	9547					

Table 6.1: The results of the manual evaluation

mechanisms to it. The normal degradation mechanism removes each method call of the selected type usage once and checks for each new version if our implementation of the DMMC system detects the known anomaly. To retain a reasonable performance and execution time, we only apply this procedure to a portion of the entire dataset (around 10%) and then calculate precision and recall as described above.

For the evaluation of RQ 4, we also support a different process of degradation. Instead of just removing one method call from one type usage, we randomly sample additional type usages from the same partition as the first one and remove one of their calls as well.

6.5 Results & Interpretation

In the following, we present our results relating to each of the research questions.

For RQ 1: Manual Review In Table 6.1, we summarize the results of manually reviewing the findings of the DMMC variation. In the entire dataset, on average there are 2.64 anomalous type usages per application, in our sample there are 1.7 findings per application. Of the total 17 findings, 3 are true positives (bug or smell) meaning that 17.65% of findings are true findings.

Most of the applications do not exhibit any anomalous type usages and those that do only have a few. There does not seem to be a relationship between the number of type usages and the number of findings per application. One of the applications is responsible for the majority of findings and also for the majority of false positives. However, it is also the only one in which a finding indicates a true bug. In fact, we reported this bug

App	$DMMC_{noContext}$ Type Usages	Findings	B	S	H	FP	$DMMC_{class}$ Type Usages	Findings	B	S	H	FP
1	11241	41 (24)	1 (1)	1 (1)	20 (15)	19 (7)	4602	28 (14)	1 (1)		8 (5)	19 (8)
2	12659	16 (3)				16 (3)	1560	10 (8)			1 (1)	9 (7)
3	80	0 (0)					57	0 (0)				
4	87	0 (0)					74	0 (0)				
5	3645	8 (6)			3 (1)	5 (5)	1918	7 (2)				7 (2)
6	13547	23 (0)			7 (0)	16 (0)	299	0 (0)				
7	880	21 (2)				21 (2)	427	21 (2)				21 (2)
8	8350	4 (4)				4 (4)	730	2 (2)			1 (1)	1 (1)
9	1675	2 (1)			2 (1)		863	1 (1)			1 (1)	
10	9547	15 (9)				15 (9)	3876	4 (3)				4 (3)

Table 6.2: Comparison of $DMMC_{noContext}$ and $DMMC_{class}$. The numbers in brackets indicate findings on the project itself.

and the maintainer of the application already confirmed and fixed it[3].

In Table 6.2, we present the results of applying $DMMC_{noContext}$ and $DMMC_{class}$ to the same applications. When ignoring the context, the total number of findings grows to 130 of which only 2 are true positive (1.54%). Since this are a lot of anomalies, we apply a filter to the results that removes any anomaly that does not originate from a class that belongs to the application itself. As a developer these are the most interesting anomalies, because they can be fixed immediately. Other anomalies that originate in libraries or other external code will be much harder to fix as they are usually maintained by different people. The number of findings that remain after applying this filter are displayed in brackets. The filter reduces the total number of findings to 49 of which 2 (4.08%) are true positives.

The class merge variation detects a total of 73 anomalies with 1 true positive among them (1.37%). By excluding the type usages that do not originate form the application itself, we reduce the total number of anomalies to 32 of which one is a true positive. And to clarify, the bug that all of the variations uncover is always the same.

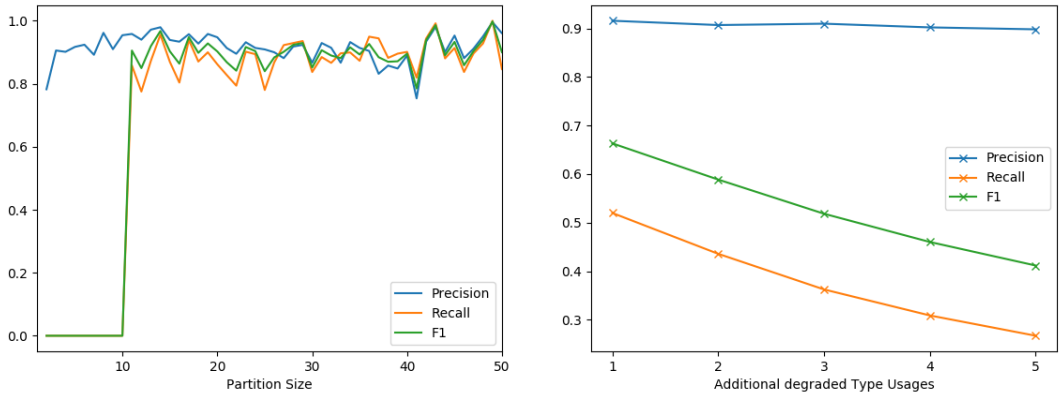
For RQ 2 and 3: Benchmark First, we present the benchmark results of the different variants DMMC, $DMMC_{noContext}$ and $DMMC_{class}$ in Table 6.3. The original variation has the best precision, but its recall is a little bit worse than that of $DMMC_{noContext}$. Nonetheless, it narrowly reaches a better F_1 score. The class merge variation produces by far the worst results.

We have plotted the development of precision and recall in dependency of the partition size in Figure 6.3a. Especially small partitions with fewer than 10 type usages have very bad recall, we explain the reason for this in the discussion. Because of this, we have also

[3] `https://github.com/raulhaag/MiMangaNu/issues/535`

	All Partitions			Partitions > 9		
	Precision	Recall	F_1	Precision	Recall	F_1
DMMC	88.96%	43.52%	58.45%	90.12%	80.5%	85.04%
$\text{DMMC}_{\text{noContext}}$	59.22%	57.55%	58.37%	60.06%	64.03%	61.98%
$\text{DMMC}_{\text{class}}$	38.89%	24.18%	29.82%	47.27%	40.05%	43.36%

Table 6.3: Benchmark Results



(a) Metrics depending on Partition Size (b) Impact of degrading more Type Usages

Figure 6.3: Benchmark Results for RQ 2 and 3

calculated the performance of our implementation for partitions that contain at least 10 type usage, they are indicated on the right side of Table 6.3. For these partitions, the superiority of DMMC over the two other variations is even more clear.

For RQ 4: Robustness In Figure 6.3b, we have plotted the evolution of precision and recall as we degrade more and more type usages. Note that these are the results of the DMMC variant on partitions that contain at least 10 type usages. While the precision stays relatively constant, the recall is decreasing rapidly as we introduce more errors into the input data.

For RQ 5: Other Anomalies The results of applying the two variations $\text{DMMC}_{\text{wrong}}$ and $\text{DMMC}_{\text{superfluous}}$ to the randomly selected applications are depicted in Table 6.4. Both methods detect a lot of anomalies. During the manual review we did not find a single true positive among them or even a smell or hint, that is, all of these anomalies

Nr	Type Usages	$DMMC_{superfluous}$ Findings	$DMMC_{superfluous}$ On Project	$DMMC_{wrong}$ Findings	$DMMC_{wrong}$ >1 Call	$DMMC_{wrong}$ On Project
1	11241	51	43	95	18	16
2	12659	22	19	46	5	5
3	80	1	1	1		
4	87	6	6	8	2	2
5	3645	22	20	33	9	9
6	13547	8	1	1		
7	880	4	4	2		
8	8350	5	2	2		
9	1675	4	4	5	1	1
10	9547	26	11	39	9	9

Table 6.4: Anomalies flagged by $DMMC_{superfluous}$ and $DMMC_{wrong}$

are false positives. We again filtered the findings to only include those type usages that originate from the application and not from libraries, however this only reduces the number of findings a little bit. Before filtering, $DMMC_{superfluous}$ flags a total of 149 type usages as outliers and after, 111 remain.

The high number of findings of $DMMC_{wrong}$ is due to a general flaw with the definition of A'_{wrong}, which we only realized during the evaluation. The problem is that all type usages that have the same type and make just one call are considered almost equal to each other. Since the majority rule considers type usages with many almost equal instances to be outliers, this causes the strangeness score of type usages with only one call to be extremely high. All in all, there is a lot of noise in the data (even if we take the context into account) and the results are nearly meaningless. Thus, we filter all type usages with only one call from the anomaly list, which reduces them considerably. However, even the filtered anomalies contain only false positives.

6.6 Discussion

For RQ 1: True vs False Positives The results of the manual review show that some of the anomalous type usages indeed indicate true bugs or smells. The number of anomalies per app is relatively small, such that a developer could easily evaluate them himself. While the manual review was time-consuming, this is partially because we do not know the applications, a developer that is familiar with them would be much faster. Because of the small sample size, we consider the actual percentages of true positives to

be of little import.

Besides manually reviewing the anomalies in 10 random applications, we also performed a side study on the top 50 anomalies of all applications. Over its course we did not uncover any additional bugs, but instead a lot of false positives. Most of these false positives arise out of a small number of failure modes such as:

- suggesting a missing call to `<init>` (basically never valid, usually the object is a parameter of the method or created by a static function)
- not following static functions (in which the suggested missing method is called)
- detecting not existing patterns with `StringBuilder`
- problems with method chaining (does not consider the returned object to be the same as before)

Some of these could be fixed rather easily, e.g., we could filter all type usages related to `StringBuilder` or anomalies that suggest a call to `<init>` is missing. Such filters would reduce the number of false positives, but they are also antithetical to the initial idea of avoiding hand crafted rules. Other problems are more difficult to remedy. Correctly detecting method chaining seems hard or will at least have some trade-offs. It seems possible to consider static function evaluations, but we expect it to be expensive in terms of performance.

As for RQ 1.1, regarding the comparison between the different variations, it seems clear that the variants $\text{DMMC}_{\text{noContext}}$ and $\text{DMMC}_{\text{class}}$ are inferior to the normal DMMC system. Not only do they flag more anomalies, the anomalies that they discover are also less relevant. They find a lot of anomalies in libraries and external code, but even after filtering these out, DMMC is still better.

For RQ 2: Benchmark Results For this question, we compare the results of the benchmark with those of the manual evaluation. In the manual review, DMMC is the superior method by far. From the benchmark results this is not as clear, it has the best precision but the recall is worse than that of $\text{DMMC}_{\text{noContext}}$. If we consider the benchmark results for partitions that have at least 10 type usages, DMMC becomes the sole leader. This is due to the fact that $\text{DMMC}_{\text{noContext}}$ does not have many partitions that are smaller than 10 and, as we will explain in the next paragraph, 10 is the crucial minimal size for this method to detect anything. Thus, it seems fair to conclude that the results of the manual evaluation align with the benchmark results. This is not enough to conclude that the benchmark itself is a good or realistic measure for the performance of different missing method call detectors. However, at least our results at least do not indicate the opposite: that the benchmark is a bad measure of their performance.

somewhere give a GENERAL aussage like: its good or not so much!

fyi the filter only removes 2 FP anomalies form the normal variation

not super happy with the end...

For RQ 3: Input Size The graph in Figure 6.3a shows that the systems accuracy increases with the size of the partition. Especially for partitions smaller than 10, the recall is terrible. To explain this, consider the strangeness score of a true bug b. If there are no instances equal to b, then $|E(b)| = 1$ and S-score$(b) = 1 - \frac{1}{1+|A(b)|}$. We consider type usages with a strangeness score of more than 0.9 to be anomalies, so b can only be flagged as anomalous as soon as there are at least 9 instances that are almost similar to it. It follows that there must be at least 10 type usages within the same partition to have even a theoretical chance of detecting any outliers among them.

Now consider that the DMMC variant that takes the context into account has shown the best performance and the distribution of partitions presented in Figure 6.2a. Even in our large dataset of 625 applications, most partitions are still rather small. In fact, a majority of type usages belong to partitions that are smaller than the necessary minimum of 10, meaning that our system can not actually detect any bugs or other outliers among them. All in all, it seems that it is difficult to obtain the necessary rich dataset, even when taking a lot of code into consideration.

more clearly work towards conclusion: it is difficult to get the right kind of data!

For RQ 4: Robustness The graph in Figure 6.3b makes it plain that the accuracy of our implementation decreases tremendously when we introduce additional degraded type usages into the data. Since we are not confident in the applicability of the benchmark, we also make a mathematical consideration with regards to the behavior of the S-score. In the previous paragraph, we already considered the true bug b with no equal instances. We need at least 9 almost similar instances to detect it. Now, consider another true bug b' that has one exactly similar neighbor, that is $|E(b')| = 2$ and thus S-score$(b') = 1 - \frac{2}{2+|A(b)|}$. This means, we can only detect these anomalous type usages if there are at least 18 instances that are almost equal to them, double as many as before.

The more errors there are in our dataset, the more data we need to detect them. The impact of this becomes especially clear if one consults the graph in Figure 6.2a displaying the distribution of partition sizes in our dataset. Most of them are small and thus having an additional mistake in the dataset can easily prevent our implementation from flagging both as an anomaly. All in all, we tentatively conclude that the majority rule is relatively sensible to erroneous input data.

For RQ 5: Other Anomalies Our implementations of $\text{DMMC}_{\text{superfluous}}$ and $\text{DMMC}_{\text{wrong}}$ both detect a relatively high number of findings without any actual success. More damningly, upon manual review of the findings it becomes clear that the patterns which they extract do not make any sense. None of the suggested changes, be it removing a method call or exchanging one, does have any causal justification and it seems clear that they

are only artefacts of randomness in the data. All in all, we do not consider our naive application of the majority rule useful for detecting superfluous or wrong method calls.

6.7 Threats to Validity

There are several internal and external threats to the validity of this study.

As for the internal threats, the strongest one seems to be that our manual evaluation is biased and subjective. Unfortunately, we do not have the resources to let multiple developers review the anomalies. To prevent judging the results too harshly, we introduce the hint category which contains many findings that are not bugs or even smells, but rather subtle pointers for something that could cause problems. If we acknowledge them as true positives, we get a favorable picture of the capabilities of our implementation. On the other hand, we can also exclude the hints and thus, obtain a lower bound. The truth probably lies somewhere between the two.

The only parameter that our implementation uses is the strangeness value above that a type usages is considered an anomaly. We could set it lower to improve the recall or higher in an attempt to improve the precision. We cannot present any hard data on this, but during the manual review, we often observed that even type usages with a strangeness value much higher than 0.9 were not really true positives. Because of this, we deem it unlikely that reducing the cutoff to a lower value would improve the results. As for increasing the cutoff, we think that the average number of findings is acceptable and does not need to be lowered.

We already mention the problems of the automatic benchmark in Section 6.1. It is simply not clear how similar degraded type usages are to missing method calls in the wild. Thus, to prevent relying too much on the benchmark, we also take some mathematical considerations into account before answering RQ 3 and 4.

Regarding the external threats, we have to review our study objects. We use a large dataset and choose the applications for manual review at random, to avoid getting results that rely on outliers. To hedge against underrating our implementation, we performed a side study on the top 50 anomalies among all applications. However, of course all study objects are Android applications and open source. It is feasible that this method works much better on another OOP language or simply on another Java framework. Finally, it is not clear if open source applications are comparable to commercial software in terms of size or quality.

7 Conclusion

In this thesis, we investigate a method for detecting missing method calls that was proposed by Monperrus et al. Detecting missing method calls would be useful over the entire lifetime of a software, be it when developing a new program or when maintaining old and mature software. The method we implement does not require any input besides the source code itself as it extracts patterns from the data it receives. This makes it superior to other bug detection systems, because it is not necessary to keep any hand-crafted detection rules up to date.

We improve upon the implementation of Monperrus et al. in that we add a database backend and enhance the extensibility. Furthermore, we propose several variations to their method with the goal of better detection of missing calls on the one hand and detecting different kind of errors on the other. To get an all-round impression of the results that their method produces and how our variations behave in comparison, we evaluate them on a large dataset of Android applications.

As for detecting missing method calls, the results of benchmark and manual review agree that the original version proposed by Monperrus et al. is better than the variations that we suggest. Neither ignoring the context, nor merging the type usages on class basis improve the results. Our implementation finds one true bug in the random applications that we reviewed, but most of the anomalies were in fact false positives. Apart from the mediocre quality of the suggested findings, our results also hint at the method being quite sensible to erroneous input data. Additionally, it seems difficult to gather enough data for the system even when applying the system to a large software ecosystem like Android.

improve this sentence!

All in all, we conclude that the majority rule can detect missing method calls under the right circumstances. However, the necessary prerequisites are not easy to achieve and the false positive rate is high. As for using the majority rule to detect superfluous or wrong method calls, at least our simple ideas do not work well. Our implementation finds a lot of anomalies and the manual review shows all of them to be false positives.

7.1 Future Work

There are two dimensions along which we envision future research. The first dimension concerns itself with improving the detection of missing method calls in some way. The

second dimension is about applying the majority rule to other types of anomalies.

With regards to improving the missing method call detection, we would first like to reference our analysis of typical failures in Section 6.6. There, we already address a number of potential improvements as well as their drawbacks. Especially filters seem like an easy way to decrease the number of false positives and thus increase the precision, albeit they are a bit antithetical to the original idea of staying away from hand-crafted rule sets. Another way to improve the precision could be to use a completely different anomaly detection algorithm. We did actually invest quite some time into research in this direction but the preliminary results were disappointing, presumably because the type usage abstraction does not contain a lot of data from which to learn from. However, we still see potential there, especially when also considering additional features, for example, the interfaces a class implements or the methods it overrides.

We question how valid it is to partition the type usages by the context they appear in and suggest two other options for it ($\text{DMMC}_{\text{noContext}}$ and $\text{DMMC}_{\text{class}}$). While our evaluation shows that grouping them by context produces the best results, there might be even better ways of organizing them. We would be especially interested to see if the results can be improved by factoring in the inheritance hierarchy of the classes that the type usages appear in.

Since the majority rule is essentially looking for deviations from the norm, having more data on what the norm is, could certainly improve the finding quality. We already consider a sizeable dataset in our evaluation, however the ecosystem of existing Android applications is by far larger than that. One could envision analyzing a large portion of the Google playstore and extracting all of the type usages to obtain more accurate results. We do not expect the performance of our implementation to be a problem any time soon, however, as soon as any problems in that area emerge, it would be easy to optimize it by using a dedicated database server and Cython or any compiled language for the analysis phase.

Besides these measures to improve the detection of missing method calls, one could also envision using the majority rules to detect other types of anomalies. Monperrus et al. suggest applying concept of almost similarity to execution traces to detect runtime defects or to conditional statements to improve the resilience of software to incorrect input.

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.