DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Nils-Jakob Kunze

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Maschinelles Lernen von Analyseregeln zur Erkennung von Fehlermustern

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

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

Munich, May 15, 2018 Nils-Jakob Kunze

Acknowledgments

don't forget!

mention: torben 4 grammarly the readers maybe thesis workshop (parents?, roommates, ...)

Abstract

Software reuse is a central pillar of software development, enabling developers to rely on frameworks and libraries to do the main work. Because of the great complexity and functionality provided by current frameworks, often their application programming interface (API) cannot be trivial. This and the fact that their documentation is often not kept up to date can led to errors when developers invoke these APIs.

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

To investigate this, we analyze more than 600 open source android applications. We manually evaluate the proposed findings of X randomly selected apps and also use an automatic benchmark which relies on artificially degrading existing code. We compare the originally proposed technique to some slight variations of it and find [...]. The results are kind of mixed, but point in the direction of this method being marginally useful.

more concrete research question

describe results + concrete numbers

Contents

1 Introduction

Developers often outsource a significant amount of work to existing libraries or software frameworks, which expose their functionality through an application programming interface (API). These APIs would ideally be simple and easy to use, but there is always a trade-off between usability and flexibility. Especially more powerful frameworks cannot provide a trivial API without limiting developers ability to use their functionality in the way most appropriate for the use case at hand. As a result, invoking an API correctly often requires a lot of knowledge about it. It can be of constraints or requirements which are not clear from the outset, but which have to be heeded to avoid serious bugs or complications, or about an involved interplay between different parts of the library. In the worst case, this knowledge might not even be contained in the API documentation.

Even if the documentation is of high quality, the complexity of some libraries makes it inevitable that there will be erroneous invocations of their APIs. The mistakes can relate to parameter choice, method order, or many other factors (e.g., some methods must be invoked in an extra thread, some specific precondition has to be satisfied, or some setup work must be performed). 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 which the correct invocations have in common. These patterns can take a lot of different forms and shapes, for example "call method `foo` before calling method `bar`", "if an object of type `Foobar` is used as a parameter to method `baz`, condition X has to be fulfilled", "never call method `qux` in the GUI thread", etc. For the Java examples mentioned above they could take the form of "an object which implements `equals()` must also implement `hashCode()`", "if object A equals object B, their `hashCode()` must also be equal" or "a call to `next()` on an iterator should be preceded by a call to `hasNext()` to ensure that it actually contains another object".

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

1.1 Motivation

consider the structure suggested by Williams: prelude, shared context, problem, solution, not yet 100 percent happy

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

Developers will fail to make important method calls when they are using classes with which they are not very familiar. As this can happen whether they are working with a new library, an extensive framework or even a whole platform (e.g., Android), we will use these words interchangeably through this work. These unfamiliar classes can be from an external source, but this is not a necessity. In large software projects it is common that one developer does not know the whole codebase, and thus, they will often encounter classes which they do not know how to use correctly. Altogether it seems simple to make an error related to missing method calls.

We can also confirm this intuition with hard data. In an informal review, Monperrus et al. found bug reports[1] and problems[2] related to missing method calls in many newsgroups, bug trackers, and forums. The issues range from runtime exceptions[3] to problems in some limit cases, but generally reveal at least a code smell if not worse. In addition to the informal review, Monperrus et al. [8] also did an extensive analysis of the Eclipse bug repository. First, they searched for syntactic patterns which they deemed related to missing method calls, such as: "should call", "does not call", "is not called" or "should be called". Manual inspection then confirmed 117 of the 211 (55%) obtained bug reports as indeed related to a missing method call.

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

[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`

and more manageable.

1.1.1 Detection - but how?

A simple and straightforward approach for 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 `foo()` also call `bar()`"
- "when calling `next()` on an Iterator always call `hasNext()` (before)"

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

To circumvent this problem, we would like to automatically detect locations in a code base where a method call is potentially missing 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. While the discovered locations will probably not be as accurate as those discovered by a hand-crafted list of rules, they could then be examined by an expert who would determine the severity of the finding and issue a fix if necessary.

1.2 Contribution

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

We evaluate this concept by applying it to a data set of more than 600 open source android applications and performing a manual evaluation of the results. We further

experiment with small changes to their initial idea and put them under the scrutiny of an automated benchmark. Additionally, we compare the results of the manual evaluation against those of the automated one and consider what this means for future research.

actually include this?

mention some results!

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 some variations which we will compare in the evaluation. We explain the inner workings of our implementation in Chapter 4, before summarizing the details and results of the evaluation in Chapter 5. The last Chapter 6 concludes this work and gives an outlook for future research.

2 Related Work

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

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

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

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

2.1 Finding Code Smells with Static Analysis

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

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

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

links!

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

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

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

2.2 Android specific Smell Detection

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

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

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

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

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

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

2.3 Inferring Properties from the Code at Hand

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

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

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

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

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

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

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

In a more general attempt Fontana et al. [4] compare 16 different machine learning techniques for code smell detection. As their training data they manually evaluated 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, however J48 and random forest have the best results, while support vector machines fare the worst.

2.4 Previous work on detecting missing method calls / object usages

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

results of monperrus et al.:

Before this, Wasylkowski et al. [14] introduced a method to locate anomalies in the order of methodcalls. First, they extract usage models from Java code by building a finite state automata for each method. The automata can be imagined similarly to the control flow graph of the method with instructions as transitions in the graph. From these they mine temporal properties, which describe if a method A can appear before

another method B. One can imagine this process as determining if there exists a path through the automata on which A appears before B, which in turn implies that a call to A can happen before one to B. Finally, they are using frequent itemset mining [5] to combine the temporal properties into patterns.

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

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

mention more related work from detecting mmc paper?

2.5 Other Ideas

change title

[too detailed?] As already mentioned, incorrect API documentation can be a big problem. It often occurs when an API developer changes or improves 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 even worse, the documentation will not only be unhelpful, it might even make matters worse. To alleviate this problem, Zhou et al. [15] 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. One important assumption they make is that the code is correct, because contrary to the documentation 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 oder logic formula. Finally the resulting formulas are fed into an SMT (satisfiability modulo theories) solver [2] to detect inconsistencies between them. Using this, they are able to detect four classes of documentation defects. 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 some other packages and find 1188 defects of which 659 are true positives (a precision of 55.5%). Overall it is worrying (maybe not surprising), that there are so many inconsistencies in the JDK documentation, which has a reputation for being of rather high quality. Less well known APIs will probably have a inferior documentation and thus even further increase the risk of API usage relates bugs.

mention them?

3 Detecting Missing Method Calls

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

better intro...

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

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

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

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

3.1 Foundation: The Majority Rule

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

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

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

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

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

3.1.1 Type Usages

make paragraphs instead of subsections?

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

Formalized, given some code the type usages are extracted as follows: For each variable x, note its type $T(x)$ and the context $C(x)$ in which it occurs. Additionally, save the list of methods which are invoked on x within $C(x)$: $M(x) = \{m_1, m_2, \ldots, m_n\}$. If there are two variables of the same type, this results in two type usages being extracted (unless they refer to the same object, more on this in Section 4.1.1).

```
class A extends Page {
    Button b;

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

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

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

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

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

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

3.1.2 Exact and Almost Similarity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

mention it is NOT symmetric!

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

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

mention ignoring the context here?!

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

3.1.3 The Strangeness Score

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

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

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

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

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

mention that monperrus hold 0.9 as treshhold for smth being an anomaly and that this will be our cutoff oä

3.1.4 Which Calls are missing?

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

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

do not like the formulation, improve!

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

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

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

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

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

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

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

Figure 3.3: Example for computing the Likelihood of missing Calls

again reference source of this example!!!

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

3.1.5 Ignoring the context

move this section down below extensions, but state explicitly, that they DID do this, only the rest is mine?

The reason for integrating the context into the definition of similarity and almost similarity is the idea that 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. Monperrus et al. [8] find in their evaluation that not considering the context adds a lot of noise (not relevant items in $E(x)$ and $A(x)$) and ultimately using it improves the precision of their experiments. However, one might 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 generally? 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 artefacts of randomness.

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

how to call it?

We will shed some more light on the relative performance of the different variants in Chapter 5.

3.2 Extensions

more clearly state that this here is my work?

In the previous section we summarized the work of Monperrus et al. [7][8], in this one we will explore some possible modifications to their method. For instance, it might be possible to leverage the majority rule not only for detecting missing method calls, but also to identify superfluous or flat out wrong method invocations. Additionally, there are different ways of organizing the type usage data, which could yield better results.

In this section we will give the motivation and theoretical background for those modifications, in Chapter 5 we will explore further if they actually outperform the original.

3.2.1 Class-based merge

basically the claass based merge is kinda the same as the inheritance relation right? (different split to dataset, either method/not methodlevel (context) or compelte class, or smth which inherits from X?)

The $DMMC_{noContext}$ variant seeks to answer the question, if it makes sense to differentiate the type usages based on the method they appear in. Following this line of thought, it is also possible to 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 those methods invoked in one method body will belong to one type usage. It is possible though, that 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. Even if its not all of them, a class based type usage offers a much bigger window into the objects utilization and potentially this makes it easier to detect if some method is missing.

question another assumption behind the std variant

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

To formalize $DMMC_{class}$, we need the new operator Cls(x) which denotes the class from which a type usage was extracted. The set of type usages with the same type and class as x is defined as:

$$TC(x) = \{x_i \mid T(x) = T(x_i) \wedge Cls(x) = Cls(x_i)\}$$

Then we can calculate the set of new type usages as follows:

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

Recall that for exact and almost similarity we will use the version which ignores the context, so $C(x')$ is not actually that important.

mention some effects (such as much smaller dataset, blabla?) or somewhere else?

not super clean, what is x_0 :/, smth like rotate through all the tus until all of them have been used?

3.2.2 Investigate 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 also be easily adapted to look for two different anomalies related to method calls. First off, it is possible to simply flip around the notion of almost similarity and investigate if there exists a superfluous method call somewhere. Additionally, one can envision a developer to simply make the *wrong* call, rather than forgetting one or adding one too many.

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

Superfluous Method

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

maybe best rationalization: it is there, before trying it we don't actually know if its bad, could be that this is actually a type of error which has some impact

include some example? (either for use case or for formalization?)

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

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

Recall that the almost similar relation A' is not actually 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 all those type usages which call one method less than the input type usage to be almost similar. With the definition of S-score and ϕ as above, the scores will now indicate which type usage is calling anomalously many methods and which method might be the one that is too much.

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 is quite possible that she chooses the wrong method for a given task. Of course, it is possible that this type of error results in noticeable bugs much faster than a missing call, but again we do not know this before investigating.

example? maybe some sort of conversion bug?

do i even wanna mention this?

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

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

can I define this a bit better or not?

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 $M(y)$ are identical, that is, the sets of method calls they make differ but only by exactly one method.

anything more here?, better description?

4 Implementation

Generally: don't make this section too long I'd say!

after laying out the theoretical foundation of the dmmc method we examine (in this thesis) in the last chatper this chapter focusses on the implementation side details about decisions taken, pitfalls that had to be overcome and tradeoffs to be made and of course things which didn't work out

the first section focusses on the internal workings of the system we developed. explain the steps from program (code) to list of potential anomalies smth smth The second one revolves around the benchmark system, the basic idea behind it and how it is designed to ensure maximum flexibility

4.1 Procedure

our system is (primarily) a system for detecting missing method calls it works by statically analyzing some software application, extracting the type usages which are present and finally determining if any of them is anomalous by the Majority rule as described in Section + ref The end result should be a list of places which are potentially missing a method call

Given a software system, which shall be tested for missing method calls, conceptually this needs to be done:

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-score, together with the calls they are potentially missing

(the ones with a score above XX are considered to be an anomaly)

using a database + flexible python benchmarking / analysis -> proper overview of resulting system: java analysis -> db -> python -> somewhere a proper overview of the concrete steps that are taken + explanation of them?! - similar to monperrus2010 p7

these considerations give rise to the system design which is displayed in figure + ref actual practical implementation: first a java applications reads the byte code of the application and then iterates over all method definitions to extract the type usages which are present those type usages are then persisted in a database the actual anomaly detection is handled by relatively simple python program it requests the currently relevant set of tus (more on that in +ref) and calculates their strangeness score after iterating through all sets of an application it can output the results

nice image: java part - db part - python part 1. bytecode einlesen + type usages extracten 2. type usages abspeichern und nach anfrage rausgeben 3. tus anfragen und sets ausrechnen

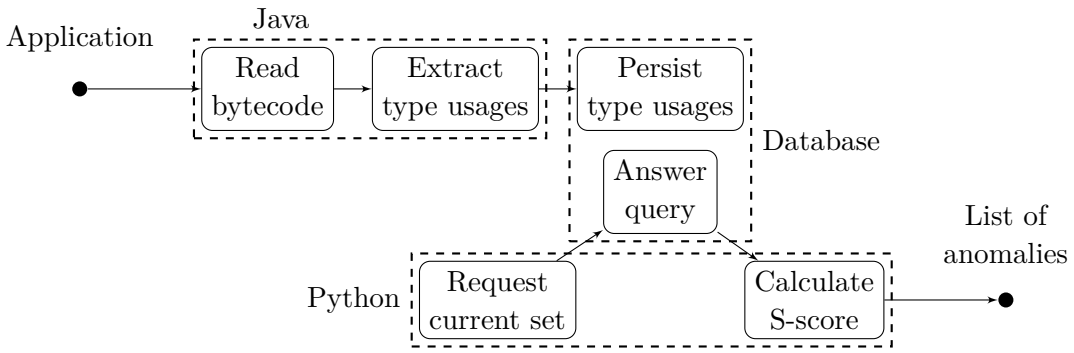

Figure 4.1: System overview

The following sections give more in depth information on the separate steps. smth mroe?

make a bit nicer (alles bisi auseinander ziehen?), better position dash box labels

4.1.1 Bytecode Analysis

maybe rename section? somewhere: if necessary: compile program, then analyze!

soot as analysis framework -> what is soot -> reference: `https://sable.github.io/soot/` originally java optimization framework wide range of usages, analyze, instrument, optimize and visualize java (and by extension android) applications provides call graph constructions, points-to analysis, def/use chains, inter and intra-procedural data-flow analysis, taint analysis we will use it for XY

operates by statically analyzing a piece of software (not on the source code, but rather on the compiled byte code -> why: easier) in theory soot would support sourcecode analysis(?), but actually the bytecode analysis is recommended (source?) easier for experiments more standardized (jvm byte code) [borrow smth from this one paper (don't remember which one...)]

transforms given programs into intermediate representation on which it operates there are four immediate representations (depending on use case) we use Jimple, the primary representation, typed 3-address intermediate rperesnetation (especially suitable for optimization, but also for our zweck)

soot is very powerful and one can customize it to carter to your specialized needs Mostly we will analyze android apps and the settings we are using reflect this -> table with settings + explanation?

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-c-j (apk-class-jimple)	Sets source precedence to format files
`-process-multiple-dex`	-	Process all DEX files found in APK
`-android-jars`	[path]	Use *path* as the path for finding the android.ja

explain the changes I made to their code

4.1.2 Extracting Type Usages

some more on extraction! + the algorithm! some information on what kind of further analysis is attempted (local must alias, bla)

where is it registered in soot and what does that mean? soots execution divided into a number of phases each phase has a specific task like for example the aggregation of local variables Phases are further grouped into Packs and the registered packs are applied succesevely during the analysis we can register custom parts which will apply some work in the specified phase register custom `BodyTransformer` with soot. in Pack: jtp -> stands for Jimple Transformation Pack and is simply applied to every method under analysis (unmodified soot has no transformations in this pack)

generally: applied to each method body first iterates through all statements and checks if they are a method call, if so remember them (aka extract list of method calls happening in this method) given this list of method call, iterate through it and group them to type usages this is done by checking first on which VARIABLE the call was made additionally, check if other locals point to the same underlying object -> group the calls made on these locals into one tu then group the calls which were made on the same object and be

happy (more details necessary? I don't really think so)

inaccuracy about "object" vs "variable" of a particular type? "LocalMustAliasAnalysis attempts to determine if two local variables (at two potentially different program points) must point to the same object. The underlying abstraction is based on global value numbering" -> this is a soot feature checking if two locals point to the same object can be expensive + seems a little bit leaky in test runs on big applications this has let memory requirements explode, thus we made it optional however for android was fine (even mention this?)

for android we chose to exclude some classes from analysis (do not step into them!, still record type usages!) because not the real code -> not really interested also performance reasons (quite important actualy) in theory soot has a mechanism for this, but it isn't working correctly packages thusly excluded are: `java.*`, `sun.*`, `android.*`, `org.apache.*`, `org.eclipse.*`, `soot.*`, `javax.*` (probably only mention java, javax and android?)

4.1.3 Storing Type Usages

after the transformer has analyzed and extracted a list of type usages form the current method body, they are persisted into a database using hsqldb -> link SQL relational database written in Java Provides small, multithreaded and transactional database engine in memory and disk based tables reasons for using this one? easy setup, small, fast permissive license, lots of features + we also invested some time into exploring the posibility to use postgres, however it turned out that the performance was worse -> see section (dead ends)

monperrus et all saved data in a text based format however there are a couple of disadvantages to this -> need to parse again for analysis, slow, large data storage using a database has many positive ramifications flexibility, future extensibility, easier to prototype fast (use python for analyis eg) queries! -> much better, need not hold everything in memory, especially for bigger datasets, can access data selectively also speed

somewhere(here?) mention, that it is not only possible but even SMART to not only incorporate the type usages of ONE application into the analysis but if at all possible, all of the software which uses the same kind of framework/library + explain why obviously of course it's useful to ALSO analyse tus over classes which are private to my application (if there is enough data) however, mostly interesting are the framework classes which are used by many applications when analyzing the usages of framework etc classes, we want to have as much information as possible

-> some sort of sql graphic? -> use the one I get from Intellij! + describe obviously (+ what was made how why)

4.1.4 Anomaly Detection

only part affected by the different variants presented in chapter 3 retrieving relevant type usages for the current variant + potentially modifying them + throwing them into the detector -> explain a bit the python class structure I developed?

maybe another img or flow chart for python ablauf (or too much detail?)

4.1.5 Improvements and Dead Ends

should this be a section or a subsection?

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

evaluation of other db system + why it didn't work (basically pure db stupidity)

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

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

static functions evaluation! something to fix the dotchaining problems

4.2 Benchmark

some more subsections? (different ways for degrading type usages?)

this only to be about the automatic benchmark stuff!

building benchmarking infrastructure flexibility with python class setup

5 Evaluation

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

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

5.1 Methodology

5.1.1 Qualitative Evaluation / Android Case Study

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

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

5.1.2 Automated Benchmark

degrading type usages + checking if they will be detected

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

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

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

5.2 Results

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

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

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

große frage: bringt das Verfahren etwas?

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

5.2.1 Dataset Overview

rausziehen (extra section)

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

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

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

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

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

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

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

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

5.2.3 RQ2: How "useful" is this technique?

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

change title...

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

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

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

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

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

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

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

only if time!

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

5.2.4 RQ3: How meaningful are the benchmarking results?

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

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

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

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

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

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

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

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

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

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

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

5.3 Discussion

5.4 Threats to Validity

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

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

6 Conclusion

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

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

6.1 Contribution

-> even have this section?

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

6.2 Future Research

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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