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

In the context of object-oriented programming, one of the errors related to API usages are missing method calls. They occur when a developer correctly instantiates some object but forgets to call one or more of the necessary methods. We study a new method for detecting missing method calls in Java applications which was proposed by Monperrus et al. [17]. The main goal is to understand how good this method is at detecting code smells and true bugs when applied to a large software system.

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

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

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

more concrete research question

describe results + concrete numbers

Contents

1 Introduction

Developers often outsource a significant amount of work to existing libraries or software frameworks, which expose their functionality through an application programming interface (API). These APIs would ideally be simple and easy to use, but there is always a trade-off between usability and flexibility. 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 [25] 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. [18] 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. [17] and further refined in a follow-up publication [18]. 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

ensure that everything is in present or only rarely past tense

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 + adoctor as well

It is difficult and time-consuming to fix bugs in software. This is because the maintainer has to realize that there is a bug, identify its cause and understand the steps necessary to remove it. Especially for larger software systems automatically detecting low-quality code and improving it can contribute significantly to the maintainability of the code and prevent bugs in the future. Because of this, there has been a lot of interest in approaches for automatically detecting bugs, or even just smells in code.

In this chapter, we present a number of different approaches for smell and bug detection. We start with some "conventional" methods which mostly rely on hard-coded rules and patterns. Since we are applying the method studied in this thesis to some Android apps in Chapter 5, 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 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 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.

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

In their work Ayewah et al. [3] apply it 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 not possible. In their analysis Findbugs detects 379 medium and high priority warnings which they classify as follows:

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

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

To alleviate the problem of a high false positive rate among the findings reported by Findbugs, Shen et al. [26] propose a ranking method which attempts to rank true bugs before less important warnings. It is based on a principle they call "defect likelihood", which is essentially the probability that a finding is a real finding. They calculate this probability using the findings of a large project (in this case the JDK), each of which they manually flag as either a true or a false positive. It 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%.

[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`

[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/`

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

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

some more info?

2.2 Android specific Smell Detection

smth more here? some bayesian stuff that actually works on android [19]

While not investigating Android specific code smells, Verloop [27] used Java refactoring tools (e.g. PMD and JDeodorant) to detect basic code smells like large class or long method in mobile applications. An especially interesting finding is that smells can appear at different frequencies in classes inheriting from the Android framework than in other classes, for example the long method smell was detected nearly two times as often in classes which do inherit from the framework.

some more info? + references for pmd and jdeodorant

Hecht et al. [12] present an approach they call PAPRIKA which operates on compiled Android applications but tries to infer smells on the source code level. From the byte-code analysis, it builds a graph model of the application and stores it in a graph database. The nodes of the graph are entities such as the app itself, individual classes, methods and even attributes and variables. These nodes are then further annotated with specific metrics relevant to the current abstraction level, e.g. “Number of Classes” for the app, “Depth of Inheritance” for a class or “Number of Parameters” for a method. Finally, they extract smells using hand-crafted rules written in the Cypher query language[7]. This enables them to recognize four Android specific smells and four general, OOP related smells.

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

After Reimann et al. [23] presented a catalog of 30 Android specific code smells, Palomba et al. [21] developed a tool to detect 15 of them. It works on the abstract syntax tree of the source code and uses specifically constructed rules to detect each of the smells. For the evaluation of their approach, they analyze 18 Android applications and compare the results of their tool against a manually built oracle. The oracle was

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

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 case 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 check if the developer used one of two popular compression libraries. However, recently a competing compression library has been gaining in popularity. Their rule does not include it and, thus, falsely flags the usage of this library as the "missing compression" smell. This is an excellent example of a case where hard-coded rules fail because they are not kept up to date (new library becomes popular, interface changes, ...), do not consider a special case or conditions change in some other way.

2.3 Inferring Properties from the Code at Hand

General: read the summary paper [25] again and check which approaches are there and which could / should be mentioned + for more information: check this summary paper / the 2017 one? / both? + do i want to merge this section and the next one?! (+ the engler paper kinda obviously belongs to the other section?)
not necessary only learning from the code at hand: [11] unsupervised machine learning over bug FIXES?!

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.

Why is this potentially better than other approaches (actually learning an api vs static rules comes to mind) some more übergang?!

Engler et al. [6] were probably the first to propose the general idea of learning from the code at hand instead of using predefined static rules. In their paper they rely on contradiction and common behavior to "find what is incorrect without knowing what is correct". To do this, they use static analysis to infer what a programmer believes about the system state and then check these beliefs for contradictions. They provide several "templates" for beliefs the programmer must or may hold. Each template is catered for a specific type of belief, they can be about a wide range of things (a pointer being null or not being null, a lock being 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

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

bugs related to the different bug types that their templates can discover.

Pradel et al. [22] noticed that in statically typed languages the compiler helps the programmer with passing method arguments in the correct order. However, the compiler cannot ensure the correct order of equally typed arguments (such as `setEndPoints(int low, int high)`. Confusing the order of these arguments can cause unforeseen and hard to detect errors. Their idea is to detect mixed up method arguments using only semantic information in the source code, namely variable and parameter names. They extract these and compare the names found at the call sites to the names from the implementation body using string similarity metrics. If reordering the names yields a significantly better similarity score, they report an anomaly. In their evaluation on 12 Java programs from the DaCapo Benchmark suite [5] (more than 1.5 Million lines of code), they reach a precision of 72% and recall of 38% on seeded (i.e. "faked") anomalies. In real code, they found 29 anomalies of which 22 revealed true problems (76%).

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

2.4 Focus on Object Usages

title sucks?! -> api misuses?

Mention MUBench?! [1] -> useful classification of api misuses Dataset of bugs related to API misuses
stuff related to "usage diversity" of classes [15], again monperrus involved though...
uses n-grams so intersting cause different approach? [28] + check related work section for some stuff that could be important!
also mention overview paper [2]

It is often difficult to correctly invoke API methods, not only because of their complexity, but also because of bad documentation. To circumvent this problem, developers often search for examples of how to use a particular API. To aid with this search, Zhong et al. [30] built a search tool for example code snippets. Their tool, MAPO, 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 objects usages and using those to find potential bugs. Most relevant for this work and its primary inspiration are two papers by Monperrus et al. [17][18] that propose a technique for detecting missing method calls. It is based on the intuition that an instance is probably an anomaly if it is alone in a large number of instances which do something similar but not exactly the same. They call this the "majority rulw" and we explain it in detail in Chapter 3.

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

The evaluation they present is split into two parts. The first is based on a simulation which creates defects by degrading real software. For the simulation they remove singular method calls from the code and check if they can detect these known missing calls. They apply this simulation to several software packages, among them the Eclipse user interface, Apache derby and Apache tomcat. Their system can answer between 54% and 76% of these simulated queries (depending on the package) and of the queries it answers 73% to 89% of the answers contain the actually missing call, albeit not necessarily as the first recommendation. All in all, it reaches precision between 59% and 83% and recall of 41% to 68%. However, it is questionable how relevant these numbers are, because it is not clear in how far real bugs will share characteristics with artificially created cases.

link?

For this reason, they also perform an additional, qualitative evaluation. They apply their tool onto 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. It can also be just a code smell, i.e., a situation which could be rewritten or refactored for more clarity, or related to software aging, for example code using an old version of the API. While they reported 17 issues and got 9 patches accepted, there are also some negative results, where the anomaly is just that: an outlier, which is totally correct, even if a majority of other usages behave differently.

Before this, Wasylkowski et al. [29] introduced a method to locate anomalies in the order of method calls. First, they extract usage models from Java code by building a finite state automaton for each method. The automata can be imagined similarly to the control flow graph of the method with instructions as transitions in the graph. From these, they mine temporal properties, which describe if a method A can appear before another method B. This process determines if there exists a path through the automata on which A appears before B, which in turn implies that a call to A can happen before one to B. Finally, they are using frequent itemset mining [10] to combine the temporal properties into patterns.

In this work, an anomaly also occurs when many methods respect a pattern, and only a few (a single one) break it. In their experiments, they find 790 violations when analyzing several open source programs. Manual evaluation classifies these into 2 real defects, 5 smells and 84 "hints" (readability or maintainability could be improved). This adds up to a false positive rate of 87.8%, but with the help of a ranking method, they can obtain

the 2 defects and 3 out of 5 smells within the top 10 results.

In a related work, Nguyen et al. [20] use a graph-based representation of object usages to detect temporal dependencies. This method stands out because it enables detecting dependencies between multiple objects and not just one. Here the object usages are represented as a labeled directed graph in which the nodes are field accesses, constructor or method calls and branching occurs because of control structures. Thus, the edges of the graph represent the temporal usage order of methods and the dependencies between them. They mine patterns using a frequent induced subgraph detection algorithm which builds larger patterns from small patterns from the ground up, similar to the way merge sort operates. Here an anomaly is also classified as a "rare" violation of a pattern, i.e., in relation to its size it does not appear frequently in the dataset. In an evaluation case study the authors find 64 defects in 9 open source software systems which they classify as 5 true defects, 8 smells and 11 hints, equalling a false positive rate of 62.5%. Using a ranking method the top 10 results contain 3 defects, 2 smells and 1 hint.

is this merge sort thing true?!

more work on temporal properties [9] mention more related work from detecting mmc paper?

2.5 Other Approaches

change title?

include this [24] (at all / in this section?) generally rethink the section structure...

As already mentioned, incorrect API documentation can be a big problem. It often occurs when an API developer changes the code but forgets to adapt the documentation accordingly. If a user of the API then relies on the documentation, she can run into unforeseen bugs, and what is worse, the documentation might even hinder her. To alleviate this problem, Zhou et al. [31] propose an automated approach for detecting "defects" in the API documentation. They consider two situations: either the documentation is incomplete in describing the constraints at hand, or the description exists, but it does not match the realities of the code. They make one crucial assumption: the code is correct because contrary to the documentation 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. Finally, the resulting formulas are fed into an SMT (satisfiability modulo theories) solver [4] to detect inconsistencies between them. With this approach, can 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 (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 relates bugs.

too detailed?

3 Detecting Missing Method Calls

In the introduction, we explored errors related to missing method calls. Especially when a developer is working with complicated frameworks and libraries and is not very experienced with them, it is easy to miss an important call. We would like to automatically detect instances of missed method calls to make maintenance easier and cheaper.

In this chapter, we give a proper explanation of the method proposed by Monperrus et al. [17][18] which is based on the majority rule. The main idea is to extract patterns from correct usages and then look for violations of these patterns. For this we define the notions of type usages, as well as their similarity and almost similarity. We 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 explore some tweaks of the method proposed by Monperrus et al. We suggest using their method to also detect anomalies related to superfluous or wrong method calls and a variation which uses a slightly different input ordering. How suitable these variants are for detecting errors we explore in Chapter 5.

3.1 Foundation: The Majority Rule

The intuition behind the majority rule is that if most or all instances do something in a particular way, this is probably the correct way. By implication this also means that if all or a great many behave differently than you, you are probably doing it wrong. This gives rise to the definition of an anomaly by the majority rule: instances which have few or none "equal" but a lot of "slightly different" neighbors. What exactly equal and slightly different means 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 of them the method `setText()` is invoked to describe the functionality of the button instance, but only 99 of them also call `setFont()`. There is one button which is missing this call and, therefore, uses the default font. It seems highly likely that this one button which is an exception to the rule "each button which invokes `setText()` should also call `setFont()`" is a mistake

and that 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 the standard Comic Sans!

The remainder of this section formalizes the different notions that are required to use the majority rule for detecting locations in code which are likely missing a method call.

intro still feels slightly lacking, but not sure WHAT is missing

3.1.1 Type Usages

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 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 we extract 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 4.2).

```
class A extends Page {
    Button b;

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

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

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

Figure 3.1: Example illustrating the Extraction of Type Usages (taken from [18])

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

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 or 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. Recall that the context is the name of the containing method together with the type of its parameters. The reason we call these type usages "similar" instead of "equal" because several things are not the same: variable names, surrounding or interspersed code, method order and the actual location (class) are all disregarded and might indeed differ.

explain better what a SIMILAR method is?

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 context and the list of method calls of y is the same as the method calls of x plus one additional call.

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

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

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

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

Figure 3.2: Examples of similar and almost similar Type Usages (also inspired by [18])

Consider the example code snippets in Figure 3.2. The type usages of `Button` in classes `A` (top left) and `B` (top right) are exactly similar, because they not only appear in the similar method `createButton`, but also invoke the same methods `setText` and `setColor`. Observe that variable names and method parameter have no impact on this. Additionally, it is not influenced by the order of methods or any interspersed code which 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 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 in comparison: `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., $xA'y \not\iff yA'x$ since one of the two type usages has to 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 looking for 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: $|M(y)| = |M(x)| + k, k \geq 1$ This means that we would look for type usages missing k calls instead of just one.

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 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 exactly 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. 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 $|A(b)| = 99$. 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.

3.1.4 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 collect the additional method calls that each almost similar type usage invokes. Recall,

that an almost similar type usage will make exactly one additional call in comparison to the anomalous one. Given a set of these additional method calls, we then take the frequency of unique calls among them and use this as the suggestion likelihood.

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

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

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

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

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

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

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

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

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

3.1.5 Ignoring the Context

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

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

3.2 Extensions

In the previous section, we summarized the work of Monperrus et al. [17][18], in this one we propose some possible modifications to their method. For instance, it might be possible to leverage the majority rule not only for detecting missing method calls, but also to identify superfluous or flat-out wrong method invocations. Additionally, there are different ways of organizing the type usage data, which could yield better results. In this section, we give the motivation and theoretical background for these modifications, in Chapter 5 we will explore further how they perform in comparison with the original.

3.2.1 Class-based merge

The $DMMC_{noContext}$ variant seeks to answer the question if it makes sense to group the type usages based on the method they appear in. Following this line of thought, 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 only the 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. Thus, 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.

We will denote this variant as $DMMC_{class}$. To obtain the class-based type usages, we extract the type usages at the method level, exactly as before. However, before calculating the strangeness score, we merge all those type usages which have the same type and originate from the same class. 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 $\text{Cls}(x)$ which 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:

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

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

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

a bit cleaner that we merge the tus in each of the partitions into one tu or smth like that

Recall that for exact and almost similarity we will use the version which ignores the context, so the context of the new type usage $C(x')$ is not that relevant.

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

3.2.2 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 look for 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 to invoke the *wrong* method, rather than forgetting a call or adding one too many.

some more?

Superfluous Method

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

include some example?

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

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

Recall that the almost similar relation A' is not symmetric and observe that in comparison to $A(x)$, here x and y are flipped in the application of A'. Thus, this definition of almost similarity considers 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 entirely possible that he chooses the wrong method for a given task. Of course, it is possible that this type of error results in noticeable bugs much faster than a missing call, but again we do not know this before investigating.

example? maybe some sort of conversion bug? (either for use case or for formalization?)

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:

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

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 $M(y)$ are identical, that is, that the sets of method calls they make differ by exactly one method. Using this definition of almost similarity, the majority rule can flag those type usages that make a unusual method call as an anomaly.

anything more here?, better description?

4 Implementation

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

After laying out the theoretical foundations of detecting missing method calls using the majority rule in the last chapter, 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.

4.1 Overview

Our system is (primarily) a system for detecting missing method calls. It works by statically analyzing a given software application, extracting the type usages which it contains and finally determining if any of them are anomalous by the Majority rule as described in Section 3.1. The end result should be a list of locations which are potentially missing a method call. Given a software system, which shall be tested for anomalies, conceptually 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-score, together with the calls they are potentially missing

make a bit nicer (alles bisi auseinander ziehen?), better position dash box labels | POTENTIALLY add the retrieve sets

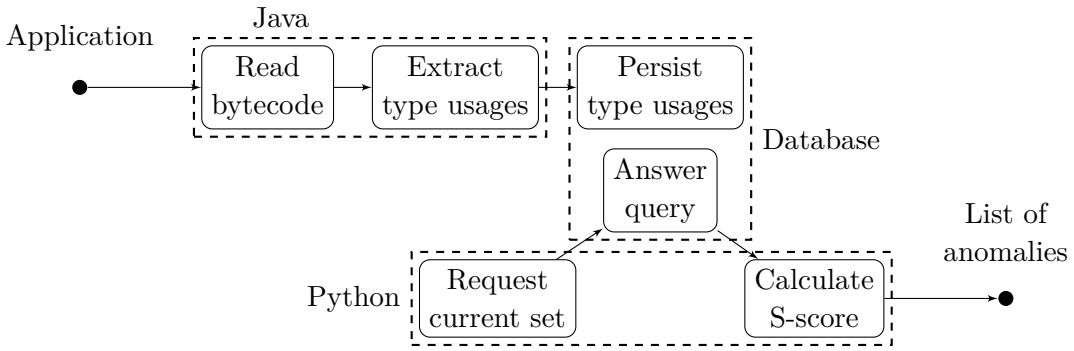

Figure 4.1: System overview

However, this simple outline does not represent the actual realities of the system. Instead of a singular process with [flat / not branching / ...] flow, it is split into three different parts which are laid out in Figure 4.1. First, a Java application reads the bytecode of the program under analysis and iterates over all method definitions to extract the type usages which are present. The extracted type usages are then persisted in a database to ensure flexibility and high performance in the following analysis phase. For the actual anomaly detection, we use a relatively simple python program to enable fast prototyping and again flexibility. It goes through all sets of type usages (more on those sets in Section 4.6), requests the type usages in the set and calculates their strangeness score. 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.

partitions instead of sets?

smth more?

4.2 Bytecode Analysis

maybe rename section?

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 usages. One can use it to 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 functionality, because we merely use it to statically extract type usages from the input application.

It would be possible to extract type usages from the source code, but extracting them from compiled code has some advantages. The JVM bytecode is very standardized, and

[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 4.2: The parameters passed to Soot

so is the Dalvik bytecode found in Android applications, which facilitates the analysis. While Soot supports source code analysis in theory, it only works up to Java 7 and bytecode analysis is the recommended approach. 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 simply compile it before analyzing it.

Soot operates by transforming the given program into intermediate representations which can then be optimized or analyzed. It provides four intermediate representations with different use cases; 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 carter quite specialized requirements. We will mostly analyze Android applications and the settings we are using reflect this. In Figure 4.2 we have listed the settings we are using to configure Soot.

more detailed description of settings? i think not

4.3 Extracting Type Usages

some more on extraction! + the algorithm! some information on what kind of further analysis is attempted (local must alias, bla) general / not sure where yet: explain the changes I made to their code ie proper refactoring, etc

The Soot execution is divided into 'phases' each of which has a specific task like for example the aggregation of local variables. The actual work is done by so-called transformations, which can modify the input they receive but are not required to do so. The phases are further grouped into 'packs', and the registered packs are applied

successively during the execution. We register a custom piece of code in the Jimple transformation pack, which is applied to every method in the analyzed program.

This custom transformation receives as input a Jimple representation of the method body it is currently analyzing. To extract the type usages, it iterates through all the statements in the method and marks those which are invoking a method. In the end, it has extracted a list of all method calls that are happening in this method. It now iterates through this list of method calls and groups them into type usages. To do so, it first checks on which variable or object the current call was invoked and checks if there is already a type usage for this object. If there is, it adds the call to the type usage and advances to the next method call in the list. If there is no type usage for the object so far, it creates a new one. After completing this process for all method invocations in the method body, the analysis is complete, and it returns the list of type usages it extracted.

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 if the calls made on these locals should be grouped into one type usage. The underlying abstraction is based on global variable numbering and follows the ideas presented by Lapowsky et al. [13] with some minor adaptions. This analysis is a Soot feature, and in test runs on big applications, we noticed that it can be very expensive in terms of memory and time. Because of this, we made it optional, however, for our benchmarks and evaluation, we were able to use it.

One additional detail is that we are excluding some classes from the Android analysis for performance reasons. 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. This means that we do not store type usages that occur *inside* of those classes, we do, however, still record usages of them.

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

4.4 Storing Type Usages

After the transformer has analyzed the current method body and extracted a list of type usages from it, we persist them into a database. For this, we are using HSQLDB[2], a SQL relational database written in Java, which provides a multithreaded and transactional database engine with in memory or disk-based tables. We chose HSQLDB because it is

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

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

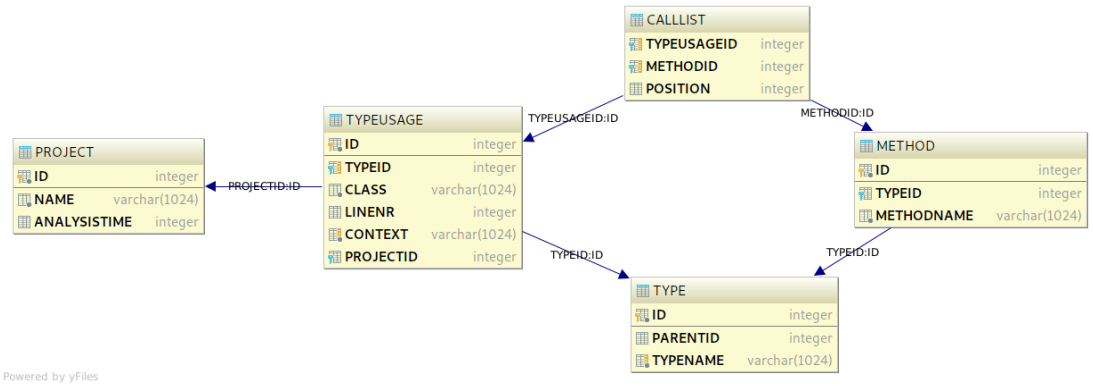

Figure 4.3: Overview of Database Tables

small and has good 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, but this means 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 off, 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.

Analyzing one application for missing method calls in separation can already yield useful results, especially if it is a large application. However, our real interest lies with the framework and library classes that are used in many applications. By analyzing many applications which use the same framework, we can build a large dataset of type usages on these classes and incorporate much more information into the decision if a type usage is an anomaly or not. Having a database backend facilitates building such a large dataset because it can store a lot of data and we can later query it selectively. Of course, it is still useful to analyze the type usages of classes which are private to some application (if there is enough data), but the framework classes are our primary focus.

more like our primary aim / goal / or even more they seem easier to hit?

REMOVE the xfiles watermark

Figure 4.3 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 what type the type usage has. Each type holds a reference to its parent type (if one exists) to enable rebuilding the inheritance hierarchy if so desired. Additionally, 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 holds the information which of the method calls that *could* be invoked on the type are actually called on the type usage in question.

+ some of the considerations made (x. normalenform, sowas?) -> ask torben? -> is there a problem with class and context (potentially duplicate information!, should actually belong to a class table i guess?)

4.5 Improvements and Dead Ends

Monperrus et al. released their code and data on GitHub[3]. We took their implementation and refactored it extensively 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. For their 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 without ensuring that $M(x)$ does not include any additional 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 it calls the additional method Z, even though they clearly do not fulfill the formal definition of almost similarity. However, we reran their experiments after correcting this small inaccuracy, and the results did not change in any significant manner.

add link to my code!

even mention this or just drop it?

After refactoring the code and adding the database backend, all of the type usage data was already in the database. For this reason, we thought it would be smart 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 for calculating the strangeness score was quite complicated with many joins over views containing many joins. Even on a small test dataset, it was quite slow. We attempted many improvements to the query, including better indices, smart cached subtables, etc. but all in all the query remained slow and memory intensive. We even invested some time into exploring the possibility of using PostgreSQL[4], but the results were disappointing.

include the db queries for the strangeness score calculation here? I am mostly thinking no

(on a application of medium size -> mention LOC of teamcale, resulting in one dataset with about X type-usages)

[3] `https://github.com/stg-tud/typeusage`
[4] `https://www.postgresql.org/`

not sure if this here or in next section?

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 a 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 candidates for exact or almost similarity are those type usages that have the same type and context as the type usage we are currently investigating. This gives rise to natural partitions of the dataset, namely those sets of type usages which share the same type and context (or just the same type for the variant $DMMC_{noContext}$). Within these partitions, we need to compare each type usage with all of the others, but it will be much fewer type usages to compare. Additionally, the database is already perfectly suited for obtaining these subsets. With this realization we can finally continue to the next step of the analysis: detecting anomalies.

change to more active way of description?

4.6 Anomaly Detection

need another picture? or fine without? i'm not sure i need that much detail... maybe another img or flow chart for python ablauf (or too much detail?)o somewhere explain the "sets of tus" + peformance issues before

this paragraph seems terrible?

Some relatively simple python scripts handle the anomaly detection. The first step is to obtain the possible partitions of the dataset, that is depending on the variant all types in the dataset or all valid combinations of type and context. Observe that the database makes it very simple to obtain these partitions with a simple query. In its outer loop, the script iterates through all partitions and retrieves the type usages belonging to the current partition. Again, querying the database for these type usages is simple and also quite fast because of an index on [column]. In the inner loop, the script iterates through all the type usages in the current partition and compares the methods that they invoke. By comparing each type usage with each other type usage only once similar to filling an upper triangle matrix, we can again save 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. We also save a list of the results for further analysis and evaluation by an expert.

include actual query? - i think yes, all of them in one figure?

refer to query + which column?

The anomaly detection is the only part that of the implementation that is affected by the different variants presented in Chapter 3. We already mentioned the differences

between including the context or not. For the class merge variant, we perform some simple preprocessing after loading the data with type partition and merge those type usages which originated in the same class before starting the analysis. For the detection of superfluous or wrong method calls, we can use exactly the same setup and only need to change the almost equal check to the new variant.

mention in the end that i tried a Better anomaly detection (is the anomality rule actually a GOOD measure for this kind of anomalies, or should we use something totally different?)) clustering detector try (+ hypersphere idea?) but all in all it didnt work out / not enough time + vermutungen warum es nicht funktioniert -> dataset exploration + size of method and typeusage lists per partition!

4.7 Other Ideas / Problems

not sure i need this section + what should be included

static functions evaluation! something to fix the dotchaining problems -> for both basically I'm just saying that i looked into it and it was too much work / impossible?

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

The XXX section revolves around the benchmark system, the basic idea behind it and how it is designed to ensure maximum flexibility.

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?)

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

5.1.2 Automated Benchmark

building benchmarking infrastructure flexibility with python class setup degrading type usages + checking if they will be detected some more subsections? (different ways for degrading type usages?)

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

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) number of partitions (types+context / types) and their sizes! 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)

mention somewhere: the monperrus2013 p11 highlight? (at least equivalent for my dataset)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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