DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Nils-Jakob Kunze

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Machine Learning of Bug Pattern Detection Rules

Maschinelles Lernen von Analyseregeln zur Erkennung von Fehlermustern

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

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

Munich, May 15, 2018 Nils-Jakob Kunze

Acknowledgments

Abstract

should for sure contain: - main idea of the thesis - my opinion or point of view - purpose of the thesis - answer to the research question (results!) (- element of surprise -> smth that is interesting and engaging and perhaps not expected) - CLARITY!

Contents

1 Introduction

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

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

include frameworks as well in some better way

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

Patterns like these are also called API usage patterns [9] and they can be used to detect potential defects. If some code in a software project deviates (too much) from the usual patterns when using an API, this hints at a bug or problem or is at least a code smell which should probably be corrected. This makes it interesting to detect these unusual instances.

1.1 Motivation

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

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

1.1.1 Examples from the Real World

Unclear APIs or frameworks and the resulting missing method calls are a real problem developers struggle with. Consider for instance this example from Monperrus et al. [4]: The developer Alice wants to create a dialog page for Eclipse. After some searching, she finds the corresponding class `DialogPage` in the API reference. She creates a new class using the Eclipse helper and ends up with the following boiler-plate code:

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

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

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

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

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

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

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

However, developers not only spend time during development on bugs related to missing method calls, but this kind of bugs also survive development and get checked into the code repository where they cause problems in the future. As an example, consider this[3] bug report on Apache Torque, an object-relation mapper for Java, which is intended to

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

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

[3] `https://issues.apache.org/jira/browse/TORQUE-42`

facilitate the access and manipulation of data stored in relational databases. This is the diff for the patch which was issued to fix the problem:

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

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

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

The example above is hardly the only bug report related to missing method calls in established software projects. In an informal review, Monperrus et al. found bug reports[4] and problems[5] related to missing method calls in many newsgroups, bug trackers, and forums. The issues range from runtime exceptions to problems in some limit cases, but generally reveal at least a code smell if not worse. In addition to the informal review, Monperrus et al. [5] also did an extensive analysis of the Eclipse bug repository. First, they searched for syntactic patterns which they deemed related to missing method calls, such as: “should call”, “does not call”, “is not called” or “should be called”. Manual inspection then confirmed 117 of the 211 (55%) thus obtained bug reports as indeed related to a missing method call.

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

[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=222305`
[5] `https://www.thecodingforums.com/threads/customvalidator-for-checkboxes.111943/`

code, not only to save expensive developer time but also to make maintenance cheaper and more manageable.

1.1.2 Detection - but how?

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

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

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

To circumvent this problem, we would like to automatically detect locations in a code base where a method call is potentially missing without needing any further input besides the code itself. Such an approach would adapt to changing libraries without requiring additional work from a developer and could also be applied to proprietary code which is not open to the public. While the discovered locations will probably not be as accurate as those discovered by a hand-crafted list of rules, they could then be examined by an expert who would determine the severity of the finding and issue a fix if necessary.

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

1.2 Contribution

In this thesis, we present a thorough reevaluation of the type usage characterization first introduced by Monperrus et al. [4] and further refined in a follow-up publication [5]. A type usage is the list of method calls which are invoked on an object of some type and occur in the body of some method (the context). The general idea behind the technique by Monperrus et al. is to check for outliers among the type usages by using the majority rule: If a type is used in one particular way many, many times (that is, in the majority of cases) and differently only one (or a few) times, this probably indicates a bug.

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

mention some results!

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

FINALLY: Summary of the other chapters of this thesis

2 Related Work

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

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

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. [10] 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. 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. [8] presented a catalogue of 30 Android specific code smells, Palomba et al. [7] 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 [2] as probably the first paper which proposed the general idea behind DMMC -> learning from the code at hand, instead of using static predefined rules
General: read the summary paper [9] again and check which approaches are there and which could / should be mentioned + check to read stuff... + for more information: check this summary paper
Remember the general machine learning approaches which were not super successful, but at least learned a LOT especially like general code smells for example

2.4 Previous work on detecting missing method calls / object usages

There have been a number of works concerned with finding patterns in object usages and using those to find potential bugs. Most relevant for this work are two papers by Monperrus et al.[4][5] 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. [11] 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 [3] 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 [6] 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?

3 Detecting Missing Method Calls

thinking back to example in introduction, it is easy to see that there are bugs out there which occur because of missing method calls additionally Monperrus et al. showed in their informal evaluation that real software systems have many bugs related to missing method calls

INTRO WHAT? - why is it interesting to detect them + what method are we using oä?

In this chapter we will first give a proper explanation of the method proposed by Monperrus et al.. This includes a proper qualification of the notions of type usages as well as their equality and almost equality, which have already been mentioned in the previous chapter. We will then explain how the strangeness score, which describes the amount 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

3.1 The Majority Rule

-> existing work

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

intuition: a piece of code is likely to contain a defect, it: very few exactly similar instances, lots of slightly different instances

analogy for the idea of the majority rule as follows imagine being the waiter preparing the tables in a restaurant there are 100 seats and each of them has a plate, a knife on the right (?) and a fork on the left however only 99 of them have a spoon on the top, one 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 singular seat where it is missing

the rest of this section extends this idea to object oriented software and formalizes the different notions required to implement a system detecting places like such here: spoon = method call, ie extend the concept of the majority rule to type usages

3.1.1 Type Usages

abstraction over analyzed code ignores order! / abstract away list of method calls on some object of a particular type occurring in body of some method [Show example + describe!!!] important information which is saved: the list of methods, the calls being made, the context = method in which tu appears context = name of the method in which the tu appears + type of the parameters (i think the return type is ignored?)

for each variable x extract T(x) => the type the context C(x), which is whole method name in which the variable is used + type of parameters (i think return type is ignored?) and list of invoked Methods: $M(x) = m_1, m_2, \ldots, m_n$ (within C on x) if there are two variables of the same type, this results in two type usages extracted (unless they refer to the same object) [again example: code + the corresponding values for T(x), C(x), M(x), maybe same figure as example above]

3.1.2 Exact and Almost Similarity

hate the title right now, "equal and almost equal" sounds better but is not as precise, what else?

consider talking about similarity instead (variable names, surrounding code, location, blabla) might be different

informally: exactly similar -> same type! type usage appears in a similar method (ie context is the same) the invoked methods = list of methodcalls is identical [include example + describe] "Similar" instead of "equal" because several things are not the same: variable names, surrounding code, method order, LOCATION!, ...

almost similar -> again the type and context are the same list of method calls is identical + one additional call [again include + describe example, probably in same figure?]

to formalize define binary relationships over type usages relationship for Exact similarity (Equality) E two type usages x and y are exactly-similar if and only if:

$$\begin{aligned} xEy \iff & T(x) = T(y) \land \\ & C(x) = C(y) \land \\ & M(x) = M(y) \end{aligned}$$

then the set of exactly similar type usages of x is defined as

potentially change alignment to make look a bit nicer

$$E(x) = \{y \mid xEy\}$$

this holds true for the identity, ie xEx is always true and E(x) is always at least x (and |E(x)| >= 1)

For almost equality 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:

$$\begin{aligned} xAy \iff & T(x) = T(y) \wedge \\ & C(x) = C(y) \wedge \\ & M(x) \subset M(y) \wedge \\ & |M(y)| = |M(x)| + 1 \end{aligned}$$

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

$$A(x) = \{y \mid xAy\}$$

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

furthermore, it is possible to adapt the definition of almost similarity instead of looking for type usages which have the same method calls + one additional ones, we could also look at k additional ones then in the definition of A the last line changes as follows: $|M(y)| = |M(x)| + k, k \geq 1$

3.1.3 The Strangeness Score

assumption behind system: a type usage is abnormal, if a small number is exactly similar, but a large number are almost similar Informally: a few places use this type in exactly the same way, but a big majority use it in a way which is only slightly different assuming the majority is correct, the tu under scrutiny is deviant and potentially erroneous

to capture concretely how anomalous an object is, we need a measure of strangenss this will be the Strangeness score it allows to order type usages, to identify the "strangest" TUs, which are most interesting for evaluation by a human expert

we define the S(trangeness)-Score as:

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

convince yourself that this definition correctly handles extreme cases: given one type usage a without any additional exactly similar or almost similar tus: $|E(a)| = 1, |A(a)| = 0 \rightarrow S - score(a) = 1 - \frac{1}{1} = 0$ so: one unique type usage without any "neighbors" is totally normal and not strange

given a type usage b with 99 almost similar and NO other exactly similar TUs: $|E(b)| = 1, |A(b)| = 99 \rightarrow S - score(b) = 1 - \frac{1}{1+99} = 0.99$ by intuition such a type usage is very strange and the s score supports the intuition

3.1.4 Which Calls are missing?

only detecting the type usages which are somewhat strange, is not enough we would also like to present the developer with some candiate suggestions, WHICH method call might be missing

Intuition: look at all the almost similar type usages and the method call which they add then take the frequency of each methodcall they add and take this as suggestion likelihood [example!!! with a couple of tus and their methodcalls -> calculate the likelihood]

formally: the calls to recommend are the calls that are present in the tus in A(x) but not in M(x)

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

use this equation or the one in their paper (should be identical, which one is easier to follow?) + make sure about klammersetzung

then for each of these methods m in R(x), we can calculate the 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 freuquency / share of almost similar TUS which use this method among all almost similar tus

3.1.5 Caveats

really include here or only in Evaluation section? 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

3.2 Extensions

-> My work theoretical explanation of the different approaches

3.2.1 Modifying the Input Data

stupid title, change it!

load type usages in different ways per class type usages / ignore context working with the inheritance hierarchy!

3.2.2 Other errors

look for different things (wrong method call, superfluous method)

3.2.3 Better Anomaly Detection

clustering approach (hypersphere) something as of now unknown from anomaly detection -> do I really need this section? maybe only in Implementation?

4 Implementation

details about implementation and pitfalls that had to be overcome

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

4.1 Bytecode Analysis

maybe rename section?

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

4.2 Procedure

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

4.3 Improvements

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

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

4.4 Dead Ends

why some solutions where discarded (eg pure database / could be revisited if it turns out to be the best anyways - performance) clustering detector try static functions evaluation! something to fix the dotchaining problems

4.5 Benchmark

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

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

5 Evaluation

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

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

5.1 Methodology

5.1.1 Qualitative Evaluation / Android Case Study

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

5.1.2 Automated Benchmark

degrading type usages + checking if they will be detected

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

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

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

5.2 Results

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

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

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

große frage: bringt das Verfahren etwas?

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

5.2.1 Q0: How does the type usage notion fare in the Android ecosystem?

what kind of dataset do we have? what are typeusages in Android apps like? (length of methods, 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? possible merge this with Q1

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, strangeness values (also for normal loader), etc) (correctly bin histogram!) verteilung of strangeness scores, tu method list sizes - also check their paper, q mas?

5.2.2 Q1: Does the Strangeness Score behave as expected on Android Apps?

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 what do those assumptions mean? -> they expect some kind of uniformity to the type usages, probably mostly present in GUI etc frameworks

5.2.3 Q2: How “useful” is this technique?

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

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

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 Q3: How meaningful are the benchmarking results?

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

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

5.2.5 Q3.1: 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

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 Q3.2: How big does the input have to be?

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 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

List of Figures

List of Tables

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] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.