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

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Motivation 2
1.1.1 Examples from the Real World 2
1.1.2 Detection - but how? 5
1.2 Contribution 6

2 Related Work **7**
2.1 Finding Code Smells with static Analysis 7
2.2 Android specific Smell Detection 8
2.3 Inferring Properties from the Code at Hand 9
2.4 Previous work on detecting missing method calls / object usages 10

3 Detecting Missing Method Calls **12**
3.1 Majority Rule - existing work 12
3.2 Extensions / my work 12

4 Implementation **13**

5 Evaluation **14**

6 Conclusion **16**
6.1 Contribution 16
6.2 Future Research 16

List of Figures **17**

List of Tables **18**

Bibliography **19**

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 [10] 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. [5]: 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,

[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-`

consider this[3] bug report on Apache Torque, an object-relation mapper for Java, which is intended to 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. [6] 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

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

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 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. [5] and further refined in a follow-up publication [6]. 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

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 section 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, as 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.

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 Googles Java codebase. Their

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

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 isn't possible. In their analysis Findbugs finds 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

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. [11] 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 3 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 Section 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.

quick mention of the PAPRIKA paper mentioned in aDoctor intro? [4] operates on byte code, but tries to infer smells on the source code level builds graph model of the application from byte code and stores it in graph db (nodes are entities such as the app itself, classes, methods, attributes and variables, further annotated with specific metrics) uses cypher query language to detect the smells -> basically hand coded rules as well only recognizes 4 android specific smells (and 4 general ones) Evaluation: specifically developed wittness application whith 62 known smells using the right kind of metrics they whereable to reach precision and recall of 1 on this witness application further they apply their tool to a bunch of free android apps in the playstore and make some assertions about the prelevance of antipatterns in publicly available applications findings: some antipatterns, especially related to memory leaks appear in up to 39% of applications, "big" ones like facebook or skype

After Reimann et al. [9] presented a catalogue of 30 android specific code smells, Palomba et al. [8] 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. These cases are exactly those where the hard coded detection rules do not consider a special case, a new library or otherwise changing criteria.

[explain shortcoming a bit better!] example where they only included 2 compression libraries in their rules, but there is a third new one

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.

[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 [10] again and check which approaches are there and which could / should be mentioned + check to read stuff...

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

-> api missuse detection

there have been a number of works concerned with finding patterns in object usages and using those to find potential bugs

short summaries of the two DMMC papers: [5] and [6] most important for this works are the two papers by Monperrus et all (cite) which introduced the notion of almost similarity and are the primary inspiration for this work they consider 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 object is used and its signature (type of its 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 identical and almost equal means the same only that the method list can contain one additional method if there are a lot of almost equal type usages and very few equal ones, the tu under scrunity is probably an anomaly. more details on their method can be found in the next section

results of monperrus et all:

Before this, Wasylkowski et al. [12] 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 an 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 fals 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 [7] 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 classifed 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.

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

3.1 Majority Rule - existing work

intutition behind 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

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 load type usages in different ways per class type usages / ignore context look for different things (wrong method call, superfluous method)

naive bayesian learner as baseline? something as of now unknown from anomaly detection clustering approach (hypersphere) working with the inheritance hierarchy!

4 Implementation

details about implementation and pitfalls that had to be overcome soot as analysis framework -> what is soot why bytecode over sourcecode . . . why some solutions where discarded (eg pure database / could be revisited if it turns out to be the best anyways - performance) static functions evaluation! something to fix the dotchaining problems . . .

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

details on the different detectors I implemented

— generally: (in this section or implementation? or somewhere totally else or not at all?)

3 main parts: Feature Extraction, Learning, Results - for each tons of different options and combinations FE: what are the features, restricted to method calls?, granularity? (including parameters?) method order?

Learning: Struktur finden und anomalien ermiteln consider what the expected structure MEANS (eg bruch expects a kind of uniformity to object usages, which is probably mostly present in GUI etc Frameworks) maybe some method can also detect multiple missing calls, order, ...

Results: how is the "quality", under which circumstances? some kind of statement about the initial assumptions behind idea and

5 Evaluation

roter faden: results of manual analysis vs benchmark (ie. is context better than no context)

sections: methology (android apps, bla bla) qualitative (manual) evaluation automatic benchmark Results (would like to answer the following questions) Q1 (-> explanation of question, then results + interprettation) Q2 . . . General takeaway

Which questions are we trying to answer with the evaluation? qualitative evaluation: are the findings useful in practice? quantitative evaluation: given some assumptions, how useful can we expect this technique to be? make sure I will be able to a) answer those questions and b) the answers are “interesting”

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 how "useful" is this verfahren -> 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) does it make sense to evaluate this method using the benchmarking as proposed by Monperrus et al.? -> comparison between manual evaluation of findings vs would love to have the robustness question in here, however how to evaluate if it’s working vs not working?

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

checking if their “mistakes” / inaccuracies make a difference or not

(Case Study - if time...) ANDROID?! would be super sexy to apply the best method to some library / framework and see what happens

subsection: threats to validity!!!

qualitative evaluation: pick a couple of high scoring findings and explain the failure modes (eg doesn't take branching into account, etc)

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

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

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) Performnance (shouldn't be a problem?)

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.