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, in the case of large frameworks or very powerful libraries, often a lot of knowledge is required to invoke the API correctly. This can be because of constraints or requirements which are not clear from the outset, but which have to be heeded in order to avoid serious bugs or complications, or because of a complex interplay between different parts of the library. In the worst case, this knowledge might not even be explicitly documented.

The fact that APIs are not trivial and have 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 a number of other factors (e.g. some methods must be invoked in an extra thread, some specific precondition has to be satisfied, etc.). An example which comes to mind in Java are an iterator on which the programmer calls `next()` 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 manifold 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 [Rob+13] and they can be used to detect potential defects. If 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, also known as API misusages(?).

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 normally used by invoking some of its methods. Some types will then have underlying patterns such as: "when methods A and B are invoked, then method C is called as well" or "Methods X and Y are always used together". If we have an object of this type 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 where only X or only Y is invokec, 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 where she creates a button and assigns it some text to display by calling `setText()` but forgets to also call `setFont()` this would be a missing method call and a bug in the code. Imagine all buttons in the app using a special and beautiful font but this one button sticking with the standard ComicSans!

In an informal review Monperrus et al. [MBM10] found bug reports and problems related to missing method calls in many news groups, bug trackers and forums. The issues caused range from runtime exception to problems in some limit cases, but generally reveal at least a code smell if not worse.

[Example] https://issues.apache.org/jira/browse/TORQUE-42 Some stuff about real life situation where missing method call can be a problem -> find different bug report then the one the Majority Rule papers uses, but something like that

Consider for instance the following stack overflow post: Here a developer spent many hours of their valuable time on debugging a relatively simple problem related to just one method call which was missing.

However, developer not only spent time during development on bugs related to missing method calls, these kind of bugs also survive development and get checked into the code repository where they cause problems in the future. In their extensive analysis of the Eclipse bug repository Monperrus et al. [MM13] show that even in mature code bases there are many bugs related to missing method calls. 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 easier.

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

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

Well crafted and thought-out rules like this 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 were software is constantly changing and improving. While such work 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. 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.

-> express some more much better than fixed preprogrammed rules, can adapt to changing system, be specific for own not open library, etc

the approach chosen in this work bases of recommender systems / learning (Mention the ideas of [Bru12], chapter 2 as an inspiration / the way to the idea - maybe) first find likely recommendations, for writing, then realize, if something is super likely given a particular situation, but it is not there, it seems like a good indicator of an error

1.2 Contribution

In this thesis we present a thorough reevaluation of the type usage characterization first introduced by Monperrus et al.[MBM10] and further refined in a follow up publication [MM13] type usage: list of method calls invoked on variable of a given type which occur in the body of a specific method Majority rules idea: if a type is used in one particular way many many times and differently only one (or few) times, this probably indicates a bug

Further evaluation of the concept with comparison test using different similarity measures, application to big data set of open source android applications, . . . mention some results!

FINALLY: Summary of the other chapters of this thesis

2 Related Work

How to split this in the best way? -> check interesting related work and split it by topic / used technique / goal

General: read the summary paper [Rob+13] again and check which approaches are there and which could / should be mentioned + check to read stuff...

here: short intro to related work in general Relevance of code smells and finding them! first an overview of the different approaches that are commonly used for code smell detection this includes static rules and bla [summary of the other section]

2.1 general approaches for bug detection / code smells

What is the general state of the art (maybe extra section?) static stuff, rules based, etc mention findbugs?

2.2 Section on approaches which learn from the given code

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 [Eng+01] as probably the first paper which proposed the general idea behind DMMC -> learning from the code at hand, instead of using static predefined rules

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

2.3 Previous work on detecting missing method calls / object usages

-> api missuse detection

short summaries of the two DMMC papers: [MBM10] and [MM13] less about the method as it is explained more in depth in the next section but about results

mention the related work from detecting mmc paper

This [WZL07] goes in the direction of detecting method order and seems quite interesting quite old, but has implementation and also an interesting related work section! extract object usage models from Java code - uses finite state automata for the usage model build automata for each method, similarly to control flow graph of the code with intsructions as the transitions mine temporal properties out of those object usage models -> method a can appear before method b, iff there is a path through the automata whera a appears before b For this they are using Frequent Itemset Mining (reference to J. Han and M. Kamber. Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2006. ?) -> short explanation of FIM these temporal properties are then combined into patterns and deviations of those patters are flagged as an anomaly an anomaly occurs if many methods respect the pattern and only a few (a single one) break it in experiments: 790 Violations, by manual analysis 2 defects, 5 smells, 84 "hints" (could be improved with regard to redability / maintainability) -> large number of false positives (87.8%?! says next paper), but with an additional ranking methdod able to to get the 2 defects and 3/5 smells within top10 results already project dependent patterns as it extracts them from the existing code

The paper on graph based object using pattern mining seems also super related [Ngu+09] graph-based representation of object usages of one or even multiple objects -> this makes it stand out also about temporal dependencies labeled directed graph: nodes = object constructor calls, method cals, field accesses and brnching via control structures edges are temporal usage orders and dependencies between them patterns are mined using a frequent induced subgraph detection algorithm which start building larger patterns from smaller patterns ground up (similar to merge sort maybe?) anomaly also when a pattern violation is "rare" ie doesn't appear often in the dataset in relation to the size evaluation via casestudy finds 64 defects in some software with 5 defects, 8 smells, 11 hints -> 62.5% false positive rate, top 10 has 3/2/1

2.4 Section on Anomaly Detection

2.5 Section on Android specific stuff

3 Detecting Missing Method Calls

3.1 Majority Rule - existing work

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)

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

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”

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! …

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 . . .

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

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 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

[Bru12] M. Bruch. “IDE 2.0: Leveraging the wisdom of the software engineering crowds.” PhD thesis. Technische Universität, 2012.

[Eng+01] 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.

[MBM10] 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.

[MM13] 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.

[Ngu+09] 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.

[Rob+13] 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.

[WZL07] 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.