Project-Specific Deletion Patterns

— Short Position Paper —

Yana Momchilova Mileva
Saarland University
Saarbrücken, Germany
mileva@cs.uni-sb.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-sb.de

ABSTRACT

We apply data mining to version control data in order to detect project-specific *deletion patterns*—subcomponents or features of the software that were deleted on purpose. We believe that locations that are similar to earlier deletions are likely to be code smells. Future recommendation tools can warn against such smells: "People who used `gets()` in the past now use `fgets()`. Consider a change, too."

1. INTRODUCTION

Having available the code history of a project, stored into a version control system, we can learn a lot about the processes that took place in the past—and use the mined data to predict the way the project is going to evolve in the future. We are particularly interested in the reasons behind *deleting* code parts. When something was deleted, it was most probably deleted for a reason and it should not reappear again. We would like to focus on detecting project-specific deletions. The main question we are working on is: *Can a deletion indicate quality issues in the remaining project code?*

Our hypothesis is that we can learn project-specific behavioral patterns based on the deletion history of the project. We would like to explore these deletion patterns on a fine granularity level, e.g., observe the evolution of variables and method calls—and extend it to general program features, such as changes in module dependencies, control structures, or data flow.

2. BACKGROUND

Analyzing the evolution of project code has recently become a very active research area. People have been concerned with what can be learned from a change. Such research leads to valuable results when it comes to finding common error patterns [2], fix-inducing changes [1] or suggesting similar change locations [3]. A change in the code usually means adding or deleting code or both. In this paper we focus on the deletion part; we are looking for *code smell patterns* based on the available deletion information.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RSSE '08, November 10, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-228-3 ...$5.00.

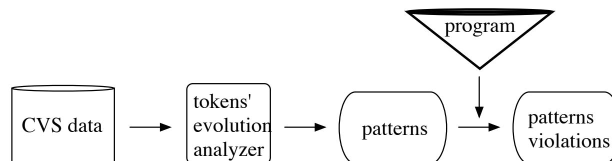

Figure 1: From CVS data to detection of patterns and their violation.

3. LEARNING FROM DELETIONS

We are interested in the evolution of fine grained tokens and in particular what can be learned from the deletion of those tokens. Here we use the Java definition for *token*, where consecutive characters are grouped into tokens, e.g., class names, operators, etc. In order to extract those tokens from the revision history of a project, we use the APFEL tool [4]. APFEL provides us with useful information regarding the tokens, e.g., token location, time of deletion of the token, number of occurrences of the token, etc. Having the tokens extracted, we can analyze the tokens' evolution and learn project-specific patterns from it. Once we have detected these patterns, we can check a version of the project against these patterns and detect possible violations. This general approach is illustrated in Figure 1.

In the evolution of tokens we focus on *deletions.* Deletion is what we call the process of deliberately removing tokens from the code. This happens because:

1. something had to disappear from the code and it got deleted (Section 3.1);

2. something had to be replaced in the code and it got deleted and substituted (Section 3.2).

As mentioned above, we are currently occupied with finding out what can be learned from deletions of fine grained tokens like variables and method calls. However, it would be interesting to explore further levels of granularity as well as the possibility of improving existing defect prediction techniques by combining them with deletion evolutionary data.

3.1 Deletion of tokens

During the development of a software project code gets deleted occasionally, e.g., some methods may get partially

deleted token	remains N times	was present N times	replacement token	replaced N times
getNextSibling	2	142	getNext	122
badArg	0	3	argBug	3
getShort	12	31	getIndex	8

Table 1: Deletion patterns: *getNextSibling* is now called *getNext*.

or completely erased from the code. If the rate of usage of some particular token drops, we can state that this token should no longer be used in the code. We will be then able to use this information for defect prevention as well as for detection of unknown defect locations.

Let us elaborate a bit more on these two cases. We will be able to prevent new defects from appearing, by warning the developer while she is actually typing her code that the token she is planning to use is no longer recommended for usage in the project. We could also run the current version of the program against the list of deleted tokens and detect locations where such tokens still exist and can indicate a potential defect. Part of our initial results can be found in Table 1. The sample data is taken from the *Rhino*[1] open-source project. As it can be seen from Table 1, *getNextSibling* was used 142 times in the past, while now it is used only 2 times. This indicates that the usage of this token is not recommended in the future and the locations where it still remains are possible code defect locations.

3.2 Replacement of tokens

In many cases something in the code gets deleted in order to be substituted by something else. Using our approach, we are able to detect in most of the cases which token got substituted with which token. Such information could add more value to the above mentioned recommendations, by not only informing the developer that some tokens should no longer be used in the project, but also by recommending a matching substituting token.

A recommendation tool that will serve this purpose should provide the developer with the name of the deleted token, the number of instances of this token that still exist in the code after the deletion (if any), the number of instances of this token that existed before the deletion, the name of the replacing token (if any) and the number of times this substitution was performed. As it can be seen from our initial results in Table 1, *getNextSibling* existed 142 times in the code, but it got deleted and replaced on 122 locations with *getNext*.

4. EXPECTED CONTRIBUTIONS

We expect the following contributions:

- **Knowledge about the relationship between code quality and deleted data:** At the end of our research, we will be able to give a first insight whether deleted data contains valuable information. We are interested in proving that this data is useful and can lead us to discovering project-specific patterns violations.
- **Current defect detection level:** Our method will add its value to the current defect detection approaches and will help developers discover code defects that could not be detected by other technique.
- **A code change recommendation tool:** We are planning to implement a tool that is based on our token patterns detection approach, which will be fully automatic and will most probably come as an Eclipse plug-in.

[1]Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users.

5. CONCLUSION

This work is still at its early stage. The most important questions are: Which are the features that get deliberately deleted over time? Is every such feature a code smell, or is it just a consequence of the software adapting to its changed environment? How can we characterize such features, and how can we detect similarities? How can we make recommendations that are the most helpful to users? All of these questions address a different type of software maintenance category: corrective, adaptive and preventive; how do we distinguish these? All of the stated questions are open issues that we would love to discuss with RSSE participants, and we look forward to fruitful exchanges.

Acknowledgments. Yana Mileva is funded by the Max-Planck-Institut Informatik and by Microsoft Research Cambridge Lab. We would like to thank Andrzej Wasylkowski and Thomas Zimmermann for giving valuable comments on earlier revisions of this paper as well as the anonymous reviewers for providing useful feedback on how to improve this paper.

6. REFERENCES

[1] S. Kim, K. Pan, and J. E. Whitehead. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006. ACM.

[2] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305. ACM, September 2005.

[3] M. M. McIntyre and R. J. Walker. Assisting potentially-repetitive small-scale changes via semi-automated heuristic search. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 497–500, New York, NY, USA, 2007. ACM.

[4] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *Eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2006. ACM.