Using Source Code Comments to Detect Self-admitted Design Technical Debt

Everton da S. Maldonado, Nikolaos Tsantalis and Emad Shihab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
e_silvam@encs.concordia.ca, nikolaos.tsantalis@concordia.ca, emad.shihab@concordia.ca

***Abstract*—During the development and maintenance of a software system, developers face unpredictable difficulties or pressures, and in many cases are forced to apply unconventional solutions to overcome these difficulties. For example, they might adopt insufficiently tested or temporary solutions (i.e., workarounds and hacks), neglect good design practices, and introduce inaccurate or incomplete documentation due to time constraints and pressure to meet deadlines. This phenomenon has been explained through the metaphor of Technical Debt.**

Prior work has shown that one of the most impacting types of technical debt is design debt and that code comments embedded in the code can be used to detect *self-admitted* technical debt. Therefore, in this paper our main goal is to study Self-admitted Design Technical Debt.

I. Introduction

Developers often have to deal with conflicting goals that require software to be delivered quickly, with high quality, and on budget. In practice, achieving all of these goals at the same time can be challenging, causing a tradeoff to be made. Often, these tradeoffs lead developers to take *shortcuts* or use *workarounds*. Although such shortcuts help developers in meeting their short-term goals, they may have a negative impact in the long-term.

Technical debt is a metaphor that has been used to express sub-optimal solutions that are taken consciously in a software project in order to achieve some short-term goals. Generally, these decisions allow the project to move faster in the short-term, but introduce an increased cost (i.e., debt) to maintain this software in the long run [1], [2]. Prior work showed that technical debt is widespread in the software domain, is unavoidable, and can have a negative impact on the quality of the software [3].

Due to the importance of technical debt, a number of studies empirically examined it and proposed techniques to enable its detection and management. The main findings of the prior work is that 1) there are different types of technical debt, e.g., defect debt, design debt, testing debt, and that design debt has the highest impact [4], [5]; and 2) statically analyzing the source code can help detecting technical debt [6]–[8]. In particular, these works use metric thresholds to detect code smells, which are considered as proxies for technical debt.

One major drawback of using metrics to detect technical debt is that no one knows if the detected smells really constitute technical debt, or if they correspond to problems that the developers care about. Therefore, more recently, our work showed that using code comments can be effective in identifying self-admitted technical debt [9]. This work uses comments to detect *generic* technical debt, and did not focus on any specific type of technical debt.

In this paper, we build on the promising approach of using code comments to detect one of the most impacting types of debt, namely *design technical debt*, which we call Self-admitted Design Technical Debt. We manually examine more than 17,000 code comments to extract comment patterns that can be used to detect Self-admitted Design Technical Debt. To examine the effectiveness of our approach, we perform an empirical study on ten open source projects. Finally, we compare our approach to state-of-the-art approaches and examine the effectiveness of using automated refactoring techniques in mitigating Self-admitted Design Technical Debt.

Based on our manual examination of the code comments, we derive 176 different comment patterns that can be used to detect Self-admitted Design Technical Debt. These patterns are able to detect Self-admitted Design Technical Debt with a precision ranging in 74.0-96.30% and a recall ranging in 10.87-83.87%. Moreover, we find that the design technical debt found with our approach is different than the design technical debt found using metric-based approaches [8]. Finally, we find that automated refactoring can address up to 24.58% of the methods containing Self-admitted Design Technical Debt.

The rest of the paper is organized as follows: Section II presents a motivating example. Section III details our approach. We present our case study results in Section IV, followed by a discussion in Section V. Section VI presents the related work. The threats to validity of our work are discussed in Section VII. Section VIII lists the conclusions of our work.

II. Motivating Example

As mentioned earlier, one of the first works on self-admitted technical debt was the work by Potdar and Shihab [9]. Their work showed that it is possible to identify self-admitted technical debt using source code comments. However, in their work, Potdar and Shihab studied *generic* technical debt, i.e., they did not discriminate between the different types of technical debt. For example, technical debt can be in the form of design debt, testing debt, defect debt, and documentation debt.

Since our work focuses on Self-admitted Design Technical Debt, we first examined the effectiveness of using the general

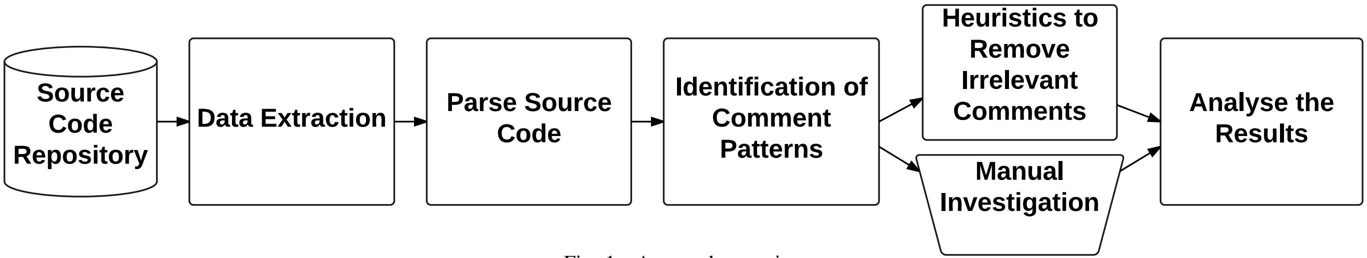


Fig. 1. Approach overview

comments used by Potdar and Shihab to detect design technical debt. We applied the comment patterns that we derived (which we present later in the paper) and the comment patterns from Potdar and Shihab on the studied open source projects. As expected, the results produced by the general comment patterns identified all types of technical debt, indicating the need for more specific comment patterns that can be used to effectively identify design technical debt.

To illustrate our point, we show some example comments flagged by Potdar and Shihab's approach in the first column of Table I. The second column of the table shows the comments that are detected by the comment patterns we propose in this paper, which focus on Self-admitted Design Technical Debt. A comparison of the comments in Table I clearly shows that the more specific comment patterns detect design issues.

This simple example shows that comment patterns that specifically target design technical debt are needed. Simply using the general comment patterns may yield unfavourable results. We elaborate more on the performance of using the general comment patterns to detect Self-admitted Design Technical Debt in Section IV.

III. Approach

The main goal of our study is to extract comment patterns that can be used to effectively identify Self-admitted Design Technical Debt. Figure I shows an overview of our approach. The following subsections detail each step of our approach.

A. Data Extraction

To perform our study, we obtain the source of ten large open source projects, namely Apache Ant, Jakarta Jmeter, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JRuby and SQuirrel SQL Client. We chose the aforementioned projects, since they belong to different domains, and vary in size (e.g., LOC), and in the number of contributors.

Table II provides statistics about each of the projects used in our study. In total, we obtained more than 258,878 comments, found in 16,249 files. We also include the release used, the number of classes, and the total lines of code (LOC). In our study, we only use the Java files to calculate the LOC. It is important to notice that the number of comments shown for each project does not represent the number of commented lines, but rather the number of individual line, block, and Javadoc comments.

B. Parse Source Code

After obtaining the source code of all projects, we extract the comments from their source code. We use JDeodorant [10], an open-source Eclipse plug-in, to parse the source code and extract the code comments. Once extracted, we store all comments in a relational database to facilitate the processing of the data.

C. Identification of Self-admitted Design Technical Debt Comment Patterns

Once we store all comments in the database, our next step is to identify the Self-admitted Design Technical Debt comment patterns. Since we are dealing with natural language in the comments, it is challenging to automatically determine what comments indicate design technical debt. Therefore, we opted to use two different approaches to determine comment patterns that indicate design technical debt. First, we use the terms mentioned in prior work [11]–[13] (i.e., code smell and anti-pattern names) as indicators of design problems to determine comments that are indicative of design technical debt. Second, we manually examined and classified all comments of one project i.e., Apache Ant, in order to determine comment patterns that are indicative of Self-admitted Design Technical Debt. After analyzing the results, we found that combining comment patterns from the two aforementioned approaches provides the best results. We detail the steps taken to achieve each of the two approaches.

1) Applying Heuristics to Eliminate Irrelevant Comments: When applying our first approach, i.e., using the terms in the prior work to identify comments that are indicative of Self-admitted Design Technical Debt, we found that we are able to flag comments that indicate design issues, but also flag many false positives. We analyzed the false positives to see whether we can gain any insight into why they appear and how we can eliminate them.

We identified three main types of false positives. First, license comments, containing copyright information and legal rights. Second, commented source code containing Java keywords, e.g., “big” and “long”. Finally, Javadoc comments were flagged, however, they often had no relation to design issues. As a result, we came up with three heuristics and a post-processing step to reduce the number of false positives.

- **Heuristic to remove license comments.** When license comments are added to the Java files in a project they are generally placed in the first lines of the file, before the

TABLE I
EXAMPLE OF GENERAL/DESIGN SELF-ADMITTED TECHNICAL DEBT COMMENTS

General Self-Admitted Technical Debt	Self-Admitted Design Technical Debt
remove this code once bug 62405 is fixed for the mainstream GTK	*This can lead to code smell, meh! Do we care*
FIXME - This caching thing should not be here; it's brittle.	*This is an absurdly long method! Break it up.*
FIXME compat: updateActionBars : should do something useful	*there should be an interface, instead of the AbstractMessageFolder*
FIXME this does not actually set the default since it is the wrong	*rethink where exactly some of the following methods belong (GenModel or GenPackage)*
TODO: - please add some javadoc - ugly classname also	*Cyclic dependency with PersistenceManager*

TABLE II
CASE STUDY PROJECT DETAILS AND STATISTICS

Project	Release	LOC	Classes	Comments	Contributors	Description
Apache Ant	1.7.0	115,881	1,475	21,587	70	A Java library and command-line tool to build Java applications.
Jakarta Jmeter	2.3.2	81,307	1,181	20,084	32	An application to measure performance and assert functional behavior.
ArgoUML	0.34	176,839	2,609	67,716	87	An UML modeling tool.
Columba	1.4	100,200	1,711	33,895	9	A desktop email client written in Java.
EMF	2.4.1	228,191	1,458	25,229	28	Eclipse Modeling Framework.
Hibernate	3.3.2 GA	173,467	1,356	11,630	216	An Object Relational Mapping framework.
JEdit	4.2	88,583	800	1,6991	55	A light weight text editor.
JFreeChart	1.0.19	132,296	1,065	23,123	18	A Java library to display graphics and charts.
JRuby	1.4.0	150,060	1,486	11,149	291	Is the implementation of the Ruby language using the Java Virtual Machine.
SQuirrel	3.0.3	215,234	3,108	27,474	40	A graphical SQL client written in Java.

TABLE III
NUMBER OF COMMENTS AFTER THE APPLICATION OF EACH HEURISTIC

Project	Initial no. of Comments	After license heuristic	After comment code heuristic	After Javadoc heuristic	After post processing
Apache Ant	21,587	20,421	20,268	6,239	4,436
Jakarta Jmeter	20,084	18,840	18,530	12,360	8,126
ArgoUML	67,716	28,180	27,848	13,972	10,303
Columba	33,895	14,600	14,256	9,095	6,825
EMF	25,229	24,355	24,093	8,861	5,868
Hibernate	11,630	10,446	10,277	4,908	3,071
JEdit	16,991	16,128	16,037	13,118	11,232
JFreeChart	23,123	22,114	22,047	5,902	4,449
JRuby	11,149	10,274	10,080	6,887	5,176
SQuirrel	27,474	25,566	25,196	13,713	8,627

class declaration. Based on this knowledge we created a heuristic that eliminates comments that are placed before the class declaration. To validate the result of this heuristic we examined a sample of the comments being removed to check if they were indeed license comments. We noticed that some comments were placed before the class declaration although they were not license comments. To mitigate the risk of eliminating important comments, we added one more condition: If the comment contains one of the task-reserved words (e.g. "todo", "fixme", or "xxx") we do not remove the comment.

- **Heuristic to remove commented source code.** If a commented piece of source code contains Java keywords like "long" or "big", it will increase the number of false positives of our approach. Commented source code can be found for several different reasons. One of the possibilities could be that the code is not being currently used, or if the particular piece of code is used to debug the program. Since commented code does not have Self-admitted Design Technical Debt, we remove commented source code using a regular expressions that captures typical Java code structures.
- **Heuristic to remove Javadoc comments.** The Javadoc comments contain information about the purpose and use of methods and classes. That said, Javadoc comments rarely mention Self-admitted Design Technical Debt. Therefore, we create a heuristic that removes Javadoc comments. To mitigate the risk of eliminating some correct cases, we added one exception - if the comment contains one of the task-reserved words (e.g. "todo", "fixme", or "xxx") we keep that Javadoc comment.
- **Post processing technique to merge multiple line comments** Another problem that we found while analyzing the comments was that some times developers make long

comments, using multiple single-line comments instead of a Block comment. Treating every single line of a long comment as an individual comment causes us to miss important context details that could be recovered by treating all single-line comments as a single block comment. Therefore, we create a post processing technique that searches for consecutive single-line comments and groups them.

The steps mentioned above significantly reduced the number of comments in our dataset and helped us focus on the most applicable and insightful comments. For example, in the Apache Ant project, applying the above steps helped reduce the number of comments from 21,587 to 4,436 comments.

2) Manual investigation of identified Self-admitted Design Technical Debt comments: In addition to using the words that indicate design issues to detect Self-admitted Design Technical Debt, we also manually examine our dataset to extract comment patterns that indicate Self-admitted Design Technical Debt comments. We started by examining all of the 4,436 comments for the Apache Ant project and classified each comment as being related to Self-admitted Design Technical Debt or not. Since our focus in this work is on design debt, comments related to other types of technical debt were not labeled as Self-admitted Design Technical Debt comments. The classification of the Apache Ant comments took approximately 32 hours and was performed by the first author of the paper.

Manual Examination of Comments to Identify Self-admitted Design Technical Debt Comment Patterns

In the end of the classification we identified 93 Self-admitted Design Technical Debt related comments out of 4,436 comments in Apache Ant project.

Our next goal was to abstract the comments and come up with a set of *comment patterns* that indicate Self-admitted Design Technical Debt. Comment patterns are general patterns that represent one or more comments. Simply using a single word to identify Self-admitted Design Technical Debt comments can be misleading since the context that the word appears in can completely change the meaning of that word. In order to address this issue, we take into consideration some of the other words that appear in the same sentence to combine them into what we call comment patterns.

By the end of this step, we had identified the comment patterns that indicate Self-admitted Design Technical Debt. **In total, we had 176 comment patterns that can be used to detect Self-admitted Design Technical Debt**. To facilitate future work in the area, we make our dataset and the comment patterns publicly available [1].

Table IV provides a sample of the comment patterns that we used to identify Self-admitted Design Technical Debt comments. The '%' symbol indicates that the pattern uses the SQL language wildcards. Wildcards make the query to match anything before or after the wildcard symbol. For example, "dependen%" would result in positive results for comments that contains the words "dependency" or "dependencies".

[1]http://users.encs.concordia.ca/ e_silvam/publications.html

TABLE IV
SAMPLE SELF-ADMITTED DESIGN TECHNICAL DEBT COMMENT PATTERNS

Related Comment Patterns
'%future%may%'
'%future%better%'
'%future%enhance%'
'%future%change%'
'%dependency%cycle%'
'%todo%dependenc%'
'%fixme%dependenc%'
'%xxx%dependenc%'

Once we derive the 176 comment patterns that indicate Self-admitted Design Technical Debt, we use these patterns to answer our research questions, which we detail in the next section.

IV. CASE STUDY RESULTS

RQ1. What comment patterns indicate self-admitted design technical debt? How are these comment patterns different than previously proposed comment patterns?

Motivation: Approach: Results:

RQ2. Can we effectively detect self-admitted design technical debt using the proposed comment patterns?

Motivation:
Approach: Results:

RQ3. How much of the detected self-admitted design technical debt can we automatically address with state-of-the-art refactoring techniques?

Motivation:
Approach: Results:

V. DISCUSSION

VI. RELATED WORK

Our work uses code comments to detect Self-admitted Design Technical Debt. Therefore, we divide the related work into three categories: source code comments, technical debt, and code smell detection.

A. Source Code Comments

A number of studies examined the co-evolution of source code comments and the rationale for changing code comments. For example, Fluri *et al.* [14] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent. Tan *et al.* [15] proposed a novel approach to identify inconsistencies between Javadoc comments and method signatures. Malik *et al.* [16] studied the likelihood of a comment to be updated and Found that call dependencies, control statements, the age of the function containing the comment, and the number of co-changed dependent functions are the most important factors to predict comment updates.

Other work used code comments to understand developer tasks. For example. Storey *et al.* [17] analyzed how task annotations (e.g., TODO, FIXME) play a role in improving team articulation and communication. The work closest to ours is the work by Potdar and Shihab [9], where code comments were used to identify technical debt.

Similar to some of the prior work. we also use source code comments to identify technical debt. However, our main focus is on the detection of Self-admitted Design Technical Debt. As we have shown, our approach yield different and better results in detection Self-admitted Design Technical Debt. Furthermore, we propose comment patterns, that are derived from source code comments, to detect Self-admitted Design Technical Debt.

B. Technical Debt

A number of studies have focused on the study of, detection and management of technical debt. Much of this work has been driven by the Managing Technical Debt Workshop effort. Fore example, Seaman *et al.* [1], Kruchten *et al.* [2] and Brown *et al.* [18] make several reflections about the term technical debt and how it has been used to communicate the issues that developers find in the code in a way that managers can understand. Other work focused on the detection of technical debt. Zazworka *et al.* [8] conducted an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding the detection of technical debt. They found that there is small overlap between the two approaches, and thus it is better to combine them than replace one with the other. In addition, they concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt. In follow on work, Zazworka *et al.* [19] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of design debt, namely God Classes. They found that God Classes are more likely to change, and therefore, have a higher impact in software quality. Fontana *et al.* [20] investigated design technical debt appearing in the form of code smells. They used metrics to find three different code smells, namely God Classes, Data Classes and Duplicated Code. They proposed an approach to classify which one of the different code smells should be addressed first, based on a risk scale. Also related here, Potdar and Shihab [9] used code comments to detect technical debt.They extracted the comments of four projects and analyzed more than 101,762 comments to come up with 62 patterns that indicates self-admitted technical debt. Their findings show that 2.4% - 31% of the files in a project contain self-admitted technical debt.

Our work is different from the work that uses code smells to detect design technical debt since we use code comments to detect design technical debt. Also, our focus is on *self-admitted* design technical debt. As we have shown in the discussion section, there is very little overlap between the Self-admitted Design Technical Debt that our approach detects and the design technical debt detected using code smells (in particular God classes).

C. Code Smell Detection

Other work build tools and techniques to facilitate the detection of code smells. Moha *et al.* [21] proposed DECOR, a tool that incorporates a set of techniques to identify code smells in the source code. They used a domain specific language (DSL) to specify code smell detection rules. Their approach automatically generates detection algorithms based on the code smell specifications. They evaluated their techniques in 11 open-source projects and found that DECOR can effectively detect code smells, with an average precision of 60.5% and recall of 100%. Palomba *et al.* [22] proposed an approach to identify code smells based on the evolution of the source code. In order to do that they mined the history change from the source code repository and then they searched for bad smells. They show that using their approach (HIST), they are able to identify 5 different bad smells. Tsantalis *et al.* [23] proposed a methodology that identified Feature Envy bad smells and evaluated the refactoring to remove the bad smell.

Our work complements the prior work on code smell detection, since we propose the use of code comments to detect Self-admitted Design Technical Debt. In particular, we propose 176 comment patterns to identify Self-admitted Design Technical Debt. Analyzing the source code comments using our approach in addition to the source code analysis techniques already employed in prior work can lead to optimal results, since our analysis showed that our approach yields results that are complementary to what code smell approaches detects.

VII. Threats to Validity

Internal validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. The comment patterns derived by us heavily relied on manual analysis of the code comments from Apache Ant. Like any human activity, our manual classification is subject to personal bias. To reduce this bias, any comment that was questionable was discussed between the three authors of the paper. When performing our study, we used well-commented Java projects. Since our technique heavily depends on code comments, our results and performance measures may be impacted by the quantity and quality of comments in a software project.

When we investigate if there are refactoring recommendations to address the detected Self-admitted Design Technical Debt, we essentially examine if the methods in which design debt is found participate in any of the refactoring opportunities suggested by JDeodorant. The presence of a refactoring opportunity for a given method, may not necessarily address the same kind of design debt described in the comment. In the future, we plan to investigate in a more fine-grained level the applicability of the suggested refactorings to Self-admitted Design Technical Debt.

When calculating the precision and recall values, we needed to manually examine the comments and label them as related to Self-admitted Design Technical Debt or not. Any errors in our labeling may impact the precision and recall values reported.

External validity consider the generalization of our findings. All of our findings were derived from comments in open source projects. To minimize external validity, we chose open source projects from different domains. That said, our results may not generalize to other open source or commercial projects. In particular, our results may not generalize to projects that have a low number or no comments.

VIII. Conclusion and Future work

The term Technical Debt is often used to express some kind of inadequacy in the source code in a way that is understandable to management. But this metaphor can represent many different kind of things, inappropriate or temporary solution to meet a deadline, error prone code, lack of tests and documentation and even design flaws or workarounds. Sometime, developers are aware of these problems and they may express their concern through comments in the source code. Therefore, in this study we propose an approach to identify such comments in the source code. Our findings show that:

References

[1] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*, M. V. Zelkowitz, Ed. Elsevier, 2011, vol. 82, pp. 25–46.

[2] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 5, pp. 51–54, Aug. 2013.

[3] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Software*, vol. 29, no. 6, pp. 22–27, Nov 2012.

[4] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spinola, "Towards an ontology of terms on technical debt," in *Proceedings of the Sixth International Workshop on Managing Technical Debt*, 2014, pp. 1–7.

[5] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, Sept 2012.

[6] ——, "Detection strategies: metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 350–359.

[7] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 274–275.

[8] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013, pp. 42–47.

[9] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[10] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 329–331.

[11] M. Fowler and K. Beck, *Refactoring : improving the design of existing code*. Reading, MA: Addison-Wesley, 1999.

[12] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis*, 1st ed. John Wiley and Sons, 1998.

[13] R. C. Martin and J. O. Coplien, *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009.

[14] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[15] S. H. Tan, D. Marinov, L. Tan, and G. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[16] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. Hassan, "Understanding the rationale for updating a function comment," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2008, pp. 167–176.

[17] M. Storey, J. Ryall, R. Bull, D. Myers, and J. Singer, "Todo or to bug," in *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 251–260.

[18] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, 2010, pp. 47–52.

[19] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the Second Workshop on Managing Technical Debt*, 2011, pp. 17–23.

[20] F. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012, pp. 15–22.

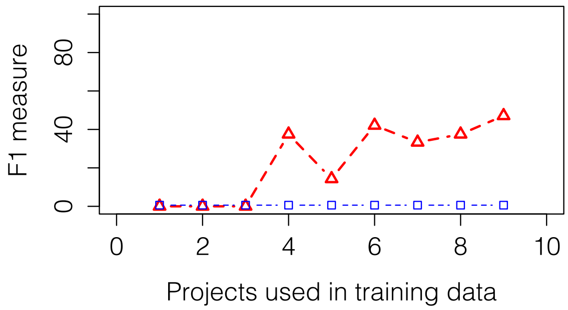
[21] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, Jan 2010.

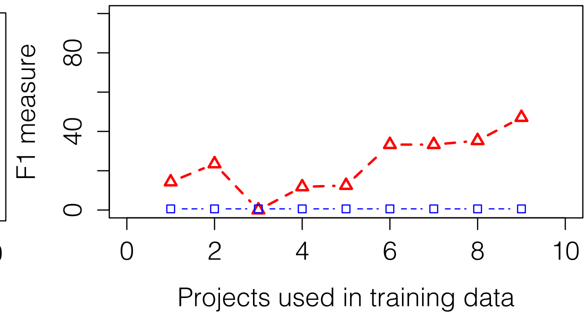
[22] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering*, 2013, pp. 268–278.

[23] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, May 2009.

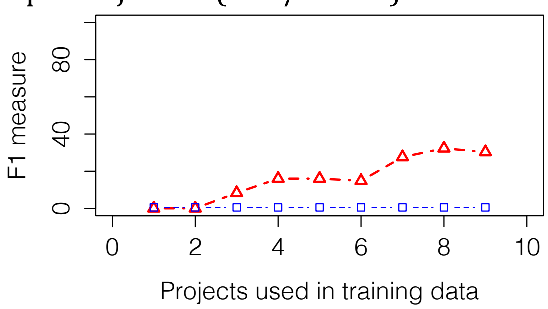
Defect

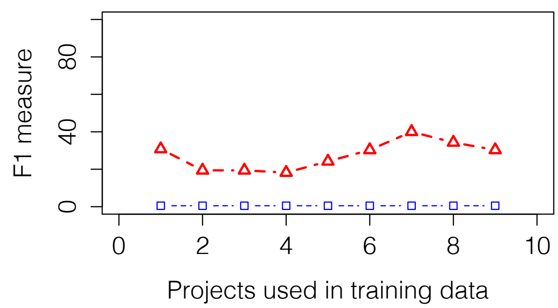
Apache-ant (cres/decres)



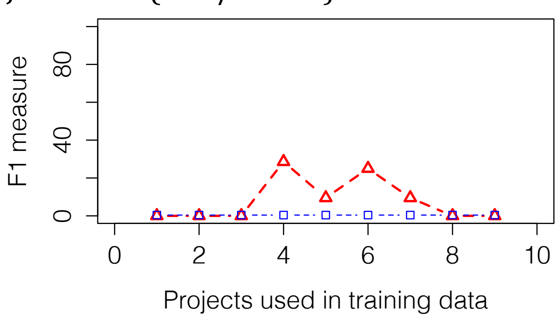


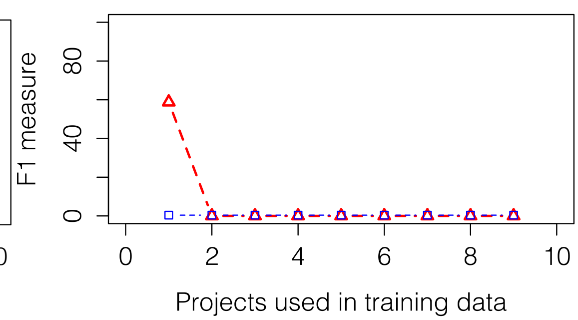
Apache-jmeter (cres/decres)





Jfreechart (cres/decres)





Argouml (cres/decres)