A New approach for fixing bugs in Code Clones: Fix It There Too (FITT)

Vaibhav Pratap Singh Saini
Department of Informatics
Bren School of ICS
University of California, Irvine
vpsaini@uci.edu

Aftab Hussain
Department of Informatics
Bren School of ICS
University of California, Irvine
aftabh@uci.edu

ABSTRACT

Code cloning, the process of reusing similar code segments in different parts of a project, has been a prevalent practice among software developers. However as projects mature, developers lose cognizance about the existing clones in their projects. This is a problem particularly in situations where the cloned code contains bug patterns. In fact, it has been shown in a recent study that in 75% of cases involving code duplication, bug-patterns associated with a code is propogated "as-is" to atleast one of the clone-siblings of the code. Therefore, when fixing bug-patterns in a code segment, it would save the developers a lot of time and effort in finding other similar bug-patterns if they knew about the presence of clones of the code segment. In this paper, we propose a plugin for Eclipse that will proactively inform the developer about the clones of a method in a Java project while the developer is working on the method. This tool will thus improve developers' awareness about the presence of clones in their projects, which will enable them to take appropriate measures while they are correcting any of the clones.

Keywords

clone detection, extending IDEs, information retrieval

1. INTRODUCTION

Code cloning via copy-and-paste is a common practice in software engineering for quick code reuse. Since existing bug pattern can be carried forward when cloning, we present a tool that will seamlessly enable developers to detect clones while they are coding, which would as a result simplify the task of bug fixing among clones. However, before discussing the main motivation of this work for detecting clones, it is important to elaborate on the present stance of software engineers and researchers on the use of clones.

Traditionally, the practice of code cloning has been considered harmful and a symptom of the ignorance of important design abstractions. As such, many previous studies suggest approaches to facilitate the discovery, removal, and refactoring of clones. But recent studies have shown that cloning is not always harmful and has advantages like rapid development, reuse of tested code, separation of concerns, etc. These characteristics have recently attracted researchers to conduct empirical studies and present evidence about the harmfulness of code cloning. Such attempts have questioned our conventional wisdom about the harmful nature of clones. For example, Kasper and Godfrey [14] presented evidence that clones may be intentional and improve developer productivity. Kim et al. [15] found that most of the clones are short lived and hence investment made in refactoring them may not be worth the effort. Toomim et al. [26] showed that managing clones via linked editing to edit multiple cloned regions without much programmer intervention can be an efficient way of dealing with clones. Rahman et al. [22] conducted a study to assess the impact of clones on defect occurrence of software products. They conducted the study on four subject systems and did not find any evidence that cloning is harmful. Such findings suggest that cloning may not be as harmful as perceived to be.

Given that clones are prevalently used, regardless of the attitudes of developers towards them, detecting clones could be useful in a number of aspects. In this paper, we are interested in how code clone detection could help us in bug fixing. The motivation to pursue this interest came from a recent study by Hitesh et al. [24] which shows that in about 75% of cases, the bug pattern associated with the code is duplicated as-is when the code is cloned. Also, while fixing a bug in a piece of code, it is very unlikely that the developer is aware of all the places where this code has been cloned, if any, in the project. While many clone detection approaches have been proposed, which are discussed in Section 5, we propose a plugin for Eclipse that will inform the developer about the clones of a method in a Java project while the developer is working on the method, without imposing any significant cognitive burden on the developer.

Structure of the paper. In Section 5, we present the related work in the area of clone detection: we discuss the extent of the problem of clones based on previous studies and the theoretical approaches that have been adopted to tackle the problem of detecting tools. In Section 3, we describe our tool and its underlying mechanism in details. In Section 2, we present the findings of a survey we carried out to elicit the demand for such a tool among software developers. In Section 4, we present the evaluation plan of our tool. We conclude our paper in Section 6, giving ideas on future efforts

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.
CONF Irvine, California
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

in the area.

2. SURVEY OF TOOL'S DEMAND

In this section, we present the findings of a survey we carried out in order to find out the existing need for an integrated clone detection plugin in the development environment among developers and others involved in software development. In Subsection 2.1, we describe the underlying goals of the survey. In Subsection 2.2, we present the results of our survey. A discussion of the results is provided in Subsection 2.3.

2.1 Survey Goals

The survey was designed to address the following hypotheses,

- *Hypothesis 1.* Reuse of code through copy-pasting is a common practice.

- *Hypothesis 2.* When fixing bugs in a particular code segment of the project, developers often look into other similar code segments that may exist in the project.

- *Hypothesis 3.* A tool that can detect similar code segments in an automated fashion would be desirable.

Although Hypothesis 1, which refers to a common type of cloning practice, has already been well established by virtue of previous studies, it was necessary to ascertain whether it holds true with current practitioners. Hypothesis 2 states the fundamental motivation behind this work that underscores the relationship between bugs and clones in software code. Finally, by testing for Hypothesis 3 we verify our approach for simplifying bug fixing through clone detection.

2.2 Survey Results

A 5-question survey was prepared and was sent out over social media. A total of 72 responses were obtained. Figures 1-5 show the results for each of the questions.

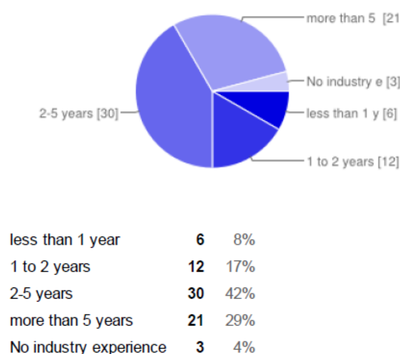

Figure 1: Demographic information of respondents.

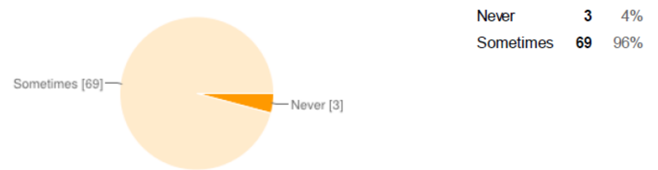

Figure 2: The practice of code copying.

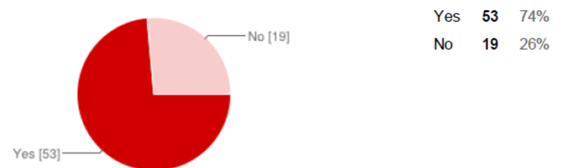

Figure 3: The practice of searching for similar code while bug fixing.

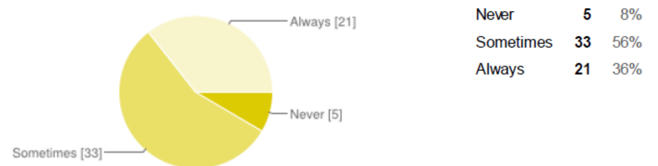

Figure 4: The frequency of searching for similar code.

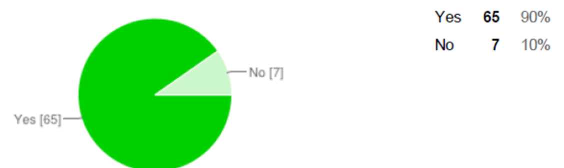

Figure 5: The demand for a clone detection plugin.

2.3 Discussion on Results

From Figure 1 we see that 91% of the respondents had at least a year's experience at the time of completing our survey, which boosts the relevance of the rest of the questions of the survey with respect to our hypotheses. Figure 2 directly addresses and confirms Hypothesis 1. Figures 3 and 4

explore the relation between bugs and clones in development practice. With 74% of the respondents confirming that they search for code segments similar to the one in which they are fixing bugs (of whom 36% do so always), there is enough reason to believe that bugs are carried forward through code reuse (via copy and paste). These findings also indicate that the developers are aware of the possibility of having bugs in clones, which is why they search for similar code when bug fixing (Proving Hypothesis 2). Having confirmed Hypotheses 1 and 2, we had to investigate whether a tool that assists in this search whilst programming would be found useful. Figure 5 gives a 90 % approval for the need of such a tool, hence proving hypothesis 3.

3. PLUGIN DESCRIPTION

In this section we elaborate on our plugin, FITT. We explain our underlying algorithm of FITT's clone detection technique in Subsection 3.1. The plugin's interface and the rationale behind the interface's design decisions are discussed in Subsection 3.2. Finally in Subsection 3.3 we present the implementation details of FITT.

3.1 Clone Detection Algorithm

To detect the clones in the Java projects, we are using the efficient index-based algorithm as proposed by Hitesh et al. in [24]. Figure 6 shows the algorithm and the work flow is shown in Figure 7. The algorithm uses a filtering technique that has shown to significantly speed up the clone detection process when compared the other naive based techniques. We are using Java methods as a code block, however algorithm poses no restriction on selecting other entities like Java class or packages as code blocks.

When the plugin gets activated, it traverses the entire project using Eclipse JDT (Java Development Toolkit) and feeds the code blocks to the indexer where their partial inverted index and forward index gets created, as shown in Figure 7. The detect clones module uses the current method where the developer is currently working on as an query block and then searches the inverted index to obtain the candidate clones. For each candidate clone, detect clones module uses the forward index to obtain all its terms and then calculates the similarity between the candidate clone block and the query block. If the similarity value satisfies the user defined similarity threshold value, the candidate clone gets reported as a clone.

3.2 Plugin User Interface

The Eclipse Java editor acts as the primary user interface for FITT plugin. The plugin is designed to have a minimalistic design in order to minimize the cognitive burden on the user for using and understanding a new utility in the environment. Figure 8 shows a red colored annotation signifying that the FITT has found clones for the method the developer is currently working on. Annotation can have one out of three colors: a) *Red* - signifies there are more than 10 clones of the current method in the project; b) *Yellow* - signifies there are five to 10 clones and c) *Green* - signifies the existence of less than five clones. A click on the annotation would show a view-part, as shown in Figure 9, listing all the clones of the current method. This list can be used to navigate to the clones of the current method.

3.3 Implementation Details

```
1  buildIndex(B, sim, θ)
2  begin
3      I, R ← {}
4      for each code block b1 in B do
5          b1 ← sort (b1,globalTermPosotionMap)
6          i ← 0
7          while i ≤ (|b1|-⌈ θ |b1| ⌉+1) do
8              It ← It ∪ id(b1)
               /* Here t is a term at b1[i]
9              b1[i] ← 0
10         end
11     end
12     return R
13 end
14 buildForwardIndex(B)
15 begin
16     FI ← {}
17     for each code block b1 in B do
18         FI_id(b1) ← terms(b1)
19     end
20 end
21 detectClones(b1, I, sim, t)
22 begin
23     candidatesList ← new ArrayList()
24     cloneSet ← {}
25     b1 ← sort (b1,globalTermPosotionMap)
26     for each term t in prefix(b1) do
27         for each id(b2) in It do
28             candidatesList.add(id(b2))
29         end
30     end
31     for each id(b2) ∈ candidatesList do
32         simVal ← sim(b1, FI_id(b2))
33         if simVal ≥ θ then
34             cloneSet ← cloneSet ∪ {id(b1), id(b2), simVal}
35         end
36     end
37     return cloneSet
38 end
```

Figure 6: Algorithm for efficient Index-based approach to detect clones.

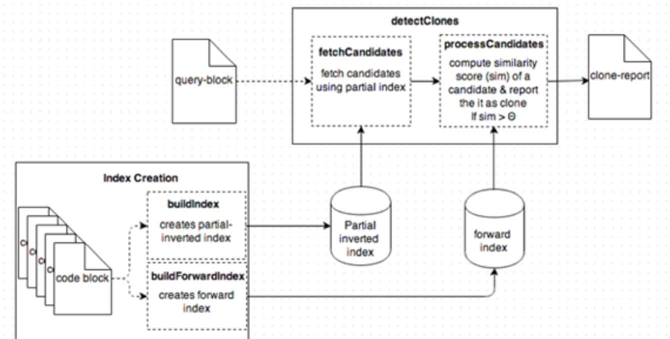

Figure 7: Workflow of efficient index-based approach.

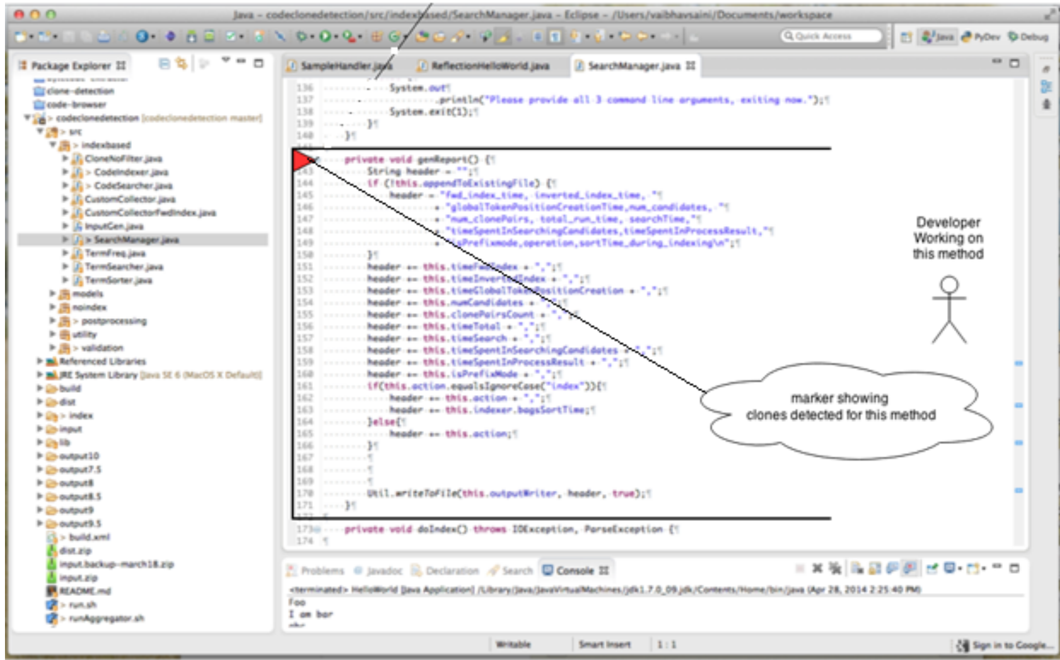

Figure 8: Red colored annotation signifying the clones of the current method have been detected.

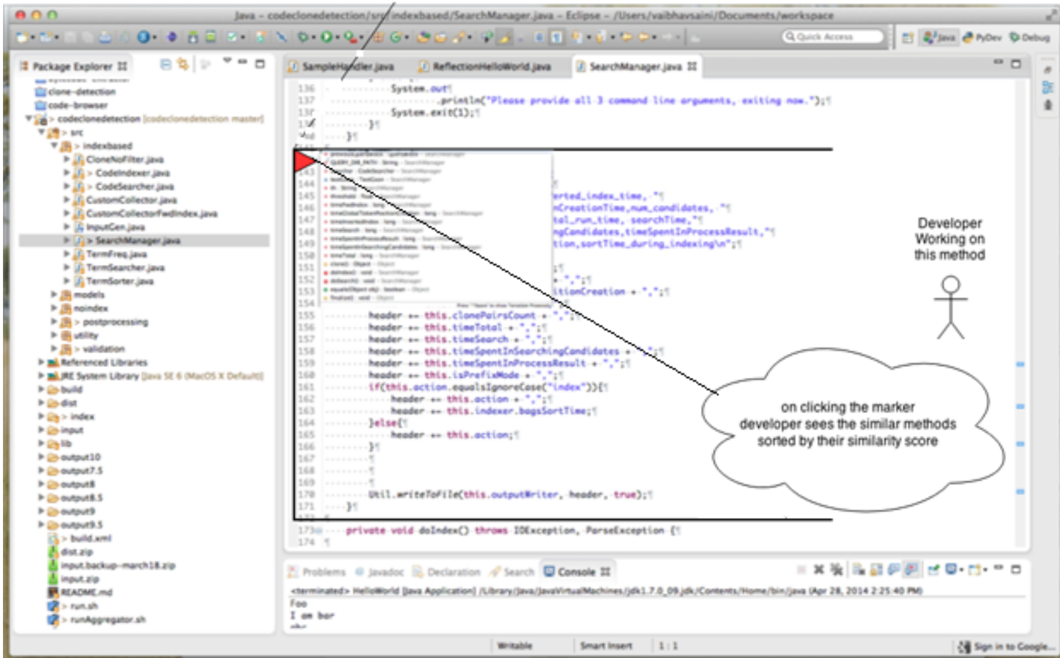

Figure 9: Context menu showing detected clones.

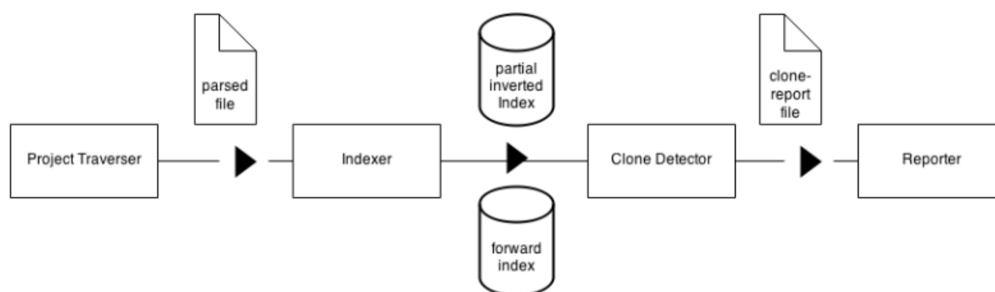

Figure 10: FITT Modules

FITT has four modules as shown in Figure 10. In the following, we discuss each of the modules.

3.3.1 Project Traverser Module: creating the input for the plugin

On activation of the plugin, it traverses the current project the developer is working on and while traversing it creates a file where each line represents a bag. A bag is an abstraction for a java method. It contains terms and their frequency in the method. We use EclipseâĂŹs JDT (Java Development Toolkit) library to traverse the abstract syntax tree of the current project.

3.3.2 Indexer Module: creating indexes using the input

The file created by the Project Traverser Module is used as an input to the Indexer Module which then creates a partial inverted index and a forward inverted index. The partial inverted index is used to search the candidate clones whereas the forward index is used to verify if the candidates are clones or not.

3.3.3 Clone Detector Module: detecting the clones

The clone detector module listens to the selection changed event on the eclipse editor. It gets the current method information from the event object using which it creates a query block using the current method and then searches for the clones using the indexes created by the indexer Module. It reports the clones of the current methods in a clone-report file.

3.3.4 Reporter Module: reporting the clones

After the clone detector module detects the clones of the current method, the reporting module does 3 tasks; i) create an annotation on the editor (where the line numbers are written). This annotation signifies the presence of clones of the current method in the project. ii) read the clone-report file as created by the clone detector and iii) creates a view part to display the information about the clones.

4. FUTURE EVALUATION

In this section, we present the evaluation methodology we look forward to carry out on our tool. We shall focus on the following evaluation criteria: performance (4.1) and ease-of-use (4.2), where we shall a comparison with previous CP-Miner [7] and CnP [8] will be made based on usage experience.

4.1 Performance Evaluation

We shall evaluate the performance of the tool by recording the startup times the tool needs for projects of varying sizes (the largest portion of the tool's startup time is used for building the repository index of the project). This evaluation will help us determine the degree of scalability of our tool. Next we shall record the time the tool needs to enlist the detected clones in the user interface.

4.2 Ease-of-Use Assessment

In order to evaluate the ease-of-use of our tool, we will hire two developers having at least two years of industry experience who are well-versed in using the Eclipse IDE. We shall provide them three plugins altogether: FITT, CnP, and CP-Miner. We will install the plugins in their IDEs and shall ask them to work on their projects using the plugins for 30 days, so that they dedicate 10 consecutive days for each individual plugin. After each 10-day period we shall ask them to complete a survey in order to capture their experiences with the tool. Our survey questions will form around the following notions:

- Work interference (e.g., how distractive the tool was while they were coding).
- Context-switching between tool output and coding editor
- Usefulness of the tools' suggestions (e.g. how often did you find bugs in the clone methods suggested by the tool?).

A scale based answering scheme will be used for the survey, which would enable us to compare the survey results of each tool in a consistent manner.

5. RELATED WORK

In this section we present the works in the field of clone detection. We have categorized the works across four different dimensions. In Subsection 5.1, we elaborate on the theoretical and empirical evidences that are available in the literature related to the problem of cloning. In Subsection 5.2, we discuss previous approaches that have been used for clone detection, and emphasize upon how our tool relates to their approaches. In Subsection 5.3, we stress upon the core concepts that were followed in the previous approaches. Finally in Subsection 5.4, we elaborate on a tool that we found to be most similar to our tool, which was yet different from ours in a number of important ways.

5.1 Studies related to theoretical and empirical evidences of cloning

Reusing code fragments via copy-and-paste, with or without modifications or adaptations, also known as code cloning, has become a common behavior of software engineers [5]. Although pervasive, code cloning has traditionally been criticized by researchers and leading practitioners alike. Parnas [4] said "if you use copy and paste while you're coding, you're probably committing a design error." Indeed, if instead of copying code, we move it into its own method, future modifications will be easier because we will need to modify the code in only one location. The code will be more reliable because we will have only one place to ensure that the code is correct. Consequently, a considerable amount of research in cloning is concentrated on detecting clones in existing source code [3, 2, 5, 11, 13, 17, 19], removing and refactoring them [25], emphasizing the need of clone detection tools to assist developers in identifying the similar code blocks.

5.2 Previous clone detection approaches

Toomim et al. [26] showed that managing clones via linked editing to edit multiple cloned regions without much programmer intervention could be an efficient way of dealing with clones. A limitation of their work is that it requires the developer to manually select and mark up code fragments as clones. Furthermore, Linked Editing is only applicable when identical changes are required for all of the clones. Our work comes closest to Patricia Jablonski's work [8] where they also propose a tool named CnP, which will detect copy-and-paste in the IDE (Integrated Development Environment) in real time. When selected statements are copied, CnP can determine the contents of the clipboard and use this information later on for error detection and other purposes. Once the statements are pasted, CnP then, highlights the statements in the IDE's editor so that the developer can keep track of both the pasted code and its origin. Our tool differs from Patricia's work. We use a clone detector to detect all the existing clones and it doesn't require capturing the copy and paste events. Our tool can be used to detect clones in an existing project including legacy systems.

5.3 Core concepts behind previous clone detection techniques

Code clone detection aims at finding exact or similar pieces of code known as code clones. Several techniques have been proposed for clone detection over many years [22, 6, 16, 3, 20, 9, 21, 10, 12, 1, 27, 18]. These techniques differ in many ways ranging from the type of detection algorithm they use to the source code representation they operate on. Techniques using various representations include tokens [13], abstract syntax trees (AST) [3, 10] program dependence graphs [6, 16], suffix trees [12, 13, 18], text representations [1], hash representations [27], etc. Each of these different approaches has their own merits and is useful for different use-cases. For example, AST based techniques have high precision, and are useful for refactoring, but may not scale. Moreover, token-based techniques have high recall but may yield clones, which are not syntactically complete [23]. They are useful where high recall is important.

5.4 A similar tool

Li et al. [7], have created a tool called CP-Miner that has attempted to detect copy-paste errors in the context of traditional clone detection. CP-Miner has error detection capabilities top. CP-Miner uses a token-based approach with data mining and a gap constraint. CP-Miner determines a mapping relationship between identifiers in copy-and-pasted code fragments and then applies a heuristic to conclude whether the fragments are consistent or not. We are using an efficient index-based technique, which falls under the information retrieval approach.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented FITT, an easy-to-use clone detection tool for the Eclipse IDE. The main motivation behind building this tool was the fact that in a majority of code cloning cases, as established by a recent study, it has been found that existing bugs in a code segment are carried forward "as is" when the code segment is copied. In addition to that coders are generally unaware of the existing clones of a code segment in their project. Consequently, a tool that proactively informs developers about existing clones of a code segment while they are working on it would greatly reduce the chances of missing similar bugs that may exist elsewhere in the project. A pre-assessment survey confirmed the demand of such a tool among software developers. Currently, our tool can successfully detect clones within an Eclipse project at the method-level, for code written in Java.

For future implementations, we look forward to extend FITT to detect class-level clones and work with other programming languages such as Python. The present implementation only uses the initial index of the project created during tool-startup, and does not take into account changes made during a coding session. The next step would be to customize the tool's index generation phase to execute intermittently. Since index generation would take a considerable time particularly for large projects, it is important these executions are implemented in a multi-threaded fashion such that the work flow of the developer is not interrupted while the index generation takes place.

With regard to the user interface of FITT, the present implementation uses a categorized marker-coloring scheme for indicating the pervasiveness of the current method, which might restrict the user's view of how widespread the method is. An improvement would be to use a graded coloring scheme calibrated with a pervasiveness metric. Another area of improvement would be the output list (context menu) of the clone method names. Currently, the list is ordered solely based on the degree of similarity between the current method and the clone, with the most similar clones appearing in the beginning of the list. A ranking that also takes into consideration the relevance clone method with respect to the current method could be used, where relevance could be based on the location of the clones within a project (e.g. clone siblings in the same class as the current method could more relevant than other clone siblings).

7. REFERENCES

[1] M. Asaduzzaman, C. K. Roy, and K. Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *Proceedings of International Conference on Software Maintenance '13*, pages 230–239, 2013.

[2] B. Baker. A program for identifying duplicated code. In *Computing Science and Statistics*, pages 24–49, 1992.

[3] I. Baxter, A. Yahin, L. Moura, M. SantâĂŹAnna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of ICSM '98*, page 368, 1988.

[4] S. Ducasse, M. Rieger, and S. Demeyer. On the criteria to be used in decomposing systems into modules. In *Commun. ACM*, pages 1053–1058, 1972.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 109–118. IEEE Computer Society, 1999.

[6] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[7] J. Ferrante, K. Ottenstein, and J. Warren. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[8] P. Jablonski. Managing the copy-and-paste programming practice. In *Proceedings of OOPSLA '07*, pages 933–934. ACM, 2007.

[9] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita. Kclone: A proposed approach to fast precise code clone detection. In *Proceedings of IWSC '09*, 2009.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE '07*, pages 96–105, 2007.

[11] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research and Software Engineering*, pages 171–183. IBM Press, 1993.

[12] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *Proceedings of ICSE '09*, pages 603–606, 2009.

[13] T. Kamiya, S. Kusumoto, and K. Inouey. "ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[14] C. Kasper and M. Godfrey. "cloning considered harmful": patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *In Proceedings of FSE*, pages 235–255, 2005.

[16] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proceedings of 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169. ACM, 2000.

[17] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS '01 Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. ACM, 2001.

[18] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of CSMR 2012*, pages 309–318, 2012.

[19] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309. IEEE Computer Society, 2001.

[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of ICSM '96*, page 244. IEEE, 1996.

[21] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of International Conference of Software Maintenance '09*, pages 491–494, 2009.

[22] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[23] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[24] H. Sajnani and C. Lopes. A parallel and efïňĄcient approach to large scale clone detection. In *IWSC*, pages 46–52. IEEE, 2013.

[25] M. Shomrat and Y. Feldman. Object-oriented programming. In *Proceedings of ECOOP 2013, Lecture Notes in Computer Science*, pages 502–526, 2013.

[26] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 173–180, 2004.

[27] S. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of 13th Working Conference on Reverse Engineering.*, pages 13–22, 2011.