Supporting Clone Analysis with Tag Cloud Visualization

Manamu Sano[†], Eunjong Choi[†], Norihiro Yoshida[‡], Yuki Yamanaka[†], Katsuro Inoue[†]

[†]Osaka University, Japan
{m-sano,ejchoi,y-yuuki,inoue}@ist.osaka-u.ac.jp

[‡]Nagoya University, Japan
yoshida@ertl.jp

ABSTRACT

So far, a lot of techniques have been developed on the detection of code clones (i.e., duplicated code) in large-scale source code. Currently, the code clone research community is gradually shifting its focus of attention from the detection to the management (e.g., clone refactoring). During clone management, developers need to understand how and why code clones are scattered in source code, and then decide how to handle those code clones. In this paper, we present a clone analysis tool with tag cloud visualization. This tool is aimed at helping to understand why code clone are concentrated in a part of a software system by generating tag clouds from a collection of identifier names in source code.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement-Restructuring, reverse engineering, and reengineering

Keywords

Code clone, Tag cloud

1. INTRODUCTION

A code clone is a code fragment that has identical or similar code fragments to it in the source code [16, 18]. If developers modify a code fragment, they have to determine whether or not to modify its code clones in source code [21]. In recent decades, a lot of techniques have been developed on the detection of code clones (i.e., duplicated code) in large-scale source code [16, 18]. Currently, the code clone research community is gradually shifting its focus of attention from the detection to the management (e.g., clone refactoring) [6].

A number of code clones are typically detected from large-scale source code by a code clone detection tool [16, 18]. During management of those detected code clones, it is difficult for developers to check all of them manually [19, 24]. However, at the beginning stage of source code reconstruction for maintainability improvement, it is unnecessary to check all of them [24]. According to Rieger et al. [24] and our experience in collaboration with industry, at the beginning stage of source code reconstruction, developers firstly investigate parts of source code that involve a number of code clones. And then they consider the reasons why the parts involve a number of code clones. Finally, they decide how to handle those code clones. In this process, they have to focus on only the parts that involve a number of code clones, and it is unnecessary to check all of the detected code clones.

For efficient grasp of parts that involve a number of code clones, several code clone visualization techniques have been developed so far [19, 24, 17]. Scatterplots (dotplots) is a two-dimensional plots in which the axes represent the files or directories of the system or systems and the points represent the presence or absence of a clone relation between them [7, 9, 27]. Developers readily know in which files or directories code clones exist by scatterplots.

However, since scatterplot provides only the location information of code clones, it is difficult for developers to understand the reason why code clones are concentrated in parts of source code. In other words, even if developers use scatterplot, they have to check source files involving code clones in order to understand the reason why those code clones exist in the files. Although lexical information (e.g., variable, function, type names in code clone) can give a hint for understanding why those fragments are code clones, existing scatterplots do not use lexical information of code clones directly.

In this paper, we present a code clone analysis tool **CloneCloud** that supports understanding of code clones based on tag cloud visualization. In general, *tag cloud* depicts keyword metadata for efficient understanding and information retrieval of given data. In order to support intuitive understanding and analysis of code clones, CloneCloud generates *tag cloud* from identifier names (e.g., variable, method, type name) in detected code clones (see Figure 2(b)). A *tag cloud* that is generated by CloneCloud helps developers to grasp implementations that lead code clones, and get a clue to the reason why code clones exist. Also, once developers get an idea of the reason for code clones from the tag cloud, they can retrieve code fragments of code clones by tags (i.e., identifier names) and verify their idea.

CloneCloud provides also scatterplots for the selection of interested a single or a pair of directories (see Figure 2(a)). A *tag cloud* is depicted from identifier names of code

InnoSWDev'14, November 16, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3226-2/14/11...$15.00
http://dx.doi.org/10.1145/2666581.2666586

clones in the selected directories (see Figure 2(b)). Each tag (i.e., identifier name) in the *tag cloud* is link for opening the source code view for code clones involving the tag (see Figure 2(c)). The combination of the scatterplots, the *tag cloud*, and the source code view efficiently supports developers who need to understand the reason why code clones exist and decide how to handle them. Developers who use only traditional scatterplots have to investigate a number of code clones in selected source files without keyword metadata that supports instinctive understanding of those code clones.

2. ARCHITECTURE OF CLONECLOUD

CloneCloud takes source files of Java system as input and visualizes code clones using *Live Scatterplots* [7] and *tag cloud* based on the detection result of CCFinder. The architecture of CloneCould is illustrated in Figure 1. As shown in this figure, CloneCloud provides, three principal views, Scatterplot View, Tag Cloud View and Source Code View.

When CloneCloud is initially executed, first of all, it internally invokes CCFinder that detects code clones from the input source files. Then, based on the results of the detection, it renders Scatterplot View that represents location and density of the code clones using *Live Scatterplots*. After a pair of directories are selected in the Scatterplot View, Tag Cloud View and Identifier Table, which represent tag could of identifier names and information of identifier names in tabular form respectively, are popped up. In Tag Cloud View, identifier names of red fonts mean that they are included in code clones. After identifier names is selected in Tag Cloud View, finally, Source Code View, which shows source code of code clones that include the selected identifier name, is opened.

The following subsections explain the detailed of Scatterplot View, Tag Cloud View and Source Code View in CloneCloud.

2.1 Scatterplot View

Figure 2(a) shows Scatterplot View that represents result of Apache Ant revision 1486439. For better overview, we labeled the important features in this figure.

Scatterplot View provides scatterplot (label 2) of the input source files using *Live Scatterplots*. In the scatterplot, both the vertical and horizontal axes represent directories of the input system. When code clone exists between the vertical and horizontal directories, the cell where the vertical and horizontal directories are met is colored based on the *Clone Density* between vertical and horizontal directories. Suppose that LEN_{Clone} represents the number of token sequences of code clones between the directories, LEN_{All} represents the number of token sequences of overall source code between the directories, then, the *Clone Density* between the directories is calculated as follows:

$$Clone\ Density = \frac{LEN_{Clone}}{LEN_{All}}$$

In the scatterplot, blue cell represents that *Clone Density* is lower than others. Meanwhile, red cell represents higher. Moreover, absolute paths of the vertical and horizontal directories where mouse is located in scatterplot are shown in (label 1). Scatterplot View provides two customization options, *Clone Density* and *RNR(S)* metric [13]. *RNR(S)* metric is proposed by our laboratory. Higo et al. found out that *RNR(S)* metric is effective to filter out uninteresting code clones (i.e., a code clone whose existence information is useless when using code clone information in software development or maintenance) [13]. Moreover they found that f-value reached its greatest when the threshold was 0.7. It is the ratio of non-repeated code sequence in clone set (i.e., a set of code clones that are identical or similar to each other) between vertical and horizontal directories. Suppose that the clone set S includes n code clones, $c_1, c_2 \ldots, c_n$, $LOS_{whole}(f_i)$ represents the length of whole token sequence of code clone c_i, and $LOS_{n-repeated}(f_i)$ represents the length of non-repeated token sequence of code clones c_i, then,

$$RNR(S) = \frac{\sum_{i=1}^{n} LOS_{n-repeated}(c_i)}{\sum_{i=1}^{n} LOS_{whole}(c_i)} \times 100$$

A clone set whose *RNR(S)* is lower means that code fragments in clone set S mostly consist of repeated token sequences. In most cases, repeated token sequences are involved in language-dependent clones (e.g., code clones that involve consecutive if or if-else blocks, case entries, variable declarations). The threshold for *Clone Density* and *RNR(S)* can be changed by moving scroll bar in (label 3, 4) respectively. This view also provides customizing options (label 5) for Tag Cloud View as follows:

- The maximum number of identifier names shown in Tag Could View.
- A criterion that decides the size of identifier names in Tag Could View (The criteria are frequencies of identifier names, TF-IDF (Term Frequency Inverse Document Frequency), and arbitrary).
- The minimum sequence length of identifier names shown in Tag Could View.
- The minimum IDF values of identifier names shown in Tag Could View

Using Scatterplot View, a user can comprehend the density of code clones in the directories. Consequently, she can find out candidates for clone refactoring by focusing on the directories where code clones are highly concentrated such as red cell in Figure 2(c).

2.2 Tag Cloud View

Tag Cloud View shows identifier names in the selected directories in Scatterplot View using *tag cloud* [1]. Using *tag cloud*, readability of this view can be improved, because *tag cloud* effectively displays frequency of identifier name with different color. Figure 2(b) shows Tag Cloud View for "optional\clearcase" directory. This view displays identifier names in source code for the directories. Among them, identifier names of black font mean that they are contained only in the source code of the directories. Meanwhile, identifier names of red font mean that they are included in code clones in the directories. Only these identifiers names provide hyperlink to the Source Code View.

[1]implemented by WordCram `http://wordcram.org/`

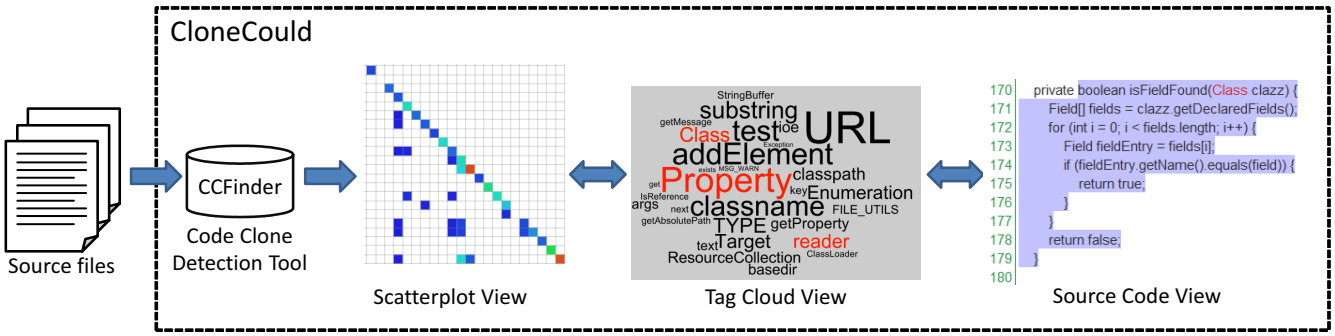


Figure 1: Architecture of CloneCloud

Using the Tag Cloud View, a user instinctively understands the role of the directories containing code clones. For example, in Figure 2(b), a user can take a cue for understanding of source files in "optional\clearcase" directory implement the functionality for ClearCase command. Moreover, she can take a cue for understanding of code clones that are included in the source code of argument creation for command line by identifiers 'Commandline' and 'createArgument'.

2.3 Source Code View

Source Code View shows the source code of the code clones that include the selected identifier names in the Tag Cloud View. Source Code View about identifier 'createArgument' is illustrated in Figure 2(c).

This view displays the absolute path of the directories which contain the select identifier names. It also provides information of code clones which contain the selected identifier name and source code of the selected code clones. Moreover, information of code clones who are belong the same clone set of the selected code clones are shown with source code of the selected code clone. In the source code, selected identifiers are shown in red font. In addition, identifiers that are also represented in the Tag Cloud View are shown in green font.

Using Source Code View, a user can confirm the source code that includes the selected identifier name. Accordingly, once she read the source code in this view, she can take a cue for understanding of the code clones that she is interested in.

3. USAGE EXAMPLE

We assume that a developer would like to perform reconstruction of large-scale Java system. She dose not know about the source code, therefore, at first, she would like to identify the similar parts of the source code in course-grained level. She decides to find the directory-level similarity because directory-level similarity provides two valuable options to her; the first option is that if she finds a similar directory pair, she can merge the pair into one directory. The other option is that if she figures out a directory that includes a great amount of code clones, she can remove these clones.

In order to find similar directories, she executes CloneCloud. Scatterplot View is rendered based on the result of the clone detection from CCFinder. She can change the threshold of *RNR(S)* and *Clone Density* in the Scatterplot View to find out the target directories for reconstruction. Next, she selects the cell with red color because this directory contains a higher number of code clones compared to the other directories.

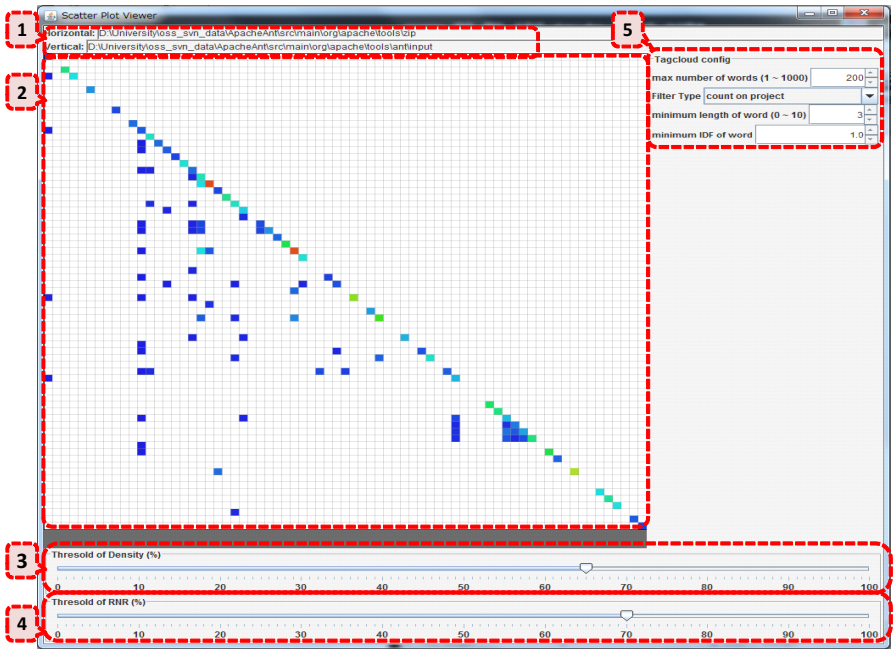
After she selects the target directory, she can comprehend identifier names in the selected directories via Tag Cloud View. Also, she can refer Identifier Table that shows the detailed information of the identifier names. With the Tag Cloud View and Identifier Table, she understands the role of the selected directories as well as code clones in the directories.

After that, she can select the identifier names in the Tag Cloud View. She can select identifier name that is shown in bigger font compared to the others. An identifier name with bigger font means frequently appeared one. For example, identifiers 'Commandline', 'cmd' and 'createArgument' in Figure 2(b) is promising to be important keywords. By selecting a specific identifier name in the Tag Cloud View, Source Code View is popped up. In this view, she can understand source code of code clones that include the selected identifier names in-detail. It helps her to identify a single or pair of directories to be reconstructed.

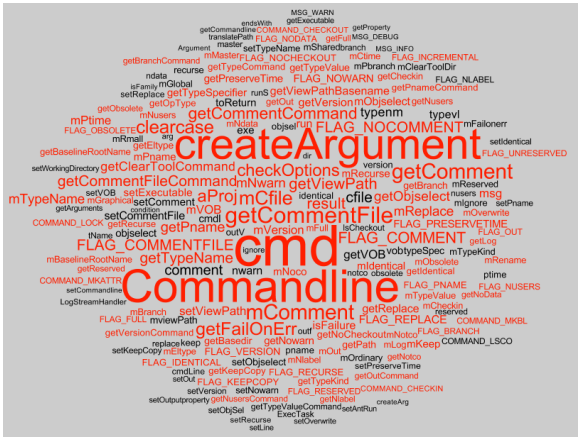
4. RELATED WORK

Many studies have been proposed for detecting code clones. Baker proposed an approach for detecting code clones then developed a tool named Dup. It detects Type-1 and Type-2 code clones based on the similarities of token sequences [3]. CP-Miner detects Type-1, Type-2, and Type-3 code clones based on the similarities of token sequences [3]. To detect code code clones, it adopted frequent subsequent mining which is an association analysis approach to discover frequent subsequences in a collection of sequences [22].

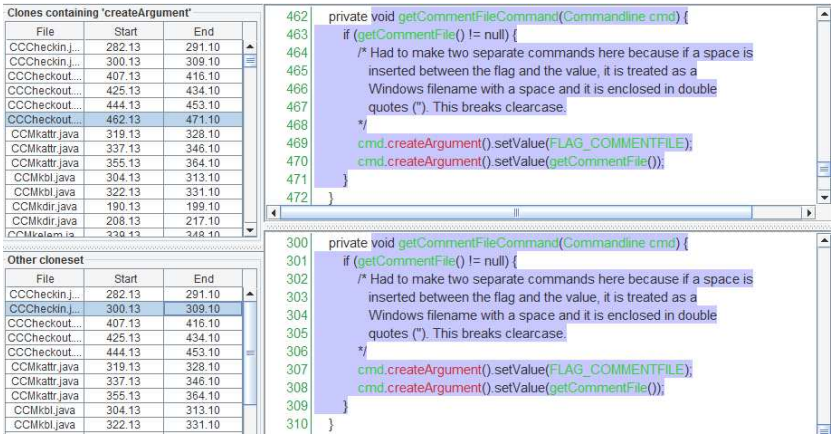
Several approaches using CCFinder have been proposed. Sasaki et al. proposed an approach that detects identical files and then investigate the characteristics of them [25]. This study identifies identical files without any (or just slight) modifications in comments or headers using MD5 hash values of the tokenized files. A tool named D-CCFinder implemented by Livieri et al. detects code clones at distributed environment based on CCFinder [23]. D-CCFinder partitions the clone search for very large systems into smaller pieces to be distributed to detect code clones with high speed. This goal of study is the same between their study and this study, but our approaches detect code clones on the single PC, whereas D-CCFinder uses distributed approach.



(a) Scatterplot View



(b) Tag Cloud View



(c) Source Code View

Figure 2: Screenshot of CloneCloud

Several approaches have been proposed on the identification and the categorization of clone refactoring opportunities in source code. Balazinska et al. [1] proposed an approach for supporting clone refactoring by categorizing code clones based on the differences of them. Baxter et al. [2] have developed a clone detection tool CloneDR based on AST similarity. CloneDR derives only syntactically-complete clones that can be easily refactored. Hotta et al. [15] focused on Form Template Method refactoring pattern [10], and proposed a specialized approach to identifying its opportunities. For the prioritization of clone refactoring opportunities, Higo et al. [14] and Choi et al. [5] proposed metric-based approaches respectively. Also, search-based approaches have been proposed for scheduling clone refactoring based on its benefit and effort [4, 20, 28].

Lucia et al. investigated how source code artifact labeling performed by information retrieval techniques would overlap from labeling performed by humans[8]. Haiduc et al. investigated the suitability of several automated text summarization techniques[26], mostly based on text retrieval methods, to capture source code semantics in a way similar to how developers understand it[12]. Gethers et al. developed CodeTopics, an Eclipse plug-in that shows the similarity between source code and high-level artifacts (e.g., requirements) also highlights to what extent the code under development covers topics described in high-level artifacts[11].

5. SUMMARY AND FUTURE WORK

In this paper, we presented a code clone analysis tool CloneCloud that supports understanding of code clones based on tag cloud visualization. In order to support intuitive understanding and analysis of code clones, CloneCloud generates *tag cloud* from identifier names (e.g., variable, method, type name) in detected code clones (see Figure 2(b)). A *tag cloud* that is generated by CloneCloud helps developers to grasp implementations that lead code clones, and get a clue to the reason why code clones exist.

As future work, we need to perform an experiment for confirming the usefulness of CloneCloud. We plan to compare the behaviors of participants who use CloneCloud and the existing clone visualizers.

6. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers 25220003 and 26730036.

7. REFERENCES

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of METRICS '99*, pages 292–303, 1999.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, pages 368–377, 1998.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[4] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *Proc. of GECCO*, pages 1885–1892, 2006.

[5] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *In Proc. of IWSC*, pages 7–13, 2011.

[6] R. K. C.K. Roy, M. F. Zibran. The vision of software clone management: Past, present and future. In *Proc. of IEEE CSMR/WCRE*, pages 18–33, 2014.

[7] J. R. Cordy. Live scatterplots. In *Proc. of IWSC*, pages 79–80, 2011.

[8] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *Proc. of ICPC*, pages 193–202, 2012.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM*, pages 109–118, 1999.

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[11] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. CodeTopics: Which topic am I coding now? In *Proc. of ICSE*, pages 1034–1036, 2011.

[12] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proc. of WCRE*, pages 35–44, 2010.

[13] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, 49(9-10):985–998, sep 2007.

[14] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution*, 20(6):435–461, 2008.

[15] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proc. of CSMR*, pages 53–62, 2012.

[16] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *Proc. of ICSE*, pages 96–105, 2007.

[17] E. Juergens. *Why and How to Control Cloning in Software Artifacts*. PhD thesis, Technische Universität München, 2011.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[19] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *Proc. of ICSM*, pages 305–314, 2005.

[20] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Journal of Software: Practice and Experience*, 41(5), 2011.

[21] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. In *Proc. of ICSE*, pages 310–320, 2012.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale

software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[23] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proc. of ICSE*, pages 106–115, 2007.

[24] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proc. of WCRE*, pages 100–109, 2004.

[25] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in FreeBSD ports collection. In *Proc. of MSR*, pages 102–105, 2010.

[26] K. Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481, Nov. 2007.

[27] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. of METRICS*, pages 67–76, 2002.

[28] M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proc. of SCAM*, pages 105–114, 2011.