JCCD: A Flexible and Extensible API for Implementing Custom Code Clone Detectors

Benjamin Biegel
University of Trier
Department of Computer Science
54286 Trier, Germany
biegel@uni-trier.de

Stephan Diehl
University of Trier
Department of Computer Science
54286 Trier, Germany
diehl@uni-trier.de

ABSTRACT

Code clone detection is an enabling technology for plenty of applications, each having different requirements for a code clone detector. In the tool demonstration we present JCCD, a code clone detection API, which is based on a pipeline model. By combining and parameterizing predefined API components as well as by adding new components, the pipeline model does not only facilitate to build new custom code clone detectors, but also to parallelize the detection process.

Categories and Subject Descriptors

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

General Terms

Algorithms

1. INTRODUCTION

A code clone is a code fragment in source code which is identical or similar to another code fragment—this description is likely to be the most popular text clone in code clone literature. Nevertheless, it is still an open issue to find a more accurate definition for this fundamental term. By now, over 40 approaches [10] have been introduced to detect code clones—without having a consistent notion of what a code clone actually is.

Due to the ambiguity of the term *code clone*, our main idea is to put the user in control. To this end, we introduced a generic pipeline model that coordinates the interplay of all required steps in a code clone detection process. Based on this model, we developed the Java Code Clone Detection API (JCCD). It allows to adapt code clone detectors to the requirements of higher-level analyses. In other words, detectors for different notions of code clones can be developed. Code clone detectors implemented with JCCD can be easily integrated into other Java applications. JCCD allows to re-implement state-of-the-art techniques as well as to realize new ideas and concepts. In particular, the open-closed principle [7] guided the design of the API. Thus, without changing the overall pipeline architecture, whole parts of the detection process can be replaced and extended. Additionaly, each phase can be adjusted through user settings.

Copyright is held by the author/owner(s).
ASE'10, September 20–24, 2010, Antwerp, Belgium.
ACM 978-1-4503-0116-9/10/09.

Despite the overhead caused by the various mechanisms for customizing and extending JCCD, the performance of JCCD can compete with other code clone detection tools.

2. THE JCCD PIPELINE

An overview of the pipeline which serves as a basis for JCCD is given in Figure 1. By splitting the analysis into several intermediate steps, JCCD achieves a flexible and easily extensible architecture. Without exception, every step of the pipeline implementation can be supplemented or replaced by alternative approaches.

Parsing: The pipeline gets plain source code as an input. The task of the parsing step is to make this source code suitable for further steps of the analysis. The parsing divides the input into a set of *source units*. A source unit might be a subtree of an abstract syntax tree (AST), a line, a subgraph of a program dependence graph, or the like. The representation of a source unit depends on the used approach (e. g. text-based, token-based, AST-based, or metric-based). Every approach requires different techniques like a line extractor, a lexer, or a parser.

Preprocessing: Normalization of a set of source units or an AST in JCCD turns them into a regular form and thus makes different source units more similar. The goal of the preprocessing is twofold: to normalize a set of source units and to add additional annotations. Preprocessing in JCCD is actually implemented by several cascaded preprocessors. Every preprocessor gets an (preprocessed) AST as input and returns a preprocessed AST as output. The user is free to select which preprocessors are to be used. A preprocessor is able to annotate, remove, collapse, and group AST-nodes. Some preprocessors normalize the AST with the above-mentioned operations, such as removing modifiers, generalizing variable names, or simplifying fully qualified identifiers. Preprocessors can also compute new annotations based on the annotations set by previous preprocessors. These annotations can either be important for both subsequent steps or further preprocessors. For example, annotations can enable to remove getter and setter methods, to remove redundant parentheses, to mark the scope of variables, or to parameterize variable names consistent within a subtree.

Pooling: The pooling step enables a preselection of candidates which might form a code clone. Preprocessed source units are grouped into different sets, called *pools*, based on user-defined criteria. Usually, these criteria are characteristics that can be directly read from the source unit and its annotations without comparing it to another one. For

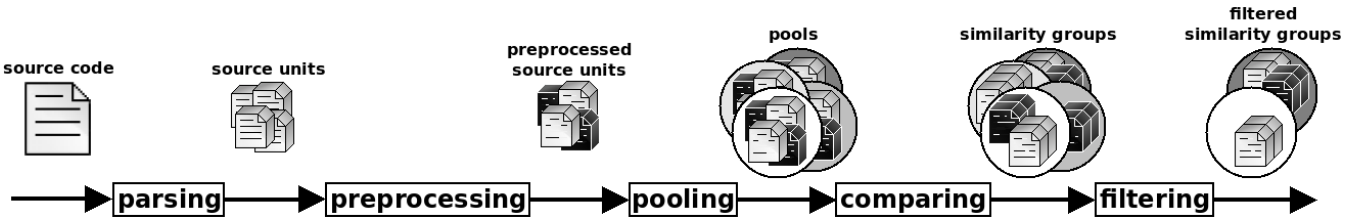

Figure 1: Proposed Generic Code Clone Detection Pipeline.

example, source units might be put into the same pool if they have the same variable name, the same numeric value, the same syntactical function within the source code, or the same annotations (computed in the preprocessing). Source units which are not in the same pool are not considered as candidates for clone pairs.

Comparing: The division of source units into pools allows the comparing step to apply a divide-and-conquer strategy. All of the given pools will be processed sequentially by comparing all contained source units recursively. In JCCD the comparison is realized by using a user-defined set of comparators. Each comparator decides if two subtrees satisfy particular characteristics. Two subtrees are called *similarity pair* only if the combination of all comparators is true.

Filtering: At the end, the filtering step is responsible for removing non-relevant similarity pairs out of the result set. As before, the filter criteria are selected by the user. We consider these final similarity pairs as *code clones*.

The user is free to control the behaviour of the pipeline by removing, replacing, or adding components. For the current version of JCCD we implemented more than 30 comparators, over 40 preprocessors, and three different pooling strategies.

Due to the pipeline model, code clone detectors implemented with JCCD can easily exploit parallel processing on a multi-core computer. For example, the comparing step may process all pools one after another. By splitting this task into multiple threads a multi-core system is able to process a set of pools simultaneously. During the evaluation of JCCD we have adapted the comparing step to this functionality. Multiple test runs show that the parallelization has a significant impact on the total runtime.

3. RELATED WORK

In a recent survey [11] Roy and Cordy found more than 40 different approaches to code clone detection. They can be roughly categorized by the kinds of information they process: strings [2], tokens [5], trees [1], program dependence graphs [6], metrics [8], or hybrid approaches [3].

CloneDetective implements a pipelined approach for extensible token-based code clone detection [4]. In their approach the actual detector is a single component. In contrast, JCCD is AST-based and the detection process is further subdivided into phases.

4. CONCLUSIONS

JCCD is a versatile API for implementing code clone detectors. It was originally developed to provide a customizable code clone detector for our refactoring identification framework [12], recently it has also been used to enable detection of more complex refactorings [9].

We are currently writing and translating the JCCD user documentation and have released JCCD into open source (see `http://jccd.sourceforge.net`) under the new BSD license—placing almost no restrictions on the users of the API and its source code. JCCD may enable other researchers to easily implement their own code clone detection approach or to realize more high-level analyses on top of JCCD.

5. REFERENCES

[1] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society, 1998.

[2] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the Frontiers of Software Maintenance, 25th IEEE International Conference in Software Maintenance (ICSM'09)*, pages 109–118, 1999.

[3] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 321–330. ACM, 2008.

[4] E. Jürgens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 603–606. IEEE Computer Society, 2009.

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

[6] 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.

[7] R. C. Martin. The open-closed principle. C++ Report, 1996.

[8] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, pages 244–253, 1996.

[9] D. Neu. AST-basierte Erkennung von komplexen Refactorings. Diploma thesis (in German), University of Trier, Germany, 2009.

[10] C. K. Roy and J. R. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'2008)*, pages 153–162. IEEE Computer Society, 2008.

[11] 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.

[12] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 231–240. IEEE Computer Society, 2006.