EasyMerge - A New Tool for Code Clones Refactoring

Shengying Pan
School of Computer Science
University of Waterloo
s5pan@uwaterloo.ca

Haocheng Qin
School of Computer Science
University of Waterloo
h7qin@uwaterloo.ca

Yahui Chen
School of Computer Science
University of Waterloo
y556chen@uwaterloo.ca

ABSTRACT

Code clones are common in medium to large scale software projects. Oftentimes, unnecessary clones cause troubles to code base maintenance and code reusability. Over past decades, many techniques and approaches have been proposed to detect code clones. However, how to refactor clones is still a very challenging topic to software engineers. Even text-wise identical code clones can be semantically different when they are referring variables and calling functions outside. And the problem is more complex when scopes and dependencies are considered. Furthermore, not all clones can be refactored as they may be part of tests or needed to maintain dependencies across multiple libraries. As a result, we need adaptive clone refactoring tools that can locate unnecessary clones and alert possible implications in the procedure to help software engineers be more efficient and make less mistakes in the refactoring process.

In this paper, we introduce EasyMerge, a new tool to refactor code clones. EasyMerge is built on top of AST-based anti-unification clone detection algorithm and refactors software projects written in Python. It adds intelligence to code clone refactoring by categorizing clones and generating refactoring recommendations based on evaluation of external variable references and function calls. It copes with features of Python language and can deal with clone segments in functions, classes, and across different scopes. It ensures functionality-wise consistency before and after refactoring and provide warnings when potential refactoring could lead to unwanted complexity or cause readability issues.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Software Engineering, Recommendation System

Keywords

Software Engineering, Clone Detection, Code Refactoring, Recommendation System, Python

1. INTRODUCTION

In software development, it's very common seeing developers reuse code fragments by copying and pasting with or without minor adaptation. Moreover, for large scale projects, developers are often too lazy to browse existing source files so that they may rewrite similar or even identical functions which were already in the code base. As a result, software systems often contain sections of code that are very similar, called code clones.

Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical software system has been cloned [1] [9]. Many code clones in code bases are unnecessary duplications. Code duplication can be a significant drawback, leading to bad design, and increased probability of bug occurrence and propagation. As a result, it can significantly increase maintenance cost, and form a barrier for software evolution. By detecting, categorizing, and removing code clones, we can produce easier to understand, cleaner, and more reusable code.

Clone detection has been an avid research topic in the field of software engineering for decades. Fortunately, several automated techniques for detecting code clones have already been proposed. However, how to deal with detected clones, e.g. how to distinguish necessary clones from unnecessary ones and how to refactor code to remove unnecessary clones still remain a big problem in not only commercial but also academic domain. As a result, in this paper, we try to classify code clones and build a recommendation system called EasyMerge to help developers merge unnecessary clones on top of current state-of-the-art clone detection approach.

More specifically, we pick CloneDigger [4], an anti-unification duplicate code detection tool as our underlying clone detection approach. CloneDigger is one of the best available clone detection tools currently for its overall performance, coverage of multiple clone types, and availability. EasyMerge integrates CloneDigger as the pre-processing tool, analyze its output clone pairs, and recommend possible merges which can remove unnecessary clones without changing functionality of code base, creating reference conflicts, nor causing troubles to future code understanding and development.

The rest of the paper is structured as follows: we first go through the basics, background, and current state of clone detection and code clone refactoring in general. Then we introduce and discuss the fundamentals of CloneDigger and the anti-unification algorithm it is using to detect clones. Afterwards, we explain EasyMerge's work flow and underlying techniques. And at the end, we set up testing environment and discuss the experimental results of running EasyMerge against several open source projects of different scales.

2. BACKGROUND

2.1 Clone Detection

Roy, Cordy, and Koschke have done a great work [10] writing an overview paper explaining the basics of clone detection, and providing a complete comparison of essential strengths and weaknesses of both individual tools and techniques and alternative approaches in general. It gives us all the needed preliminaries to focus on clone refactoring rather than spending time working on the detection part. We begin with a basic introduction to clone detection terminology in Roy's paper.

Definition 1. (Code Fragment). A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple ($CF.FileName$, $CF.BeginLine$, $CF.EndLine$).

Definition 2. (Code Clone). A code fragment $CF2$ is a clone of another code fragment $CF1$ if they are similar by some given definition of similarity, that is, $f(CF1) = f(CF2)$ where f is the similarity function (see clone types below). Two fragments that are similar to each other form a clone pair $(CF1, CF2)$, and when many fragments are similar, they form a clone class or clone group.

Definition 3. (Clone Types). There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Type 1 to 3)[3] and functional (Type 4)[6, 8] similarities:

- **Type-1:** Identical code fragments except for variations in whitespace, layout and comments.
- **Type-2:** Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-3:** Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
- **Type-4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

There are many different techniques to detect code clones. In Roy's paper [10], they covered tools of textual approaches, lexical approaches, syntactic approaches, semantic approaches and hybrids. Different tools have advantages and disadvantages for different facets (usage, interaction, language, clone information, technical aspect, adjustment, processing, and evaluation) and behave quite differently for some scenarios. As clone detection is still an on-going research, it's hard to say one tool is the best at the moment. However, for the purpose of building EasyMerge, we want to pick a tool that is cross-platform, freely available, efficient, covering multiple clone types, and can handle different editing scenarios decently. Thus we chose CloneDigger [4] after a comprehensive comparison.

2.2 Clone Refactoring

On the other hand, for clone refactoring, most proposed solutions use some pre-defined metrics as the refactoring guidance to help users determine whether the code clones are suitable for refactoring. For example, Balazinska [2] proposed to use 21 metrics to measure the suitability for refactoring. Higo [7] developed ARIES tools, using the code detection tool CCFinder and environment analysis tool Gemini to find the clones can be refactored. And Schulze [11] proposed a new metric DIST (distance of the code clones). Unlike them, instead of calculating distance between code segments and only merging close clones, EasyMerge try to fix the differences and merge as many clones as possible without introducing extra complexity.

For the actual merging process, currently existing research mainly focus on using refactoring patterns [5], especially "Extract Method" and "Pull Up Method". "Extract Method" means that a fragment of source code is extracted and redefined as a new method. "Pull Up Method" means that the same methods defined in child classes are pulled up to its parent class. In general, they are both merging similar code blocks into one newly created function, and "Pull Up Method" only differs from "Extract Method" as it places the newly created function into the super class when code blocks are inside functions sharing the same parent class.

Unfortunately, code clones may be coupled with surrounding code, its inheritance tree, or other independent classes/code blocks. For instance:

3. CLONEDIGGER

4. EASYMERGE

5. EXPERIMENTAL RESULTS

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] B. Baker. On finding duplication and near-duplication in large software systems. *in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995*, pages 86–95, 1995.

[2] M. Balazinska, E. Merlo, and M. Dagenais. Advanced clone-analysis to support object-oriented system refactoring. *in: Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE 2000*, pages 98–107, 2000.

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

[4] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. *Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008*, pages 4–7, 2008.

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

[6] 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, 2008.

[7] 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: Research and Practice*, 20(6):435–461, 2008.

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

[9] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software systems. *in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008*, pages 81–90, 2008.

[10] 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, Special Issue on Program Comprehension, ICPC 2008*, 74(7):470–495, 2009.

[11] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. *in: Proceedings of the 2nd Workshop on Refactoring Tools, WRT 2008*, page 6, 2008.