JDeodorant: Clone Refactoring

Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, Zackary Valenta
Computer Science and Software Engineering
Concordia University, Montreal, Canada
{d_mazina, tsantalis}@cse.concordia.ca, {raphistein, zackary.valenta}@gmail.com

ABSTRACT

Code duplication is widely recognized as a potentially harmful code smell for the maintenance of software systems. In this demonstration, we present a tool, developed as part of the JDeodorant Eclipse plug-in, which offers cutting-edge features for the analysis and refactoring of clones found in Java projects. https://youtu.be/K_xAEqIEJ-4

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Refactoring, Code duplication, Refactorability analysis

1. INTRODUCTION

Duplicated code (also known as code clones) is considered as a potentially serious problem, which may negatively affect the maintainability and evolvability of a software system. Clone management [4] comprises all activities aiming at 1) preventing the creation of new clones in the system (*proactive*), 2) avoiding the negative impacts of clones that cannot be removed from the system for some valid reasons (*compensatory*), and 3) removing existing clones from the system (*corrective*).

However, the corrective aspect of clone management is still not adequately supported. The current tool support for the refactoring of clones is rather immature. For instance, the state-of-the-art IDEs, such as Eclipse and IntelliJ IDEA, allow the user to select a piece of code within a method, and give the option to replace all other occurrences of the same code with calls to the extracted method. However, this refactoring support is restricted only to duplication instances within the same file, and to duplicated code fragments with identical Abstract Syntax Tree (AST) structures, tolerating in some cases differences in variable identifiers. However, there is evidence that software clones can be dispersed among different files [6] (e.g., subclasses of the same inheritance hierarchy or unrelated classes), and they may have non-trivial variations (e.g., differences in method calls), reordered statements, or even gaps (i.e., added, deleted, or modified statements).

Several researchers attempted to improve the tool support for the refactoring of clones. Tairas and Gray [5] extended the Eclipse refactoring engine to enable the processing of more types of variations among duplicated code fragments, such as differences in field accesses, string literals, and method calls without arguments. Hotta et al. [1] proposed an approach to refactor clones with gaps located in different subclasses by introducing an instance of the Template Method design pattern. More recently, Meng et al. [2] proposed a technique for automated clone refactoring based on systematic edits (i.e., similar edits to different locations in the source code). Each one of the aforementioned approaches suffers from limitations, which restrict its applicability to clones with specific characteristics. For instance, [5] supports only clones located in the same file, [1] supports only clones located in subclasses of the same inheritance hierarchy, and [2] supports only clones containing variations in expressions inside statements, or variations in method calls covering entire statements (i.e., it does not support clones with gaps, or reordered statements).

In [6], we proposed a novel approach that takes as input any kind of clones (regardless of their relative location, or their variations), and examines whether they can be safely refactored in a way that preserves program behavior (i.e., without any side effects). The proposed approach has been implemented as an Eclipse plug-in, which is part of the JDeodorant[1] code smell detection and refactoring suite. In this demonstration, we present the key features offered by this tool, and explain how these features can help developers to understand better the differences between clones (clone comprehension and visualization), and if possible eliminate them (clone refactoring).

2. TOOL FEATURES

2.1 Clone Import and Group Analysis

Over the last years, several techniques and tools have been devised for the detection of clones. Each approach has its own advantages and disadvantages with respect to the types of clone variations being supported, the syntactic correctness (complete or incomplete abstract syntax trees) and granu-

[1]https://github.com/tsantalis/JDeodorant

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-4205-6/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2889160.2889168

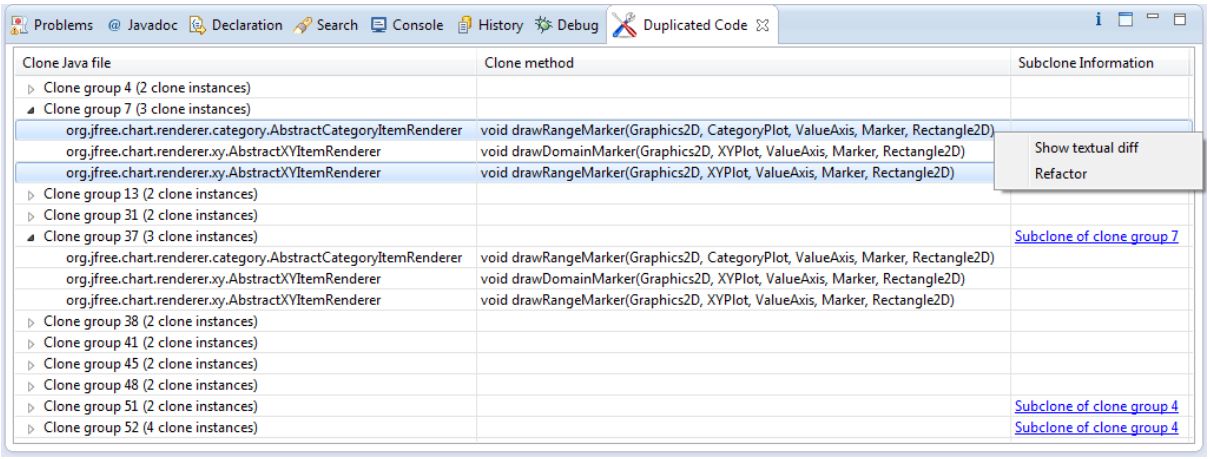

Figure 1: Presentation of the imported clone groups and clone instances.

larity (file/function-level) of the detected clones, and finally the efficiency or scalability of the detection process.

Our tool relies on external clone detection tools for finding duplicated code within a Java project, and currently supports the import of results from five popular tools, namely CCFinder, ConQAT, NiCad, Deckard and CloneDR. These tools report a clone instance based on the starting and ending lines or offsets of the duplicated code fragment inside a file. While importing the results, our tool automatically checks the syntactic correctness of the clone fragments, and fixes any discrepancies by removing incomplete statements and adding the missing closing brackets from incomplete blocks of code. Additionally, the tool filters out clone instances that extend beyond the body of a method (i.e., class-level clones), and clone groups that contain the same clone instances with another group (i.e., repeated clone groups).

The imported results are presented to the user in a tree-like view, as shown in Figure 1. At first level, each detected clone group appears with a unique ID and some basic information about the number of clone instances it contains, and whether it is a *subclone* of another group. Group A is a subclone of group B, if every clone instance in A is a subclone (i.e., a partial code fragment) of an instance in B. The subclone information appears as a link in the last column of the clone group table to help the user navigate between clone groups having a subclone relationship.

The user can further explore the clone instances inside a group of interest by expanding it. Double-clicking on a clone instance opens the corresponding Java file and highlights the duplicated code fragment in the Eclipse editor. The user can also select any pair of clone instances within a group and perform a textual comparison by clicking on "Show textual diff", or perform a refactorability analysis [6] by clicking on "Refactor" (more details about this feature will be given in Section 2.2).

After the application of a refactoring, the clone group table is automatically updated by disabling the refactored clone instances and the corresponding clone group, if it does not contain any other clone instances, as well as all groups being subclones of the refactored group. Additionally, the offsets of other clone instances belonging to the same refactored Java files are automatically updated. In this way, the user can continue with the inspection and refactoring of other clone groups without having to import new clone detection results from external tools.

2.2 Clone Refactorability Analysis

In [6], we presented a technique that takes as input a pair of duplicated code fragments or duplicated methods and applies three steps to determine if they can be safely refactored. In the first step, our approach tries to find the largest possible code fragments within the input clones having an identical nesting structure. The assumption is that two code fragments can be unified only if they have an identical nesting structure, and therefore each pair of matched nesting subtrees will be treated as a separate refactoring opportunity. Ideally, if the input clones have the same nesting structure, then the entire clone fragments will be treated as a single refactoring opportunity. In the second step, for each refactoring opportunity resulting from the previous step, our approach finds a mapping between the statements of the corresponding clone fragments that maximizes the number of mapped statements and minimizes the number of differences between the mapped statements. From this step, we take as output a set of mapped statements along with the differences that might exist between them, and a set of unmapped statements that could not be mapped to a statement from the other clone fragment (i.e., statements appearing in clone gaps). In the final step, the differences extracted in the previous step are examined against a set of preconditions [6] to determine whether they can be parameterized without changing the program behavior. The preconditions are rules that examine if the parameterization of a difference breaks existing data or control dependencies. A clone pair is considered refactorable if there are no precondition violations.

The outcome of the refactorability analysis is presented to the user as shown in Figure 2. The analyzed clone fragments appear as two side-by-side tree structures, where each pair of tree nodes in the same row represents a pair of mapped statements in the first and second clone fragment, respectively. The differences between the mapped statements are highlighted in yellow.

If there are multiple refactoring opportunities found in the input clones, the user can inspect each one of them separately by selecting the corresponding option in the "Select Refactoring Opportunity" combo box (Figure 2, point 1).

By hovering over a pair of mapped statements a tooltip appears (Figure 2, point 2) providing semantic information about the type of each difference based on the program elements (e.g., variables, method calls, literals, class instan-

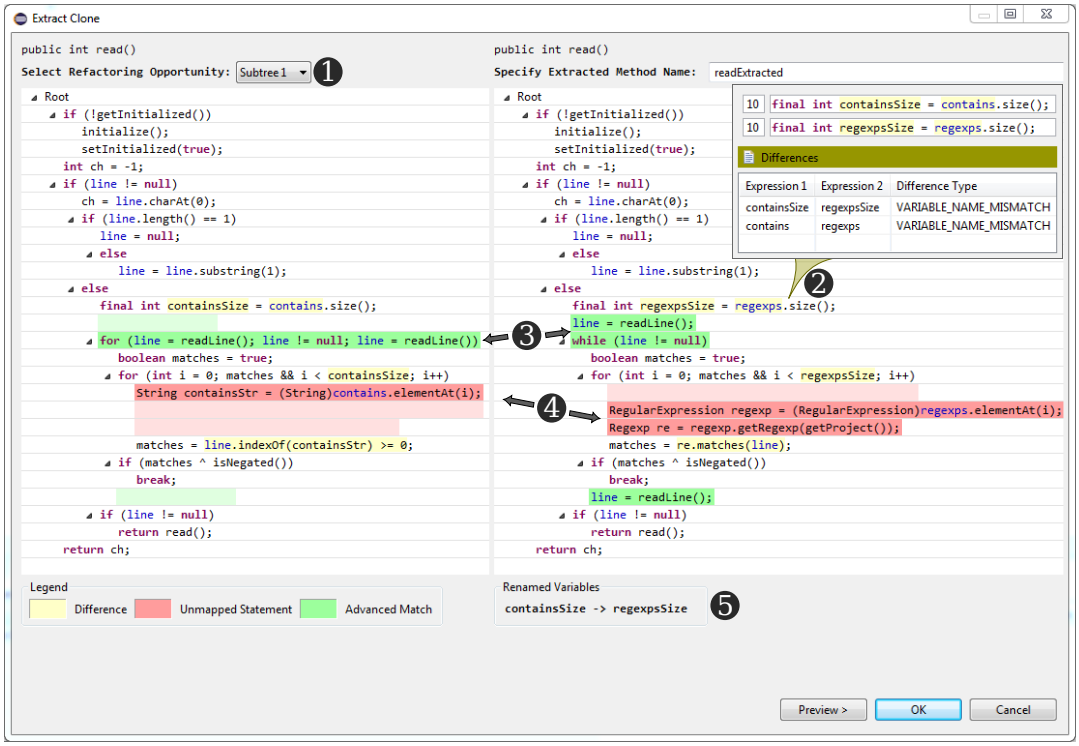

Figure 2: Clone pair visualization and refactorability analysis.

tiations) appearing in the difference. Currently, our tool supports 20 difference types, including some more advanced ones, such as the replacement of a field access with the corresponding getter method call, and the replacement of a field assignment with the corresponding setter method call, which are both semantically equivalent differences (i.e., their AST structure is different, but the effect from the execution of these expressions on program behavior is the same). The tooltip may also include information about precondition violations, if the expressions appearing in the differences cannot be safely parameterized. Not all differences need parameterization. Semantically equivalent differences, and renamed variables are not examined against preconditions, since they should not be parameterized. Our tool automatically detects the local variables that have been consistently renamed between the clone fragments, as shown in Figure 2, point 5.

Our tool also supports the matching of semantically equivalent statements [3]. As we can observe in Figure 2, point 3, the first clone fragment contains a `for` loop, where the initializer expression `line = readLine()` and the updater expression `line = readLine()` are embedded in the loop declaration, while the second clone fragment contains a `while` loop, where the same initializer appears in a separate statement right before the `while` loop, and the same updater appears as the last statement inside the body of the `while` loop. Although these two loops are different in terms of their AST structure, they perform exactly the same iteration (i.e., they are semantically equivalent loops). The statements being part of semantically equivalent statements are not considered as unmapped statements, and are highlighted in green indicating an *advanced match*. Our tool currently supports the matching of semantically equivalent loop variants (enhanced `for`, traditional `for`, `while`, and `do-while` loops), the matching of an `if-then-else` statement with a semantically equivalent statement using the conditional expression `condition ? true_expr : false_expr`, and the matching of a statement declaring and initializing a variable with two separate statements (one declaring a variable and another one assigning a value to the same variable).

Finally, our tool handles unmapped statements (highlighted in red), i.e., statements that do not have a matching statement in the other clone fragment (Figure 2, point 4), in two possible ways. First, it determines if they can be safely moved before or after the execution of the extracted method containing the mapped statements without breaking existing data or control dependencies. If this is not possible, it examines if the unmapped statements located at the same nesting level in both clone fragments can be abstracted into methods with identical signatures (i.e., same input parameter types, return type, and thrown exception types), and therefore replaced with identical method calls.

2.3 Clone Refactoring Support

If there are no precondition violations the user can proceed with the refactoring of the clone fragments by clicking on the "Preview" or "OK" buttons (Figure 2). The former option will first show a preview of the changes that will be applied by the refactoring, while the latter option will directly apply the refactoring on the source code. The tool automatically determines the appropriate refactoring that should be applied based on the relative location of the clones. In general, our tool supports three refactoring scenarios.

Extract Method is applied when the clones are located in the same method or in different methods of the same class. The duplicated code is extracted in a private method within the same class. If there exist unmapped statements that can be abstracted into methods with identical signatures, our tool introduces a functional interface as a parameter

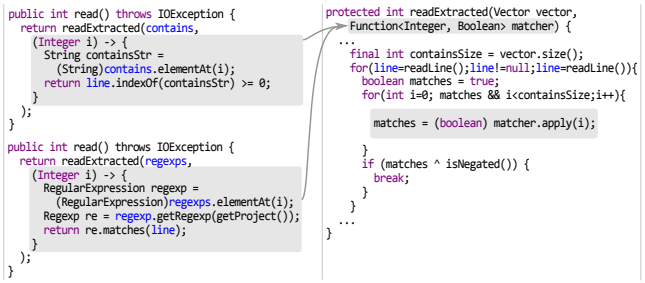

Figure 3: Refactoring using Lambda expressions.

to the extracted method. The original methods containing the clones pass a Lambda expression as an argument for the introduced parameter when calling the extracted method. Each Lambda expression contains inside its body the unmapped statements of the corresponding clone. The clones in Figure 2 are refactored as shown in Figure 3 using Lambda expressions. In this case, our tool used the predefined `java.util.function.Function` functional interface, since both code fragments inside the Lambda expressions require one input argument (`int i`) and return one output result (`boolean matches`). If there is no suitable functional interface type predefined by Java, our tool defines a custom functional interface type using the `@FunctionalInterface` annotation.

Extract and Pull Up Method is applied when the clones are located in methods belonging to different subclasses within the same inheritance hierarchy. If the common superclass is inherited by only these two subclasses, then the duplicated code is placed in the superclass. If there are other subclasses extending the common superclass, then the duplicated code is placed in a newly created intermediate class, which extends the common superclass and is inherited by the subclasses containing the clones. In this way, we ensure that the other subclasses will not inherit functionality that they do not need. Finally, if the clones are accessing fields and methods declared in the subclasses that have an identical AST structure (i.e., called methods being exact duplicates, or containing only renamed variables, and fields having the same type, name, and initialization), then these fields and methods are also pulled up in the superclass where the duplicated code is placed. On the other hand, if the clones are accessing methods declared in the subclasses that have an identical signature but different body, then an abstract method with the same signature is added in the superclass where the duplicated code is placed. Therefore, after the application of the refactoring, the unified code calls the newly introduced abstract method, which is overridden in the two subclasses originally containing the clones. In that case, the refactoring is essentially introducing an instance of the *Template Method* design pattern.

Extract Utility Method is applied when the clones are located in methods of unrelated classes, and they do not access any instance variables or methods. In this case, the duplicated code is extracted into a static method and placed in a newly created utility class.

3. VALIDATION

In [6], we evaluated the correctness of our clone refactoring engine by executing refactorings on 610 clone pairs found in project *JFreeChart* (version 1.0.10) that were covered by unit tests and were assessed as refactorable by our approach. After each refactoring, we were running all unit tests of the project to check whether the refactoring introduced any test failures. Only 13 refactorings led to test failures, which were all related to deserialization failures due to commonly accessed fields being pulled up from the subclasses to the superclass. We fixed our refactoring implementation to avoid pulling up non-transient fields from Serializable classes.

In order to automate the testing of our refactoring engine, we developed a separate project[2] that executes the entire workflow we applied in our previous experiment without any human intervention. We are currently using this project to test more extensively the new features added in our clone refactoring engine, such as the use of Lambda expressions.

4. CONCLUSIONS

In this paper, we presented a tool, implemented as an Eclipse plug-in, that can help developers to refactor non-trivial clones in Java projects. More specifically, it examines the *refactorability* of a clone pair selected by the user, and presents the results of the analysis in an interactive visualization highlighting the differences that can or cannot be safely parameterized. If there are no precondition violations, the tool can automatically apply the appropriate refactoring based on the relative location of the clones.

As future work, we plan to support multi-clone refactoring (i.e., refactoring of clone groups containing more than two clone instances), and also develop a refactoring cost analysis technique to guide the developers when there are multiple alternative ways to refactor a group of clones (i.e., refactor possible sub-groups of clones having less differences, or apply a single refactoring for all clones in the group). In addition, we will further investigate potential applications of Lambda expressions in the refactoring of clones with more advanced differences.

5. ACKNOWLEDGMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Faculty of Engineering and Computer Science at Concordia University.

6. REFERENCES

[1] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012.

[2] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 392–402, 2015.

[3] G. Qiao. Mining and analysis of control structure variant clones. Master's thesis, Concordia University, April 2015.

[4] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 18–33, 2014.

[5] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, Dec. 2012.

[6] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, Nov 2015.

[2]https://github.com/tsantalis/jdeodorant-commandline