Clone Refactoring with Lambda Expressions

Nikolaos Tsantalis, Davood Mazinanian, Shahriar Rostami
Computer Science and Software Engineering
Concordia University, Montreal, Canada
tsantalis, d_mazina, s_rostam@encs.concordia.ca

***Abstract*—Lambda expressions have been introduced in Java 8 to support functional programming and enable *behavior parameterization* by passing functions as parameters to methods. The majority of software clones (duplicated code) are known to have behavioral differences (i.e., Type-2 and Type-3 clones). However, to the best of our knowledge, there is no previous work to investigate the utility of Lambda expressions for parameterizing such behavioral differences in clones. In this paper, we propose a technique that examines the applicability of Lambda expressions for the refactoring of clones with behavioral differences. Moreover, we empirically investigate the applicability and characteristics of the Lambda expressions introduced to refactor a large dataset of clones. Our findings show that Lambda expressions enable the refactoring of a significant portion of clones that could not be refactored by any other means.**

***Keywords*-Refactoring; Code duplication; Lambda expressions**

I. Introduction

The refactoring of software clones can help to reduce the size of the code-base [1], [2], [3], avoid software defects due to inconsistent clone updates [4], [5], and minimize the negative effect of duplicated code on maintenance effort and cost [6], [7]. A recent study involving GitHub contributors has shown that they are seriously concerned about code duplication [8]. Among the top reasons they applied Extract Method refactorings was to reuse existing code (i.e., avoid duplication) and remove already existing duplication [8]. However, clone refactoring is not trivial, because the majority of clones are not identical code fragments, i.e., developers tend to modify the copied code fragments to adjust them to another context, or to different requirements. As a matter of fact, there are more copy/pasted fragments in which minor to significant editing activities took place than identical clones in repositories [9].

Motivation: The current state-of-the-art clone refactoring techniques merge clones by introducing parameters in the extracted method for the expressions being different among the merged clone fragments [10], [11]. After the refactoring, the parameterized expressions are passed as arguments to the extracted method calls, and thus these expressions are evaluated before the execution of the extracted duplicated code. However, this parameterization approach could cause a change in the program behavior, especially when the parameterized expressions are method calls and object creations, due to *side-effects* on the state of the objects involved in these expressions.

In a previous work [12], we developed a *refactorability analysis* approach that can automatically determine whether the parameterization of the expressions being different between two clones could change the program behavior. The proposed technique performs a sophisticated static source code analysis to extract intra- and inter-procedural data dependencies between the statements inside the clone fragments to be merged, and examines whether these data dependencies would be preserved after the refactoring of the clones. If the parameterization of the clone differences breaks at least one of the originally existing dependencies, then the refactoring is not behavior-preserving, and thus the clones are assessed as *non-refactorable*. Using this refactorability analysis tool, we performed a large-scale empirical study on clones detected by four different clone detectors in nine open-source projects [12], where we found out that around 94% of the detected clones belong to either Type-2 (i.e., structurally/syntactically identical code fragments with variations in identifier names, literal values, and types), or Type-3 (i.e., copied fragments with statements changed, added or removed in addition to Type-2 differences) categories. Only 14% of these Type-2 and Type-3 clones were assessed as safely refactorable using the standard parameterization approach. Among the top reasons preventing these clones from being refactorable are (1) the presence of different expressions within matched statements whose parameterization would change the program behavior, (2) the presence of unmapped statements within the clone fragments, also known as clone gaps, and (3) the presence of different method calls having a `void` return type (Java does not allow parameters of `void` type).

Java 8 introduced Lambda expressions as a means to support functional programming. A Lambda expression in Java 8, is an anonymous function containing either a block of statements or a single expression that can be passed as a parameter to a method, thus achieving *behavior parameterization* [13]. One of the most important features of a Lambda expression is that it executes *in the context* of its invocation (i.e., inside the method in which it is passed as an argument), and thus it can use the values of the variables that are defined in that context. This makes Lambda expressions ideal for the parameterization of the three behavioral differences discussed in the previous paragraph preventing the refactoring of Type-2 and Type-3 clones, since with Lambda expressions we can parameterize behavior that is different among the clone fragments without changing the original execution order.

Contributions: In this paper, we propose a technique and a tool that utilizes Lambda expressions to enable the refactoring of Type-2 and Type-3 clones having behavioral differences that cannot be parameterized with regular parameters. In

particular, we developed an algorithm to examine whether it is possible to introduce Lambda expressions for the unmapped statements (i.e., clone gaps) and expressions assessed as non-parameterizable by our previously proposed refactorability analysis approach [12]. To the best of our knowledge, this is the first work to investigate the use of Lambda expressions as a means to refactor software clones with behavioral differences. To evaluate the efficacy of the proposed technique, we first assess its correctness by refactoring and testing 12,602 clone pairs covered by unit tests, and reporting the compilation errors and test failures that occurred. Next, we apply the proposed technique on a dataset of 46,765 Type-2 and Type-3 clone pairs, and report the percentage of the clone pairs whose refactoring was enabled using Lambda expressions, and the characteristics of the introduced Lambda expressions. Finally, we make publicly available the dataset of the examined clones to facilitate future research on clone refactoring [14].

II. Motivating Examples

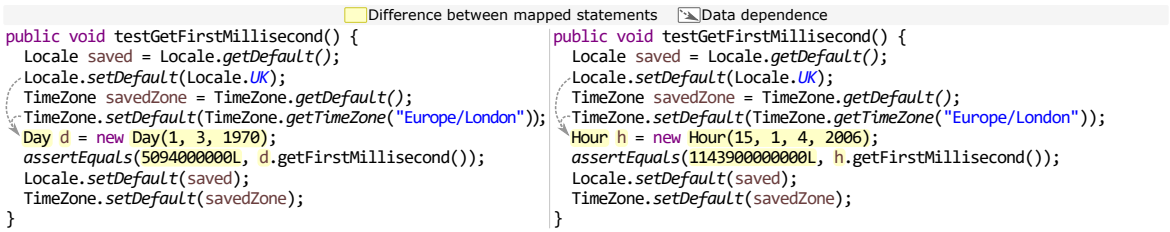
In this section, we use two examples of clones from open-source projects to motivate the reader about the usefulness of Lambda expressions in the refactoring of clones.

Example 1: Figure 1a shows two duplicated methods found in the test code of the JFreeChart project (version 1.0.10) that create a mock object of `Day` and `Hour` type, respectively, to test its method `getFirstMillisecond()`. Both test methods initially save the current `Locale` and `TimeZone` of the system into temporary variables `saved` and `savedZone`, respectively, and set the `Locale` to `UK` and the `TimeZone` to `Europe/London`, before instantiating the mock object. After testing an assertion with `assertEquals()`, the `Locale` and `TimeZone` of the system is reset to the original values saved in the temporary variables. The reset of the `Locale` and `TimeZone` is necessary to avoid affecting other unit tests whose execution follows. The duplicated test methods are Type-2 clones and have only two differences (highlighted in yellow), namely (1) the type of the mock object being instantiated, and (2) the literal of `long` type passed as the first argument in `assertEquals()`. It should be emphasized that the same test method is duplicated 9 times in the JFreeChart test code-base, each time testing a different subclass type of the `RegularTimePeriod` superclass type, namely `Day`, `Hour`, `Millisecond`, `Minute`, `Month`, `Quarter`, `Second`, `Week`, and `Year`, with exactly the same differences.

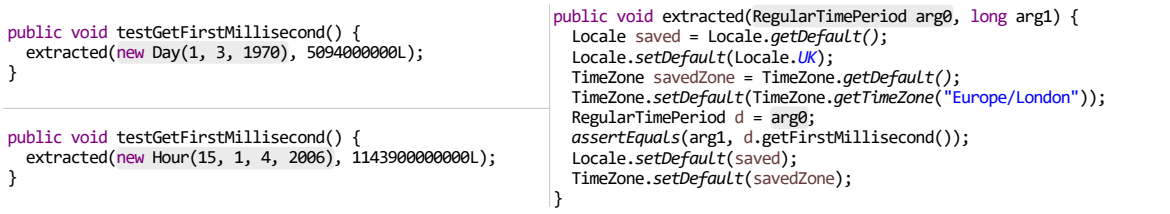
If we assume that the developers of these test methods decide to test the mock objects with another `Locale` and/or `TimeZone`, they would have to apply the same changes 9 times, which requires significant effort and time. Therefore, we can argue that this is an interesting case for refactoring, assuming that future test maintenance activities are probable. In fact there is evidence that developers read, write, and maintain test code [15], [16], [17], and thus clone refactoring could facilitate test code reuse, understandability, and maintainability.

To refactor these test methods there are two alternative approaches. The first approach is to extract the identical functionality (i.e., the first four statements constitute the common *set up* logic of the tests, while the last two statements constitute the common *tear down* logic of the tests) into separate methods. However, this approach might not be always feasible (e.g., the *set up* logic of the tests should return two variables, namely `saved` and `savedZone`, and Java allows methods to return at most one value), and does not eliminate completely the duplication (e.g., the two statements initializing the mock objects and testing the assertion will remain duplicated). The second approach is to parameterize the differences in the test methods. Figure 1b shows the parameterization of the test methods using two regular parameters. Note that in the extracted method the types of the mock objects have been generalized to the common superclass type `RegularTimePeriod`, and method `getFirstMillisecond()` is polymorphically called. This parameterization is changing the program behavior and makes the refactored tests fail. The reason is that the mock objects are instantiated before the execution of the extracted method, when they are passed as arguments to the extracted method, and thus their instantiation is not using the appropriate `Locale` and `TimeZone` configured in the *set up* logic of the tests. Through inter-procedural program dependence analysis, we can see that `Day` and `Hour` constructors are using the static variables `Locale.defaultLocale` and `TimeZone.defaultTimeZone`, defined by statements `Locale.setDefault()` and `TimeZone.setDefault()`, respectively, in the *set up* logic, and thus there are inter-procedural data dependencies from these statements to the constructor invocations. Figure 1c shows the parameterization of the test methods using Lambda expressions to pass the two different class instantiations as arguments to the extracted method. Note that the type of the first parameter in the extracted method is the `java.util.function.Supplier` functional interface (more details about functional interfaces will be given in Section III-B), and the Lambda expressions passed as arguments are executed by calling the `get()` method over the `Supplier` parameter `arg0`. This parameterization is preserving the program behavior and makes the refactored tests pass, because the mock objects are instantiated in exactly the same execution point as before the refactoring.

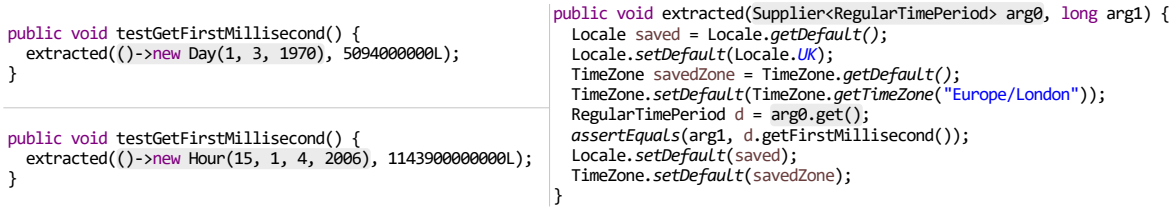
Example 2: Figure 2a shows two duplicated methods found in the Apache Ant project (version 1.7.0) that belong to classes `LineContains` and `LineContainsRegExp`, respectively. Both methods implement a similar logic, i.e., they read a stream line-by-line (the current line is saved in the `line` field), and the first method (left side of Figure 2a) examines if `line` contains one of the words inside the `contains` vector, while the second method (right side of Figure 2a) examines if `line` matches with one of the regular expressions inside the `regexps` vector. The duplicated methods are Type-3 clones and have two main differences, namely (1) the types of the elements inside the vectors (`contains` is a vector of `String` objects, while `regexps` is a vector of `RegularExpression` objects), and (2) the logic behind the matching of `line` as indicated by the unmapped statements (highlighted in red) inside the inner `for` loop.



(a) A pair of clones found in the tests of the JFreeChart open-source project



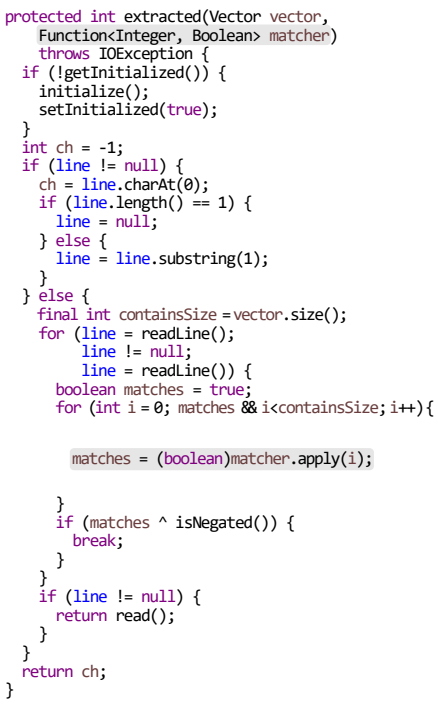
(b) Parameterization with regular parameters changing the program behavior



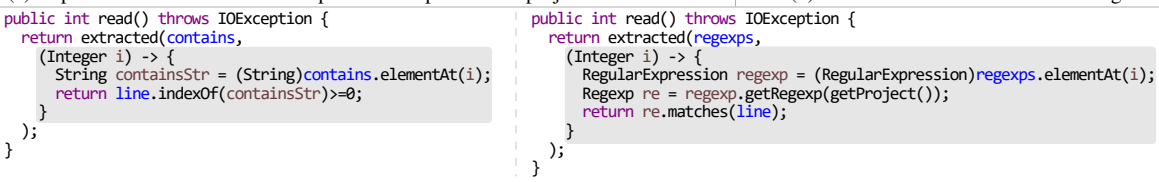
(c) Parameterization with Lambda expressions preserving the program behavior

Fig. 1. Using Lambda expressions to enable the refactoring of Type-2 clones.

(a) A pair of clones found in the Apache Ant open-source project



(b) Extracted method after refactoring



(c) Clones after refactoring with Lambda expressions

Fig. 2. Using Lambda expressions to enable the refactoring of Type-3 clones.

To refactor these methods one could first try to convert them into Type-2 clones by moving the unmapped statements before or after the common code. In our previous work [12], we observed some typical patterns of Type-3 clones that can be easily converted into Type-2 clones by inlining temporary variables that appear in only one of the clones, or by moving statements declaring a variable at a different nesting level. However, in the example of Figure 2a it is not possible to move the unmapped statements, because they depend on the value of variable `i` that increments after each iteration of the inner `for` loop (i.e., there is a data dependence from the `for` loop, which declares and increments variable `i` to the first unmapped statement that uses `i` to retrieve the next element from the vector). In turn, the statements that follow inside the inner `for` depend on the variable declared by the first unmapped statement (i.e., `containsStr` and `regexp`, respectively), as shown by the data dependencies. Therefore, we need a parameterization approach that preserves the execution of the unmapped statements inside the inner `for` loop.

Figure 2c shows the parameterization of the methods using Lambda expressions to pass the two different `line` matching behaviors as arguments to the extracted method (Figure 2b). The Lambda expressions (Figure 2c) take as input parameter `i` and return the matching result as a `boolean`. The type of the second parameter in the extracted method is the Java predefined `java.util.function.Function` functional interface, and the Lambda expressions passed as arguments are executed by calling the `apply(i)` method through the `Function` parameter `matcher`. This example demonstrates that Lambda expressions are executed in the context of their invocation, and thus they can use the values of the variables that are defined in that context (i.e., variable `i`).

Alternatively, these methods could be parameterized by applying the Template Method design pattern [18], i.e., introducing a new abstract method in the common superclass where the extracted code (i.e., template) will be pulled up, replacing the uncommon code with a call to the abstract method, and finally overriding the abstract method in the subclasses containing the original clones with the corresponding behavior. However, it is not always possible to apply the *Form Template Method* refactoring [19], because (a) the clones might exist in the same class or in classes not having a common superclass, (b) it might not be always possible to make the common superclass abstract, if the project contains objects instantiated from the common superclass, (c) the common superclass might have other subclasses that must provide an implementation for the newly introduced abstract method without necessarily needing its functionality, and (d) it might not be always possible to introduce a new intermediate common superclass (i.e., extract superclass) to host the template method, if the subclasses containing the original clones belong to different levels of the inheritance hierarchy. On the other hand, there are no such restrictions for the application of Lambda expressions.

With these examples, it becomes clear that *Lambda expressions can enable the refactoring of clones whose parameterization is not feasible or behavior-preserving by other means.*

III. Approach

A. Input

The input of our approach is the output of the clone refactorability analysis tool from our prior research [12], which takes as input a pair of clone fragments, and returns a mapping between the statements of the clone fragments, along with the differences (i.e., pairs of expressions being different) inside the mapped statements, and a list of unmapped statements (i.e., statements that could not be matched with any statement from the other clone fragment due to incompatible AST structure) for each clone fragment. In a nutshell, the solution proposed in our prior research [12] applies a *maximum common subgraph* algorithm [20] on the Program Dependence Graphs [21] of the clones in a divide-and-conquer fashion by breaking the initial statement mapping problem to smaller subproblems based on the control dependence structure of the clones. In this way, it explores the search space of alternative mapping solutions in order to maximize the number of mapped statements and minimize the number of differences between them. Additionally, it can properly handle clones in which matching statements have been reordered, in contrast to token-based differencing approaches, such as *MCIDiff* [22], that do not consider the syntactic structure of the program.

A visual representation of the extracted information is shown in Figure 2a, where the differences between the mapped statements are highlighted in yellow and the unmapped statements are highlighted in red. In the final phase, a list of preconditions is examined to determine whether the detected differences can be safely parameterized with regular parameters, and whether the unmapped statements can be safely moved before or after the common code. If the examined preconditions fail, the corresponding differences or unmapped statements are associated with a *precondition violation*, and the clones are assessed as non-refactorable.

Our current work focuses on the differences and unmapped statements associated with precondition violations and proposes a method to examine whether their parameterization with Lambda expressions is feasible.

B. Lambda Unification

Since Lambda expressions are anonymous functions, the unification of two code fragments into a common function (or a functional interface following the Java terminology) requires that these code fragments can be abstracted to two functions with an identical signature. More specifically, there are three conditions that should be met regarding the signature:

1) The code fragments should return at most one variable of the same type, or different types that can be generalized to a common type.
2) The code fragments should require the same input parameter types from the extracted duplicated code.
3) The code fragments should throw the same exception types.

Our approach can be divided into 4 parts:

1. Handling of statement gaps:

A statement gap consists of two sets of consecutive unmapped statements S_i and S_j, which are nested at the same

level. To determine whether a statement gap can be parameterized in the form of Lambda expressions, we propose Algorithm 1. In a nutshell, the proposed algorithm is a recursive function that can expand backwards and/or forwards the original sets of statements (if necessary), so that the final sets of statements require the same input parameters and produce the same output. Additionally, when the aforementioned conditions are finally met, it also examines a list of preconditions to ensure that the introduction of the Lambda expressions is feasible and behavior-preserving.

Algorithm 1 Recursive function determining if a statement gap can be parameterized in the form of Lambda expressions

Input: A statement-level gap [S_i, S_j]
Output: `true` if Lambda refactoring is feasible, `false` otherwise, updated sets [S'_i, S'_j] if expansion occurred

```
1:  function LambdaRefactorable(S_i, S_j)
2:      UV_i = usedVariables(S_i)
3:      UV_j = usedVariables(S_j)
4:      if commonParameters(UV_i, UV_j) then
5:          RV_i = returnedVariables(S_i)
6:          RV_j = returnedVariables(S_j)
7:          if validReturnType(RV_i, RV_j) then
8:              if checkPreconditions(S_i, S_j) then
9:                  return true
10:             end if
11:         else if [S'_i, S'_j] = forward(S_i, S_j) ≠ [S_i, S_j] then
12:             return LambdaRefactorable(S'_i, S'_j)
13:         end if
14:     else if [S'_i, S'_j] = backward(S_i, S_j) ≠ [S_i, S_j] then
15:         return LambdaRefactorable(S'_i, S'_j)
16:     end if
17:     return false
18: end function
```

Function *usedVariables* (lines **2**, **3**) takes as input a set of statements and returns the set of variables that are collectively used in the input statements. Function *returnedVariables* (lines **5**, **6**) takes as input a set of statements and returns the set of variables that are modified in the input statements and are used by code executed after the input statements. Function *commonParameters* (line **4**) takes as input two sets of variables and examines if the types of the corresponding variables are the same. It should be noted that the input sets do not need to have the same cardinality, as long as one of them is a *proper* (or *strict*) subset of the other, and the additional variables of the superset are declared before the clone fragment to be extracted (e.g., parameters of the method containing the clone fragment). This relaxation was necessary in order to deal with *overloaded* method calls accepting a different number of arguments. Function *validReturnType* (line **7**) takes as input two sets of variables and examines if the cardinality of both sets is equal to zero or one (i.e., one variable can be returned at most). When the cardinality of both sets is equal to one, it additionally examines if the types of the variables are the same, or if they are sub-types of a common superclass.

Function *checkPreconditions* (line **8**) takes as input two sets of statements and examines the following list of preconditions:

1) The statements contained in S_i and S_j should collectively throw the same exception types.
2) Sets S_i and S_j should not include branching statements (`break`, `continue`) without including the corresponding target `switch` or loop statement.
3) Sets S_i and S_j should not include conditional `return` statements (i.e., `return` without an expression causing a direct exit from the method containing the clone).
4) If sets S_i and S_j contain `return` statements with an expression returning a value, then all possible execution flows should end with a `return` statement.
5) If sets S_i and S_j contain statements using variables declared in the code remaining in the original methods after refactoring, then these variables should be *final* or *effectively final* (i.e., a variable or parameter whose value is never changed after it is initialized [23]). This condition applies only to local variables and parameters of the methods containing the clone fragments, but not to accessed fields, and it is required because Java handles Lambda expressions as single-method *anonymous classes* [24].
6) Sets S_i and S_j should not include all statements of the original clone fragments. This condition is necessary to avoid cases where the expansion might cover the entire clone fragments.

Finally, function *backward* (line **14**) adds the statement executed right before the first statement in S_i and S_j, respectively, as long as this statement is nested at the same level. Function *forward* (line **11**) adds the statement executed right after the last statement in S_i and S_j, respectively, as long as this statement is nested at the same level and contains differences, which cannot be parameterized with regular parameters.

2. Handling of expression differences:

An expression difference consists of two expressions found in two mapped statements of the clone fragments, which may have a different syntactic structure (i.e., different types of AST nodes), but are evaluated to the same type or types having a common superclass. For example, in the clones shown in Figure 2a mapped statements 15 and 16 from each clone, contain an expression difference highlighted in yellow. The expressions have a different AST type (on the left clone fragment we have an infix expression `line.indexOf(containsStr) >= 0`, while on the right one we have a method invocation expression `re.matches(line)`), but both of them are evaluated to type `boolean`. As a result, the condition *validReturnType* is met by default, and the only remaining conditions that should be examined to determine if the two expressions can be parameterized as Lambdas is *commonParameters* and *checkPreconditions* (only preconditions 1 and 5 are applicable).

3. Merging of overlapping gaps and expression differences:

After all statement gaps and expression differences are processed, we apply a post-processing step to determine if some of them can be merged in order to minimize the

number of Lambda expressions that should be introduced. More specifically, we apply three merging operations:

1) If the statements corresponding to an expression difference are included inside an expanded statement gap, then only the statement gap is parameterized.
2) If a statement gap is subsumed by another one (i.e., all its statements are included within a larger statement gap), then only the larger one is parameterized.
3) If two statement gaps have an overlap (i.e., they share some common statements, but do not subsume one another), then they are merged into a single statement gap. If the merged statement gap is refactorable according to Algorithm 1, then the original overlapping statement gaps are not parameterized, and only the merged one is parameterized.

4. Creating appropriate functional interfaces:

For each one of the final statement gaps and expression differences an appropriate functional interface should be added as a parameter to the extracted method. Java 8 provides a list of predefined functional interfaces in the `java.util.function` package, which are used whenever it is possible:

1) `Function<T,R>`: The Lambda expressions take as input one argument (T) and return a result (R).
2) `Supplier<T>`: The Lambda expressions take no input arguments and return a result (T).
3) `Consumer<T>`: The Lambda expressions take as input one argument (T) and return no result.

All aforementioned functional interface types do not support exception throwing. Therefore, when the Lambda expressions take as input two or more arguments and/or throw one or more exceptions, a custom functional interface needs to be introduced within the same class that the extracted method will be placed in:

```
@FunctionalInterface
interface CustomInterface {
  T0 apply(T1 arg1, T2 arg2, ...) throws E1, E2, ...;
}
```

C. Running Example

In this subsection, we will demonstrate the application of Algorithm 1 on the motivating example we used in Section II, shown in Figure 2a. The initial input to the algorithm is the sets of statements $S_i = \{14\}$ and $S_j = \{14, 15\}$. Both sets of statements are nested at the same level under the mapped `for` loops (with IDs 13 and 13, respectively). The sets of used variables are $\text{UV}_i = \{\texttt{int i}\}$ and $\text{UV}_j = \{\texttt{int i}\}$, and thus the function *commonParameters* returns `true`, and there is no need for a backward expansion of the sets S_i and S_j. However, the sets of returned variables are $\text{RV}_i = \{\texttt{String containsStr}\}$ and $\text{RV}_j = \{\texttt{Regexp re}\}$, and thus the function *validReturnType* returns `false`, because the types of the variables differ and cannot be generalized to a common super-type. Therefore, the function *forward* is executed and updates the original input statements to $S'_i = \{14, 15\}$ and $S'_j = \{14, 15, 16\}$. After the forward expansion, the sets of used variables UV_i and UV_j remain the same, while the sets of returned variables are becoming $\text{RV}_i = \{\texttt{boolean matches}\}$ and $\text{RV}_j = \{\texttt{boolean matches}\}$. As a result, now the sets S'_i and S'_j contain statements that require the same input parameters and produce the same output, and thus no further expansion is needed.

IV. Evaluation

To assess the correctness and applicability of the proposed technique for enabling the refactoring of clones with non-trivial behavioral differences, we designed a study aiming to answer the following research questions:

RQ1: Does the proposed approach refactor clones without introducing compile errors or changing program behavior?

RQ2: What portion of the clones that cannot be parameterized with regular parameters, becomes refactorable with Lambda expressions?

RQ3: What are the characteristics of the introduced Lambda expressions?

A. Experiment Setup

To investigate our research questions we created a large and diverse dataset of clones extracted from 9 open source projects using 4 different clone detectors, consisting of 46,765 clone pairs having statement gaps (Type-3) and/or expression differences (Type-2) that were assessed as *non-refactorable* using the standard parameterization approach [12].

Subject Selection: To avoid bias in the selection of projects and enable the comparison of our results with previous studies, we adopted the systems used in the studies conducted by Tairas and Gray [10] and Tsantalis et al. [12], shown in Table I.

TABLE I
EXAMINED PROJECTS

Project	Domain	Age†	KLoC
Apache Ant 1.7.0	Java application build tool	6½	67
Columba 1.4	email client	1½	75
EMF 2.4.1	modeling framework	5½	118
JMeter 2.3.2	server performance testing	7¼	54
JEdit 4.2	text editor	5	51
JFreeChart 1.0.10	chart library	7½	76
JRuby 1.4.0	programming language	3½	101
Hibernate 3.3.2	Java persistence framework	7½	209
SQuirreL SQL 3.0.3	universal SQL client	8	141

† years of development from the initial release to the examined release

The dataset includes projects from 9 different application domains, having a different development history ranging from 2 to 8 years. We assume that these two variation points affect the variability of the clones present in these systems, and thus the generalizability of our findings.

Clone Detector Selection: We used 4 popular clone detection tools, namely CCFinder [25], Deckard [26], CloneDR [27], and NiCad [28]. The selection criteria, tool descriptions, and configuration settings are detailed in [12]. We also kept in the dataset only one instance of the clones reported by multiple tools to avoid having duplicate data points.

Data Collection: Figure 3 shows the process we followed for collecting the data required to investigate our questions.

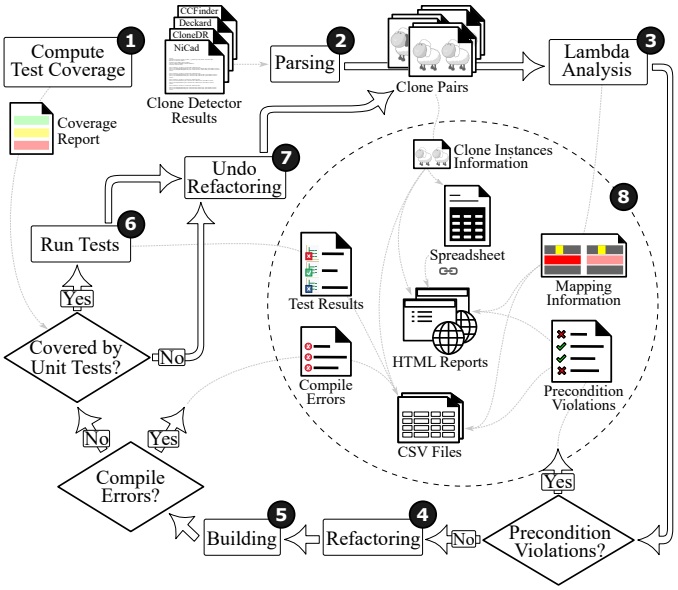


Fig. 3. The workflow applied for the collection of the experimental data.

We have fully automated the data collection process, by developing an Eclipse command-line application [29] that:

1) Computes the unit test code coverage for the project under analysis using the JaCoCo library [30].
2) Parses the output of clone detection tools and determines if the clones are fully, partially, or not covered by unit tests.
3) Executes the proposed technique implemented as part of the JDeodorant Eclipse plug-in [31] in headless mode for each pair of clones reported by a clone detection tool.
4) Applies the appropriate refactoring for each pair of clones assessed as Lambda-parameterizable by the proposed technique, based on the relative location of the clones.
5) Builds the project after each refactoring and reports any compilation errors.
6) Executes the test suite of the project for each refactored pair of clones that is covered by unit tests, compares the test results after refactoring with those before refactoring, and reports any differences.
7) Restores the project in its original state after each refactoring, by undoing the applied refactoring and rebuilding the project.
8) Stores the collected information in CSV files (used for statistical analysis), HTML reports (containing statement mapping information and precondition violations for a pair of clones), and a spreadsheet (containing information about all clones examined in the project and links to the reports).

B. Correctness of Lambda Refactorings (RQ1)

Motivation: The goal of RQ1 is to ensure that the proposed approach can safely refactor clones using Lambda expressions without introducing compilation errors and changing program behavior. This kind of evaluation is essential to strengthen the validity of the collected empirical data that will be used to answer RQ2 and RQ3. Our hypothesis is that if our approach assesses the behavioral differences appearing between a pair of clone fragments as Lambda-parameterizable, then it should be possible to refactor these clones without causing any compilation errors, and all tests of the project should pass after the application of the refactoring, assuming the clones are covered by unit tests.

Approach: The execution of the entire test suite of a project after each refactoring is rather time consuming. Therefore, for the investigation of RQ1, we narrowed down our analysis on a single project, namely *JFreeChart*. The reason we selected this project is that it has one of the highest unit test source code coverage (e.g., 54.4% coverage for the `src` directory and 69.5% including tests), and the fastest test suite execution time (less than 10 seconds) among the examined projects. Moreover, *JFreeChart* contains clones with diverse relative locations, allowing to test all clone refactoring scenarios supported by our refactoring engine (i.e., *Extract Method*, *Extract and Pull Up Method*, *Form Template Method*, *Extract Utility Method*). We collected 12,602 clone pairs that were assessed as Lambda-parameterizable by our approach and were covered by unit tests, out of which 230 (1.8%) are located in the same method, 1,821 (14.5%) are located in the same class, 9,585 (76%) are located in different classes having a common superclass, and 966 (7.7%) are located in unrelated classes. Each clone pair was refactored and tested by executing the entire test suite of the project after each refactoring. To the best of our knowledge, this is one of the most extensive evaluations of a clone refactoring engine, even comparable to the evaluation of tools specialized in the testing of refactoring engines, such as *SafeRefactor* [32] and *ASTGen* [33].

Results: None of the 12,602 applied refactorings caused a compilation error, and only one refactoring led to a test failure. The clones involved in the test failure are the methods `writeObject()` in classes `XYShapeAnnotation` and `DialValueIndicator`, respectively. The problem is actually caused from the way that the statements of these methods are matched by [12] and the special role of these methods in the serialization of objects. More specifically, the methods include statements calling special functions (e.g., `SerialUtilities.writePaint()`, `SerialUtilities.writeStroke()`) that write the field values of the classes into a stream. For the deserialization phase, the methods `readObject()` located in the same classes are used to restore the serialized field values by reading them in a specific order from the stream. However, the statement mapping algorithm [12] matches the statements of the methods in a different order than their original order in an attempt to maximize the number of mapped statements. This is causing the field values to be inserted in a different order into the stream after the refactoring, and thus the deserialization fails. We consider that this issue can be fixed by improving the mapping algorithm [12] to assume the presence of data dependencies between statements reading/writing from/to a stream.

Conclusion: We found that the proposed clone refactoring approach is very reliable by testing it on 12,602 pairs of clones. Our clone refactoring engine did not introduce any compilation errors and caused only one test failure.

C. Applicability of Lambda Expressions (RQ2)

Motivation: The goal of RQ2 is to assess the applicability of Lambda expressions for the refactoring of Type-2 and Type-3 clones. Answering this question will help us understand how useful are Lambda expressions for parameterizing the behavioral differences existing in typical Type-2 and Type-3 clones detected by clone detection tools. Moreover, we break down the collected results taking into account three dimensions: (1) the source code type (clones in production vs. test code), (2) the relative location (clones within the same method, same class, same inheritance hierarchy, or unrelated classes), and (3) the clone type (Type-2 vs. Type-3). This will help us understand if Lambda expressions are more useful for a particular category of clones. Finally, we investigate the percentage of the clone pairs for which the Template Method design pattern can be safely applied as an alternative solution to Lambda expressions.

TABLE II
RELATIVE CLONE LOCATION AND LAMBDA APPLICABILITY

Source Type	Clone Location	#Pairs (%)	#Applicable (%)
All	Same method	2832 (6.1%)	1340 (47.3%)
	Same class	10723 (22.9%)	7644 (71.3%)
	Same Java file	141 (0.3%)	113 (80.1%)
	Same hierarchy	25025 (53.5%)	15824 (63.2%)
	Unrelated classes	8044 (17.2%)	2297 (28.6%)
	Total	**46765 (100%)**	**27218 (58.2%)**
Production	Same method	2343 (7.5%)	1209 (51.6%)
	Same class	8715 (28.0%)	5800 (66.6%)
	Same Java file	129 (0.4%)	101 (78.3%)
	Same hierarchy	12315 (39.5%)	6663 (54.1%)
	Unrelated classes	7661 (24.6%)	2238 (29.2%)
	Total	**31163 (100%)**	**16011 (51.4%)**
Test	Same method	489 (3.1%)	131 (26.8%)
	Same class	2008 (12.9%)	1844 (91.8%)
	Same Java file	12 (0.1%)	12 (100%)
	Same hierarchy	12710 (81.5%)	9161 (72.1%)
	Unrelated classes	383 (2.5%)	59 (15.4%)
	Total	**15602 (100%)**	**11207 (71.8%)**

Approach: Our analysis focused on 46,765 clone pairs having statement gaps (Type-3) and/or expression differences (Type-2) that were assessed as non-refactorable by [12]. This means that it is not possible to move the statement gaps before or after the common code, and introduce regular parameters for the expression differences, due to data dependencies between the code in the gaps/differences and the common code to be extracted. The third column of Table II shows the number and percentage of clone pairs in 5 different categories of relative location, for all clone pairs, those detected in production code, and those detected in test code, respectively. Two thirds of the examined clone pairs belong to production code, while one third of them to test code. We can also observe that most of the clone pairs are located in different subclasses of the same inheritance hierarchy (especially for clones in test code), while the second most frequent relative clone location is within the same class declaration. There is also a significant number of clone pairs located in unrelated classes of the production code. Regarding the types of the examined clone pairs, as shown in the second column of Table III, 60% of them are Type-2 clones, while the remaining 40% are Type-3 clones.

All clone pairs went through Lambda analysis (step 3 in Figure 3), and those without precondition violations were refactored and compiled (but not tested). We consider that a clone pair refactored with Lambda expressions can be safely refactored using the Template Method design pattern in two cases: 1) when the clone is extracted to a new superclass, because it can be declared `abstract` and thus we can add abstract methods for the behavioral differences that will be overridden by its subclasses containing the original clone fragments, and 2) when the clone is extracted to an existing superclass that is already `abstract` and is only inherited by the two subclasses containing the original clone fragments.

TABLE III
CLONE TYPE AND LAMBDA APPLICABILITY

Clone Type	#Pairs (%)	#Applicable (%)
Type II	28363 (60.65%)	16173 (57.02%)
Type III	18402 (39.35%)	11045 (60.02%)

Results: As we can observe in the last column of Table II, the parameterization with Lambda expressions is feasible in 58.2% of the examined clone pairs. Moreover, Lambda expressions tend to be more applicable to clones detected in test code than clones detected in production code (71.8% vs. 51.4%). We believe this is due to the nature of the differences appearing in test clones, which are mostly related to the instantiation of the objects being tested and the assertions being examined. We also found that the clones located in the same class or different subclasses in the same inheritance hierarchy tend to be more refactorable with Lambda expressions than clones located in unrelated classes. This result shows that the behavioral differences found in more "distant" clones (i.e., located in unrelated classes) are more difficult to be abstracted into Lambdas, possibly because the implemented requirements are different. Regarding the applicability of Lambdas with respect to clone types, as we can observe in the last column of Table III, the percentage of Type-2 and Type-3 clones that can be parameterized using Lambda expressions is very similar (57% vs. 60%). Finally, out of the 27,218 clone pairs refactored with Lambda expressions, we found that the Template Method design pattern could be applied in only 34.2% of them.

Conclusion: We found that Lambda expressions is a promising Java feature for the refactoring of clones with behavioral differences, especially for those located in test code. In addition, Lambda expressions tend to be more beneficial for clones located in the same class or subclasses in the same inheritance hierarchy. Moreover, Lambda expressions can equally benefit both Type-2 and Type-3 clones. Finally, the Template Method design pattern can be used as an alternative approach to Lambdas in only one third of the refactored clone pairs.

D. Characteristics of Lambda Expressions (RQ3)

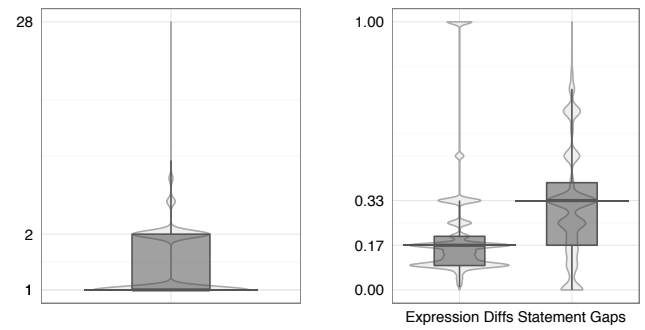
Motivation: In the previous research question, we saw that Lambda expressions can be very effective in the refactoring of clones with behavioral differences. However, this refactoring tends to disperse the duplicated code from two locations originally to three locations after refactoring, i.e., the code

corresponding to behavioral differences remains in the original locations and is passed as Lambdas to the extracted method, while the common code is extracted to another method in the same class, a superclass, or a utility class. The goal of RQ3 is to assess the effect of Lambda expressions on the dispersion of the original code. According to best practice guidelines, Lambda expressions should be short [34], and developers should prefer the standard functional interfaces [35]. When the clone fragments have a small number of behavioral differences with a small size that can be parameterized using the predefined functional interface types provided by Java, the resulting refactored code will be less dispersed. The smaller the number and size of the differences, the more the code is concentrated to a single location after refactoring. The more predefined functional interface types are applicable, the less new custom types will be introduced after refactoring.

Approach: Our analysis focused on the 27,218 clone pairs refactored with Lambda expressions. We used the following indirect measures for code dispersion:

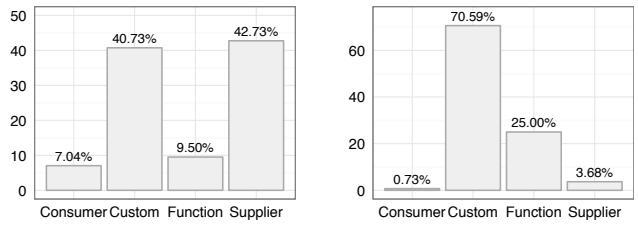
1) The number of Lambda expressions required to parameterize the behavioral differences of the clone fragments.
2) The relative size of the Lambda expressions to the size of the clone fragments.
3) The functional interface types used to parameterize the behavioral differences of the clone fragments.

For each clone pair, we count the number of Lambda expressions required to parameterize its behavioral differences (Figure 4a). For each Lambda expression, we compute the ratio of its size (i.e., number of statements) over the size of the clone fragment (Figure 4b). In particular, for expression differences, we approximate their size to one statement, although an expression is always part of a statement. Finally, for each Lambda expression, we record the functional interface type (Function, Supplier, Consumer, or custom) used to parameterize the corresponding behavioral difference (Figure 5a). Moreover, since the Java predefined functional interface types do not support exception throwing, our refactoring implementation introduces a new custom type whenever the code inside the Lambda expressions is throwing an exception, even if the number of arguments meets the requirements of a predefined type. Therefore, we investigate the percentage of the custom types that could be potentially converted to the Java predefined types, if they supported exception throwing (Figure 5b).



(a) Number of Lambda Expressions (b) Size of Lambda Expressions

Fig. 4. Characteristics of the Lambda Expressions in refactored clone pairs.



(a) Actual Functional Interface types (b) Break down of Custom types

Fig. 5. Distribution of Functional Interface Types.

Results: As we can observe in the box plot shown in Figure 4a, the median value for the number of Lambda expression per clone pair is equal to 1, while the third (upper) quartile is equal to 2. This means that 50% of the refactored clone pairs require only one Lambda expression, while 75% of them require two or less. Regarding the size of the Lambdas, for expression differences the median value is equal to 1/6 of the clone fragment size (box plot on the left side of Figure 4b), while for statement gaps the median value is equal to 1/3 of the clone fragment size (box plot on the right side of Figure 4b). Finally, as we can observe in Figure 5a almost 60% of the Lambda expressions can be parameterized using the Java predefined functional interface types (i.e., Function, Supplier, Consumer), and thus there is no need to create custom types. As we can observe in Figure 5b, an additional 30% of the custom types could be converted to the Java predefined types if they supported exception throwing, increasing the Lambdas that could be potentially covered by the predefined types to 72%.

Conclusion: The majority of the refactored clone pairs require a small number (two or less) of Lambda expressions to parameterize their behavioral differences. On average, the size of the Lambda expressions is relatively small covering 1/6 of the clone fragment for expression differences, and 1/3 of the clone fragment for statement gaps. 60% of the Lambda expressions can be parameterized with the Java predefined functional interface types. This could increase to 72% if the Java predefined types supported exception throwing.

E. Threats to Validity

Internal: *How reliable is the used statement mapping algorithm?* The algorithm used for the mapping of the statements within the clone fragments [12] definitely affects the results of this study, since it produces the expression differences and statement gaps used as input by the proposed Lambda refactoring approach. This algorithm maximizes the number of matched statements and handles the matching of reordered statements between the clone fragments. Despite the fact that it is using data dependence information (coming from interprocedural data flow analysis) as well as type binding information (coming from the compiler), to guide the matching of the statements, there might be some cases where the statement mapping returned by the algorithm is not the most accurate due to missing dependencies resulting from an incomplete analysis. However, the extensive testing we performed on 12,602 clone pairs (Section IV-B) gives us the confidence that the used statement mapping algorithm is reliable and accurate.

How does the clone-pair granularity affect the results? The results of this study are reported at clone-pair level, because the used statement mapping algorithm does not support the analysis of clone groups (also known as clone classes) containing more than two clone fragments, but only pairs of clones. Whenever a clone group is analyzed, we apply our technique on every possible combination of clone pairs (i.e., $\binom{n}{2}$ for a group with n clone fragments). The majority of the clone groups analyzed in this study contain 2 or 3 clone instances (the median is equal to 2, the third quartile is equal to 3, and the mean size is 2.9). This means that for most clone groups the number of clone pair combinations is either 1 or 3, and thus reporting the results at clone-group level would not significantly affect our findings.

External: *How representative are the examined clones?* The clone detectors used in the study have several configuration options, related to the size and similarity of the clones. Even minor changes in the settings can affect significantly the characteristics of the detected clones [36]. Wang et al. introduced a search-based approach to determine the configurations (from the space of all configuration choices) that maximizes tool agreement [36]. However, in our study it is important to get as more diverse clones as possible from each clone detector. Therefore, we used the default or recommended configuration options for each tool. These settings have been consistently used in a large number of empirical studies and can be considered as standards.

How representative are the examined projects? We included in our study 9 projects coming from 9 different application domains, but also having a different Java version compatibility. For example, the projects EMF and JRuby are using the newer language features, such as Generics and varargs, but other projects like Ant are compatible with previous Java versions. This diversity in the domain and the language features used by each project makes possible to generalize our findings to several other open-source projects with similar characteristics. Moreover, we adopted the exact same systems used in previous studies [10], [12] to avoid bias in the selection of subjects.

Verifiability: To make the results of this study reproducible, we provide online all the artifacts required to reproduce and replicate the experiments, including the source code of our tools [31], [29], the results of the clone detection tools used in the experiment, and the R scripts we developed to obtain the results of the experiments [14].

V. Related Work

Clone Refactoring: Meng et al. [11] proposed a technique for automated clone refactoring based on systematic edits (i.e., similar edits to different locations in the source code). One major difference is that Meng et al. [11] rely on systematic edits, while our approach takes as input the results of clone detection tools. Relying on systematic edits for clone refactoring is a rather intuitive solution, since it is believed that clones being frequently and consistently modified during the evolution of a software system should have a higher priority for refactoring [2]. However, systematic edits cannot capture clones that are not consistently updated, mainly because the developers are not aware of their existence, or clones that are never updated. On the other hand, an approach based on the input of clone detectors is more generic and universal, since existing tools can detect both consistently and inconsistently modified clones depending on their configuration (i.e., similarity threshold), as well as stable clones. Other clone refactoring techniques have a more restricted scope. For instance, Tairas and Gray [10] focus only on Type-2 clones having expression differences whose parameterization is free of side-effects. Hotta et al. [37] and Juillerat et al. [38] focus only on clones that can be refactored by applying the Template Method design pattern. Our technique has a much broader scope being able to handle clones with diverse relative locations and behavioral differences. Finally, there are works based on machine learning that recommend clones for refactoring [1], [2], [3].

Lambda Refactoring: Gyori et al. [39], [40] developed a tool, named LambdaFicator, which automates two refactorings introducing Lambda expressions. The first refactoring converts anonymous inner classes to lambda expressions. The second one converts `for` loops that iterate over `Collections` to functional operations that use lambda expressions. The goal of this work is to make the code more succinct by eliminating anonymous classes, and enable unobtrusive parallelism by applying functional operations like `map` or `filter` to iterations. However, our goal is to leverage Lambda expressions in order to eliminate code duplication.

Lambda-related Studies: There is a very limited number of studies investigating the effect of Lambda expressions on development, debugging and testing effort. A recent controlled experiment comparing the use of C++ lambdas with iterators [41] has shown that participants spent more time with compiler errors, and had more errors, when using lambdas as compared to iterators, suggesting difficulty with the syntax chosen for C++. However, the experience level of the participants had a large effect on the experiment results, since professionals were more likely to complete tasks, with or without lambdas, and could do so more quickly than students. Therefore, the experience of the developers, the language syntax, and tool support (e.g., for debugging lambda expressions) play a significant role on the understandability of the code.

VI. Conclusions

In summary, the main conclusions and lessons learned are:

1) Lambdas are effective for parameterizing behavioral differences (58% applicability), especially for test clones (72%).
2) Lambdas are equally beneficial for parameterizing Type-2 and Type-3 clones (57% vs. 60% applicability).
3) The majority of clones require one or two Lambdas.
4) 60% of the Lambdas can be parameterized using just three of the Java predefined functional interface types.
5) Extending these functional interface types to support exception throwing, would increase the aforementioned percentage to 72%.

REFERENCES

[1] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 331–340.

[2] M. Mondal, C. Roy, and K. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, 2014 Software Evolution Week*, 2014, pp. 114–123.

[3] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic identification of important clones for refactoring and tracking," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 11–20.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 485–495.

[5] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming*, vol. 77, no. 6, pp. 760–776, 2012.

[6] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 227–236.

[7] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[8] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE '16, 2016.

[9] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[10] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, Dec. 2012.

[11] 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*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 392–402.

[12] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, Nov 2015.

[13] R.-G. Urma, M. Fusco, and A. Mycroft, *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2014.

[14] N. Tsantalis, D. Mazinanian, and S. Rostami. Clone Refactoring with Lambda Expressions – Dataset. [Online]. Available: http://tiny.cc/ICSE17

[15] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 23:1–23:49, Sep. 2015.

[16] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 547–558.

[17] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, ser. XP 2001, 2001, pp. 92–95.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[20] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[22] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 164–174.

[23] Oracle. The Java™ Tutorials – Local Classes. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

[24] ——. The Java™ Tutorials – Lambda Expressions. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[25] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[26] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 96–105.

[27] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 368–377.

[28] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

[29] Tsantalis. JDeodorant – Commandline. [Online]. Available: https://github.com/tsantalis/jdeodorant-commandline

[30] EclEmma. JaCoCo Java Code Coverage Library. [Online]. Available: http://www.eclemma.org/jacoco/

[31] Tsantalis. JDeodorant – JLS8-support branch. [Online]. Available: https://github.com/tsantalis/JDeodorant/tree/JLS8-support

[32] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, Feb 2013.

[33] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, pp. 185–194.

[34] JetBrains IntelliJ IDEA Blog. Java 8 top tips. [Online]. Available: https://blog.jetbrains.com/idea/2016/07/java-8-top-tips/

[35] Baeldung. Lambda expressions and functional interfaces: Tips and best practices. [Online]. Available: http://www.baeldung.com/java-8-lambda-expressions-tips

[36] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 455–465.

[37] 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*, 2012, pp. 53–62.

[38] N. Juillerat and B. Hirsbrunner, "Toward an implementation of the "form template method" refactoring," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 81–90.

[39] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 543–553.

[40] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "LAMBDAFICATOR: From imperative to functional programming through automated refactoring," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1287–1290.

[41] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of C++ lambdas and programmer experience," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 760–771.