Applicability of Lambda Expressions in Refactoring Near-Miss Micro-Clones in Java

Mehedi Hasan Shanto[1*], Manishankar Mondal[2], Muhammad Asaduzzaman [3],

[1] Computer Science and Engineering Discipline, Khulna University, Bangladesh
[2]School of Computer Science, University of Windsor, Canada

*Corresponding author e-mail id: shanto1832@cseku.ac.bd[1]
mshankar@cseku.ac.bd[2]
masaduzz@uwindsor.ca[3]

Abstract. Duplicate code fragments, known as code clones, are a widespread phenomenon in software development, often causing difficulties in the maintenance and evolution of software systems. Micro-clones refer to short code fragments (i.e., less than five lines in length), which are particularly prevalent in the codebase. This study investigates the application of lambda expressions—a Java 8 functionality that enables the generation of concise and functional code segments, as a refactoring strategy for micro-clones. We focused and identified near-miss micro-clones of across five distinct subject systems using the NiCad clone detector. We have chosen 425 near-miss micro-clone pairs to ensure a 95% confidence level with a 5% confidence interval, based on the clone detector results for thorough investigation. Our findings indicate that 69.4% of these pairs can be refactored utilizing lambda refactoring technique. We also built a classifier using significant features based on Mann-Whitney-Wilcoxon U Test results from the micro-clones that achieved 78.91% accuracy. The effectiveness of lambda expressions in minimizing repetition, enhancing maintainability, and parameterizing the small changes in behavior among micro-clone fragments was verified by their feedback. We enhance the overall understanding of refactoring techniques and offer developers real insights to optimize their code bases.

Keywords: Micro-clone, Refactoring, Lambda Expression, Java, Classification, Software Maintenance.

P. Ranjan et al. (eds.), *Proceedings of the International Conference on Recent Advances in Artificial Intelligence for Sustainable Development (RAISD 2025)*, Advances in Intelligent Systems Research 196,
https://doi.org/10.2991/978-94-6463-787-8_49

1 Introduction

1.1 Background:

Code clones, which are the copies of the code fragments that are present in the code base, are a well-known problem in software engineering that may cause several maintenance issues [1] [2]. The existence of these clones may lead to code obtrusion, may enhance the possibilities of containing bugs, and may make the software difficult to maintain in the future [3]. Conventional refactoring, as well as specialized techniques such as lambda expression refactoring, typically target bigger fragments of duplicated code (i.e., regular clones), where the advantages of refactoring are more readily evident [4]. One kind of redundancy that is both common and extremely sensitive is micro-clones, which are pieces of code that are shorter than five lines. These micro-clones can potentially be present in large numbers throughout the entire project, leading to technical code debt and a decline in quality of the codebase in general [5]. However, micro-clones are often overlooked because they are small or because refactoring them is too time-consuming as compared to the size of the clones. However, many resources could be spent on maintenance due to the existence of micro-clones [5].

1.2 Problem Statement:

Refactoring micro-clones is challenging, so the management of micro-clones needs to be better studied. Micro-clones often involve straightforward methods or common behavior patterns that may not justify object-orientated refactoring, as doing so could introduce unnecessary complexity. Because of their existing limitations on object state, refactoring approaches may change the behavior of programs, particularly when parameterized expressions contain method calls and object construction [4]. Since Java 8, lambda expressions have opened up new ways for refactoring small, repetitive code segments [6].Readability and maintainability can differ between traditional constructs and lambda expressions, as shown in recent study [7]. As lambda expressions are still somewhat new, their adoption in concurrent object-oriented programming remains limited [8]. According to Alqaimi et al. [9], lambda expressions are used in 11% of Java GitHub projects, yet only 6% of these have source code comments attached. In comparison to 2015, there was a 54% rise in lambdas added per extra line of code in 2016, according to research [6]. Petrulio et al. investigated the adoption of lambda expressions in Java, analyzing how developers integrate these features into existing codebases[10]. Despite their potential benefits, lambda expressions have not been tested for refactoring micro-clones. As far as we are aware, this is the initial research to investigate the role of lambda expression in micro-clone refactoring. It would be helpful for managing codebases on a large scale if we knew when and how to use lambda expressions to refactor micro-clones.

1.2 Contributions:

The main goal of this study is to examine the suitability of lambda expressions for refactoring micro-clones in the Java programming language. More precisely, this work seeks to assess the feasibility of using lambda expressions for refactoring micro-clones (less than 5 lines of code), which is still unexplored. We also had a validation session with 11 professionals on refactoring techniques with lambda expressions, especially for microclones. Their input was integrated to help in evaluating the research questions of our group. Our comprehensive investigation of several Java projects revealed the following findings:

- In the 425 near-miss micro-clone pairs, around 69.4% of them can be successfully refactored by using lambda expressions.
- Intra-file micro-clones are less complex to refactor using lambda abstractions than clones that span multiple files.
- It is also found that Type-3 micro-clones are better candidates for refactoring with lambda expressions compared to Type-2 micro-clones.
- Our identified features were used to predict refactorable micro-clones using lambda expressions with an average level of accuracy of 78.91%.

Our aim is not to compare traditional clone refactoring methods with our proposed approach for refactoring micro-clones. In contrast, our objective is to investigate whether there is empirical evidence that substantiates the efficacy of employing lambda expressions in this particular refactoring task.

The remainder of this paper is organized as follows: The 3 section presents a use case scenario of micro-clone existence in a codebase, lambda refactoring techniques, and the use of lambda expressions in software engineering. The 4 section defines the methodology employed to find clones across several Java projects and the criteria for assessing their appropriateness for lambda-based refactoring. The sections 5 and 6 define the experimental settings and outcomes of our investigation, including the proportion of micro-clones amenable to successful refactoring via lambda expressions. In the 7 part, we examine the implications of these findings for software development methodologies. The 8 section examines pertinent literature relevant to our subject. The 10 section clarifies potential future research avenues arising from our investigation and summarizes the key contributions of our research.

2 Terminology: Micro-Clones vs. Regular Clones

2.1 Regular Clones:

These clones are larger, duplicated code fragments (typically more than four lines) that may involve complex logic and structures. Regular clones are usually more straightforward to detect and refactor using traditional methods.

2.2 Micro-clones:

They are defined as code fragments with fewer than five lines; microclones often involve simple, repetitive operations. Despite their small size, they contribute to technical debt and are harder to manage due to their dispersed nature.

3 Use Case Scenario: Super Shop Billing System with Discount for Students and Teachers

The scenario encompasses the refactoring of micro-clones within a SuperShop billing system, which offers discounts to two categories of customers: students and teachers. Different customer types receive varying discounts on food and non-food items.

```
class Student {
        // Method for student discount public void Bill(intfp, intnfp) { double food = fp -
        fp * 0.10; //discount for food double nonFood = nfp - nfp * 0.10; //discount for
        non-food
                double TPrice = food + nonFood;
                System.out.println(Student Bill + TPrice);
        }
}
class Teacher {
        // Method for teacher discount public void Bill(intfp, intnfp) { double food = fp -
        fp * 0.10; // discount for food double nonFood = (nfp> 200) ? nfp - 50 : nfp; //
        fixed 50
       credit discount if non-food item price is greater than 200 double TotalPrice = food +
                nonFood;
                System.out.println(Teacher Bill+TPrice);
        }
}
```

Listing 1.1. Refactored Student and Teacher Bill Calculations with Lambda Expressions

The provided Java code defines two classes, Student and Teacher, each implementing a Bill method to calculate and print the final bill for food and non-food items. Both classes apply a 10% discount on food items. The Student class also applies a 10% discount on non-food items, while the Teacher class applies a fixed 50-credit discount on non-food items if the total price exceeds 200; otherwise, no discount is applied. This difference made these two code fragments Type 3 (near miss) micro-clones of each other (highlighted in red). Here the similarity is preserved by both code fragments of Student and Teacher classes is 75%. After calculating the individual prices, both methods of both classes print the food price, non-food price, and the total bill.

3.1 Examples of Micro-Clones

Within the codebase, discount offers may be available for both teachers and students in several locations. Additionally, there could be other discount offers for professionals like doctors, soldiers, and so on. These discount offers consist of short, recurring code sequences that are dispersed throughout various classes.

3.2 Refactoring Using Lambda Expressions

This section builds upon the previous version of the SuperShop billing system but introduces a more elegant and reusable approach using lambda expressions and a functional interface to handle discount calculations.

```
// Functional interface for discount calculation interface
Discount { double applyDiscount(intprice);
}
class DiscountCalculator {
        // Common method to apply discount logic using a lambda expression public void Fi-
        nalBill(intfoodPrice, intnonFoodPrice, Discount
       nonFoodDiscount, String CustomerType) { double FoodPrice = food-
                Price - foodPrice * 0.1; double NonFoodPrice = nonFoodDis-
                count.applyDiscount(
       nonFoodPrice); double TPrice = FoodPrice
                +NonFoodPrice;
                System.out.println(CustomerType+Bill+TPrice);
                        }
}
class Student extends DiscountCalculator { public void
        Bill(intfp, intnfp) {
                // Define lambdas for student discounts
                Discount nonFoodDiscount = price -> price - price * 0.10; //
       10% discount on non-food
                FinalBill(fp, nfp, nonFoodDiscount, Student);
        }
}
class Teacher extends DiscountCalculator { public void
        Bill(intfp, intnfp) {
                Discount nonFoodDiscount = price -> (price > 200) ? price - 50
         : price; // fixed 50 credit discount if non food tem price is greater than 200
                FinalBill(fp, nfp, nonFoodDiscount, Teacher);
        }
}
```

Listing 1.2. Refactoring of Student and Teacher Bill Calculations using Lambda Expression

This Java code uses a functional interface and lambda expressions to refactor discount calculation logic for Student and Teacher classes. A functional interface Discount is

created to apply different discounts. The class Discount Calculatordefines a common method Final Billto compute the total bill. Both Student and Teacher classes inherit from Discount Calculator. The food item discount (10%) is common for both, but non-food discounts differ:

- Students receive a 10% discount on non-food items.
- Teachers get a fixed 50-credit discount if the non-food item price exceeds 200 credits.

This logic is implemented using lambda expressions passed to the FinalBillmethod (highlighted in red), which prints the total bill for both customer types.

4 Approach

4.1Refactoring with Lambda Expressions:

The preliminary stage was to employ the NiCad [11] clone detector, an acknowledged tool for pinpointing code clones, to identify possible clones in the systems under study. For regular code clones and micro-clones, NiCad is able to search for code clones between one and 2500 lines in length. Through this research, we confined ourselves to micro-clones, these are the segments of the code that do not exceed five lines of code. The micro-clones were manually analyzed to determine the applicability of lambda expressions as a refactoring technique in Java. Lambda expressions in Java provide a concise way to express instances of single-method interfaces, simplifying code for similar scenarios. Since lambda expressions are anonymous functions, they must be abstracted to two functions that have the same signature in order to be combined into a single function. Three primary elements were taken into consideration while analysing method signatures for lambda unification in the clone refactoring with lambda expression analysis by [4]: throw the same exception types, need the same kinds of input parameters, and only allow one variable of the same type to be returned. The following standards were also the subject of our manual analysis:

1. We checked whether the micro-clones were able to support similar or the same argument, necessary for refactoring with lambda expressions. To reproduce the refactored code functionality and coherence, it is required to keep uniformity of input parameters.
2. We verified the generalizability or identity of the return types of the micro-clones. Successful refactoring had required that the lambda expressions had consistent return types.
3. We looked at the exception handling techniques used in micro-clones in order for those to be consistently handled by refactored lambda expression.

The manual analysis consisted of refactoring each micro-clone pair with lambda expressions when appropriate. This method adhered to optimal principles in lambda expression utilization, guaranteeing that the refactored code was both succinct and maintainable while remaining efficient.

4.2 Classification of Micro-Clone Pairs that are Suitable for Lambda Refactoring:

We also explore refactorable and non-refactorable clone pairs and define a metric that encompasses a variety of attributes and groups pairs into various categories based on a combination of their properties and the code components they measure. A rationale explaining the reason for selecting a specific cluster is provided for each cluster.

Features Collection:

In this paragraph, we discuss the features that we gathered to understand the differences between refactorable and non-refactorable micro-clone groups. The complexity of code pieces is influenced by their dimensions, and refactoring is greatly impacted by them[12],[13]. Smaller, more concise code blocks are more amenable to transformation into lambda expressions, which are intended for simple, single-operation functions. Hence we selected the total number of characters in each code fragment along with the lines of code *(LOC)* as features. Less maintainability and more trouble with refactoring ([14],[15]) is often found on code with high amounts of *cyclomatic complexity* and *nesting depth*. Complex code structures don't mesh well with lambda expressions, as they are easier to use than the complex ones.

Code that is similar in syntax and meaning should be refactored as duplicates or close to each other [16],[17]. High similarities mean they can be put into each other and they can do similar things. We covered both:

- **Syntactic Similarity**: Cosine similarity was based on a simple bag-of-words representation.
- **Semantic Similarity**: At the same time, we measured Jaccard similarity [18] of the method names and variable naming consistency.

A complete list of the matrices with their reasoning is provided in Table 1.

We derived various code features using the previously extracted features for the differences between instances of code fragments CF1 and CF2. The calculated code features are arranged in Table 2 (*diff* means numeric difference between two code fragments).

Subsequently, we categorized the data into refactorable and non-refactorable groups and conducted a Mann-Whitney-Wilcoxon test. The U test [19] is employed to calculate the p-value for each feature, with consideration given to those features that are statistically significant ($p < 0.05$) (refer to second column of Table 2). Classifiers were trained using selected features, and the results are presented in Table 5

Table 1. All Extracted Features and Their Rationale

Clone Volume Features, Data Types	Rationale
(cf1 loc, cf2 loc),(cf1 characters, cf2 characters)	It measures the size and code volume of each code fragment measured in lines of code.
Structural Complexity Features, Data Types	**Rationale**
(cf1 cyclomatic complexity, cf2 cyclomatic complexity), (cf1 conditionals, cf2 conditionals),(cf1 loops, cf2 loops),(cf1 depth, cf2 depth),(cf1 return statements, cf2 return _ statements, cf1 exception handlers, cf2 exception handlers)	It quantifies the intricacy of a code's control flow, significantly influencing the ease of employing lambda expressions for refactoring.
Code Structure Features, Data Types	**Rationale**
(cf1 parameters, cf2 parameters) (cf1 method calls, cf2 method calls) (cf1 assignments, cf2 assignments) (cf1 presence super, cf2 presence super, cf1 presence this, cf2 presence this, cf1 presence asserEquals, cf2 presence asserEquals) (cf1 variables, cf2 variables, cf1 keywords, cf2 keywords)	We are able to analyze features such as number of parameters, method calls, and assignments of a program, as well as whether specific keywords are present in order to characterize coding style and the functionality present in a program because they tell how complex and refactoring adaptable is code fragment containing lambda expressions.
Similarity Features, Data Types	**Rationale**
(*similarity method names*, *variable naming consistency*, *cosine similarity*)	We analyze semantic connections in code fragments and give hints to the classifier about which pairs may be good for refactoring in terms of intermediate code changes.

4.3 Validation Through Expert Feedback:

To validate our findings and methodology, we ran a feedback session with 11 participants of various backgrounds, including software engineers, educators, and PhD and MSc students. All the participants were proficient with lambda expressions, having undertaken a course on Software Evolution and Maintenance. During the session, we showed our manually refactored clones and their original code, followed by a survey designed to gather their opinions and insights.

We first conducted a session with the professionals to familiarize them with our approach. We designed our questionnaires not only to validate our approach but also to explore alternative approaches for refactoring our created dataset of micro-clones. Our questions were as follows:

(a) Whether the approach for refactoring micro-clones is appropriate.

(b) Do they have alternative approaches for refactoring these micro-clones?

The questionnaire enquired about participants' ability to propose alternative methods for refactoring the gathered micro-clones and their perception of the suitability and efficacy of lambda expressions. The fact that most participants cannot provide different refactoring techniques emphasizes even more the need for lambda expressions for this specific kind of code duplication. Their feedback further added another degree of validation with the potency of our method and the relevance of our findings in practical software development contexts.

Specifically, our approach consisted of automatic clone recognition and manual analysis followed by expert validation to fully understand the possibility of lambda expressions' fit for refactoring micro-clones. This methodical approach led us to derive important results about the possibilities and limitations of lambda-based refactoring within the Java environment.

Table 2. Derived Features from Table 1 and Selected Features considering P value less than 0.05 of MWW Test

Derived Features	Selected Features
Code Volume Features	–
loc _diff, *total characters diff*	
Complexity Features	**Complexity Difference Features**
cyclomatic complexity diff	*cyclomatic complexity diff*
num conditionals diff	*num conditionals diff*
num loops diff	*num loops diff*
max nesting depth diff	*max nesting depth diff*
num exception handlers diff	–
Code Structure Features	**Structural Difference Features**
num parameters diff	–
num method calls diff	*num method calls diff*
num assignments diff	–
num variables diff	*num variables diff*
num keywords _diff	–
presence super diff	*presence super diff*
presence this diff	*presence this diff*
presence assertEquals diff	*presence assertEquals diff*
–	**Similarity-Based Features**
–	*cosine similarity*, *similarity method names*
–	*variable naming consistency*

Table 3. List of Studied Subject Systems with their Domain and Number of LOC in the lastRevision

Subject Systems	Lang.	Domains	KLoc
Apache-Ant	Java	Java application build tool	67
Jmeter	Java	server performance testing	54
Jabref	Java	Reference Management	46
DnsJava	Java	Dns Protocol	23
JFreeChart	Java	chart library	76

5 Experimental Settings

We developed the study to evaluate the accuracy and suitability of lambda expressions for refactoring micro-clones with complex behavioral variations. The study aimed to address two research questions. A dataset comprising 425 near-miss micro-clone pairs exhibiting Type 3 and Type 2 was developed utilizing the NiCad clone detector to facilitate the examination of our study concerns. The dataset was obtained from five open-source GitHub projects.

5.1 Subject Systems:

We chose five distinct subject systems from GitHub representing various domains and coding styles to conduct a thorough examination. Size, complexity, and accessibility of source code were the determining factors in the selection of the systems. By analysing a broad variety of micro-clones provided by these systems, we investigated the versatility of lambda expressions in various coding styles and domains (see Table 3).

5.2Micro-Clone Identification:

We used NiCad [11] to run on the five selected subject systems and extracted clones that satisfied our near-miss micro-clone criteria. Code fragments with up to four lines and exactly 75% similarity were selected because they were likely to be variants of the same logic or structure and therefore potential refactoring candidates. For each of the five systems, we then collected the first 85 pairs of micro-clones giving 425 near-miss micro-clone pairs in total. NiCad [20] detects function- or block-level code clones. NiCad settings detected two sorts of clones. With standard NiCad parameters, high accuracy and recall are achieved [21][22][23].

5.3 Manual Refactoring and Collection of Data to Answer Research Questions:

In Figure 1, we show the methodology we used to gather the data necessary to answer our research questions.

1. We began by downloading subject systems from GitHub in several domains and applied NiCad, a clone detector, to detect both regular and micro-clones.

2. We analyze manually the clones and use method clones with 4 maximum lines and 25%maximum dissimilarity.

3. For each of the subject systems, 85 micro-clone pairs were selected and placed into adataset of 425 micro-clone pairs. A sample size of 385 would be needed to get a 95% confidence level with a margin of error of 5% [24]. After filtration, we analyze the first 85 clones of each subject system, yielding a total of 425 near-miss micro-clone pairs. Manual refactoring of each clone is difficult, so we chose only 85 micro-clones from each system for refactoring. 4. To each clone pair, we applied lambda expressions and labelled the refactorable and non-refactorable clones, respectively.

4. Using features with a $p - value$ <0.05, we identified the significant features and built aclassifier.

5. Finally, we conducted a user study to validate our approach and findings.

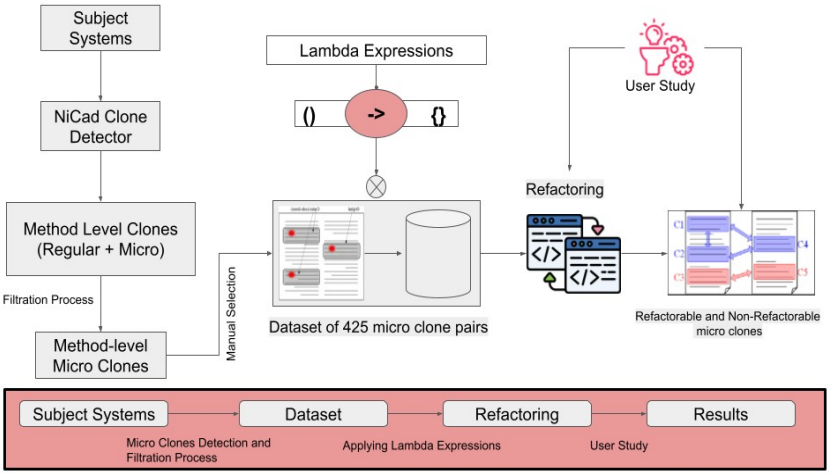

Fig.1. An overview of our proposed approach

Table 4. Refactorability metric in Clone Types Centric Analysis and Their Relative

FilePath Analysis						
Systems	Apache-Ant	Jmeter	Jabref	DnsJAVA	JFreeChart	Total
RMCP	58	67	65	58	47	295
NRMCP	27	18	20	27	38	130
(%) of RMCP	68.2%	78.8%	76.4%	68.2%	55.2%	69.41%
(%) of NRMCP	31.7%	21.1%	23.5%	31.7%	44.7%	30.59%
(%) of CP in Type 2	70.59%	61.18%	63.53%	40.00%	54.12%	57.88%
(%) of CP in Type 3	29.41%	38.82%	36.47%	60.00%	45.88%	42.12%
(%) of RMCP in Type 2	70.00%	73.08%	72.22%	70.60%	47.83%	67.07%
(%) of RMCP in Type 3	64.00%	87.88%	83.87%	66.67%	64.10%	72.62%
Analysis in File Locations						
(%) of RMCP in Similar File	81.82%	92.31%	88.00%	65.00%	65.00%	79.44%
(%) of RMCP in Different File	52.66%	72.88%	60.00%	71.11%	52.31%	62.04%

6 Results

6.1 Applicability of Lambda Expressions

RQ1: To what extent can we apply lambda expressions for refactoring micro-clones?

Motivation:

The primary goal is to evaluate the usefulness of Lambda expressions for refactoring near-miss micro-clones across different types (Type 2 and Type 3) and locations (same file vs. different files). Addressing this question sheds light on the practicality of using Lambda expressions to parameterize observed behavioral differences in clone detection tool output, helping to identify whether this refactoring approach is more beneficial for specific clone types or locations.

Approach:

We collected 425 near-miss micro-clone pairs from 5 different subject systems with 75% similarity detected by NiCad. These pairs included 246 Type 2 clones (57% of the dataset) and 179 Type 3 clones (43%). We manually applied Lambda expressions to these clones, as described in Section 4, to determine their refactorability. Additionally, we analyzed the applicability of Lambda expressions based on clone locations, differentiating between clones within the same file and those across different files. The total number of refactorable and non-refactorable micro-clone pairs, along with their percentages, are presented in Table4.

Result:

In Table 4, **RMCP** and **NRMCP** mean Refactorable micro-clone pairs and Nonrefactorable micro-clone pairs, respectively. Our analysis shows that Lambda expressions can be used to refactor 69.4% (RMCP) of near-miss micro-clone pairs. Specifically, 73% (RMCP in Type 3) of Type 3 clones and 67% (RMCP in Type 2) of Type 2 clones were deemed refactorable. Additionally, Lambda expressions were found to be more relevant for clones located in the same file (79.44%) (RMCP in Similar File) compared to those in different files (62.04%) (RMCP in Different File). These findings align with Tsantalis et al. [4], who demonstrated that Lambda expressions are more applicable to smaller code clones and those located within the same file. However, unlike Tsantalis et al., who employed automated techniques, our study relied on manual validation by 11 experts, providing a focused analysis on 425 near-miss micro-clones. This demonstrates the considerable potential of Lambda expressions for refactoring micro clones, particularly Type 3 clones and those in the same file, thereby extending the scope of existing research to micro-clones.

6.2 Prediction of Refactorable Micro Clones

RQ2: Can we predict suitable refactoring candidates?

Motivation:

The design of this research question is based on the findings of RQ1. According to the RQ1 the refactorable and non-refactorable near-miss micro-clones are structurally different. The RQ3 is to find and extract statistically important characteristics from the labelled dataset of 425 microclones and construct a classifier. It makes the automated classification of micro-clones appropriate for refactoring with Lambda expressions feasible. However, this is necessary for developers working with these micro-clones to identify appropriate refactorable clones. Examining this question would allow developers to quickly manage their maintenance procedures by guiding them with simple decisions laid out on the basis of code feature analysis. If included in IDEs, a classifier may help developers identify the micro-clones that can be refactored. Additionally, during Lambda refactoring, developers may disregard parts of the code that are not suitable for refactoring.

Approach:

A labelled dataset of 425 near-miss micro-clone pairs was classified. Firstly, we extracted numerous features from pairs of code snippets. This analysis attempts to study the multitude of attributes of the code, e.g., size, complexity, similarity, and uses of particular programming constructs. Such features consist of lines of code, cyclomatic complexity, the presence of certain key words, and cosine similarity among pairs of codes. After feature extraction, we utilized statistical tests (Mann-Whitney U test [19]) to select features that significantly differentiate code pairs that can be refactored

using Lambda expressions from those that do not. As a result, this selection process aims to select the most informative features in the detection of refactorable micro-clones. The data was split into the training and testing sets; the training data comprised 70% while the testing consisted of 30%, and different machine learning models were trained with the selected features. We have about 70% refactorable micro-clones, which is a significantly higher portion than the portion of non-refactorable clones; their number is skewed. Faced with this problem, we designed an oversampling algorithm called SMOTE [25] for the training dataset.

Result:

The performance of all five classifiers in our investigation is reported in Table 5. Additionally, we demonstrate that our SVM, Logistic Regression, and Random Forest classifiers provide competitive accuracy values of 78.91%, 77.34%, and 76.56%, respectively, thereby showcasing the predictive performance of these classifiers over the dataset. Then XGBoost and Decision Tree follow at 74.22%, 76.56%.

In this classification challenge, Logistic Regression, Random Forest, and XGBoost work well on the class Refactorable, R. Logistic regression has a precision of 0.87, recall of 0.80, and balanced F1-score of 0.83. In this class too, these models do a great job, and Random Forest scored 0.90 precision and 0.74 recall, whereas XGBoost scored 0.89 precision and 0.72 recall. As for ensemble approaches, the decision tree performed with less precision (0.89) as well as recall (0.75); however, it performed well balanced.

For class NR (non-refactorable), SVM has the highest recall (0.77), F1-score (0.69), and precision (0.62). Random Forest results suggest precision (0.58) and recall (0.82) are well balanced, consequently decreasing the false negatives. Logistic regression performs similarly with a precision (0.61) and recall (0.72) for class NR. Among this class, XGBoost has 0.79 recall but the lowest precision (0.55). Random Forest is very similar in performance with Decision Tree, also displaying 0.79 recall for Class NR with 0.58 precision.

Table 5. Comparison of Precision, Recall, and F1-Score for Different Classifier

Classifier	Class	Precision	Recall	F1-Score
Logistic Regression	NR	0.61	0.72	0.66
	R	0.87	0.80	0.83
		Accuracy: 0.7734		
SVM	NR	0.62	0.77	0.69
	R	0.89	0.80	0.84
		Accuracy: 0.7891		
Random Forest	NR	0.58	0.82	0.68
	R	0.90	0.74	0.81
		Accuracy: 0.7656		
XGBoost	NR	0.55	0.79	0.65
	R	0.89	0.72	0.80
		Accuracy: 0.7422		
Decision Tree	NR	0.58	0.79	0.67
	R	0.89	0.75	0.82
		Accuracy: 0.7656		

7 Discussion

In our study, we devoted ourselves to solving four research problems to confirm the feasibility of lambda expression, in particular, in refactoring micro-clones. Results from response to RQ 1 show that the lambda expression plays a role in refactoring most of the micro-clones detected by a clone detector. In addition, we found that micro-clones residing in the same file are more refactorable using lambda expressions than micro-clones with file locations differing between them. It also, however, brings out the fact that Type 3 micro-clones are a little more refactorable than Type 2 micro-clones. The feature selection addressed how to distinguish internal patterns of refactorable as well as non-refactorable micro-clones from each other. The results of these research questions inspired us to build a classifier to accurately predict refactorable candidates. The last study question classifies the suitable micro clones for refactoring using lambda expression, examining internal features of both refactorable and non-refactorable near-miss micro clone pairs. We further validated the findings by administering a survey to 11 specialists. This finding is especially important in software systems of large scale, where minor improvements in code quality lead to significant longterm benefits.

The use of lambda expressions facilitates the production of a clearer, more readable, more succinct, and less error-prone codebase. This is especially important in a team context in which two or more developers will work with a single codebase. However, this study also showed limitations, such as methods with different signatures and specific internal logic, and their contexts were less effective with lambda expressions. To the extent that these findings can be generalized, we suggest that lambda refactoring is indeed valuable, albeit with limited applicability: for some situations we shouldn't use it.

8 Related Work

8.1 Analysis of Micro-Clones:

The concept of micro-clones was introduced by Beller et al. [26]. The small-scale empirical investigation on the automated detection and elimination of micro-clones was carried out by Tonder et al. [27]. Mottakin et al. [28] examined context adaptation bugs in micro-clones and found that Type-1 and Type-2 micro-clones have a higher probability of harbouring context adaptation bugs compared to Type-3 micro-clones.

In a recent study by Shanto et al. [29], the change-proneness of micro-clones at different granularities (e.g., function and block) was observed. They found that micro-clones at the function level have higher change-proneness than block-level micro-clones by applying three different evaluation approaches.

Mondal et al. [30] conducted a study on micro-clones implemented in Java and C across six different systems. Their findings indicate that the proportion of consistent updates in micro-clones is dramatically greater than the proportion observed in regular clones. The researchers noted that around 80% of the regular updates throughout the whole lifespan of a software system take place in micro-clones. Furthermore, Mondal et al. [31] found that around 83% of the regular updates in micro-clones are complex. Although their work highlights the significance of micro-clones, it does not provide any refactoring protocols for eliminating micro-clones.

We see all the studies investigated micro-clones potentiality, but none of the existing studies perform refactoring using lambda expressions. This study represents an initial effort to refactor micro-clones exhibiting behavioral differences. We particularly measure the applicability of lambda expressions in the context of micro-clones and built a classifier to classify suitable refactoring candidates using lambda expressions.

8.2 Investigations on Lambda Expressions:

According to the study [32] examines the performance of lambda expressions in Java 8, highlighting that their inclusion enhances the functionality, conciseness, and reada-

bility of code. Furthermore, when the execution time is sufficient, the new lambda expressions can offer a performance benefit.

None of the studies addressed the potential of lambda expressions in micro-clones. The study by Zheng et al. [33] suggested avoiding large lambda expressions, making it worthwhile to investigate micro-clones that are a maximum of four lines in size. We focused on this part of the codebase to increase the significance of lambda expressions and program comprehension.

Tsantalis et al. [4] [34] have developed a technique to examine the applicability of lambda expressions for refactoring regular clones with behavioral differences. They found that lambda expressions enable the refactoring of a significant portion of clones that cannot be refactored using other methods. Lambdas are efficient for defining behavioral variations, especially for test clones (72%). They are comparably advantageous for parameterizing Type 2 and Type 3 clones (57% versus 60%). The majority of clones necessitate one or two lambdas, and 60% can be parameterizedutilizing only three predefined functional interface types in Java.

Our aim is to investigate the usefulness of lambda expressions for the refactoring technique, especially in the micro-clone context. Additionally, none of the studies predict the refactorable and non-refactorable micro-clones using lambda expressions.

9 Threats to Validity

We chose the NiCad clone detector in this study due to its higher precision and recall compared to other clone detectors, a critical factor as the findings heavily rely on its output. The study acknowledges that results can vary depending on the clone detector's configuration, a problem termed the confounding configuration choice by Wang et al.[35], who recommended specific configurations for NiCad. NiCad is used for several studies for detecting clones [5] [29] [28].

9.1 Manual Analysis:

The study's internal validity could be affected by biases, such as the manual refactoring of 425 micro clones across 5 subject systems, given that manual analysis of the full set of micro clones is impractical. Although the sample may not fully represent the entire population of micro-clones, the authors treated it as statistically representative, with a 95% confidence level and 5% confidence interval. Additionally, the expert validation could be a threat to validity because we engaged only 11 experts for the validation, but the selection of experts was diversified and their feedback was generalized.

9.2 Diverse Coding Styles:

Lambda refactoring may be less effective for diverse or unconventional coding styles. The findings emphasize that while generalization is limited due to the small number of systems studied, the diverse characteristics of the systems suggest that the results are significant for clone management and software evolution.

10 Conclusion

In this empirical investigation, we sought to assess the feasibility of lambda expression refactoring in micro-clones and answered four research questions. The results of our study show that 69.4% of all considered micro clone pairs can be classified as refactorable, and clones within the same file and Type-3 clones are more likely to be refactorable. To this end, we created classifiers using the selected significant features using the MNW U test (p <0.05) to predict refactorable and non-refactorable micro-clones. SVM was the best model in terms of accuracy at 78.91%, with logistic regression at 77.34%, random forest at 76.56%, and XGBoost at 74.22%. Random Forest and XGBoost had reasonable prediction accuracy with a balanced accuracy and recall rate, especially for the refactorable class.

We should mention that proposing a technique for refactoring near-miss micro-clones using lambda expressions was not our goal in this research. Our goal was to assess the refactorability of near-miss micro-clones using lambda expressions by employing experienced programmers. As we have found that near-miss micro-clones are refactorable using lambda expressions, any existing automated lambda refactoring technique (for example, the technique proposed by Tsantalis et al. [4], and JDeodorant [34]) might be applicable for automatically refactoring micro-clones using lambda expressions. In the continued work towards the automation of the present paper's methodology, potential strategies for the future involve creating machine learning programs that would identify micro-clones and propose lambda-based refactoring suggestions. In order to examine the strengths and limitations of lambda-based refactoring in different programming environments, further research could focus on lambda and other related functional languages. Another research topic is concerned with the effectiveness of lambda-based refactoring in terms of code maintainability if applied in the long run.

In light of these results, the usefulness of lambda refactoring for micro-clones in large-scale systems is underscored, and the viability of employing machine learning classifiers in supporting developers through a large code base is demonstrated.

Disclosure of Interests. The authors declare that they have no conflict of interest.

References

1. Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S.: Do code clones matter?,in 2009 IEEE 31st International Conference on Software Engineering, pp.

485–495, IEEE, (2009).
2. Bettenburg, N., Shang, W., Ibrahim, W. M., Adams, B., Zou, Y., and Hassan, A. E.: An empirical study on inconsistent changes to code clones at the release level,Science of Computer Programming, **77**, no. 6, pp. 760–776, (2012).
3. Lozano, A., and Wermelinger, M.: Assessing the effect of clones on changeability,in 2008 IEEE International Conference on Software Maintenance, pp. 227–236, IEEE, (2008).
4. Tsantalis, N., Mazinanian, D., and Rostami, S.: Clone refactoring with lambda expressions, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 60–70, IEEE, (2017).
5. Mondai, M., Roy, C. K., and Schneider, K. A.: Micro-clones in evolving software,in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 50–60, IEEE, (2018).
6. Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D.: Understanding the use of lambda expressions in Java,Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, 1–31, (2017).
7. Hanenberg, S., and Mehlhorn, N.: Two n-of-1 self-trials on readability differences between anonymous inner classes (AICs) and lambda expressions (LEs) on Java code snippets, Empirical Software Engineering, **27**(2), 33, (2022).
8. Nielebock, S., Heumüller, R., and Ortmeier, F.: Programmers do not favor lambda expressions for concurrent object-oriented code, Empirical Software Engineering,**24**, 103–138,(2019).
9. Alqaimi, A., Thongtanunam, P., and Treude, C.: Automatically generating documentation for lambda expressions in Java, in 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 310–320,(2019).
10. Petrulio, F., Sawant, A. A., and Bacchelli, A.: The indolent lambdification of Java: Understanding the support for lambda expressions in the Java ecosystem,Empirical Software Engineering,**26**, 1–36, (2021).
11. Cordy, J. R., and Roy, C. K.: The NiCad clone detector, in IEEE 19th International Conference on Program Comprehension, pp. 219–220, (2011).
12. Martin, R. C.: Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, (2009).
13. Chidamber, S. R., and Kemerer, C. F.: A metrics suite for object-oriented design,IEEE Transactions on Software Engineering, **20**(6),476–493, (1994).
14. McCabe, T. J.: A complexity measure,IEEE Transactions on Software Engineering, (4), 308–320, (1976).
15. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, (2018).
16. Roy, C. K., and Cordy, J. R.: A survey on software clone detection research,Queen's School of Computing TR,**541**(115), 64–68, (2007).
17. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J.: Distributed representations of words and phrases and their compositionality,Advances in Neural Information Processing Systems, 26, (2013).
18. Niwattanakul, S., Singthongchai, J., Naenudorn, E., and Wanapu, S.: Using of Jaccard coefficient for keywords similarity, in Proceedings of the International Multiconference of Engineers and Computer Scientists, **1**, 380–384, (2013).

19. The Mann–Whitney U: A test for assessing whether two independent samples come from the same distribution,(2007).
20. Roy, C. K., and Cordy, J. R.:NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in 2008 16th IEEE International Conference on Program Comprehension, pp. 172–181, (2008).
21. Mondal, M., Roy, C. K., and Schneider, K. A.: An empirical study on clone stability,SIGAPP Appl. Comput. Rev., **12**, 20–36, (2012).
22. Mondal, M., Roy, C. K., and Schneider, K. A.: A survey on clone refactoring and tracking, Journal of Systems and Software,159, 110-429, (2020).
23. Islam, J. F., Mondal, M., and Roy, C. K.: A comparative study of software bugs in micro-clones and regular code clones, in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 73–83, IEEE, (2019).
24. Calculator.net, Sample size calculator, n.d. Accessed: (2024-10-04).
25. Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P.: SMOTE: Synthetic minority over-sampling technique, Journal of Artificial Intelligence Research, **16**, 321–357, 2002.
26. Beller, M., Zaidman, A., and Karpov, A.: The last line effect, in Proceedings of the 23rd International Conference on Program Comprehension (ICPC), pp. 240–243, (2015).
27. van Tonder, R., and Goues, C. L.: Defending against the attack of the micro-clones, in Proceedings of the 24th International Conference on Program Comprehension (ICPC), pp. 1–4, (2016).
28. Mottakin, S., Zaman, M. B., Mondal, M., and Shome, A.: Context-adaptation bugs in microclones, in Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC), pp. 239–248, (2023).
29. Shanto, M. H., Rony, M. A., Rameswar, D., and Mondal, M.: Granularity-based comparison of the change-proneness of micro-clones, in Proceedings of the 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), (2024).
30. Mondal, M., Roy, C. K., and Schneider, K. A.: Micro-clones in evolving software, in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 50–60, IEEE, (2018).

31. Mondal, M., Roy, C. K., Rahman, M. S., Saha, R. K., Krinke, J., and Schneider, K. A.: Comparative stability of cloned and non-cloned code: An empirical study, in Proceedings of the 27th Annual ACM Symposium on Applied Computing, 1227–1234, (2012).
32. Ward, A., and Deugo, D.: Performance of lambda expressions in Java 8, in Proceedings of the International Conference on Software Engineering Research and Practice (SERP), pp. 119, (2015).
33. Zheng, M., Yang, J., Wen, M., Zhu, H., Liu, Y., and Jin, H.: Why do developers remove lambda expressions in Java?, in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 67–78, , (2021).
34. Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A.: Ten years of JDeodorant: Lessons learned from the hunt for smells, in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 4–14, IEEE, (2018).