Analysis and Maintainability of Complex Industry Test Code Using Clone Detection

by

Wafa Othman Ahmad Hasanain

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Many companies, including Ericsson, experience increased software verification costs. Agile cross-functional teams find it easy to make new additions of test cases for every change and fix. The consequence of this phenomenon is the duplications of test code [2], which is often referred to as (test) code clone. In this thesis, we attempt to understand the prevalence of test code clones, thus contributing to the clone detection and software testing fields, and we then study how test code can be refactored to remove clones, therefore improving software testing activities.

In this thesis, we aim to achieve the following goals. Firstly, we aim to detect clones in industry test codes. We found, in the subjects shared by our industry partner, that 49% of lines of code of the entire C test code are clones, which is also equivalent to 36% of all the test cases; We found that 73% of lines of code of the entire Java test code is a clone, amounting to 94% of all the test cases. The results we report on in our study include figures about clone frequencies, types, similarity, fragments, size distributions, and the number of line differences in cloned test cases.

Secondly, following standard definitions of clone types, if the source code contains fragments that are in a Type-1 clone class, we theoretical argue and empirically confirm that those fragments also belong to a Type-2 clone class; Further, assuming the source code includes fragments that belong to a Type-2 consistent clone class, these fragments will also appear in a Type-2 blind clone class; Such inclusions also occur with Type-3 clone classes, with different threshold values to detect differences. When one relies on standard definitions of source code clone types, they are therefore running a risk of overlapping information being returned for different types of clone classes. Further, if one interprets

clone type definitions in a somewhat strict manner, whereby, for instance, fragments in a Type-1 clone class should not belong to a Type-2 clone class, the inclusions we just mentioned can be qualified as false-positives. This leads us to refine the standard terminology on clone types in order to avoid possible miss-interpretations of clone detection results, and define and implement a Clone Reduction Procedure to remove overlapping clone detection information. We then use this tool infrastructure to empirically study the extent of overlapping information in clone detection results produced by NiCad, a well-known and effective clone detection tool. In our study, we are using nine Java and nine C subject programs. Our results confirm the presence of overlapping information, sometimes in tremendous proportions. For instance, for the Java subjects, clone fragments obtained with a difference threshold of 10% are at least 80% overlapping with fragments obtained with a threshold of 0%.

The third goal is to refactor test code clones. Refactoring is about restructuring and enhancing existing code without altering behaviour in an attempt to improve maintainability. In our context, we aim to identify the clone tests that may appear to be good opportunities for refactoring test codes according to well defined or to be discovered test patterns so that refactored test code is easy to enhance, understand, and maintain. Our result shows that we used 153 times the Parameterized Test pattern, Extract method and Pull up method and Test Utility Method patterns on 43% of the total of clone groups. Also, we found that the average lines of code reduction percentage for all test suites is equal to 13.46% lines of code.

Acknowledgements

All praise to Allah without his grace and blessing, this work will not have been possible.

I would like to thank my highly inspiring and motivating supervisor, Professor Yvan Labiche, for his valuable advice, guidance, support, and various ideas. His great ideas have guided me to conduct my research properly. This thesis would not have completed without him.

I would also like to thank Dr. Sigrid Eldh from Ericsson for her guidance as well as for providing necessary information regarding the project, and also for their support in completing the project. We would like to thank the engineers at Ericsson Canada who provided useful information and feedback for this research, in particular, Mauro Martins and Victor Cao.

I would also like to thank my industry partner Ericsson Inc. for giving me the opportunity to perform a part of my research at Ericsson and for providing me with the funds and resources. I would like to thank the Natural Sciences and Engineering Research Council of Canada for financially fund this project.

Special thanks to my parents for their continuous love, encouragement and support. I would like to thank my husband, Anas, and my precious gifts from Allah, my four sons, Mohammad, Malik, Majd, and Mohannad, for brightening up my life and being always there for me.

Table of Contents

List of Tables

List of Illustrations

Chapter 1: Introduction

The agile way of working has made it possible to deliver faster and more often. The agile process simplifies adding new features to an existing product. Refactoring code is equally important in the process, but in a business-driven development, aspects of maintainability and especially on the test code is easily pushed in the future as technical debt. Many companies experience increased software verification costs. One possible reason is the constant additions of test cases. As agile cross-functional teams are aiming to show the code works and fix existing bugs, less focus may be spent on maintaining the test code and refactoring test suites. The constant addition of test cases is a result of agile cross-functional teams facing difficulties in identifying the right test case (if it even exists) among several thousands of test cases. Also, developers do not necessarily find the time to reflect on the entire test suite that is run several times a day and is instead of adding new tests repeatedly. As a result, a lot of duplicated test code is produced since the easiest way for developers to create new self-contained tests is to copy-paste from existing tests and make, hopefully, small changes when creating these new tests.

Consequently, the set of test cases, i.e., the test suite, tends to grow in size. Test suite size may even sometimes reach such a point that developers are no longer allowed to commit additional tests, or only a limited number of tests. This leads developers to attempt to trim the test suite. This is no easy task because of the lack of documentation, although the agile approach manifest recommends that “code should be documented.” The test code documentation lacks necessary descriptions of the intent of tests and is, depending on the nature of the software, difficult to interpret. Since Bavota and colleagues [9] provide evidence that test smells have a strong negative impact on program comprehension and

maintenance, and that duplication of code in tests, often referred to a test code clones, is one type of test smell, we are motivated to study the presence of clones in test codes.

In the clone detection field, one identifies different types of clones. Exact clones are pieces of code that are syntactically identical, except for variations in white-spaces, layouts and comments [4]. Near-miss clones are syntactically similar pieces of code except for differences in identifiers, literals, types, whitespaces, layouts, and comments, or changed, added or removed statements. Specifically, when code fragments are syntactically identical, except for variations in whitespaces, layouts, or comments, they are said to be Type-1 clones. When there are also differences in identifiers, literals and types, the fragments are said to be Type-2 clones. As a result of this definition, if the source code contains fragments that are in a Type-1 clone class, then those fragments also belong to a Type-2 clone class; Such inclusions also occur with other standard definitions of types of clones, with other definitions of clone types, as we will discuss later in this manuscript. When one relies on standard definitions of source code clone types, they are therefore running a risk of overlapping information being returned for different types of clone classes. Further, if one interprets clone type definitions in a somewhat strict manner, whereby, for instance, fragments in a Type-1 clone class should not belong to a Type-2 clone class, the inclusions we just mentioned can be qualified as false-positives. Others made this observation before us, though they did not study it in depth [10, 11].

We argue that understanding and reporting on the prevalence of overlapping clone information is important to make an informed decision when dealing with clones (e.g. deciding to refactor, estimating refactoring cost). Suppose, for instance, that a clone detection tool reports that a piece of software exhibits 940 function fragments that are in

Type-1 clone classes and 2023 function fragments that are in Type-3 clone classes: essentially, Type-3 clones are twice as more prevalent in the code as Type-1 clones. One would conclude that refactoring code to remove clones in this piece of software will be difficult because refactoring Type-3 clones is expensive. If, in fact, the 2023 function fragments include a significant amount of fragments that are overlapping with Type-1 clone fragments and there are (only) 895 function fragments in Type-3 clone classes that are not in Type-1 clone classes, the picture in terms of refactoring effort changes: there are roughly as much Type-1 cloning as Type-3 cloning.

Prior to studying clones in test code, we therefore first felt the need to revisit standard clone type terminology to account for such overlapping clone information. We complement a partial extension provided by Islam and Zibran [10], by defining new terminology for clone types and account for various parameters one uses when searching for clones such as renaming conventions and allowed thresholds of differences. The traditional types of clone classes are more or less permissive in the sense that they allow one (or a tool) to account for varying amounts of differences between code fragments when attempting to group them into a clone class. For instance, Type-2 is more permissive than Type-1 but less than Type-3; Using a threshold of 10% is more permissive than 0% and less than 20%; blind renaming is more permissive than consistent renaming. For each traditional clone type, including renaming conventions or threshold values, we define a corresponding Pure version that only includes clone classes for that type and omits clone classes that can be discovered by more permissive clone types. For instance, we define a Pure Type-2 clone class, a Pure Type-3 Consistent clone class, a Pure clone class with a threshold 30%, which are derived

from the usual definitions of the Type-2, Type-3 Consistent, Threshold of 30% clone classes, respectively.

We also define a generic Clone Reduction Procedure to remove overlapped information. It is generic as it does not depend on any specific clone detection tool. We implement this procedure to work on clone data returned by NiCad, a very well-known and effective clone detection tool.

We then use NiCad and our Clone Reduction Procedure to empirically study the extent of overlapping clone information in NiCad clone detection results. To validate our procedure and allow replications, we used nine C open-source subject programs, and nine Java open-source subject programs, that others have used [1] and that are part of a standard benchmark to compare clone detection technologies [12]. Our results confirm the presence of overlapping clone information, sometimes in tremendous proportions. For instance, for the Java subjects: reduction percentage results indicate that clone detection results for threshold value, 10% are 82% to 99% redundant with results obtained for threshold value 0%. NiCad produces less redundant data for C subjects: reduction percentage results indicate that clone detection results for threshold value 10% are 16% to 75% redundant with results obtained for threshold value 0%.

With clone detection technology and tool support to include (e.g. NiCad) or exclude (e.g. NiCad plus our reduction procedure) overlapping data information, we then turned our attention to clones in test code; Indeed, although there is a large body of knowledge about clones [4] in application logic code, we believe more work needs to be dedicated to studying clone prevalence in test code, especially in industry contexts. Clone detection results are likely different for test code when compared to clone detection results for

application logic because the test code and application code are different. For instance, test code typically consists of smaller functions than application code; test code structure is relatively consistent with usually a set-up/execution/tear-down pattern; Some authors argue that test code should be as simple as possible, to the extent that test code should not typically contain alternative control flow[1] [13]. Regardless, the "best" way to structure test code is not yet a solved issue, since probably other factors related to the system under test will impact that structure. Therefore, we expect, at the outset, to have different clone rates and distributions of clone types when studying different industry test code, which also warrants the need for studies like ours.

Furthermore, studying the prevalence of clones in test code will help us in the broader challenge of test suite maintenance and improvement, for instance, to reduce test code size, improve test efficiency, and reduce testing costs.

In this context, we studied clones detected by the NiCad clone detection tool and used two C projects and one Java project, comprised of a total of 1,077,552 lines of code (LOC), obtained from our industry partner. When we examined test code, we found that several test cases are typically created to verify a specific functionalities/function; therefore these test cases have a high chance of being clones. Also, we found that other test cases are system level test cases. These test cases are large because they need to execute functions from a library of functionalities, and they can therefore be considered as system-level tests, there are many aspects of the hardware to configure the testing environment before executing tests, and then there is a lot to clean up at the end of test case executions. Our

[1] The rationale is that with few control flow paths, down to no control flow, in test code, test code is easy to understand, is less prone to faults.

results show that 49% lines of code of the entire C test code are clones, which is also equivalent to 36% of all the test cases; We found that 73% LOC of the entire Java test code is a clone, amounting to 94% of all the test cases. The results we report on in our study include figures about clone frequencies, types, similarity, fragments, size distributions, and the number of line differences in cloned test cases.

Moreover, our investigation shows that a clone fragment can belong to different clone classes under different clone types. Suppose a Type 1 clone class has two fragment f_1 and f_2; f_1 and f_2 are 100% similar. Also, suppose a Type 3 clone class has two fragment f_3 and f_4; f_3 and f_4 are at least 70% similar. Suppose f_1 and f_3 are the same fragments: f_1 is 100% similar to f_2 and at least 70% similar to f_4. One may want to refactor code for each clone class separately; we instead argue that merging such clone classes of various clone types but that share fragments prior to refactoring their fragments will offer more opportunities to improve test codes. We therefore also define a Merge Clone Classes Procedure to merge different clone classes that shared clone fragments, and implement this procedure to work on clone data returned by NiCad. The results of applying the Merge Clone Classes procedure show that the percentage of reduction to the clone classes is approximately 63%, the average fragments reduction for all the test suites is equal to 54.4%, and the LOC reduction percentage is equal to 40.1%. This indicates that the Merge Clone Classes procedure helps to reduce the total clone classes, the number of fragments and total LOC significantly. Our merging procedure will help to reduce the effort of refactoring the clone fragments.

In this thesis, we focus on managing the test clones by refactoring the test clones. Clone detection and refactoring usually refer to two different tasks. We find that it is necessary

to integrate these two tasks in order to enhance the maintainability of the test codes. The integration between these two tasks is usually referred to as clone refactoring. The clone refactoring approach is a potential technique of clone management.

In this context, we study the prevalence of test clone classes, and we define 15 properties of clone classes. We also study the (test) patterns in order to find out which pattern we can use. We find that we can use the following patterns: Parameterized Test pattern, Extracted method, and Pull up method and Test Utility Method patterns. Then we associate each clone class's property with a pattern that can be used to refactor a specific clone class. We identify association guidelines between the clone class's properties and the pattern. To refactor the clone classes, we apply these guidelines on the 153 clone classes that are equivalent to 43% of the total clone classes. Also, we found that the average lines of code reduction percentage for all test suites are equal to 13.46% LOC.

Our thesis is that engineers need precise information about clones, more precise than what a tool like NiCad reports, in order to refactor test code, and this precise information must include overlapping clone detection information and the merging of clone classes with similar fragments.

1.1 Research Questions

In this section, we point out to the main research questions and objectives of this thesis.

Overlapping Information in Clone Detection Results—We find that clone detection results contain overlapping information: Results of Type-2 clone classes also include results of Type-1 clone classes; Results of Type-3 clone classes include results of both Type-1 and Type-2 clone classes. We argue that understanding and reporting on the prevalence of

overlapping clone information is essential to make the enlightened decision when dealing with clones. In this thesis, we refine the standard terminology on clone types, extending a partial extension provided by Islam and Zibran [10], by defining new terminology for clone types and account for various parameters one uses when searching for clones such as renaming conventions as well as thresholds of differences. The traditional types of clone classes are more or less permissive in the sense that they allow one (or a tool) to account for varying amounts of differences between code fragments when attempting to group them into a clone class. For instance, Type-2 is more permissive than Type-1 but less than Type-3; Using a threshold of 10% is more permissive than 0% and less than 20%; blind renaming is more permissive than consistent renaming. Blind and Consistent renaming, as another clone terminology is defined in detail in Chapter 2: For each traditional clone type, including renaming conventions or threshold values, we define a corresponding Pure version that only includes clone classes for that type and omits clone classes that can be discovered by more permissive clone types. For instance, we define a Pure Type-2 clone class, a Pure Type-3 Consistent clone class, a Pure clone class with a threshold 30%, which are derived from the usual definitions of the Type-2, Type-3 Consistent, Threshold of 30% clone classes, respectively. This would allow arguably to work in favour of a user who gets accurate information and, consequently, is not fed "false positives."

We also define a generic Clone Reduction Procedure to remove overlapped information. It is generic as it does not depend on any specific clone detection tool. We implement this procedure to work on clone data returned by NiCad, a very well-known and effective clone detection tool.

We then use NiCad and our Clone Reduction Procedure to empirically study the extent of overlapping clone information in NiCad clone detection results. To validate our procedure and allow replications, we used nine C open-source subject programs, and nine Java open-source subject programs, that others have used [1] and that are part of a standard benchmark to compare clone detection technologies [12]. Also, we perform the same study, looking at the prevalence of overlapping clone information on several other open source subject programs.

An Analysis of Complex Industry Test Code using Clone analysis—We study the prevalence of clones in test code in industry test code to help us in the broader challenge of test suite maintenance and improvement, for instance, to reduce test code size, improve test efficiency, and reduce testing costs. We studied clones detected by the NiCad clone detection tool and used two C projects and one Java project, comprised of a total of 1,077,552 lines of code (LOC), obtained from our industry partner. We have formulated and answered the following research questions:

RQ1: What is the ratio of clones in industry test code? The clone ratio reveals how much of the test code is duplicated. This measure is also traditionally used in studies of clones in application code.

RQ2: What is the distribution of clone classes' sizes, the size of a clone class being its number of fragments? This gives us another measure of the scale of cloning in test code.

RQ3: What is the ratio of cloned test cases? Since we set the granularity of the clone detection to "function" during clone detection, and in our code, tests are functions, all the clones we detected are test cases. While RQ1 will reveal the

clone ratio in terms of LOC, RQ3 is complementary as it will reveal the clone ratio in terms of test cases.

RQ4: What is the difference between clone similarity and the number of line differences? Answering this question will tell us how similar, in terms of LOC, cloned tests are. Given clone detection tools typically detect differences from transformed versions of the original code (e.g., pretty-printed code, abstract syntax tree), considering LOC differences is an interesting complementary piece of information since possibly large differences in the transformed code may actually take place in a very few numbers of lines of the original code.

RQ5: What is the number of occurrences of clone fragments for each similarity threshold? Clone detection relies on various similarity thresholds, i.e., maximum amounts of differences to identify clones. We study the occurrences of clone fragments for various similarity threshold values to find out how many clones fit under each similarity. Since such studies have been reported in the literature on application code, this data will allow us to see whether there are differences with the industry test code we measure.

Test Clone Refactoring—We define a Merge Clone Classes Procedure to merge different clone classes that shared clone fragments. And we implement this procedure to work on clone data returned by NiCad. After that, we analyze the clone tests in order to define the properties of clone classes. We have defined 15 properties of clone classes. Then we study the (test) patterns and we find that we can use the following patterns: Parameterized Test pattern, Extracted method and Pull up method and Test Utility Method patterns. We define a relationship between each clone class's property and the pattern that

can be used to refactor a specific clone class. We identify association guidelines between the clone class's properties and the pattern. To refactor the clone classes, we apply these guidelines to the 153 clone classes. We analyzed the results by answering the following research questions:

RQ1: What are the clone classes' properties that we found in the test clones? Which pattern is applied to the test clones? What is the association between the clone class's properties and refactoring decisions? Which pattern is the most frequently used? Answering this question will help us to identify the clone class's properties, which will reveal the essential purposes of cloning in test codes. We study the different patterns and the goal of each pattern, and then we find out which clone class's properties are suitable to apply a particular pattern. In particular, we are interested in finding out association guidelines that combine the pattern and the clone class's properties. The guidelines do not represent specific implications in a logic sense, but rather a possible association of the analyzed clone classes' properties. This could help to determine which specific pattern and clone class's properties pairs are particularly interesting in the sense that they explain a large number of analyzed clone classes' properties and are frequently found in the clone classes.

RQ2: What is the ratio of clone reduction in industry test code? The clone reduction ratio reveals how much of the test suite size (i.e., number of test cases) is reduced and how much of the test clones are reused after refactoring the clone classes.

RQ3: What are the effects of the clone type on refactoring decisions? And what is the relation between clone type and refactoring decision? Answering this question will reveal if the clone type influences the decision of refactoring. And we want to find out what is the most refactorable clone types and what is the most challenging clone type from the refactoring point of view.

RQ4: How does the size of the clone fragments (i.e., lines of code) affect the refactoring decision? Answering this question will help researchers to understand if the size of the clone fragment will influence the refactoring decision. Also, the answer will help to know if the fragment size increases the refactoring complexity.

RQ5: How does the size of the clone class affect the refactoring decision? Answering this question will reveal if the size of the clone class will increase or decrease the effects of refactoring the test codes. This means if we have a large number of fragments belonging to the same clone class, and we want to find out what is the possibility of refactoring this clone class, then answering this question will help us to decide to refactor such large clone classes or not.

1.2 Contribution of the thesis

The dissertation contributes the following to the domain:

First, we study the Overlapping Information in Clone Detection Results, and contribute the following:

- We formalize the notion of overlapping clone information;
- We define a generic way to define a Pure version of a traditional clone type, which does not include overlapping clone information from more permissive

clone types. In particular, we refine the standard terminology on clone types with notions like Pure Type-2, Pure Type-2 Consistent, Pure Type-2 Blind, Pure Type-3, Pure Type-3 Consistent, Pure Type-3 Blind clones;

- We define and implement a Clone Reduction Procedure to remove overlapping clone information, i.e. remedy false positives. This is essentially a procedure to trim clone information, as returned by conventional clone detection tools (for instance, NiCad), and obtain Pure clone type data. We explain how this general procedure can be applied to NiCad;
- We replicate a previously published experiment [1] using the same subject systems [12] and tool (NiCad) setup and apply our Clone Reduction Procedure. This allows us to show the prevalence of overlapping clone information experimentally;
- We perform the same study, looking at the prevalence of overlapping clone information on several other open source subject programs.
- We perform the same study, looking at the prevalence of overlapping clone information on source code provided by an industry partner.

Secondly, we show the prevalence of software clones in the industry test code. We studied clones detected by the NiCad clone detection tool (after removing overlapping information) and used two C projects and one Java project, comprised of a total of 1,077,552 lines of code (LOC), obtained from our industry partner. We have formulated and answered the research questions we discussed earlier (Section 1.1).

Thirdly, we define a Merge Clone Classes Procedure to merge different clone classes that shared clone fragments, and we implement this procedure to work on clone data returned by NiCad.

Fourthly, we study the prevalence of test clones and define 15 properties of clone classes that lead to refactoring with several well-known patterns. Then we associate each clone class's property with a pattern that can be used to refactor a specific clone class. We identify association guidelines between the clone class's properties and the pattern. To refactor the clone classes, we apply these guidelines to 153 clone classes from test code provided by our industry partner. We analyzed the results by answering the research questions we discussed earlier (Section 1.1).

1.3 Structure of the manuscript

The rest of the manuscript is organized as follows. **Chapter 2** presents some important terminology and discusses related work. **Chapter 3** describes the procedure of removing overlapped data in clone detection results. **Chapter 4** discusses the analysis of complex industrial test code using clone analysis. **Chapter 5** discusses the Merge clone Classes procedure. **Chapter 6** discuss the clone refactoring results. And we conclude this thesis in **Chapter 7**.

1.4 Publications

The work on overlapping information, which is essentially **Chapter 3:**, has been submitted as a journal paper to IEEE Transactions on Software Engineering in March 2020.

> W. Hasanain, Y. Labiche, S. Eldh, "Extending the Taxonomy of Clone Detection Types to Account for Overlapping Information in Clone Detection Results"

Parts of **Chapter 4** was published as a conference paper:

W. Hasanain, Y. Labiche and S. Eldh, "An Analysis of Complex Industrial Test Code Using Clone Analysis," 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, 2018, pp. 482-489."

Most of **Chapter 4** , that is an extension of the conference paper above-mentioned, has been submitted with the same title to the Journal of Systems and Software on December 5, 2019.

We are planning to submit **Chapter 5:** and **Chapter 6:** to journals.

Chapter 2: Background and Related Work

In this chapter, we first provide some background information. We discuss some standard code clone terms (section 2.1), discuss the clone detection process (section 2.2), provide an overview of clone detection techniques (section 2.3), and provide background on some refactoring patterns (section 2.4). We then discuss related work (section 2.5), compare some clone detection tools (section 2.6), and introduce the NiCad clone detection tool (section 2.7).

2.1 Standard Code Clone Terminology

A clone detection tool typically reports on clones in the form of Clone Pairs or Clone Classes or both. These terms are based on an equivalence relation (reflexive, transitive, and symmetric) between code fragments that define a similarity relation between them. Different equivalence relations can be used, typically referring to sequences of characters in text strings (with or without whitespaces), sequences of tokens, normalized or transformed sequences of tokens.

In this context, a **Code Fragment** (CF) [4]is any sequence of contiguous lines of code with or without comments. A code fragment is identified by using a file name, a begin-line, and an end-line. A code fragment CF_2 is a **Code Clone** [4] of another code fragment CF_1 when they are similar according to some given definition of similarity (equivalence relation), that is, $f(CF_1, CF_2)$ where f is the similarity function. When two fragments are similar to each other, these two fragments form a clone pair (CF_1; CF_2). A **Clone Class** [4] is the equivalence class formed by the maximal set of code fragments that form clone pairs. A clone class is simply the union of all clone pairs that share code fragments. A clone class is sometimes also called a clone group.

The **Clone Granularity** [4] is the granularity of the clone fragments in a clone class. The clone granularity can be either free or fixed. A free clone granularity means clones have no syntactic boundaries (e.g., code fragments in a clone class can span over function boundaries). A fixed granularity means that there are predefined syntactic boundaries such as method or block for clone fragments.

The literature has so far identified two main kinds of **Clone Types**[4], based on whether fragment similarity is syntactic (i.e., clones have similar sequences of program texts) or functional (i.e., clones have similar functionalities but are syntactically dissimilar). These notions are traditionally refined into the following clone types: In a **Type-1** clone class, code fragments are syntactically identical except for variations in whitespaces, layouts, and comments; In a **Type-2** clone class, code fragments are syntactically identical except for differences in identifiers, literals, types, whitespaces, layouts, and comments; In a **Type-3** clone class, code fragments show minor or major modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespaces, layouts, and comments; In a **Type-4** clone class, code fragments perform the same computation but are implemented by different syntactic variants. A Type-1 clone is also referred to as an **exact clone**. Type-2 and Type-3 clones are usually referred to as **near-miss clones**.

When determining whether fragments belong to the same clone class of a given type, we typically rely on a notion of dissimilarity between fragments, and a threshold to distinguish between clone class types. The **dissimilarity [4]** of two fragments is a measure of the differences between the two code fragments, typically computed by comparing them line-by-line and determining the number of unique lines in each one. The amount of

dissimilarities, when greater than an a priori **threshold** [4], determines the type of the clone class to which belong the fragments. A threshold is typically a percentage, i.e., the maximum percentage of dissimilarities between two code fragments to consider the two fragments are clones. It is important to note that such a threshold is an operational decision that depends on the tool one uses to detect clones: two different tools may use different threshold values for the same clone type (e.g., Type-3) or one may use different threshold values to identify different kinds of Type-3 clones. When reporting on clone data it is therefore important to disclose threshold values.

Table I: Example of line of code comparison

Line Number	Segment-1	Segment-2	Compare Segment-1 and Segment-2
1	for(	for(	1
2	j = 0;	j = 0;	1
3	j < 7;	j < 10;	0
4	j++)	j++)	1
Total matches			3
Total Mismatches			1

To illustrate this operational decision, suppose we have the following two original line of codes:

- Segment-1: for(j = 0; j < 7; j++)
- Segment-2: for(j = 0; j < 10; j++)

Table I shows a comparison between Segment-1 and Segment-2 where pretty-printing is used to split each fragment into four separate lines. Using a simple line-by-line text comparison of the two segments, we can precisely determine the dissimilarity between the segments. Suppose that we set the dissimilarity to be equal to 70%, we can see that Segment-1 and Segment-2 differ by only one line of code (out of 4) that is equivalent to 25%. That means the two segments are 75% similar. Thus, Segment-1 and Segment-2 form a clone pair (only the initialization value of j changes) with 75% similarity.

In general, if the threshold is set to 0.0, then one is interested in exact clones; if the threshold is set to 0.1, then one is interested in code fragments that may differ by up to 10% of their normalized lines (clone detection algorithms typically normalize strings of lines of code to perform a uniform comparison, such as pretty-printing as exemplified earlier); and if the threshold is set to 0.2, then one is interested in code fragments that differ by up to 20% of their normalized lines. It is also important to note that the line normalization technique is an operational decision and is tool specific. When reporting on clone data it is therefore important to disclose which normalization technique has been used.

Identifier renaming [4] is a strategy to ignore identifiers when determining clones from code fragments (c.f., clone types). The set of identifiers to rename includes function names, variable names, data type names and literal values. Two identifier renaming strategies are typically used: blind renaming and consistent renaming. The **blind renaming** strategy systematically renames all the identifiers to exactly one synthetic identifier string, depending on their type. As a result, the (dissimilarity) comparison algorithm completely ignores differences in identifiers. The **consistent renaming** strategy systematically and consistently renames each identifier separately: for instance, local variable “pressure” may be changed into "X" each time it appears in the code, whereas local variable “temperature” may be changed into “Y” each time it appears in the code. (With a blind renaming strategy, both “pressure” and “temperature” would be renamed into the same string, say, “Z,” each time they appear in the code.)

Illustration 1 illustrates some of these notions with three fragments (fragment A, fragment B, and fragment C) and the corresponding consistent and blind renaming strategies [4]. With the *consistent* renaming, each occurrence of a variable type, variable

name, … is consistently renamed: e.g., “int” is renamed into “x7” in all fragments, the first local variable of each fragment (i.e., “s” in fragment A, “sum” in fragments B and C) is renamed into “x3”. Thus, when comparing the resulting fragments (i.e., fragment *A-consistent*, fragment *B-consistent*, and fragment *C-consistent*), one can identify clone pairs: fragments *A-consistent* and *B-consistent* form a Type-1 clone pair; fragment C-consistent is not in this Type-1 clone class since its uses different local variable types (which are therefore renamed differently as those of fragments A and B), and the order of parameters in the last statement call is different; all three fragments are however in a Type-2 clone class. With a *blind* renaming, all identifiers are renamed to the same string “x”: fragment *A-blind*, fragment *B-blind*, and fragment *C-blind*. Thus, when comparing the resulting

`void calculate(int m) {`
`  int s = 0;`
`  int p = 1;`
`  for (int j=1; j<=m; j++) {`
`  s = s + j;`
`  p = p * j;`
`  foo(s, p);`
`  }`
`}`
	`void sumTimes(int n) {`
	`  int sum= 0;`
	`  int product = 1;`
	`  for (int i=1; i<=n; i++) {`
	`  sum = sum + i;`
	`  product = product * i;`
	`  fun(sum, product);`
	`  }`
	`}`
		`void sumTimes (int n) {`
		`  float sum = 0.0;`
		`  double product = 1.0;`
		`  for (int i=1; i<=n; i++) {`
		`  sum = sum + i;`
		`  product = product * i;`
		`  fun(product, sum);`
		`  }`
		`}`

Fragment A	**Fragment B**	**Fragment C**

`x8 x1(x7 x2) {`
`  x7 x3 = 0;`
`  x7 x4 = 1;`
`  for (x7 x5=1; x5<=x2; x5++) {`
`  x3 = x3 + x5;`
`  x4 = x4 * x5;`
`  x6(x3, x4);`
`  }`
`}`
	`x8 x1(x7 x2) {`
	`  x7 x3 = 0;`
	`  x7 x4 = 1;`
	`  for (x7 x5=1; x5<=x2; x5++) {`
	`  x3 = x3 + x5;`
	`  x4 = x4 * x5;`
	`  x6(x3, x4);`
	`  }`
	`}`
		`x8 x1 (x7 x2) {`
		`  x8 x3 = 0.0;`
		`  x9 x4 = 1.0;`
		`  for (x7 x5=1; x5<=x2; x5++) {`
		`  x3 = x3 + x5;`
		`  x4 = x4 * x5;`
		`  x6(x4, x3);`
		`  }`
		`}`

Fragment A—consistent	**Fragment B—consistent**	**Fragment C—consistent**

`x x(x x) {`
`  x x = 0;`
`  x x =1;`
`  for (x x=1; x<=x; x++) {`
`  x = x + x;`
`  x = x * x;`
`  x(x, x);`
`  }`
`}`
	`x x(x x) {`
	`  x x = 0;`
	`  x x =1;`
	`  for (x x=1; x<=x; x++) {`
	`  x = x + x;`
	`  x = x * x;`
	`  x(x, x);`
	`  }`
	`}`
		`x x(x x) {`
		`  x x = 0.0;`
		`  x x =1.0;`
		`  for (x x=1; x<=x; x++) {`
		`  x = x + x;`
		`  x = x * x;`
		`  x(x, x);`
		`  }`
		`}`

Fragment A—blind	**Fragment B—blind**	**Fragment C—blind**

Illustration 1 Code fragments illustrating the notions of blind and consistent renaming [4].

fragments (i.e., fragments *A-blind*, *B-blind*, and *C-blind*), one can identify a Type-1 clone class.

As a result of following standard definitions of clone types, if the source code contains fragments that belong to a Type-1 clone class, then those fragments also belong to a Type-2 clone class; Further, assuming the source code contains fragments that belong to a Type-2 consistent clone class, these fragments will also appear in a Type-2 blind clone class; Such inclusions also occur with Type-3 clone classes, with different threshold values to detect differences. When one relies on standard definitions of source code clone types, they are therefore running a risk of overlapping information being returned for different types of clone classes.

Further, if one interprets clone type definitions in a somewhat strict manner, whereby, for instance, fragments in a Type-1 clone class should not belong to a Type-2 clone class, the inclusions we just mentioned can be qualified as false-positives.

After making similar observations, though only for some clone type definitions, Islam and Zibran [10] defined two new notions of clone types. A set of ***Pure Type-2*** clones includes only those Type-2 clones that do not exhibit Type-1 similarity; A set of ***Pure Type-3*** clones includes only those Type-3 clones that do not exhibit Type-1 or Type-2 similarities. They do not however explore, neither from an analytical point of view (defining new types of clones for different levels of similarity for instance) nor from an experimental point of view (studying the prevalence of clone fragments that belong to clone classes of different clone classes), this phenomenon. One contribution of our work is to extend these definitions in a more general way and to empirically study the prevalence of clone information overlap, thereby justifying the new (extended) definitions.

2.2 Clone Detection Process

A clone detector attempts to find fragments of code with high similarity in a source text. Since we do not know in advance which code fragments may be duplicated, the detector must compare every possible fragment with every other possible fragment. From a computational point of view, this comparison is prohibitively expensive, and hence, a clone detector needs to take several measures to reduce the comparison domain before performing the actual comparisons. Also, after the cloned fragments are identified, further analysis is required to determine the code clones. Since it is important to detect clones, many different clone detections approaches have been presented over the previous decade. In this section, we provide a short summary of the basic steps of a general clone detection process.

2.2.1 Preprocessing

The clone detection process starts by partitioning the source code to determine the domain of the comparison. This phase consists of three primary objectives, which are:

A. **Removing uninteresting parts:** in this phase, the source code is filtered to remove all the uninteresting source code. For instance, embedded code is separated for different languages (e.g., SQL embedded in Java code, or Assembler in C code). This step is especially important if the tool is not language independent. Also, the generated code, such as LEX- and YACC-generated code is removed from the source code. Some source code sections that are expected to produce false positives such as table initialization are removed from the source code [14].

B. **Determining source units:** After the uninteresting source code is removed, the remaining source code is divided into a set of disjoint fragments, which are called source units. The source units are the largest fragments that may directly have clone relation with each other. The source units can have any granularity level such as files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

C. **Determining comparison units / granularity:** Depending on the comparison technique, the source units may be additionally partitioned into smaller units. For example, the comparison procedure may need the source units to be subdivided into lines or even tokens for comparison. Also, the source units' syntactic structure can be used to derive comparison units. For instance, an if-statement is divided into the conditional expression, the then and else blocks. Depending on the comparison technique, the order of the comparison units within their equivalent source units may or may not be important.

2.2.2 Transformation

When the comparison units are specified, the source code of the comparison units is transformed into a proper intermediate representation of comparison if the comparison technique is not textual. The process of transformation of the source code into an intermediate representation is often called *extraction* in the reverse engineering community [15]. In order to detect different clone types, the clone detection tools support further normalizing transformations following extraction. The normalizations can be very simple, such as removal of whitespace and comments [16], or can be complex, involving source code transformations[17]. These normalizations are done either before or after the extraction of the intermediate representation.

2.2.2.1 Extraction

In this step, the source code is transformed into an appropriate input to the actual comparison algorithm. The extraction is usually performed in one or more of the following steps, depending on the tool:

A. **Tokenization:** if the clone detection tool uses a token-based approach, then each line of the source code is partitioned into tokens that are consistent with the lexical rules of the programming language of interest. After that, token sequences are formed from the lines or files to be compared. Also, when the token sequences are formed, all the whitespaces and comments between tokens are removed from the token sequences. CCFinder [18] and Dup [16] are clone detection tools that use the token approach.

B. **Parsing:** in a syntactic approach, the source code is parsed in order to structure an abstract syntax tree (AST). The subtrees of the parse tree, or the AST, are represented as the source units to be compared, and the comparison algorithms search for similar subtrees to identify them as clones [19-21].

C. **Control and Data Flow Analysis:** Semantics-aware approaches produce program dependence graphs (PDGs) from the source code. The PDG nodes represent the statements and conditions of a program, and the PDG edges represent control and data dependencies. The subgraphs of these PDGs represent the source units to be compared. In order to find clones, these techniques look for isomorphic subgraphs [22, 23].

2.2.2.2 Normalization

Normalization is an optional step that is meant to remove superficial differences such as whitespaces, comments, formatting or identifier names differences.

A. **Removal of whitespaces:** whitespaces are usually eliminated in all approaches, but line-based approaches keep line breaks. However, some metrics-based approaches utilize formatting and layout as part of their comparison. The indentation pattern of pretty-printed source code is used in Davey et al. [24]as one of the features of their attribute vectors, and the layout metrics as the number of non-blank lines are used in Mayrand et al. [25].

B. **Removal of comments:** comments are eliminated and ignored in most approaches. However, Marcus and Maletic [26] employ comments as part of their concept similarity method, and the number of comments is used in Mayrand et al. [25] as one of their metrics.

C. **Normalizing identifiers:** most approaches apply an identifier normalization before comparison to identify Type-2 clones. Generally, all identifiers in the source code are substituted by the same single identifier in this normalization. However, in order to detect consistently renamed Type-2 clones, Baker [16] uses an order-sensitive indexing scheme to normalize source code.

D. **Pretty-printing of source code:** the pretty-printing is used to remove the differences in layout and spacing in source code and to reorganize the source code to a standard form. Pretty printing is usually used in text-based clone detection approaches to find clones that vary in spacing and layout. Cordy et al. [27] use an island grammar [28] to generate a separate, pretty-printed text file for each potentially cloned source code.

E. **Structural transformations:** when the structural transformations are applied to change the structure of the source code, then the minor variations may be considered

similar to the same syntactic form [17, 18, 29]. For example, Kamiya et al. [18] eliminate keywords such as static from C declarations.

2.2.3 Match Detection

Once the source code is transformed, then the transformed source code is fed into a comparison algorithm in order to compare the transformed comparison units to each other to detect matches. Often adjacent similar units are combined to form larger units. A fixed-granularity clone unit belongs to either a function or block. When the fixed-granularity techniques/tools are used, the comparison units belonging to the same granularity are combined. For the free-granularity techniques/tools, aggregation is carried on as long as the given threshold is above the similarity of the aggregated sequence of comparison units, resulting in the longest possible similar sequences. A set of candidate clone pairs is formed by the match detection output. Each clone pair appears as the source matches of each of the matched fragments in the transformed code.

Others popular matching algorithms are also used in clone detection such as suffix trees [16, 17, 30, 31], dynamic pattern matching (DPM) [32, 33] and hash value comparison [19, 25].

2.2.4 Formatting

Once the transformed code is obtained by the comparison algorithm, the source code matches of each clone pair obtained in the comparison phase are mapped to their locations in the original source files.

2.2.5 Post-processing / Filtering

In the post-processing phase, the clones are classified or filtered using a manual analysis or automated heuristics. After extracting the original source code, manual analysis

can be performed on clones by human experts to filter out any false positive clones [34]. Also, the cloned source code can be visualized in a suitable format, such as an HTML web page [17], that can support the human expert to manually filter the clones. Automated heuristics are usually defined based on length, diversity, frequency, or other characteristics of clones, so post-processors automatically rank or filter out clone candidates [18, 35].

2.2.6 Aggregation

In this phase, the clones may be aggregated into clone classes to reduce the quantity of data to report on, the required time to analyze the data and to perform statistics calculations. In spite of some tools that directly identify clone classes [17], most of the tools return the results as clone pairs [18].

2.3 Overview of Clone Detection Techniques

In the literature, many clone detection approaches are proposed. These approaches can be primarily distinguished by the level of analysis applied to the source code. Consequently, these approaches are categorized into five main categories: textual, lexical, syntactic, semantic, and hybrid. In this section, we summarize the clone detection techniques by category.

2.3.1 Textual Approaches

The text-based clone detection techniques perform no or minor transformation/normalization on the raw source code before trying to detect identical or similar (sequences of) lines of code. Typically, whitespace and comments are ignored. A fingerprinting technique is applied by Johnson [36] to compare pieces of the source code. The fingerprinting is referred to as a short string that is used to stand in for larger data for comparison objectives. Ducasse and colleagues present [32] a language-independent clone

detection tool called *duplex* that does not require any parsing of the source code. NiCad [17] is a text-based clone detection technique that also exploits the benefits of a tree-based structural analysis from a lightweight parsing to perform flexible pretty-printing, code normalization, source transformation, and code filtering.

2.3.2 Lexical approaches (or token-based techniques)

A lexical approach starts by transforming the source code into a sequence of tokens using a compiler-style lexical analysis. These token sequences are searched for duplicated token subsequences, and then the original source code is returned as clones. Lexical approaches are considered more robust than textual approaches since they need minor code changes such as formatting, renaming, and spacing [16]. Kamiya et al. [18] have developed one of the leading lexical techniques tools that is called CCFinder. CCFinder uses a lexical analyzer to tokenize the source code and then groups tokens into sequences depending on the specified granularity level. The token sequences are then fed into a token-based clone detection algorithm to compare them token-by-token.

2.3.3 Syntactic Approaches

In a syntactic approach, the source code is converted into a parse tree or abstract syntax tree (AST) using a parser, and tree-matching measures or structural metrics are used to detect clones:

1. **Tree-matching approaches:** Once the source code is transformed into a parse tree for the target language, a tree-matching approach (or tree-based technique) detects clones by finding similar subtrees, and the corresponding code fragments are returned as clone pairs. In order to perform more advanced clone detection, variable names, literal values,

and other leaves (tokens) in the source codes may be abstracted in the tree representation. Many clone detection tools apply a tree-matching approach; for example, Baxter et al.'s CloneDr [19] is considered one of the pioneering tree-matching clone detection approaches. CloneDr uses metrics to compare subtrees based on a hash function.

2. **Metrics-based approaches:** In a metrics-based technique, a number of metrics for code fragments are gathered, and metric vectors are compared rather than source code or ASTs. One popular technique comprises fingerprinting functions where metrics calculated for syntactic units such as a class, function, method, and statement that produce values that can be compared to detect clones of these syntactic units. Generally, the source code is firstly parsed to an AST or control flow graph (CFG) representation, and metrics are calculated on this representation. A metrics-based technique is applied to detect duplicates. For example, Mayrand et al. [25] use several metrics to determine functions with similar metrics values as code clones. Names, layouts, expressions, and (simple) control flow of functions is used to calculate metrics. A function clone is identified as a pair of whole function bodies with similar metrics values.

2.3.4 Semantic Approaches

In a semantic technique, static program analysis is used to provide more accurate information than simple syntactic similarity. A program dependence graph (PDG) considers the semantic information encoded in the dependency graph that holds control and data flow information. Once the PDG is created for the source code, a subgraph isomorphism algorithm is used to find similar subgraphs, which are returned as clones. One

of the leading clone detection tools that applies a semantic approach is proposed by Komondoor and Horwitz [22]. It finds isomorphic PDG subgraphs using (backward) program slicing.

2.3.5 Hybrid Approaches

Hybrids approaches use a combination of syntactic and semantic characteristics to detect clones in the source codes. Leitao [37] uses the hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques that uses call graphs, in combination with specialized comparison functions.

2.4 Background on Some Refactoring Patterns

Many test cases in a test suite perform similar things. For example, we want to execute the same test with slightly different input values and then validate the actual output accordingly. Each of these test cases would have the exact same steps. These test cases are a perfect way to achieve good code structural coverage; however, it is not the right approach from a maintainability viewpoint because if we want to change the algorithm for one of these test cases, then we must make changes to all the similar test cases. Therefore, the best solution for reusing logic is to use refactoring patterns. For example, if we have two test cases with 100% similarity under the same test suite in two subclasses, then the most straightforward reuse logic patterns that we can use are the Extract Method in both subclasses to remove the duplication in the subclasses and then the Pull up method pattern. In the following paragraphs, we discuss the test patterns that we use in our work and that have been used by others when dealing with code clones (section 2.5).

Extract Method [2] is applied for clone fragments with 100% similarity; thus, we can group these fragments using the Extract Method pattern and eliminate the test code duplication.

Pull Up Method [2] is applied when the clone fragments perform similar work under the same test suite. Then we use *Extract Method* to group these fragments and eliminate test code duplication, and then we pull up the method to superclass using *Pull Up Method***.** After that, we can call the method from the superclass in any location in the test suite

Test Utility Method [38] is applied to clone fragments that have similar test logic in many test cases; We move the test logic that appears in more than one test case into a *Test Utility Method*; after that, we can call this method from various tests within a single test. If there is any variation in the test cases, we can pass these variations as arguments to the *Test Utility Method*.

Parameterized Test [38] is applied to test cases that have similar test logic to reduce test code duplication. The *Parameterized Test* pattern is used when the whole test cases have the same test logic; this means the fixture setup, exercised SUT, result verification, and fixture teardown have similar test logic. To apply the *Parameterized Test* pattern, we use the utility method to factor the common logic that takes only the information that differs from test to test as its arguments. Since the test parameters vary from test to test, and the *Parameterized Test* requires these parameters to run the tests, these parameters are passed into the test methods. The *Parameterized Test* pattern helps to reduce the code maintainability and achieve good coverage; also, it helps the testers to add more tests as required.

2.5 Related work

We discuss the related work of this thesis. In section 2.5.1, we discuss work related to overlapping information in clone detection results. In section 2.5.2, we discuss work regarding clone detection studies in industry. In section 2.5.3, we discuss the related work on clone refactoring.

2.5.1 Related work on Overlapping Information in Clone Detection Results

Source code clone research is a very active field which includes, among other things, research on technologies, algorithms and tools for clone detection (e.g., [39]), studies on the harmfulness of clones (e.g., [40]), studies of the relation between clones and vulnerabilities (e.g., [41]), technologies, algorithms and tools for code clone refactoring (e.g., [42]). This section is not meant to discuss all aspects of research on source code clone; rather we wish to focus on work that discusses or studies topics similar to the overlapping information in clone detection results, work we can compare to, or work on the prevalence of source code clones since such aspect of code clone is necessarily related to prevalence of overlap in clone detection results.

A search for related work about what we refer to as overlapping of clone data between results for different types of clones or detection at different threshold values has been unsuccessful. There is, therefore, no work we can really compare to. There are two exceptions. First, Islam and Zibran [10] recognize, though without referring to the phenomenon as we do, that Type-2 clone detection results contain Type-1 clones and Type-3 clone detection results contain Type-1 and Type-2 clones. They suggest to refine the notions of clone types by avoiding overlapping clone data, introducing the notions of Pure Type-2 Clones and Pure Type-3 Clones as follow: A set of Pure Type-2 clones (T_2^p)

includes only those Type-2 clones (T_2) that do not exhibit Type-1 (T_1) similarity ($T_2^p = T_2 - T_1$); A set of Pure Type-3 clones (T_3^p) includes only those Type-3 clones (T_3) that do not exhibit Type-1 (T_1) or Type-2 (T_2) similarities ($T_3^p = T_3 - T_2$). (Since results of T_1 are included in results of T_2, there is no need to involve T_1 in the subset relation.) Second, Islam and colleagues [11] excluded Type-1 clone classes from Type-2 results as well as Type-1 and Type-2 clone classes from Type-3 results when exploring the effects of bug-fix changes between clone and non-clone code identified by NiCad over thousands of revisions of seven subject systems written in two different programming languages (Java and C). Both sets of authors do not precisely describe the procedure they followed to remove overlapping clone data from detection results; We do discuss a precise procedure. They also limit the notion of overlapping clone data while we provide a more extensive discussion. Lastly, they do not study the extent to which clone detection tools produce overlapping clone detection data.

2.5.2 Related work on Clone Detection Study in industry

Various studies of clones in industry and open-source systems have been reported in the literature, though mostly on production, application logic code. In our study, we focus on studying clones in the industry test code instead of the production code. We expect test code to be different from production code, for instance, because they tend to exhibit different structures: a test typically has a set-up phase, an execution part, an oracle evaluation and a tear-down. We, therefore, expect to have different clone rates in (industry) test code than in (industry) production code.

Many different studies have reported findings of code clone rates in the industry. Baker [43] performed an empirical study using the Dup clone detection tool on two large software

systems consisting of MLOC (millions of lines of code). They found a clone rate of 19% in the X windows system, which includes about 700 KLOC of C code. In a production system that consists of 1.1 MLOC, they found a clone rate of 20%. The smallest clone code fragment was 30 LOC long. Lague et al. [44] conducted an industry study to discover the benefits of using a clone detection technology in the industry software development process and identified different ways to optimize the development process based on clone data. They analyzed a large telecommunication system which consists of nearly 15 MLOC, including comments, over three years. They found that the clone rate stayed at a relatively constant level over time, ranging between 6.4% and 7.5%. Baxter et al. [19] introduced the first syntax-tree clone detection tool and applied it to identify duplications in an industry C system of 400 KLOC. They found about 12.7% cloned code overall in all the studied subsystems and 28% cloned code in three subsystems, indicating the clone rate may vary for different (sub-) systems.

Dang et al. [45] shared their experience in the design and development of a clone detection tool called XIAO at Microsoft and provided feedback from Microsoft engineers after using XIAO. They found that once engineers understand the identified clones, they were able to reduce the number of near-miss clones.

Tüzün and Er [46] performed an empirical study on large-scale industrial software systems to provide clone categorization and emphasized the effectiveness of using clone management in the industry. The targeted framework consists of 677 Java files with a total of 97,473 LOC. The clone rate they measured was 22%, and they found more than half of the files (360 files) have at least one clone. Also, they showed that it was challenging to automatically refactor all clones because of high false-positive rates, suggesting human

interaction is necessary for the refactoring process. Merlo et al. [47] reported on lessons learnt on their research experience of using the CLAN clone detection technology in an industry-size telecommunication system of about 94 MLOC written in C/C++ and Java. They did not, however, report on specific values of identified clone rates. Duplicated test code is commonly referred to as test smells or test clones. Bavota et al. [40] studied the impact of test smells in the context of test cases manually written by developers, demonstrating that test smells are spread widely in the source code, and have massive effects on software maintenance. 82% of JUnit classes in their dataset were affected by at least one test smell and that the presence of design flaws had a strong negative impact on maintainability. In our work, we do not study the presence of test smells in test cases, but we study test clones.

Also, many different studies have conducted empirical studies on open source systems and they reported their findings of code clone rates. Ducasse et al. [32] presented a textual, language-independent approach to detect clones in code. They applied this technique to four software systems written in four different languages, namely C, Smalltalk, Python, and COBOL; these systems consist of 751 KLOC in total. They report clone rates ranging from 8.7% to 59.3% of LOC. The smallest clone code fragment was identified to be 10 LOC. Roy and Cordy [1, 48] conducted empirical studies on twenty-eight open-source systems written in C, Java, C#, Python using their NiCad clone detection tool. They investigated different properties of code clones, including language, clone size, clone location, and clone density. Overall, exact-clone rates were 7.2% for Java, 6% for C#, 1.1% for C, and 7.4% for Python (percentages of LOC). The clone rate for near-miss clones

ranged from 2.8% to 24.9% depending on the specified threshold (they used 0%, 10%, 20% and 30%).

In another large-scale study, involving nine open-source projects and four clone detection tools (CCFinder, Deckard, CloneDR, and NiCad), Tsantalis et al. [49] proposed an approach for automatically assessing whether a pair of clones can be safely refactored without changing the behaviour of the program. The clone rate in the test code was between 14.3% and 81.9%, while the clone rate in application code was between 18.1% and 85.7%. They found that clones in production code tended to be more refactorable than clones in test code. They did not, however, attempt to explain why this was the case. Also, they discovered that clones with a close relative location such as the same method, type, or file tended to be more refactorable than clones in distant locations such as the same hierarchy, or in unrelated types. They found that Type-1 clones tended to be more refactorable than the other clone types, and clones with a small size tended to be more refactorable than clones with a larger size.

Since the average clone rate had been reported differently in different studies, Koschke and Bazrafshan [50] conducted an empirical study on 7,800 open-source projects written in C or C++, which consist of about 240 MLOC in total to estimate the average clone rate in open-source projects. They reported an average clone rate of 12% for open-source C and C++ programs; they found that only 20% of the projects did not have a Type-2 clone of at least 100 tokens; 44% of the projects had at least one Type-1 clone. Comparing their observations to previously published work, they concluded that clone rates reported in the literature were not representative of open-source projects.

When we compare our study to other studies, we find that other studies used either open source or industrial systems to analyze production code except for Tsantalis et al. [49], where they detected clones in production and test code of open source projects. In our study, we instead focus on clones in the industry test code.

2.5.3 Related work on Clone Refactoring

Researchers recommend managing code clones to eliminate their negative consequences. The clone refactoring approach is a potential technique of clone management. Fowler et al. [2] refer to refactoring as changing a software system's code to improve its internal structure while the external system behaviour is preserved. Clone detection and refactoring usually refer to two different tasks. We find that it is necessary to integrate these two tasks in order to enhance the maintainability of the software systems. The integration between these two tasks is usually referred to as clone refactoring. The clone refactoring refers to the merging of two or more clone fragments from the same clone class.

In the literature, we found many studies that have been done on clone refactoring. Recently, Mondal et al.'s [51] survey many studies that focus on the integration between clone detections and refactoring. We discuss these studies about clone refactoring in the following paragraphs.

Baxter et al. [19] proposed the first syntax-tree clone detection tool that detects block clones by generation ASTs (abstract syntax trees) of the C source codes and then generates macros for replacing groups of clone fragments. Their tool is capable of deciding whether the clone fragments are refactorable or not automatically. Also, this tool is capable of refactoring Type 1 and Type 2 clone pairs semi-automatically; however, they don't check

if the system behaviour is preserved after refactoring clone fragments. In our case, we want to refactor clone groups that consist of all clone types; therefore, this study is different than our work since this study is only capable of semi-automatically refactor clone pairs for Type 1 and Type 2.

In 1999, Balazinska et al. [52] proposed a clone refactoring tool called CloRT (Clone Reengineering Tool). CloRT detects Type 1 and Type 2 clones by generation ASTs (abstract syntax trees) of the Java source codes, and then uses the strategy design pattern, which is proposed by Gamma et al. [53], to automatically factorize the common parts of the cloned methods and parameterize their differences. The method clones are classified into three categories, which are identical clones, clones showing superficial differences (names of local variables, names of parameters and name of the method), and clones differing in the use of non-local variables. The authors used their previously proposed tool that is called SMC [54] to classify the detected clones into these three categories. The authors selected 28 clone methods; after that, they grouped these clone methods into 11 clone classes that fit these three categories. They have successfully refactored the clones, but their refactoring process increased the source code size. In our study, the test cases will have many different variations, so we want to refactor clone groups for all clone types; however, this study is only capable of automatically refactoring clone methods of Type 1 and Type 2 with minimum differences.

In 2001, Koni-N'Shapu et al. [55] proposed a tool called SUPREMO to assist programmers in semi-automatic refactoring of code clones. SUPREMO refactoring tool used DUPLOC [32] to detect clones in the source codes. Since DUPLOC can only detect Type 1 and Type 3 clones, then SUPREMO can only consider Type 1 and Type 3.

SUPREMO only shows programmers the code clones; after that, the programmers manually assess if the clone pair is refactorable; then, the programmers decide which refactoring pattern should be applied for refactoring the code pairs. After that, the programmers apply the refactoring patterns manually. SUPREMO does not offer tool support for applying any refactoring pattern. SUPREMO was used for refactoring clones in SMALLTALK, C++, and Java systems. In our study, we want to use a clone refactoring tool that is capable of refactoring all the clone types and a group of clones. Since SUPREMO is only capable of refactoring Type 1 and Type 3 clone pairs, our study is different than this work.

Higo et al. [56] proposed a clone refactoring tool that is called CCShaper. CCShaper analyzes the code clone that is detected by CCFinder. Since CCFinder is only capable of detecting Type 1 and Type 2 clones, then CCShaper is only capable of analyzing and refactoring Type 1 and Type 2 clones. CCShaper cannot asses the refactorability of code clones. CCShaper analyzes code clones detected by CCFinder and automatically identifies structural blocks from clone fragments. These blocks are considered appropriate for refactoring. After that, the programmers make a decision to refactor based on these structural blocks that are extracted from the code clones. The authors used CCShaper to analyze a Java open source and then applied only two patterns: Extract Method refactoring and Pull Up Method refactoring on the structural clone blocks. In our study, we want to refactor the clone groups that consist of all clone types; however, this study focuses on refactoring Type 1 and Type 2 clones only.

Higo et al. [57] proposed another clone refactoring tool ARIES which is implemented on the top of their previous tool CCShaper. ARIES have a significant improvement over

CCShaper. ARIES works by generating different metrics that help to determine the appropriate clone refactoring opportunities in Java systems only. The programmers manually need to decide to refactor the structural blocks based on the different generated metrics. ARIES is capable of helping the programmers to apply the following refactoring patterns: Extract Method, Pull Up Method, Extract Class, Form Template Method, Move Method, Parameterize Method, and Pull Up Constructor. The authors used CCFinder to detect clones in Apache Ant, and they found 154 clone classes; after that, they used the ARIES tool to refactor the clone classes. The results of their refactoring show that they applied the Extract Method to refactor 52 clone classes, and they applied the Pull Up Method to refactor 12 clone classes. In our study, we want to refactor the clone group that consists of all clone types in C language; however, ARIES is only capable of refactoring Type 1 and Type 2 clones, so our study is different than this study.

In 2009, Li and Thompson [58] presented a hybrid approach, which is a combination of token-based and AST based techniques to detect code clones in Erlang/OTP programs. Their tool called Wrangler integrated with Emacs and Eclipse. Wrangler works by offering semi-automatic help for refactoring Type 1 and Type 2 clones. The programmer highlights the code clones in the IDE; after that, Wrangler checks if the clone fragments can be safely refactored. This tool is capable of handling the following refactoring patterns: Generalize a function definition, extract function, and fold expressions against a function definition. Thus, the tool cannot automatically decide if these clone fragments are refactorable or not; the programmers need to do it manually. In our case, we want to refactor clone groups that consist of all the clone types in C language. Since Wrangler is only capable of refactoring

Type 1 and Type 2 clones and cannot refactor clone in C language, our work is different than this work as well.

Brown and Thompson [59] proposed a clone refactoring tool called HaRe (Haskell Refactorer). This tool automatically performs clone detection in Haskel source codes by using an AST based technique; After that, HaRe semi-automatically refactors Type 1 and Type 2 clones by applying an integrated refactoring technique. HaRe cannot automatically decide if these clone fragments are refactorable or not; the programmers need to do it manually. This tool is capable of identifying the following refactoring patterns: Function Folding, As-Pattern Folding, and Merging refactoring patterns. In our study, we want to refactor clone classes that contain all the clone types in C language; however, this study is different than our study since HaRe is only capable of refactoring Type 1 and Type 2 clones and cannot refactor clone in C language.

Tairas and Gray [60] developed a clone refactoring tool called CeDAR (Clone Detection, Analysis, and Refactoring). CeDAR is an Eclipse plug-in that combines the detections and refactoring of code clones in Java systems only. The authors developed a complete system for clone maintenance by combining the existing techniques of code clone detections and refactoring. CeDAR detects Type 1 and Type 2 code clones using the DECKARD clone detection tool [61]; after that, CeDAR passes the clones to Eclipse refactoring engine. The factoring engine analyzes the code clones to decide which is suitable for refactoring and then automatically refactor the code clones. The authors had evaluated the CeDAR clone refactoring tool in eight open-source systems; the programmers were able to apply the following refactoring patterns: Extract Method, Pull Up Method, and Introduce Utility Method. The results of the evaluation show that CeDAR

is capable of refactoring clones by providing a higher number of refactorable clone classes to the programmers. In our study, we want to refactor clone classes that consist of all the clone types in the C language. Since CeDAR is only capable of refactoring Type 1 and Type 2 clones and cannot refactor clones in C language, our work is different than this work as well.

Fontana et al. [62] proposed a clone refactoring tool called DCRA (Duplicated code refactoring advisor). DCRA is implemented to identify the clone refactoring opportunities in Java programs. DCRA was built on top of the NiCad clone detector; also, it includes a Clone Detailer module that determines the required information for refactoring clone-pairs. DCRA supports the refactoring of Type 1 and Type 3 clones. Although DCRA supports the refactoring of Type 3 clones, it cannot refactor Type 2 clones; however, the Type 3 clones that it can refactor is obtained from Type1 clones. Thus, DCRA primarily is only capable of refactoring clone-pairs with 100% similarity. The DCRA works as follows: it detects Type 1 and Type 3 clone-pairs using NiCad; after that, the Clone Detailer module analyzes a clone-pair and provides a ranked list of possible refactoring patterns for the clone-pair. The programmer decides which refactoring pattern wants to apply by selecting the most suitable patterns for refactoring a clone-pair. DCRA supports seven clone refactoring techniques, which are Extract Method, Replace Method with Method Object, Merge Method, Pull Up Method, Pull Up Method Object, Form Template Method, and Leave Unchanged. In our study, we want to refactor clone classes that consist of all the clone types in C language. Since DCRA is only capable of refactoring Type 1 and Type 3 clone pairs and cannot refactor clone in C language, our work is different than this work as well.

Meng et al. [63] implemented a clone refactoring tool that is called RASE. RASE is an entirely automatic clone refactoring tool that can automatically asses the refactorability of Type 1 and Type 2 clones and automatically refactors Type 1 and Type 2 clones in Java systems only. Although RASE automatically performs refactoring, the authors reported that there were cases where automatic refactoring was impossible. Only the experienced programmers are capable of performing refactoring in such cases. RASE can apply six refactoring operations: extract method, add a parameter, parameterize type, form template method, introduce return object, and introduce exit label. It can help us refactor Type 1 and Type 2 clones. In our study, we want to refactor clone classes that consist of all the clone types in C language. Since RASE is only capable of refactoring Type 1 and Type 2 clone pairs and cannot refactor clone in C language, our work is different than this work as well.

Mazinanian et al. [42] presented a clone refactoring tool called JDeodorant, which was implemented as an Eclipse plug-in. JDeodorant is capable of importing the clone detection results from five clone detectors, which are CCFinder, DECKARD, CloneDR, NiCad, ConQAT. JDeodorant provides semi-automatic support for refactoring Type 1, Type 2, and Type 3 clone in Java systems only. JDeodorant works by allowing the programmer to select a clone-pair; after that, JDeodorant automatically checks the eight preconditions to determine if the refactored code will preserve the original behaviour. If the eight preconditions are met, the tool is capable of refactoring the clone-pair, and the original system's behaviour is kept. However, if any of the eight preconditions are violated, the tool cannot refactor the clone-pair because the original behaviour of the system cannot be preserved. Checking preconditions is particularly essential when refactoring Type 2 and Type 3 clones. If the tool can refactor the clone-pair, then the tool automatically displays a

preview to the programmer containing all the changes that will occur to the code after refactoring. The programmer can then select the particular refactoring. JDeodorant also provides support for applying many refactoring patterns including extract method, pull up method, create a template method, introduce utility method, and parameterize method. An updated version of the JDeodorant is proposed [49] that supports a new refactoring technique called Lambda Expressions. This new refactoring technique significantly increases the number of refactorable clones. In our study, we want to refactor clone classes that consist of all the clone types in C language. Since JDeodorant is only capable of refactoring a clone-pair for all clone types in Java systems, our work is different than this work as well.

2.6 Comparison of Clone Detection tool

Clone detection tools are used widely to detect similar pieces of source code within software code. Developers use clone detection results to enhance and maintain software quality, reduce development risks, prevent and detect bugs, among other things [64]. Many clone detection tools have been introduced. In 2009, Roy et al.'s [65] surveyed 40 clone detection tools. In 2013, Rattan et al.'s [66] systematic review found at least 70 clone detection tools; there is a 75% increase in developing new clone detection tools between 2009 and 2013. Even with a large number of clone detection tools found, there have been a few evaluation and comparison studies of those clone detection tools. We discuss these few comparisons in the following paragraphs.

The first clone detection tools comparison was conducted by Burd and Bailey [67]. They compare three clone detection tools, CCFinder, CloneDR and Covet and two plagiarism detectors JPlag and Moss. The authors manually validated all the clone results

of a Java subject system that was obtained by all the tools. After that, the authors relied on a human oracle that was used to compare the clone detection tools in terms of precision and recall. Precision refers to the percentage of true positives (clone pairs/ classes) within a set of code pieces identified by the tool as clones. Recall refers to the percentage of true positives that are retrieved by the tool within the complete set of known clones. The result of their experiment showed 100% precision for CloneDR, recall, however, was very low (9%). The CCFinder showed the highest recall which is 72% and reasonable precision which is 72%. The Covet showed the lowest precision which is 63% compared to the other tools.

Another clone detection tools comparison is conducted by Koschke et al. [34]. They evaluated their AST-suffix-tree-based tool that is called cpdetector in terms of precision, recall and runtime against existing tools, which are dup, cloneDR, CCFinder, duplix, CLAN, duploc, ccdiml, clones and cscope. They used Bellon's experimental procedure [12] (which we discuss next) and they only used the four C systems of Bellon's experiment because their tool supports only C language. The results show that the recall of clones, cscope, and cpdetector is 30%, 33% and 26%, respectively. The ccdiml records higher recall that is equal to 53%. Also, their experiment result shows that the clones tool found 71% more clones than CCFinder.

Another clone detection tools comparison is conducted by Bellon et al. [12]. They conducted a comprehensive quantitative evaluation of six clone detectors on a number of open-source software systems which are written in C and Java. The clone detection tools are CCFinder, CloneDR, CLAN, dup, duplix, and duploc. They found that precision and recall were complementary for each of the tool except for duplix, where precision and recall

are the lowest. The CloneDR and the CLAN had high precision, but their recall was very low. The dup, CCFinder and duploc had the highest recall, but the precision was low. Also, when they compare the execution time for the tools, the experiment shows that the duplix performed very badly comparing to the other clone detection tools.

Roy and Cordy [68] proposed a mutation/injection-based framework for evaluating clone detection tools. The mutation/injection-based framework randomly generates mutated clone fragments from the original code base, and randomly injects these mutants into the codebase to obtain mutated code bases. They executed the evaluation framework to compare the different variants of NiCad that are Basic NiCad, FlexP NiCad and Full NiCad. Their study shows that the Basic NiCad has a poor recall for clones generated by mutants of Type-2 clones. FlexP NiCad also has a poor recall for clones generated by mutants of Type-2 clones. Full NiCad has much better recall to find clones generated by mutants of all types of clones. Full NiCad had 100 percent recall and over 96 percent precision for Type-2 and Type-3 generated clones.

The most recent study that compares clone detection tools is conducted by Svajlenko and Roy [69]. They evaluate and compare the recall of eleven modern clone detection tools which are CCFinderX, ConQat, CPD, CtCompare, Deckard, Duplo, IClones, NiCad, Scorpio, SimCad and Simian. They used four benchmark frameworks to validate the performance of the eleven clone detection tools. Their study indicates that all the clone detection tools have a very high recall when detecting Type-1 and Type-2 clones. iClones and NiCad have a near-perfect recall for C and Java languages.

Sajnani et al. [70] proposed a token-based clone detector called SourcererCC that can detect exact and near-miss clones from a large repository. They evaluated the scalability,

execution time, recall and precision of SourcererCC against existing tools, which are CCFinderX, Deckard, iClones and NiCad. Their results show that SourcererCC and NiCad have both high recall and precision; also, SourcererCC is able to scale to a large repository, but NiCad has a poor execution time for larger inputs.

Wang et al. [71] implemented a novel clone detecting technique called CCAligner. Also, they extend the tool evaluation experiment in the SourcererCC publication by Sajnani et al. [70]. They execute these evaluations for their CCAligner, under the same conditions used by Sajnani et al. [70]. CCAligner has a perfect or near-perfect recall for all three clone types. NiCad and SourcererCC perform similarly. CCAligner performs better than iClones and performs much better than CCFinderX and Deckard. CCAligner has similar precision to SourcererCC. Also, CCAligner has competitive scalability amongst the tools.

Based on the above studies, we observe that these studies have used different clone detection tools to evaluate and compare; therefore, we have to create a list of characteristics that includes which criteria we need to have in the clone detection tool. Since we will run the clone detection tool on test code, and we know that the testers tend to copy and paste test cases and modify the test cases to meet some requirements, then we need a clone detection tool that has a strong capability at detecting near-miss clones. Also, we want the clone detection tool to be able to detect clones in different languages. Also, we want to use a clone detection tool that is used widely in the clone detection community. To do that, we have focused our search on the most recent studies. We started this project in 2016, so we have looked at the most recent studies which are Roy and Cordy [68], Svajlenko and Roy [69], and Sajnani et al. [70]. We noticed that Roy and Cordy [68] found Full NiCad had 100 percent recall and over 96 percent precision for Type-2 and Type-3 generated clones.

And Svajlenko and Roy [69] evaluate and compare the recall of eleven modern clone detection tools; their study indicates that all the clone detection tools have a very high recall when detecting Type-1 and Type-2 clones. iClones and NiCad have a near-perfect recall for C and Java languages. The most recent study that is performed by Sajnani et al. [70] show that SourcererCC and NiCad has both high recall and precision; Consequently, we found that NiCad has, in all the three studies, high recall and precision compared to other clone detection tools; therefore, we have decided to used the NiCad clone detection tool in our work.

2.7 NiCad tool

NiCad is a hybrid clone detection tool that combines a language-sensitive parsing, currently supporting C, Java, C# and Python, with a language-independent similarity analysis to accurately detect exact and near-miss clones: We used NiCad version 4, the latest version available to us when we conducted the study. NiCad can detect both exact and near-miss clones at the function or block levels of granularity. We selected NiCad because it has been widely used in the clone detection community, and it has a record of both high precision and high recall at finding near-miss clones [68], using its default settings [65].

Table II: NiCad settings

Comment	Threshold	Renaming
Type-1	0%	None
Type-2 Blind	0%	Blind
Type-2 Consistent	0%	Consistent
Type-3	30%	None
Type-3 Blind	30%	Blind
Type-3 Consistent	30%	Consistent

NiCad must be run with a configuration setting, including a granularity (block or function), the minimum size of a code fragment to be considered as a clone, the maximum size of a code fragment to be considered as a clone, a threshold value for identifying differences between code fragments, a renaming procedure (either none, blind or consistent). A minimum fragment size of three LOC and maximum fragment size of 2,500 have shown to lead to NiCad results with high precision and high recall [68]. The granularity is set to "function" in the default NiCad configuration, and has shown to work well too; detected clones are therefore functions. Values of the other configuration inputs are typically set as indicated in Table II, and have also been shown to provide high precision and high recall.

Chapter 3: Extending the Taxonomy of Clone Detection Types to Account for Overlapping Information in Clone Detection Results

Source code is known to contain pieces that are (very) similar to each other, sometimes so much similar that they are identical. These similar pieces of code are called clones, and clones have been classified according to the level of similarity between pieces of code: Type-1 clones, Type-2 clones … Following standard definitions of clone types, if the source code contains fragments that are in a Type-1 clone class, then those fragments also belong to a Type-2 clone class; Further, assuming the source code contains fragments that belong to a Type-2 consistent clone class, these fragments will also appear in a Type-2 blind clone class; Such inclusions also occur with Type-3 clone classes, with different threshold values to detect differences. When one relies on standard definitions of source code clone types, they are therefore running a risk of overlapping information being returned for different types of clone classes. Further, if one interprets clone type definitions in a somewhat strict manner, whereby, for instance, fragments in a Type-1 clone class should not belong to a Type-2 clone class, the inclusions we just mentioned can be qualified as false-positives. We argue that understanding and reporting on the prevalence of overlapping clone information is important to make a well-informed decision when dealing with clones (e.g. deciding to refactor, estimating refactoring cost). In this chapter we first illustrate the notion of overlapping clone detection data on a real example (Section 3.1). Building on the work of Islam and Zibran [10], we extend the notion of Pure Types of clones by introducing new terminology for clone types and account for various parameters one uses when searching for clones such as renaming conventions and allows thresholds

of differences (Section 3.2). For each traditional clone type, including renaming conventions or threshold values, we define a corresponding *Pure* version that only includes clone classes for that type and omits clone classes that can be discovered by more permissive clone types.

We then define a generic Clone Reduction Procedure to remove overlapped information. It is generic as it does not depend on any specific clone detection tool (Section 3.3). We implement this procedure to work on clone data returned by NiCad, a very well-known and effective clone detection tool.

Finally, we experiment with our Clone Reduction Procedure on real open-source projects to study the prevalence of overlapping clone detection information (Sections 3.4 and 3.5). We used NiCad on nine C open-source subject programs, and nine Java open-source subject programs, that others have used [1] and that are part of a standard benchmark to compare clone detection technologies [12]. Our results confirm the presence of overlapping clone information, sometimes in tremendous proportions. For instance, for the Java subjects: reduction percentage results indicate that clone detection results for threshold value, 10% are 82% to 99% redundant with results obtained for threshold value 0%. NiCad produces less overlapped data for C subjects: reduction percentage results indicate that clone detection results for threshold value 10% are 16% to 75% redundant with results obtained for threshold value 0%. We argue that clarifying this is as important as removing false positives in static analysis to get accurate results – and the differences in the different languages also show that this is strongly related to static analysis aspects, where the tool results differ based on its implementation.

Last, we discuss threats to validity for this experiment, followed by some conclusions in Sections 3.6 and 3.7, respectively.

3.1 Illustrating Overlap Clone Information

As discussed in the previous section, Type-2 clone classes are also considered Type-3 clone classes, and we call the intersection between Type-2 clone classes and Type-3 clone classes overlapping clone data. Similarly, there are overlapping clone data in the set of Type-3 clone classes with the Type-1 and Type-2 clone classes. Illustration 2 analytically illustrates this overlap of information between clone classes with a Venn diagram: Some Type-1 clone class fragments belong only to Type-1 clone classes (point A); however, some Type-1 clone class fragments also belong to Type-2 Consistent classes (point B), Type-2 Blind classes (point C), or Type-3 classes (point D); some Type-1 clone class fragments also belong to Type-2 Consistent classes and Type-2 Blind classes as well (point E); some Type-1 clone class fragments also belong to Type-2 Consistent and Type-3

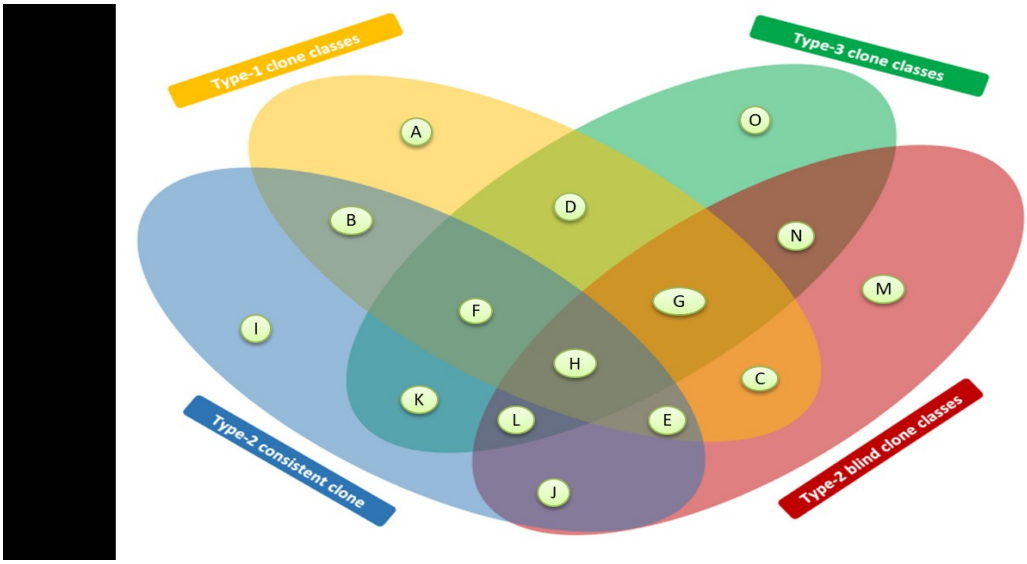


Illustration 2 Venn diagram illustrating some situations of overlapping clone data

classes as well (point F); some Type-1 clone class fragments also belong to Type-2 Blind and Type-3 classes as well (point G); and some Type-1 clone class fragments belong to Type-2 Consistent, Type-2 Blind, and Type-3 classes as well (point H). Similarly, some Type-2 Consistent class fragments belong only to Type-2 Consistent (point I), and some of them also belong to Type-2 Blind (J) or Type-3 (point K) classes; and some Type-2 Consistent clone class fragments belong to Type-2 Blind and Type-3 classes as well (point L); some Type-2 Blind clone class fragments belong only to Type-2 Blind (point M), and some of them also belong to Type-3 classes (point N); some Type-3 clone class fragments belong only to Type-3 (point O). The Venn diagram shows that there are several situations of overlapping clone information for different clone types. A more complex diagram could be produced to account for Type-3 Consistent and Type-3 Blind classes, but it would be less legible; we feel the current diagram is enough to illustrate the overlap of information graphically.

To concretely illustrate this overlap in clone detection information on real pieces of code, we used NiCad version 4, the most recent version of NiCad at the time we conducted this study, to detect clones in one open-source subject program, namely Eclipse-ant; we used the same version of Eclipse-ant as the one others have used before us, and that is part of a clone detection benchmark available online [12]. We used NiCad to find clones in Eclipse-ant's code, using NiCad's default configuration, with threshold values 0%, 10%, 20% and 30%.) The rationale for proceeding this way is to allow the reader to easily replicate the study and be convinced we have not introduced any bias in the execution of the example.

Table III shows an excerpt of the results produced by NiCad. In the first column, we give a name to the different pieces of information for easy reference in the text below. The second column shows the name of an XML file generated by NiCad. The file name also shows the threshold value that was used when executing NiCad. For example, the file name "src_functions-clones-0.10-classes.xml" contains results produced by NiCad with a threshold value of 10%. The third column shows an example clone class we extracted from the corresponding file on that row of the table. (To facilitate reading, we re-arranged the XML tags so they all appear in the same order.)

The result in file "src_functions-clones-0.00-classes.xml" (clone class we refer to as CC-1) indicates a clone class of two code fragments (nclones = "2") starting and ending at lines 93 and 95, and 163 and 165, respectively, both in the same file. Since the reported similarity of the fragments (similarity = "100") is one hundred percent, these are identical

Table III An Example of Overlapping Information between Clone Classes

Clone class	Filename	Clone class
CC-1	src_functions-clones-0.00-classes.xml	<class similarity="100" nlines="4" nclones="2" classid="1"> <source startline="93" endline="95" pcid="44" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java" /> <source startline="163" endline="165" pcid="54" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java" /> </class>
CC-2	src_functions-clones-0.10-classes.xml	<class similarity="100" nlines="4" nclones="2" classid="1"> <source startline="93" endline="95" pcid="44" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> <source startline="163" endline="165" pcid="54" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> </class>
CC-3	src_functions-clones-0.20-classes.xml	<class similarity="100" nlines="4" nclones="2" classid="1"> <source startline="93" endline="95" pcid="44" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> <source startline="163" endline="165" pcid="54" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> </class>
CC-4	src_functions-clones-0.30-classes.xml	<class similarity="100" nlines="4" nclones="2" classid="1"> <source startline="93" endline="95" pcid="44" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> <source startline="163" endline="165" pcid="54" file="/home/ewafhas/bellon-sources/systems/main/eclipse-ant/src/ant/BuildEvent.java"/> </class>

code fragments. This is expected since we asked NiCad to find clones by using a threshold of 0% (XML file name). This is a Type-1 clone class with two fragments. From Table III, we observe that, regardless of the threshold value (0% in CC-1, 10% in CC-2, 20% in CC3, 30% in CC4), NiCad returns the same clone class for fragments that are exact copies (similarity = ”100”): the same number of fragments, the same fragments in the same file. There is overlapping clone data information between results at threshold 0% and results at other, higher threshold values. Other results, not shown here, indicate that results returned by NiCad at threshold 30% include results returned at threshold 20%, which themselves contain results returned at threshold 10%, which themselves contain results returned at threshold 0%. This concrete example is G of the Venn diagram.

This is a concrete illustration of the notion of overlapping clone information which we have introduced earlier in this manuscript.

3.2 Revised Code Clone Terminology

When considering the standard definitions of clone types, we observe for instance that, by definition, if the source code contains fragments that are in a Type-1 clone class, then those fragments also belong to Type-2 and Type-3 clone classes; Further, assuming the code contains fragments that belong to a Type-2 Consistent clone class (Type-2 class with consistent renaming), these fragments will also appear in a Type-2 Blind clone class (Type-2 class with blind renaming). Such inclusions also occur with Type-3 clone classes. In summary, the results of Type-2 clone classes also include results of Type-1 clone classes, results of Type-3 clone classes include results of both Type-1 and Type-2 clone classes. Similarly, when making an operational decision to detect clones based on a similarity threshold value, one will observe inclusions between different clone detection results. For

instance, results when using a high threshold value for differences (say, for instance, 30%) will include results using a lower threshold value for differences (say, for instance, 20%).

Islam and Zibran [10] therefore defined Pure Type-2 Clones and Pure Type-3 Clones as follow: A set of ***Pure Type-2*** clones (T_2^p) includes only those Type-2 clones (T_2) that do not exhibit Type-1 (T_1) similarity ($T_2^p = T_2 - T_1$); A set of ***Pure Type-3*** clones (T_3^p) includes only those Type-3 clones (T_3) that do not exhibit Type-1 (T_1) or Type-2 (T_2) similarities ($T_3^p = T_3 - T_2$).

We extend these definitions to also account for identifier renaming:

Pure Type-2 Consistent Clones: A set of *Pure Type-2 Consistent* clones (T_{2C}^p) includes only those Type-2 Consistent clones (T_{2C}) that do not exhibit Type-1 (T_1) similarities: $T_{2C}^p = T_{2C} - T_1$.

Pure Type-2 Blind Clones: A set of *Pure Type-2 Blind* clones (T_{2B}^p) includes only those Type-2 Blind clones (T_{2B}) that do not exhibit Type-1 (T_1) or Type-2 Consistent (T_{2C}) similarities: $T_{2B}^p = T_{2B} - T_{2C}$. (Since results of T_1 are included in results of T_{2C}, there is no need to involve T_1 in the subset relation.)

Pure Type-3 Clones: A set of *Pure Type-3* clones (T_3^p) includes only those Type-3 clones (T_3) that do not exhibit Type-1 (T_1), Type-2 Consistent (T_{2C}), or Type-2 Blind (T_{2B}) similarities: $T_3^p = T_3 - T_{2B}$. (Since results of T_{2B} include results of T_1 and T_{2C}, there is no need to involve the latter two in the subset relation.)

Pure Type-3 Consistent Clones: A set of *Pure Type-3 Consistent* clones (T_{3C}^p) includes only those Type-3 Consistent clones (T_{3C}) that do not exhibit Type-1 (T_1), Type-2

Consistent (T_{2C}), Type-2 Blind (T_{2B}), or Type-3 (T_3) similarities: $T_{3C}^{p} = T_{3C} - T_3$. (Similarly, to previous cases, we only need to involve T_{3C} and T_3 in the relation.)

Pure Type-3 Blind Clones: A set of Pure *Type-3 Blind* clones includes only those Type-3 Blind clones that do not exhibit Type-1, Type-2 Consistent, Type-2 Blind, Type-3, or Type-3 Consistent similarities. Mathematically, $T_{3B}^{p} = T_{3B} - T_3$, where T_{3B}^{p} denotes the set of Pure Type-3 Consistent clones and T_{3B} denotes the set of Type-3 Blind clones.

The reader will note that our definitions of pure clone types do not refer to similarity threshold values for the identification of clones. Our definitions, similarly to the original clone type definitions (section 2.1) are independent from operational decisions one needs to make about threshold values. In other words, when using our definitions, one still needs to make operational decisions.

The original types of clone classes (section 2.1) are more or less permissive in the sense that they allow one (or a tool) to account for varying amounts of differences between code fragments when attempting to group them into a clone class. For instance, Type-2 is more permissive than Type-1, and Type-3 is more permissive than Type-2 and Type-1, which we note $Type3 \gg Type2 \gg Type1$. When factoring in a possible renaming, Blind is more permissive than Consistent, which is more permissive than no renaming: $Type3^{Blind} \gg Type3^{Consistent} \gg Type3^{None} \gg Type2^{Blind} \gg Type2^{Consistent} \gg Type2^{None} \gg Type1$. When factoring in different levels of differences when identifying Type-3 clones, using a 30% difference is more permissive than using a 20% difference, which is itself more permissive than using a 10% difference: $Type3^{30\%} \gg Type3^{20\%} \gg Type3^{10\%} \gg Type3^{0\%}$. The above "Pure" types allow us to distinguish what is obtained solely with a more permissive type, omitting what can be obtained with less permissive types. We note

$PureType3^{Blind} > PureType3^{Consistent} > PureType3 > PureType2^{Blind} > PureType2^{Consistent} > PureType2 > Type1$.

3.3 Removing Redundant Data in Source-Code Clone Detection Results

As discussed earlier, as illustrated below in this section and as indicated by our results (see experimental results next in this chapter), a clone detection tool such as NiCad produces overlapping clone detection information. In this section we discuss a general principle for our Clone Reduction Procedure, under the form of general algorithms (section 3.3.1). This procedure is one possible way to implement the definitions of Pure Types of clones presented earlier. Last, we show how the general procedure can be applied to general definitions of Type-1, Type-2 and Type-3 clones, with or without renaming, or for various similarity threshold values (section 3.3.2).

3.3.1 A General Procedure to Remove Overlapping Clone Information

To remove overlap information between clone classes, we suggest comparing code fragments in clone classes, to compare clone classes, and to compare clone data files as discussed below. Essentially, the procedure removes from a set of clone classes that is obtained with a more permissive type, the set(s) of clone classes obtained with the less permissive type(s). For instance, we remove from a clone class set obtained with Type-2, the clone class set obtained with Type-1, we remove from a clone class set obtained with a 30% difference, the clone class sets obtained with 20% difference and 10% difference. Reusing the notation we have introduced in the previous section, the procedure takes results that satisfy $Type3^{Blind} \gg Type3^{Consistent} \gg Type3^{None} \gg Type2^{Blind} \gg Type2^{Consistent} \gg Type2^{None} \gg Type1$ and generates clone class sets that satisfy

$PureType3^{Blind} > PureType3^{Consistent} > PureType3 > PureType2^{Blind} > PureType2^{Consistent} > PureType2 > Type1$.

The first step is to identify whether two code fragments are equal. We assume that a code fragment is uniquely identified by the following characteristics: a start line, an end line, and a file name. We consider that two fragments are equal if they have an equal start line, an equal end line, and an equal file name: Algorithm 1 in Illustration 3.

The second step is to identify whether two clone classes have equal fragments. We assume a clone class is defined by a similarity threshold, a size (i.e., number of fragments), and a fragment size (i.e., the number of LOC that are similar according to the threshold). All fragments in a clone class have the same number of LOC, between the same start line and the same end line. Each clone class has at least two clone fragments. We consider two clone classes are identical when the following conditions hold at the same time (Algorithm 2 in Illustration 3): they have equal sizes, their similarity values are equal, their fragment sizes are equal, and the fragments in the two clone classes are equal (see Algorithm 1). Note that, since fragments in a class have the same number of LOC between the same start and end lines, the two nested loops could be simplified as it is enough to compare one fragment of CC-1 (e.g., the first one) with one fragment of CC-2 (e.g., the first one); The nested loops is safer to accommodate for unexpected situations that could be returned by the clone detection tool that feeds data to our algorithm.

Third, we compare two clone files and remove from the second file, the clone classes that are in the first file: Algorithm 3 in Illustration 3.

3.3.2 Applying the Clone Reduction algorithm

As discussed earlier, overlap information can potentially be found when comparing different types of clone class results. Relying on previously defined algorithms, we remove overlap information from clone detection results and identify Pure Type-2 clone classes, i.e. Type-2 clone classes that are not Type-1 classes, Pure Type-3 classes, i.e., Type-3 clone classes that are not Type-2 nor Type-1 classes… We also account for renaming procedures.

```
Algorithm 1: Comparison of two clone fragments
Input: two clone fragments (cf-1 and cf-2)
Output: Boolean
function compareFragments(cf-1, cf-2)
  if (cf-1.startLine == cf-2.startLine && cf-1.endLine == cf-2.endLine &&
      cf-1.fileName == cf-2.fileName) then  return true;
  else return false;
end function
```

```
Algorithm 2: Comparison of two clone classes
Input: two clone classes (CC-1 and CC-2)
Output: Boolean
function compareCloneClasses(CC-1, CC-2)
  // CC-1.Fragments is the collection of fragments in clone class CC-1
  // CC-1.numberOfLines is the number of LOC of all the fragments in CC-1
  if (CC-1.Fragments.size != CC-2.Fragments.size || CC-1.similarity != CC-2.similarity
      || CC-1.numberOfLines != CC-2.numberOfLines) then
    return false;
  end if
  for each cfi in CC-1.Fragments, do // cfi is a clone fragment in CC-1
    Boolean found = false;
    for each cfj in CC-2.Fragments, do
      if (compareFragments(cfi, cfj)  then  found = true;  end if
    end for
    if (! found) then  return false;  end if
  end for
  return true;
end function
```

```
Algorithm 3. Comparison of two clone files
Input: two clone class files: file1, file2
Output: file2 without duplicated clone classes
function removeDuplicatedClasses (file1, file2)
  // file1.cloneClasses is the set of clone classes in file1.
  for each cpi in file1.cloneClasses do
    for each cpj in file2.cloneClasses do
      if (compareCloneClasses(cpi , cpj)), then  file2.cloneClasses.remove(cpj)
      end if
    end for
  end for
end function
```

Illustration 3 General Algorithms to Remove Overlapping Clone Information.

- Pure Type-2 Consistent: These are the Type-2 Consistent classes that do not match any Type-1 class.

  PureType2Cons = removeDuplicatedClasses(Type1, Type2Cons)

- Pure Type-2 Blind: These are the Type-2 Blind classes that do not match any Type-1 nor Type-2 Consistent classes.

  PureType2Blind = removeDuplicatedClasses(Type2Cons, removeDuplicatedClasses(Type1, Type2Blind))

- Pure Type-3: These are the Type-3 classes that do not match any Type-2 Blind class, nor Type-2 Consistent class, not Type-1 class.

  PureType3 = removeDuplicatedClasses(Type1, removeDuplicatedClasses(Type2Cons, removeDuplicatedClasses(Type2Blind, Type3)))

- Pure Type-3 Consistent: These are the Type-3 Consistent classes that do not match any Type-1 class, nor Type-2 Consistent class, nor Type-2 Blind class, nor Type-3 class.

  PureType3Cons = removeDuplicatedClasses(Type3, removeDuplicatedClasses(Type2Blind, removeDuplicatedClasses(Type2Cons, removeDuplicatedClasses(Type1, Type3Cons))))

- Pure Type-3 Blind: These are the Type-3 Blind classes that do not match any Type-1 class, nor Type-2 Consistent class, nor Type-2 Blind class, nor Type-3 class, nor Type-3 Consistent class.

  PureType3Blind = removeDuplicatedClasses(Type3, removeDuplicatedClasses(Type3Cons, removeDuplicatedClasses(Type2Cons,

removeDuplicatedClasses(Type2Blind, removeDuplicatedClasses(Type1, Type3Blind)))))

A similar procedure can be followed if different threshold values are of interest. If a clone class at a lower threshold value exists at a higher threshold value, then that class is excluded from the higher threshold value set (calls are omitted since they are similar to previous ones):

- Pure classes at threshold 10%: These are the classes obtained at threshold 10% that do not match any class obtained at threshold 0%.
- Pure classes at threshold 20%: These are the classes obtained at a threshold of 20% that do not match any class obtained at threshold 0% or 10%.
- Pure classes at threshold 30%: These are the classes obtained at threshold 30% that do not match any class obtained at threshold 0%, 10%, or 20%.

3.4 Experiment design

This section presents the design of our experiment. We discuss the subject systems we used (Section 3.4.1) and how we prepared them for analysis by using our overlap clone data reduction procedure we described earlier (Section 3.4.2).

3.4.1 Subject Systems

In this study, we have chosen the same subject systems that Roy and Cordy [1] have used to validate NiCad. This includes four C and four Java systems varying in size from 11K LOC to 204K LOC. These systems are also used in Bellon's experiment [12]. Table VI provides some details about each system (we only consider C and Java files in the

Table IV NiCad settings

Threshold	Renaming	Comment	
0%	None	Type-1	Same configurations as the one used by others [1],
10%	None	Type-2	
20%	None	Type-3	
30%	None	Type-3	

Min segment size: 3 Max segment size: 2,500 Granularity: Function

calculations). We downloaded the same subject software as the original study [1] from the original web site: http://www.softwareclones.org/research-data.php.

We used the same NiCad configurations as those used by Roy and Cordy [1]: see Table IV. They are the ones for which NiCad has been shown to perform well in terms of precision and recall (recall Section 2.7).

By using the same experimental subject software and the same NiCad configurations as Roy and Cordy, we make sure results can be compared. By ensuring the subject software has been extensively used by others, we make sure our choice of case studies is relevant to the field.

We also selected ten other open-source subject programs for analysis (Table VI): five are written in Java and five are written in C. They were selected because they are more recent, present a large codebase, are well known to users, and have been used by others [49, 60] as experimental subjects to evaluate several source code analysis techniques.

Also, we used all the test suites from two projects provided by our industry partner. The system being tested is a part of middleware in the telecom domain. We specifically focused on test code written in C. For obvious confidentiality reasons, technical details on the application code and test code cannot be revealed. Table V, however, provides some information about the test suites we analyzed. The total number of test suites in the two projects is 36, for a total of 427 C files, which are made of 286,168 LOC. The test suites of the two projects are identified with different background colours in the table.

The first project, project-1, is larger (more test suites, more test code) than the second project. We can also observe that project-1 has two test suites, TS-2 and TS-20, that have a significantly larger number of C files than the other test suites; TS-2 has the highest

Table V Test suites details

Test Suite	C-Files	LOC	Average LOC/ File	Test Suite	C-Files	LOC	Average LOC/ File
TS-1	7	1978	283	TS-19	2	527	264
TS-2	82	87818	1071	TS-20	50	22463	449
TS-3	19	9749	513	TS-21	1	48	48
TS-4	9	8672	964	TS-22	22	12303	559
TS-5	3	581	194	TS-23	12	2431	203
TS-6	10	2063	206	TS-24	1	478	478
TS-7	6	630	105	TS-25	3	2321	774
TS-8	5	6619	1324	TS-26	11	17030	1548
TS-9	34	9572	282	TS-27	4	6667	1667
TS-10	3	1138	379	TS-28	31	20608	665
TS-11	7	638	91	TS-29	6	5141	857
TS-12	10	11536	1154	TS-30	2	1100	550
TS-13	16	7566	473	TS-31	1	316	316
TS-14	6	3671	612	TS-32	13	944	73
TS-15	14	15306	1093	TS-33	5	2246	449
TS-16	15	11265	751	TS-34	3	2880	960
TS-17	7	8106	1158	TS-35	1	317	317
TS-18	5	964	193	TS-36	1	476	476
				Total	427	286168	

number of LOC. TS-2, TS-8, TS-12, TC-15, TS-17, TS-26, and TS-27 have files with on average more than 1000 LOC. TS-27 has 1600+ LOC per file on average.

3.4.2 Post-processing NiCad Results

In our experiment, since we are interested in the prevalence of overlapping information in clone detection results such as those produced by NiCad, we want to run NiCad and collect the results it produces, and then post-process those results with our clone reduction algorithm. For each subject program and for each clone type we select, we, therefore, obtain two sets of clones, with information on clone classes, clone types and clone fragments, which we compare: a set of clones as produced by NiCad and a set of clones after reduction.

3.5 Results & Discussion

Table VI Subject systems Details

	Language	Subject system	LOC
Previous study's subjects [1]	Java	Eclipse-Ant [3]	35K
		Eclipse-JdtCore [3]	148K
		j2sdk1.4.0-javax-swing [5]	204K
		netbeans-javadoc	14K
	C	Postgresql [6]	202K
		Snns [7]	94K
		weltab	11K
		Cook [8]	70K
New subjects	Java	Hibernate (3.3.2GA)	249K
		JFreeChart (1.0.10)	109K
		Apache Sqoop (1.5.8.10)	83K
		Jkarta Jmeter (2.3.2)	82K
		JRuby (1.4.0)	163K
	C	Curl (7.56.0)	54K
		Handbrake (1.8.12)	121K
		VLC (4.5.0.1)	472K
		Onion (2.5.3)	16K
		NetData (1.5.6)	48K

We first discuss results and then discuss results in more general terms, reflecting on conclusions one can draw from clone detection data before reduction and conclusions one can draw from clone detection data after reduction.

3.5.1 Results

Table VII and Table VIII show results for the subjects of the previous study and the new subjects, respectively (Table VI), with the four NiCad configuration settings we used (Table IV). Three figures in Appendix A at the end of this manuscript provide an alternative way to look at the data. For each subject and each threshold value Table VII has four groups of data: (i) previously published results on clone detection for the subjects and those NiCad configurations [1], (ii) results collected from our own execution of NiCad with the same NiCad configurations (no clone data reduction), (iii) results after using our clone reduction procedure, using the same NiCad configurations, and (v) percentages of overlap information when comparing results before and after executing our clone reduction procedure. Since we include new subjects, Table VIII has one less column; we do not have previously published results. For each clone detection procedure and percentage, each table indicates the number of clone pairs, the number of clone classes and the total number of fragments in those classes. (There is one exception: previously published results [1] do not report on numbers of fragments.) For instance, for subject Eclipse-Ant at threshold 10%, previous results indicate 365 clone pairs and 94 clone classes, our execution of NiCad identifies 252 fragments in a total of 93 clone classes and 364 pairs, and after reduction[2],

[2] A careful reader will notice some odd results in the table, such as those prior and after reduction for Eclipse-Ant. Prior to reduction, NiCad reports on 92 clone classes at threshold 0% and 120 clone classes at threshold 30%; After reduction, we report on 92 clone classes at threshold 0% and 1, 8 and 25 clone classes at threshold 10%, 20% and 30%, respectively. Given the transitive nature of overlapping information as the threshold increases, given our reduction procedure, the sum of the number of classes at all threshold values as reported after reduction should equal the number of clone classes at threshold 30% prior to reduction. This is however not the case: 92+1+8+25 does not equal 120. We investigated and discovered that (prior to reduction) NiCad reports fragments at threshold 0% that it should also report at threshold 30% but does not. We conjecture this is a glitch in NiCad, which we have shared with the main contributor of NiCad. The differences due to this glitch being small overall (a handful of fragments), they should not jeopardize our conclusions.

we find two fragments in one clone class (one pair). Illustration 4 shows a screenshot for Eclipse-Ant at threshold 10% after reduction; there are two fragments in one clone class. This results in a clone pair reduction of 99.7%, a clone class reduction of 98.9% and a clone fragment reduction of 99.2%.

We first notice slight differences between previously published results and our own executions of NiCad. Since we used the same subjects, the same subject versions, and the same NiCad configurations, we conjecture these slight differences are due to improvements to NiCad over the years. The previously published results were obtained with NiCad-3; in our experiment, we have used NiCad-4, which was the latest version available to us at the time we conducted the study. The differences being small, they should not impact our

```
<?xml version="1.0" encoding="UTF-8"?>
- <clones>
    <systeminfo threshold="10%" system="src" processor="nicad3" minlines="3" maxlines="2500"
      granularity="functions"/>
    <cloneinfo npcs="1944" npairs="364"/>
    <runinfo ncompares="462173" cputime="30000"/>
    <classinfo nclasses="93"/>
  - <class similarity="90" nlines="10" nclones="2" classid="1">
      <source startline="225" pcid="1746" file="/home/ewafhas/bellon-
        sources/systems/main/eclipse-ant/src/mail/MailMessage.java" endline="233"/>
      <source startline="315" pcid="1759" file="/home/ewafhas/bellon-
        sources/systems/main/eclipse-ant/src/mail/MailMessage.java" endline="323"/>
    </class>
</clones>
```

Illustration 4 After reduction result for Eclipse-Ant at threshold 10%: two fragments in one clone class.

conclusions. Results for threshold 0% are the same, before and after reduction because, at this threshold value, we do not execute the clone reduction procedure as there is nothing to compare against.

The results of applying the clone reduction procedure illustrate there is a lot of overlap information, whether we consider the number of clone pairs, the number of clone classes or the number of clone fragments. For instance, for subject netbeans-javadoc, pairs, classes and fragments obtained at threshold 10% are at least 96% overlapping with pairs, classes and fragments obtained at threshold 0%; It is at least 82% at threshold 20% and at least 74% at threshold 30%. The results also show a high overlapping of data regardless of

Table VII NiCad results before and after using the reduction algorithm for open source projects

Language	Subject system	Threshold	Clone Pairs	Clone Classes	Clone Pairs	Clone Classes	Fragments	Clone Pairs	Clone Classes	Fragments	Clone Pairs	Clone Classes	Fragments
			Previously Published Results [1]		*Before Using Clone Reduction*			*After Using Clone Reduction*			*Reduction Percentage*		
Java	Eclipse-Ant	0%	363	92	363	92	250	363	92	250	0.00%	0.00%	0.00%
		10%	365	94	364	93	252	1	1	2	99.7%	98.9%	99.2%
		20%	374	101	374	101	269	10	8	17	97.3%	92.1%	93.7%
		30%	426	119	428	120	319	54	25	74	87.4%	79.2%	76.8%
	Eclipse-JdtCore	0%	1427	323	1409	326	940	1409	326	940	0.0%	0.0%	0.0%
		10%	1553	377	1490	373	1044	141	66	166	90.5%	82.3%	84.1%
		20%	2126	518	2289	500	1459	805	172	548	64.8%	65.6%	62.4%
		30%	4378	660	4891	640	2023	2618	231	895	46.5%	63.9%	55.8%
	j2sdk1.4.0-javax-swing	0%	8115	516	8121	522	1843	8121	522	1843	0.0%	0.0%	0.0%
		10%	8203	558	8205	561	1929	90	47	132	98.9%	91.6%	93.2%
		20%	9978	687	9790	701	2330	1587	163	456	83.8%	76.7%	80.4%
		30%	11209	843	11197	848	2812	1414	222	709	87.4%	73.8%	74.8%
	netbeans-javadoc	0%	193	80	194	81	206	194	81	206	0.0%	0.0%	0.0%
		10%	197	82	196	83	210	5	3	7	97.4%	96.4%	96.7%
		20%	240	95	226	94	240	35	15	41	84.5%	84.0%	82.9%
		30%	304	110	292	108	284	70	23	72	76.0%	78.7%	74.6%
C	postgresql	0%	7	7	5	5	10	5	5	10	0.0%	0.0%	0.0%
		10%	24	20	22	18	38	17	13	28	22.7%	27.8%	26.3%
		20%	195	89	199	89	219	177	72	183	11.1%	19.1%	16.4%
		30%	530	203	601	195	543	405	125	370	32.6%	35.9%	31.9%
	snns	0%	109	63	126	68	165	126	68	165	0.0%	0.0%	0.0%
		10%	157	86	166	85	208	40	19	47	75.9%	77.6%	77.4%
		20%	343	143	299	124	321	133	46	130	55.5%	62.9%	59.5%
		30%	495	191	475	157	421	176	53	162	62.9%	66.2%	61.5%
	weltab	0%	46	8	46	8	27	46	8	27	0.0%	0.0%	0.0%
		10%	105	11	117	10	45	71	6	35	39.3%	40.0%	22.2%
		20%	148	17	156	17	62	39	11	39	75.0%	35.3%	37.1%
		30%	160	20	160	20	68	4	4	16	97.5%	80.0%	76.5%
	Cook	0%	7	5	7	5	11	7	5	11	0.0%	0.0%	0.0%
		10%	18	12	43	12	32	36	7	21	16.3%	41.7%	34.4%
		20%	107	56	108	44	108	65	35	83	39.8%	20.5%	23.1%
		30%	280	98	314	79	221	206	48	146	34.4%	39.2%	33.9%

threshold values. For instance, 93.2% of fragments obtained with threshold 10% for j2sdk1.4.0-javax-swing (Java) are also obtained with threshold 0%, 82.9% of fragments obtained with threshold 20% for netbeans-javadoc (Java) are also obtained with threshold

Table VIII NiCad results before and after using the reduction algorithm for other open source projects

Language	Subject system	Threshold	Clone Pairs	Clone Classes	Fragments	Clone Pairs	Clone Classes	Fragments	Clone Pairs	Clone Classes	Fragments
			Before Using Clone Reduction			*After Using Clone Reduction*			*Reduction Percentage*		
Java	Hibernate	0%	3888	736	2178	3888	736	2178	0.0%	0.0%	0.0%
		10%	3907	747	2204	19	14	32	99.5%	98.1%	98.5%
		20%	4061	850	2436	154	108	245	96.2%	87.3%	89.9%
		30%	4544	1020	2897	483	203	547	89.4%	80.1%	81.1%
	Jfreechart	0%	5565	695	2003	5565	695	2003	0.0%	0.0%	0.0%
		10%	5633	732	2091	68	47	108	98.8%	93.6%	94.8%
		20%	6302	904	2586	671	234	651	89.4%	74.1%	74.8%
		30%	10268	1007	3251	3966	267	1128	61.4%	73.5%	65.3%
	Sqoop	0%	136	83	186	136	83	186	0.0%	0.0%	0.0%
		10%	151	92	207	15	10	23	90.1%	89.1%	88.9%
		20%	302	117	283	151	30	91	50.0%	74.4%	67.8%
		30%	436	145	367	134	36	107	69.3%	75.2%	70.8%
	Jkarta Jmeter	0%	2851	220	679	2851	220	679	0.0%	0.0%	0.0%
		10%	2856	225	689	5	5	10	99.8%	97.8%	98.5%
		20%	3055	277	838	200	65	190	93.5%	76.5%	77.3%
		30%	3922	338	1080	867	93	335	77.9%	72.5%	69.0%
	Jruby	0%	7412	703	2151	7412	703	2151	0.0%	0.0%	0.0%
		10%	7474	743	2240	62	43	97	99.2%	94.2%	95.7%
		20%	9357	903	2759	1886	222	842	79.8%	75.4%	69.5%
		30%	189848	1063	3989	40908	380	2093	78.5%	64.3%	47.5%
C	Curl	0%	40	16	41	40	16	41			
		10%	89	33	89	49	19	52	44.9%	42.4%	41.6%
		20%	316	51	151	227	32	101	28.2%	37.3%	33.1%
		30%	419	74	217	103	39	128	75.4%	47.3%	41.0%
	Handbrake	0%	7	7	14	7	7	14	0.0%	0.0%	0.0%
		10%	17	15	31	10	9	19	41.2%	40.0%	38.7%
		20%	67	42	97	50	29	70	25.4%	31.0%	27.8%
		30%	153	90	212	86	55	133	43.8%	38.9%	37.3%
	VLC	0%	252	127	299	252	127	299	0.0%	0.0%	0.0%
		10%	287	145	342	35	18	43	87.8%	87.6%	87.4%
		20%	652	214	546	365	88	265	44.0%	58.9%	51.5%
		30%	1293	364	979	641	193	580	50.4%	47.0%	40.8%
	Onion	0%	221	110	265	221	110	265	0.0%	0.0%	0.0%
		10%	353	124	328	132	24	92	62.6%	80.6%	72.0%
		20%	519	190	494	166	83	213	68.0%	56.3%	56.9%
		30%	898	298	787	380	146	398	57.7%	51.0%	49.4%
	Netdata	0%	30	3	15	30	3	15	0.0%	0.0%	0.0%
		10%	7474	743	2240	3	3	6	100.0%	99.6%	99.7%
		20%	9357	903	2759	34	12	30	99.6%	98.7%	98.9%
		30%	189848	1063	3989	124	34	91	99.9%	96.8%	97.7%

10%, 76.8% of fragments obtained with threshold 30% for Eclipse-ant (Java) is also obtained with threshold 20%.

Depending on the Java subject, the clone fragments at threshold 10% are 84% (Eclipse-JtdCore) to 99% (Eclipse-ant) overlapping with those at threshold 0%; the fragments at threshold 20% are 62% (Eclipse-JtdCore) to 93% (Eclipse-ant) overlapping with those at threshold 10%; the fragments at threshold 30% are 47% (JRuby) to 81% (Hibernate) overlapping with those at threshold 20%. For C subjects, the fragments at threshold 10% are 22.2% (wetlab) to 87% (VLC) overlapping with those at threshold 0%; the fragments at threshold 20% are 16.4% (postgresql) to 98.9% (Netdata) overlapping with those at threshold 10%; the fragments at threshold 30% are 31.9 (postgresql) to 97.7% (Netdata) overlapping with those at threshold 20%. The range of overlapping information is very large, regardless of the threshold values or programming language.

Overall it appears that there is less overlapping clone data for C subjects than for Java subjects, although with a lot of variation. It may simply be because there are fewer clones in the C subjects than in the Java subjects to start with. We can only conjecture about possible reasons why C subjects lead to less overlapping clone data than Java subjects: Is it due to intrinsic characteristics of the programming languages? For instance, do characteristics of Java make it more prone to lead to clones than C? Is it due to the software development processes used to develop the software (e.g., traditional development process versus agile)? Is it due to the "age" of the subjects (e.g., the early 90s for postgresql and late 90s for swing)? Is it due to the community of developers?

Differences in terms of the amount of overlapping clone data between C and Java do not seem to be driven by the size of the code (LOC). For instance, postgresql and j2sdk

Table IX NiCad results before and after using the reduction algorithm for industry code, and Reduction Percentages

Clone Type	Clone classes	Clone fragments	LOC/ fragment	Clone classes	Clone fragments	LOC/ fragment	Clone classes	Clone fragments	LOC/ fragment
	Before using clone reduction algorithm			***After using clone reduction algorithm***			***Reduction Percentage***		
Type-1	33	84	1481	33	84	1481	0.00	0.00	0.00
Type-2 Consistent	113	321	4691	87	257	3524	23.01	19.94	24.88
Type-2 Blind	115	329	4778	6	19	163	94.78	94.22	96.59
Type-3	345	1095	54459	324	1045	53373	6.09	4.57	1.99
Type-3 Consistent	327	1021	27781	243	807	24493	25.69	20.96	11.84
Type-3 Blind	437	1695	67652	282	1258	57390	35.47	25.78	15.17
Total	1370	4545	160842	975	3470	140424	28.83	23.65	12.69

have relatively the same size (Table VI) but exhibit very different clone overlapping results. Differences do not seem to be related to the initial prevalence of clones in the code either. For instance, netbeans-javadoc and snns have similar numbers of fragments at threshold 10% but different overlapping percentages; we observe the same thing for netbeans-javadoc and postgresql at threshold 20%.

We also observe that for Java subjects, as the threshold value increases, the percentage of overlap clone data tends to decrease. This is not necessarily the case for C subjects: there is a decrease of overlap from 10% to 20% threshold and an increase from 20% to 30% for Curl, Handbrake, Postgresql, snns, Cook, whereas VLC, Onion and Netdata show a decrease from 10% to 20% and 20% to 30%, while Wetlab shows an increase from 10% to 20% and 20% to 30%.

If we turn our attention to industry code provided by our partner, this time for different threshold values and renaming (Table IX), we also observe the overlapping data in results

returned by NiCad. Approximately 20% of the Type-2 Consistent results are overlapped with Type-1 results; 95% of Type-2 Blind results are overlapped with Type-2 Consistent results. The percentage of redundant information between Type-3 and Type-2 Consistent is much smaller (e.g., 6% of classes).

3.5.2 Discussion

Roy and Cordy [1] had made the following conclusions based on the NiCad results (recall that these are conclusions without removing duplicated information). We wish to study whether one would make the same conclusions if looking at clone information without overlapping information. They concluded the following:

1. They found a large number of exact function clones in the open source systems.
2. They observed significantly higher numbers of near-miss clones than exact clones in the open-source systems.
3. They found more exact function clones in object-oriented Java than in C systems.
4. They observe that when the threshold for near-miss clones increases, the amount of clone information also increases, as expected, and that this effect is independent of the programming language (paradigm).
5. They found that there are no significant associations between the number of clones and the size of the systems.
6. They found that the largest Java system (j2sdk1.4.0-javax-swing) seems to be representative of the cloning characteristics of other systems written in a similar language.

Based on our results, we notice that the first conclusion still holds since the exact function clone information, which refers to clones with a 0% threshold value has not changed using our clone reduction procedure.

In order to investigate if the number of near-miss clones is higher than the number of exact clones, we calculated the total number of near-miss clones by summing up all clone information for threshold 10%, 20% and 30% after reduction: Table VII provides essentially the same information as Table VII and Table VIII except that we only report on data for threshold values 0% and 30% as well as for the sum of data for threshold values 10%, 20% and 30%. For all Java subjects, except Eclipse-JtdCore, Sqoop and JRuby, the number of near-missed clones (10% to 30%) is significantly lower than the number of exact clones; This is the opposite observation to the one made by Roy and Cordy. For C subjects, there are indeed more near-miss clones than exact clones, even after reduction, but the difference is smaller. When we only look at clones obtained at threshold 30%, which are sometimes referred to as near-missed (Type-3) clones [72], results for all the Java subjects indicate that there are fewer near-miss clones than exact clones (even for Eclipse-JtdCore, Sqoop and JRuby); For C subjects snns and wetlab, there are also fewer clones obtained at threshold 30% than at threshold 0%. Taking into account overlapping information in clone detection results leads to different conclusions as to the prevalence of specific types of clones in source code.

Their third conclusion was that there were more exact function clones in object-oriented Java than in C open-source systems; our results agree with their observation.

Their fourth conclusion was that if the threshold to detect near-miss clones increases, then the amount of clone information increases as well. When we analyze our results after

removing overlap clone information for clones at threshold 10%, 20% and 30%, i.e. past threshold 0%, we notice that when the threshold increases the clone measures increase as well for all Java systems except for j2sdk1.4.0-javax-swing and wetlab, as well as for all C subjects except Curl. In these three subjects, the number of clone pairs decreases when the threshold increases to 30%. Since our results show exceptions for three systems, we cannot entirely agree with their fourth conclusion.

In this study, we are not able to see if the fifth and sixth conclusions hold or not because we did not analyze the clone information after removing overlap clone information.

In general, we conclude that NiCad results contain overlapping information, sometimes in tremendous proportions: results of a higher threshold clone classes also include results of lower threshold clone classes results; for example, results of Type-2 clone classes also include results of Type-1 clone classes, results of Type-3 clone classes include results of both Type-1 and Type-2 clone classes; therefore, the conclusions that are empirically derived from NiCad results should be considered with care. We conclude that one can draw different conclusions if they are using clone detection information before or after reduction. We however believe that both results, before and after reduction, maybe independently useful pieces of information when reasoning about source code clones: after reduction, one can precisely identify the amount of clones that are really not exact clones, one can precisely discriminate between the different types of clones; by using clone detection information prior to reduction can help identify broader opportunities for refactoring since, for instance, Type-2 clone fragments may be so close to Type-1 clone fragments that they all may be refactorable together.

3.6 Threats to Validity

In the following, we discuss threats that could affect the validity of our work. The validity threats are categorized into four categories, which are conclusion validity, internal validity, construct validity, and external validity [73].

We foresee some threat to conclusion validity. This threat corresponds to the inability to draw valid conclusions. The reason for this threat is related to insufficient data to draw logical conclusions. The percentage of redundant information of all clone types for the subject systems (open source or from industry) might not be representative of all systems and test suites, and the percentage of redundant information results might not be generalizable. To overcome this threat, we used nine C subject systems and nine Java subject systems with varied sizes and complexities. We nevertheless do not make claims that the observations we make are generalizable to any software; more experiments of the kind we conducted are necessary.

Concerning some internal validity threats to our work, the tool we used to detect clones' instances could fail to retrieve some of the clones' instances in the subject systems and test suites. To ease this concern, we are aware of the effects of configurations on the performance of the tool, so we used multiple pre-defined configurations for NiCad. At the same time, there are configurations that meet the definitions of Type-1, Type-2 Consistent, Type-2 Blind, Type-3, Type-3 Consistent and Type-3 Blind clones that are used in many different publications. In the end, we used the default configuration of NiCad since they have been shown to lead to both high precision and high recall at finding near-miss clones

[68]. We disclosed two categories of odd results in what NiCad reports; the differences we observed are very small compared to the large numbers of clone classes/pairs/fragments, and they should not jeopardize our conclusions.

Table X Comparing Numbers of Near-Miss and Exact Clones

				Before Using Clone Reduction			*After Using Clone Reduction*		
Java	Subjects from previous study [1]	Eclipse-Ant	0%	363	92	250	363	92	250
			30%	428	120	319	54	25	74
			10% to 30%				65	34	93
		Eclipse-JdtCore	0%	1409	326	940	1409	326	940
			30%	4891	640	2023	2618	231	895
			10% to 30%				3564	469	1609
		j2sdk1.4.0-javax-swing	0%	8121	522	1843	8121	522	1843
			30%	11197	848	2812	1414	222	709
			10% to 30%				3091	432	1297
		netbeans-javadoc	0%	194	81	206	194	81	206
			30%	292	108	284	70	23	72
			10% to 30%				110	41	120
	New subjects	Hibernate	0%	3888	736	2178	3888	736	2178
			30%	4544	1020	2897	483	203	547
			10% to 30%				656	325	824
		JFreeChart	0%	5565	695	2003	5565	695	2003
			30%	10268	1007	3251	3966	267	1128
			10% to 30%				4705	548	1887
		Sqoop	0%	136	83	186	136	83	186
			30%	436	145	367	134	36	107
			10% to 30%				300	76	221
		JKarta JMeter	0%	2851	220	679	2851	220	679
			30%	3055	277	838	200	65	190
			10% to 30%				1072	163	535
		JRuby	0%	7412	703	2151	7412	703	2151
			30%	189848	1063	3989	40908	380	2093
			10% to 30%				42856	645	3032
C	Subjects from previous study [1]	postgresql	0%	5	5	10	5	5	10
			30%	601	195	543	405	125	370
			10% to 30%				599	210	581
		snns	0%	126	68	165	126	68	165
			30%	475	157	421	176	53	162
			10% to 30%				349	118	339
		weltab	0%	46	8	27	46	8	27
			30%	160	20	68	4	4	16
			10% to 30%				114	21	90
		Cook	0%	7	5	11	7	5	11
			30%	314	79	221	206	48	146
			10% to 30%				307	90	250
	New subjects	Curl	0%	40	16	41	40	16	41
			30%	419	74	217	103	39	128
			10% to 30%				379	90	281
		Handbrake	0%	7	7	14	7	7	14
			30%	153	90	212	86	55	133
			10% to 30%				146	93	222
		VLC	0%	252	127	299	252	127	299
			30%	1293	364	979	641	193	580
			10% to 30%				1041	299	888
		Onion	0%	221	110	265	221	110	265
			30%	898	298	787	380	146	398
			10% to 30%				678	253	703
		NetData	0%	30	3	15	30	3	15
			30%	189848	1063	3989	124	34	91
			10% to 30%				161	49	127

Construct validity is concerned with the relationship between theory and observation. We have used a procedure to remove duplicated clone classes and pairs from raw NiCad results; we have partly automated the procedure to reduce sources of mistakes. To reduce this threat, we have automated as much as possible, and we manually verified the outcome results on sample data to make sure that the procedure worked as intended.

There can be threats to external validity, as well. These threats are related to the external aspects of the experiments that can limit the generalization of the results. This threat is primarily associated with the subject systems and test suites selection for performing the analysis and the test code clones derived from them. To reduce this threat, we selected Bellon's subject systems that have already been used by Roy and Cordy to validity NiCad. We also did not cherry-pick the test code from our industry partner; the code was provided by our industry partner and includes complete suites for the application logic they verify.

3.7 Conclusion

In this chapter, recognizing that there might be some overlap of clone detection information when a clone detection tool reports on different types of clones, we study the extent of such overlapped information and to what extent having such overlapped information or removing such overlapped information impacts conclusions one can empirically derive when conducting experiments on clone detection in source code. Indeed, the standard definition of clone types allows clone detection results to overlap. For instance, in a Type-3 clone class, code fragments show minor or major modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespaces, layouts, and comments, that is in addition to modifications that define a Type-2 clone class; As a result, Type-2 clone class fragments can also be returned as

fragments of Type-3 clone classes. Given this analytical observation, this chapter first illustrates concretely with an example (open-source subject analyzed with a clone detection tool) that clone detection results overlap.

We therefore first refine standard clone type definitions to account for overlapping clone detection information. Essentially, since different types of clones are derived with more or less allowed source code changes, we define, for each clone type, a pure clone type counter-part that avoids clone detection information of less permissive clone types: for instance, since Type-1 is less permissive than Type-2, we define a Pure Type-2 clone class such that Pure Type-2 clone classes do not contain fragments that belong to Type-1 clone classes or exact clone classes. Indeed, the different pure types of clones do not have overlap data between them. We then defined and implemented a generic Clone Reduction Procedure to remove overlapped clone information from clone detection tool results, and illustrate how it can be applied to results produced by NiCad, a well-known and effective clone detection tool, thereby confirming on a simple case that there can be overlapping clone detection information.

Given anecdotal evidence of overlapping clone information in clone detection results, we confirmed the extent of such results experimentally. To that end, we selected, replicated and extended an experiment others have conducted with NiCad [1], using a total of nine large Java subjects programs and nine large C subject programs. We also performed the same study, comparing NiCad results with and without our reduction procedure, on code provided by an industry partner.

We conclude that clone detection results such as those produced by NiCad contain overlapped information, sometimes in tremendous proportions: on Java subjects, for

instance, clone fragments obtained with a different threshold of 10% are at least 80% overlapping with fragments obtained with a threshold of 0%. We observe that conclusions empirically derived from results produced by a clone detection tool such as NiCad should be considered with care: different conclusions may be obtained whether they are driven from results with overlapping clone detection information or from results without overlapping clone detection results.

Note that, in this study, we do not intend to judge the quality of NiCad. NiCad has already been shown to perform well at detecting clones. We want to raise awareness about the presence of overlapping information and show the extent of this phenomenon empirically. Also, we do not intend to debate about whether results with overlapping information are better or worse than results without overlapping information. On the contrary, we believe both can be useful. For instance, although this should be confirmed, we believe that a better idea of the refactoring efforts associated with removing clones can be obtained from clone detection results without overlapping information; And we believe that clone detection results without overlapping information are less useful than results with overlapping information to actually decide what to refactor because, when only using results without overlapping information, one may not realize that Type-2 clone fragments may be so similar to exact clones that they could be refactored together.

The researcher and practitioner communities should decide on the matter after other studies about the issue. We also believe that being aware of this overlap information, and its extent, may be useful when comparing different clone detection tools. Could various amounts of overlap information in results reported by varying tools be a reason for disagreeing empirical results? We believe raising awareness about this phenomenon is

equally important for researchers and practitioners alike since the presence of overlap information in clone detection results may lead to erroneous conclusions during experimental studies.

Chapter 4: An Analysis of Complex Industry Test Code using Clone analysis

In this chapter, we perform a case study with industry test code that aims at better understanding such duplicated test fragments, or as we call them, test clones. This chapter has the following goal: we aim to detect clones in industry test codes. We found that 49% (LOC) of the entire C test code are clones, which is also equivalent to 36% of all the test cases; We found that 73% LOC of the entire Java test code is a clone, amounting to 94% of all the test cases. The results we report on in our study include figures about clone frequencies, types, similarity, fragments, size distributions, and the number of line differences in cloned test cases. It is challenging to keep clones consistent and remove unnecessary clones during the entire testing process of large-scale commercial software.

The rest of this chapter is organized as follows. Section 4.1 discusses the experiment design. Section 4.2 introduces and discusses the results of our empirical study. Furthermore, we discuss threat to validity, followed by the conclusion in Sections 4.3 and 4.4, respectively.

4.1 Experiment Design

To answer the research questions we presented in the Introduction (Section 1.1), we developed an experimental procedure that involves the use of the NiCad tool on several commercial packages.

In this study, we used three projects provided by our industry partner. The system being tested is a part of middleware in the telecom domain. We specifically focussed on test code written in C and Java. For obvious confidentiality reasons, technical details on the

application code and test code cannot be revealed. The total number of C files in the two C projects is 427, which is made of 286,168 LOC. The third project, project-3, is larger (more test suites, more test code) than the first and second projects; it consists of 5,876 Java files, which are made of 791,384 LOC.

We reused the same setting and procedures as discussed earlier: the same version of NiCad, the same setup and configuration of NiCad (Section 2.7), the same procedure to remove duplicates (Section 3.3). Recall that the rationale for removing overlapping clone data is to obtain results that are specific to the selected clone types and solely for those types (not for other, more permissive types). We used all the test suites from the two projects that our industry partner, Ericsson Canada, contributed.

4.2 Experimental Results

In this section, we separately answer each of the research questions we presented in the Introduction.

4.2.1 The ratio of clones in industry test code (RQ1)

For each pure clone class type, Table XI shows the number of clone classes of that type, the total number of clone fragments of those classes, and the total LOC of those fragments, plus percentages over the total size of the project. For instance, there are 33 clone classes of Type-1, made of a total of 84 fragments and those fragments amount to a total of 1,481 LOC, i.e., 0.52% of the entire C project size (286,168 LOC).

For C projects, the first observation we can make is that exact clones, either as seen in the source code (Type-1) or after simple naming normalization (Type-2), i.e., first three rows in Table XI, represent a small percentage of the source code (1.8%, 5,168 LOC). These are much more prevalent in the Java test code (23.42%, with 185,338 LOC). In both

Table XI Nicad clone detection results after removing duplications

	C				Java			
Clone Type	**Clone classes**	**Clone fragments**	**LOC/ fragment (Percentage)**		**Clone classes**	**Clone fragments**	**LOC/ fragment (Percentage)**	
Type 1	33	84	1,481	(0.52%)	1,258	5521	70,030	(8.85%)
Pure Type 2 Consistent	87	257	3,524	(1.23%)	806	8136	77,866	(9.84%)
Pure Type 2 Blind	6	19	163	(0.06%)	128	3511	37,442	(4.73%)
Pure Type 3	324	1,045	53,373	(18.65%)	1,030	9014	130,972	(16.55%)
Pure Type 3 Consistent	243	807	24,493	(8.56 %)	710	11235	121,546	(15.36%)
Pure Type 3 Blind	282	1,258	57,390	(20.05%)	455	12270	140,024	(17.69%)
Total	975	3,470	140,424	(49.07%)	4,387	49687	577,880	(73.02%)

Java and C subjects, there are more Pure Type-3 clones, corresponding to a more significant proportion of the test code, than Type-1 and Pure Type-2 clones. This may not be entirely surprising since Pure Type-3 clones are determined with a higher dissimilarity threshold (30%) that does allow matching of code fragments with more differences than Type-1 and Pure Type-2 clones. It is to be expected that Pure Type-3 Consistent clones have a lower prevalence than Pure Type-3 and Pure Type-3 Blind because consistent renaming is more restrictive than blind renaming, or no renaming; this is the case for C but not entirely for Java. Overall, approximately half (49.07%) of the C code and 73% of the Java code show some duplication (clone), indicating the C and Java projects we analyze have a high level of redundancy, and redundancy is primarily due to Type-3 clones.

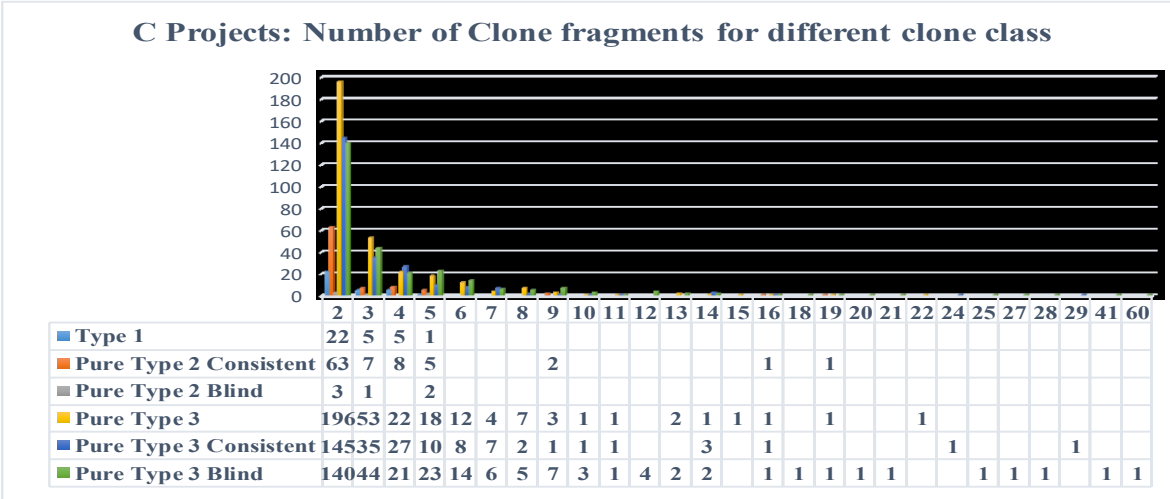
When we compare results for C with results for Java, we observe that the Java project has a higher clone percentage than C projects. Clones seem to be more uniformly distributed over various clone types in Java (i.e., between 5% and 10% for Type-1 and Pure Type-2, around 16% for Type-3) than in C (e.g., at most 1% for Type-1 and Pure Type-2, between 1% and 20% for Type-3). We can only conjecture about reasons for such differences: e.g., training of individuals contributing test code, team practices, tool support

(e.g., JUnit), specifics of the packages under test, test objectives (e.g., functional, structural)?

Comparing our work with previous work, Tsantalis et al. [49] used NiCad to find clones in the test code of open source projects and found a clone rate between 14.3% and 81.9% (of LOC). Our case study is average in comparison. We note, however, that comparing the results of their study with ours should be done with caution; indeed, they did not apply a procedure to trim the sets of reported clones as we do (Sections 4.1); We, therefore, believe that their results are inflated when compared to ours, likely making the two series of results more on par.

4.2.2 Distribution of clone class sizes (RQ2)

Clone classes have various numbers of code fragments. For C projects, Illustration 5 shows the clone class size distribution for various pure clone class types. The x-axis indicates clone class sizes (i.e., number of fragments) and the y-axis shows the number of occurrences of each clone class size, while clone types have their own colours. For



	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	18	19	20	21	22	24	25	27	28	29	41	60
Type 1	22	5	5	1																							
Pure Type 2 Consistent	63	7	8	5				2							1		1										
Pure Type 2 Blind	3	1		2																							
Pure Type 3	196	53	22	18	12	4	7	3	1	1		2	1	1	1		1			1							
Pure Type 3 Consistent	145	35	27	10	8	7	2	1	1	1			3		1						1				1		
Pure Type 3 Blind	140	44	21	23	14	6	5	7	3	1	4	2	2		1	1	1	1	1			1	1	1		1	1

Illustration 5 Clone class size distribution.

instance, we observe 22 Type-1 clone classes of size 2 (i.e., with two identical code fragments), five Type-1 clone classes of size 3 and size 4, and one Type-1 clone class of size 5. Given the granularity of the clone, detection is the function; the fragments in these classes are entirely identical functions. We observe that the vast majority of the clone classes, irrespective of the type, have a size of 2. This suggests that the code shows a lot of similarities, as discussed earlier (49% of the code shows some level of duplication), but that duplications are varied. Again, given our granularity setup, these duplications are copies of entire functions. The longest code fragments in those clone classes of size 2 are two 401 LOC long functions with a similarity of 78% making a Pure Type-3 clone class. The second-largest code fragments are two 399 LOC long functions with a similarity of 79% making a Pure Type-3 Blind clone class. Also, we observe that we only have one clone class occurrence for clone class sizes that are greater than 15. As the clone class size decreases, the clone class size occurrence increases.

We find that Pure Type-3 Blind clone classes have large numbers of clone fragments: for instance, classes with 25, 27, 28, 41, and 60 clone fragments. These (function) fragments have a modest size (between six and 16 LOC) with a similarity of at least 70%. The Pure Type-3 Consistent classes that have a large number of (function) fragments (i.e., 24 and 29) also have a modest fragment size (between five and ten LOC), with a similarity of at least 71%. Two Pure Type-3 clone classes have a large number of fragments: 19 and 22 fragments of 29 and ten LOC, respectively, with a similarity of at least 70%. Two Pure Type-2 Consistent classes have a large number of fragments: 16 and 19, with fragments of five and ten LOC respectively, and a similarity of 100%. Overall, with a few exceptions discussed above, clone fragments are short.

We have performed the same analysis on the Java project; Illustration 6 has the same structure as Illustration 5 except that we do not show the numbers below the bar chart, to not clutter the figure. The Appendix at the end of this manuscript provides the table with clone class size numbers for each pure clone type. The Java project has 730 Type-1 clone classes of size 2, 420 Pure Type-2 Consistent clone classes of size 2, and 399 Type-1 clone class of size 3. We observe that the vast majority of the Type-1, Pure Type-2 Consistent, and Pure Type-2 Blind clone classes have a size of 2 or 3. Similarly to our observation for C subjects, this suggests that duplications are varied. Also, we observe that we only have one clone class occurrence for clone class sizes that are greater than 36. As the clone class size decreases, the clone class size occurrence increases.

The longest code fragment in those classes is a 2,002 LOC function in a Type-1 clone class (similarity of 100%). We found clone classes with a large number of fragments; for instance, some Pure Type-2 Blind clone classes have 475, 531, and 570 code fragments. The size of these code fragments is between five and 11 LOC with a similarity of 100% (after blind renaming). Ten clone classes have at least 80 code fragments, of ten to 17 LOC (similarity of 100%); 22 Pure Type-2 Consistent clone classes have at least 80 code fragments. This indicates that the analyzed test suites have many very similar test cases. The size of these clone fragments is between five and 17 LOC, with a similarity of 100%.

596 Pure Type-3, 359 Pure Type-3 Consistent and 202 Pure Type-3 Blind clone classes have a size of 2. The vast majority of the clone classes, irrespective of the type, have a size of 2 or 3, again indicating highly varied duplications. We only have one clone class occurrence for clone classes with a very large number of fragments with fragments length between five and 14 LOC (similarity of at least 70%): for instance, we observe Pure Type-3 Blind clone classes with 570, 632, 3,129, and 4,844 fragments, Pure Type-3 Consistent clone classes with 513, 625, 886, 926, and 1,663 clone fragments.

Type-3 has clone classes with 1,562 and 2,050 clone fragments. The size of these clone fragments is 12 and 11 LOC, respectively, with a similarity of 70%. Also, we see that, as the clone class size decreases, the clone class size occurrence increases. We conjecture that this high frequency of clone classes that are reduced to pairs is due to copy/paste of test code in the process of creating and maintaining test cases.

Contrasting results for the C and Java packages, we conclude that data show the same trends (e.g., large number of clone classes of size 2, decrease of occurrences of clone classes as class size increases, few clone classes with large size, some clone classes with

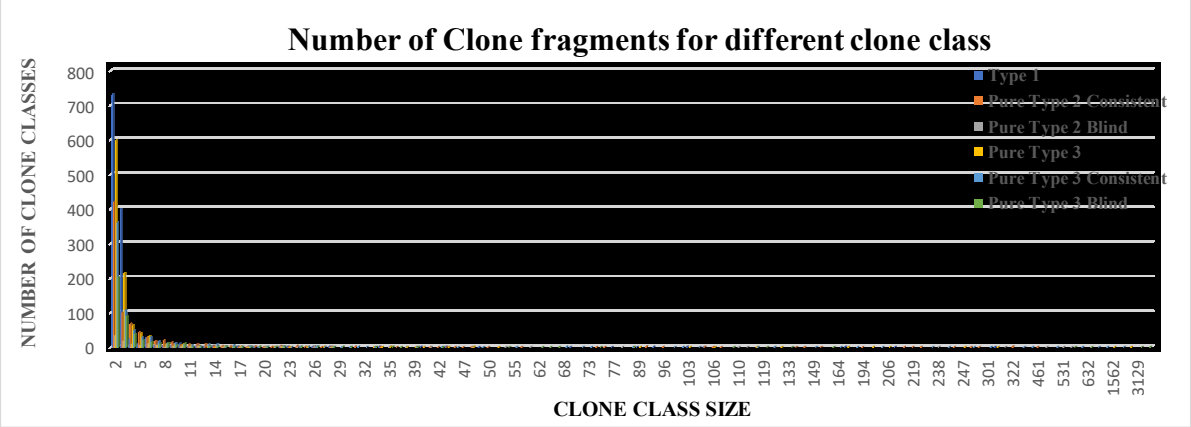


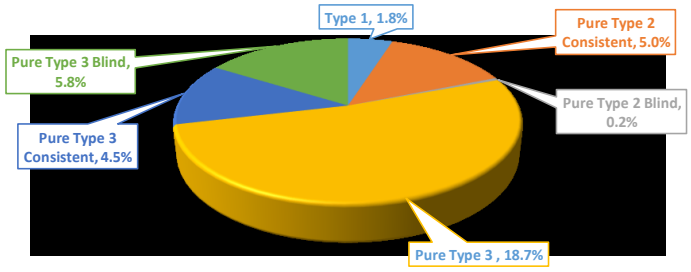
Illustration 6 Clone class size distribution for Java project.

extremely large sizes) but trends are more pronounced in Java compared to C (e.g., larger number of clone classes of size 2, much larger class sizes), and this does not seem to be entirely due to the larger size of the Java code compared to the C code.

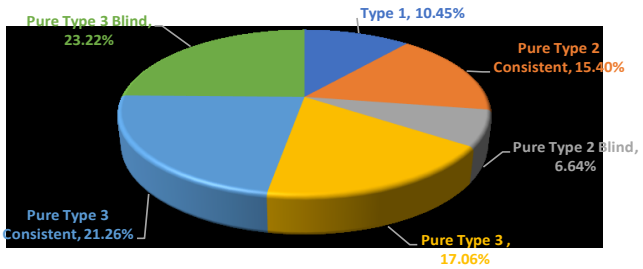
4.2.3 The ratio of cloned test cases (RQ3)

Recall (Section 2.7) that clone fragments are test cases because NiCad's analysis granularity was set to "function," and test cases are functions. Together, C projects are composed of 4,708 functions/test cases, and 36% of them are clones of some type, i.e., with at least 70% similarity. Illustration 7 (left) shows the ratio of cloned test cases for each pure clone type: 18.7% of the total number of functions are clones of Pure Type-3, which represents 52% of all cloned functions; 5.8% of the entire set of functions are clones of Pure Type-3 Blind and this type of clones makes up 16% of all cloned functions.

The Java project is composed of 52,842 functions/test cases, and 94.03% of them are clones of some type, i.e., with at least 70% similarity. Illustration 7 (right) shows the ratio



(C)



(Java)

Illustration 7 Percentage of Cloned Functions in C projects and the Java project

of cloned test cases for each pure clone type. 32.49 % of the total number of functions are exact clones (Type-1, Pure Type-2 Consistent, and Pure Type-2 Blind), which represents approximately 34.56% of all cloned functions; 61.54% of the entire set of functions are near-miss clones (Pure Type-3, Pure Type-3 Consistent and Pure Type-3 Blind) and these types of clones make up 65.5% of all cloned functions.

We conjecture that the reason for this large number of clones is that the context of our study is embedded code, that is code that is close to the hardware platform; embedded code requires sometimes complex hardware setup when being tested. In other words, these test cases need to execute several functionalities from libraries and those functionalities are often similar and they are executed in similar ways. Also, there are many aspects of the hardware to configure before executing test, and then, at the end of test case executions, there is a lot to clean up; this may explain why we have large pieces of set up and tear down that are cloned. When we examined test code, we found that several test cases are typically created to verify a specific functionality/function; therefore, these test cases have a high chance of being clones as the functionality/function may need to be triggered in a similar way (function calls) but with alternative inputs; this may explain why we have many clone classes of small size

Also, we believe that another reason for this large number of clones is the difficulty for engineers to dig into large test suites (4,708 C test functions, 52,842 Java test functions in the test code we analyzed) to identify whether one needs to add test cases during sprints. For instance, assuming these clones could be simply refactored by entirely removing duplicates, an engineer who would consider extensions to the test suites would need to look at and understand 1,694 C test functions (36% of 4,708, since 36% of functions are clones)

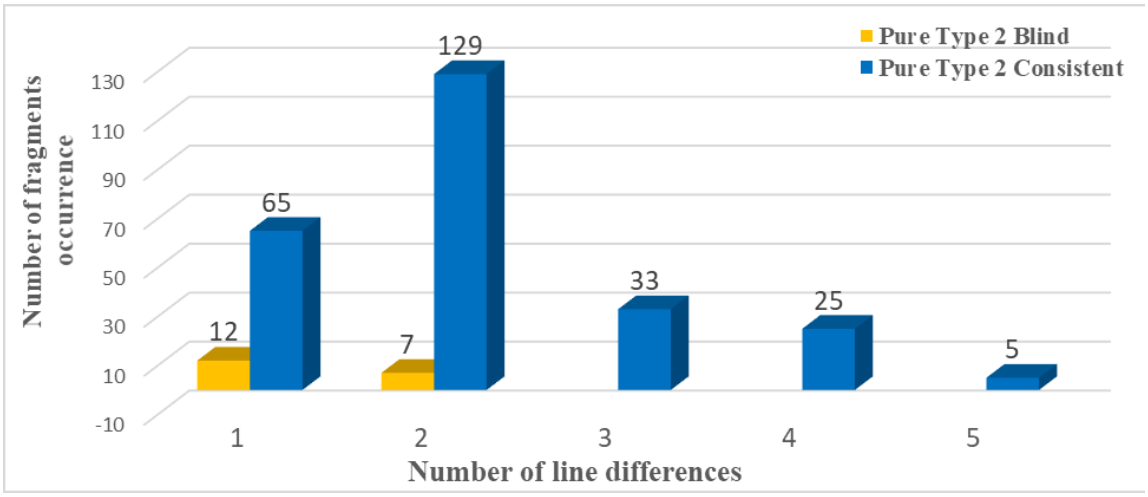


Illustration 8 Distribution of line differences between clone fragments for Type-2 Consistent and Type-2 Blind: C projects

or 18,262 Java test functions (32.5% of 52,842, since 32.5% of functions are clones). Given these test suites are very large, we believe engineers tend to prefer pure additions of test cases, at the (unknown) risk of creating test clones, to spending time trying to understand the purpose of a large number of tests.

4.2.4 Clone similarity versus the number of line difference (RQ4)

Results indicate the code includes clone fragments that are 70-100% similar. Since NiCad computes similarity on the pretty-printed version of code fragments, and pretty-printing means splitting each line of code into several pretty-printed lines, each line in the original source code fragment corresponds to several pretty-printed lines and several lines of the pretty-printed code which are identified to be 70-100% similar may correspond to only one line of the original source code. Thus, the similarity percentage computed by NiCad does not tell us exactly how many lines of actual source code fragments are similar in clone classes. A 30% difference between two pretty-printed fragments may take place in one single line of the original source code. Conversely, Pure Type-2 Blind and Pure

Type-2 Consistent clone classes are made of 100% similar pretty-printed fragments after some renaming; 100% similarity actually means some differences in original source code because of renaming. Therefore, we further study how clone class similarity translates into source code similarity, and we perform the study separately for Pure Type-2 and Pure Type-3 clone classes. We did not perform the analysis on Type-1 classes since Type-1 clones are identical, and there is, therefore, no line difference (neither in pretty-printed lines nor in original code lines).

For C projects, we first performed the analysis on Pure Type-2 Consistent and Pure Type-2 Blind clones. Illustration 8 shows the number of lines that differ on the x-axis, and the number of fragments that exhibit these amounts of difference for Pure Type-2 Blind and Pure Type-2 Consistent clone classes on the y-axis. Pure Type-2 clone fragments differ by at most five LOC in the source code, and the vast majority of those fragments differ by one or two LOC. 129 Pure Type-2 Consistent clone fragments differ by two LOC and 65 by one LOC. These are very small, minimal differences.

Since Pure Type-3 clones allow for larger differences than Pure Type-2 clones, the range of the number of line differences is larger than for Pure Type-2 clones: Illustration 9. Fragments in Pure Type-3, Pure Type-3 Blind, and Pure Type-3 Consistent clone classes differ by 1 to 89 LOC. A few of them differ by one-line; Pure Type-3 Consistent and Pure Type-3 Blind clone classes, have a record high number of fragments that differ by only two lines (181 and 205 fragments respectively). Given that Pure Type-3 clone classes are obtained with a maximum of 30% differences in the pretty-printed version of the code, this seemingly large difference only appears in at most two lines of the original code for those fragments. Also, Pure Type-3 classes have a significant number of fragment occurrences

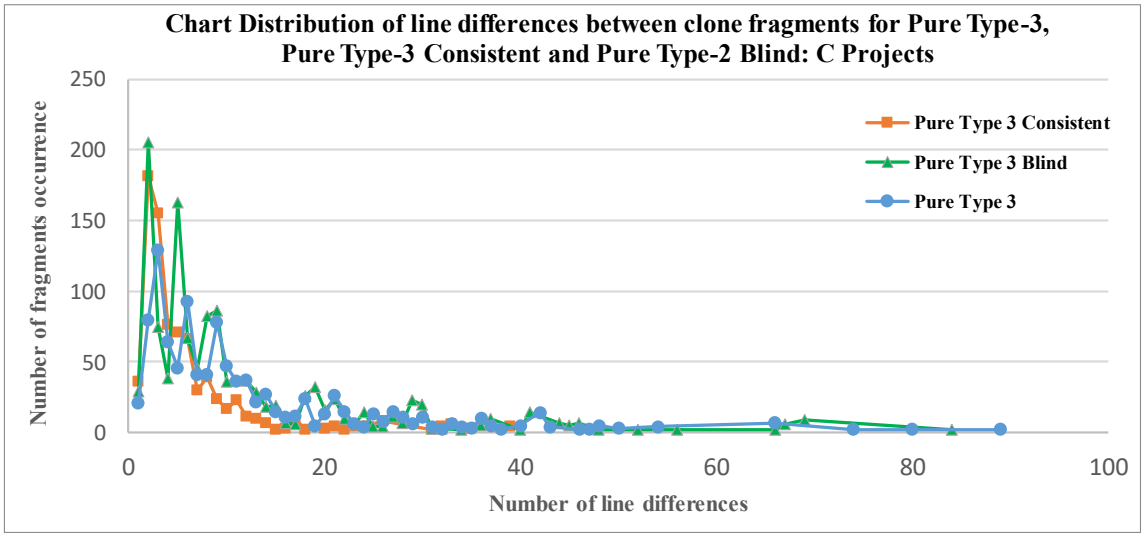


Illustration 9 Distribution of differences between clone fragments for Type-3, Type-3 Consistent and Type-3 Blind: C projects

with three lines difference: 129 clone fragments in Pure Type 3 classes, 155 clone fragments in Pure Type-3 Consistent. Furthermore, these three clone types have a considerable number of fragment occurrences that differ by two to ten lines in the source

code. There is a Pure Type-3 clone class with two fragments that differ by 89 lines; Those fragments are 401 LOC long. There is a Pure Type-3 Consistent clone class with two fragments that differ by 24 lines; Those fragments are also very large.

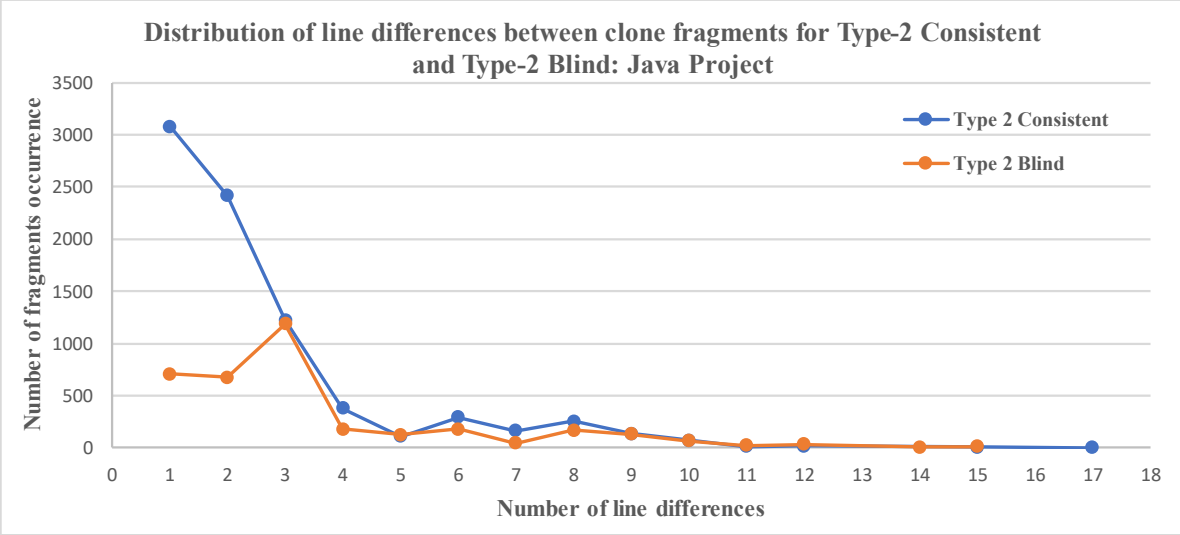


Illustration 10 Distribution of line differences between clone fragments for Type-2 Consistent and Type-2 Blind.

For the Java project (Illustration 10), fragments in Type-2 clone classes differ by 1 to at most 17 LOC. The vast majority of those fragments differ by one, two, or three LOC: 3,078 Pure Type-2 Consistent clone fragments differ by only one LOC, 2,418 differ by two LOC, and 1,224 differ by three LOC. Additionally, 706 fragments in Type-2 Blind clone classes differ by one LOC, 674 by two LOC and 1,187 by three LOC. These are all very small, minimal differences. Fragments in Pure Type-3 clones classes differ by 1 to at most 521 LOC in the source code (Illustration 11), and the vast majority of those fragments differ by one, two, three or four LOC: 1,166 Pure Type-3 clone fragments differ by one LOC, 1,200 by two LOC, 2,047 by three LOC, and 3,960 by four LOC. Pure Type-3 clone fragments record a large number of line differences such as 100, 259, 261, 341 and 521. These fragments have a large size; for instance, the Pure Type-3 clone fragments that differ by 521 lines have an overall fragment size of 2,002 LOC.

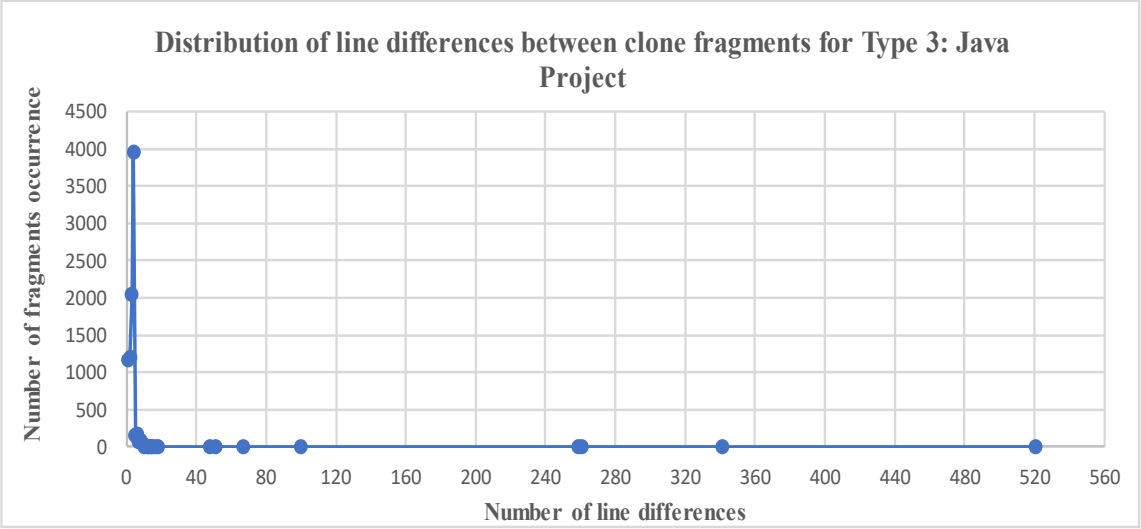


Illustration 11 Distribution of differences between clone fragments for Type-3 in Java Project

In Pure Type-3 Consistent and Pure Type-3 Blind clone classes, fragments differ by 1 to 62 LOC (Illustration 12), and, again, the vast majority of those fragments differ by two, three or four LOC. The maximum of 30% difference in the pretty-printed version of the code appears in at most two, three or four LOC of the original source code of those fragments. For instance, Pure Type-3 Consistent clone classes have 4,139 fragments with three-line differences, and Pure Type-3, Blind clone classes, have 5,108 fragments with four-line differences.

At this stage, we essentially have two indicators of fragment similarity (or difference): the difference in the pretty-printed versions of the source code, the LOC difference in the source code. We, therefore, ask ourselves whether there is a correlation between the two. The correlation coefficient is used to measure the strength of the relationship between two variables. The Pearson correlation coefficient [74] is used to measure the extent with which two variables have a linear relationship and assumes variables are normally distributed,

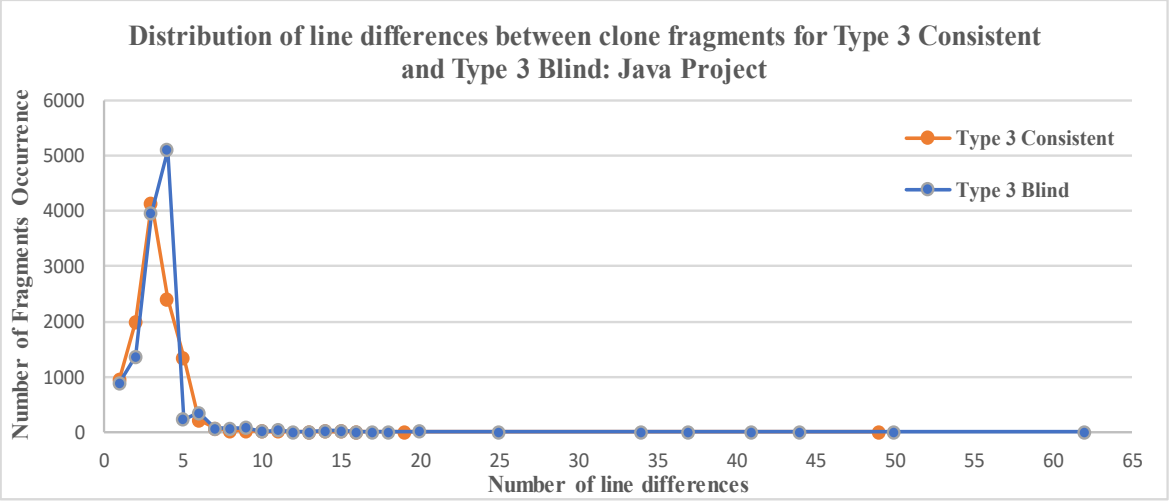


Illustration 12 Distribution of differences between clone fragments for Type-3 Consistent and Type-3 Blind in Java Project.

Table XII Shapiro-Wilk normality test for the fragments' size, line difference, and similarity differences

Clone Type	C projects P-value			Java projects P-value		
	Fragment Size	**Line Difference**	**Similarity Difference**	**Fragment Size**	**Line Difference**	**Similarity Difference**
Pure Type 2 Consistent	< 0.01	< 0.01		< 0.01	< 0.01	
Pure Type 2 Blind	< 0.01	< 0.01		< 0.01	< 0.01	
Pure Type 3	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Pure Type 3 Consistent	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
Pure Type 3 Blind	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

whereas the Spearman's Correlation [75] is used to measure the extent with which the variables have a nonlinear relationship, does not assume normal distributions and is robust to outliers. It is rare to find a perfect monotonic relationship (+1 or -1), so depending on how a result is on the scale, one must adjust conclusions. A commonly used scale is the following [76]: The correlation is very weak if the correlation value is between zero and 0.19, and the negative correlation between 0 to -0.19; A weak correlation exists if the positive correlation value is between 0.2 to 0.39, and the negative correlation between -0.2 to -0.39; A moderate correlation exists if the positive correlation value is between 0.4 to 0.59, and the negative correlation between -0.4 to -0.59; A strong correlation exists if the positive correlation value is between 0.6 to 0.79, and the negative correlation between -0.6 to -0.79; A very strong correlation exists if the positive correlation value is between 0.8 to 1.0, and the negative correlation between -0.8 to -1.

In order to decide whether to use Pearson or Spearman correlation, we first need to perform a normality test on the given data. We used the Shapiro-Wilk normality test since Razali et al. [77] concludes that it performs better than alternative normality tests and is used more often. It tests the null hypothesis that the measured sample comes from a

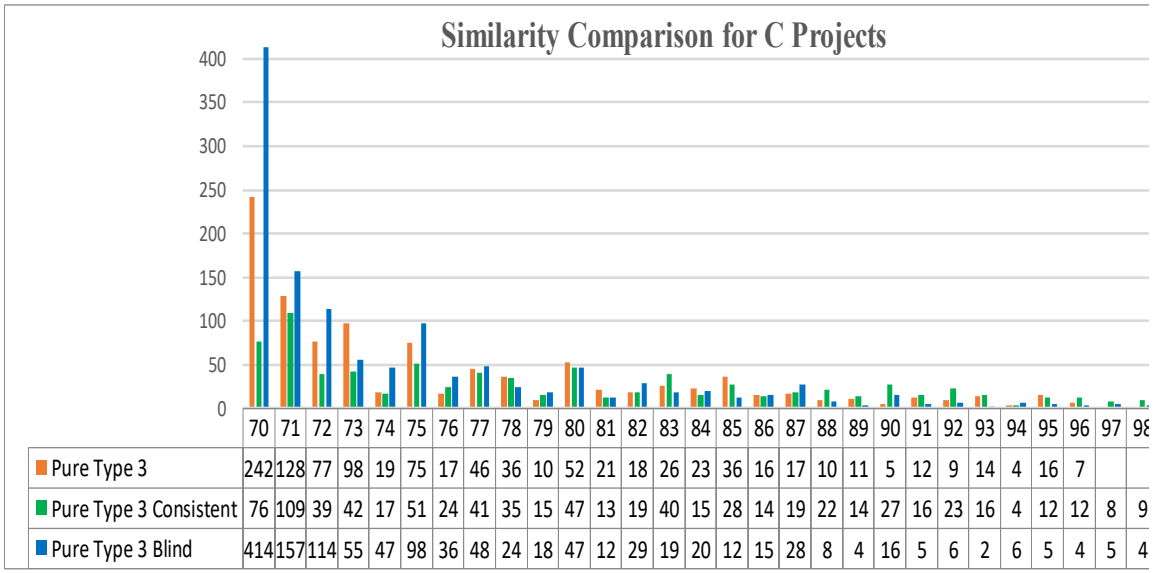
Table XIII Statistical correlation (Spearman) results

	Clone Type	C projects- Statistical Correlation	Java projects- Statistical Correlation
Correlation between fragment size and line difference	Pure Type 2 Consistent	0.79	0.36
	Pure Type 2 Blind	0.69	0.44
	Pure Type 3	0.92	0.81
	Pure Type 3 Consistent	0.80	0.86
	Pure Type 3 Blind	0.94	0.96
Correlation between fragments' percentage difference and the line difference	Pure Type 3	0.31	0.38
	Pure Type 3 Consistent	0.18	0.44
	Pure Type 3 Blind	0.1	0.36

normally distributed population. Table XII shows that the p-values for fragment's size, line differences, and similarity differences for all clone types are small; similarity values for Pure Type-2 are not provided because the similarity is 100%. Since all the p-values are small, we reject the null hypothesis and conclude that distributions are not normal. Consequently, we should use Spearman's Correlation.

Table XIII reports on correlation (Spearman) results between fragment size and the number of line differences. Results indicate a very strong positive correlation between the two variables for Pure Type-3 clone classes for both C and Java. Results differ for Pure Type-2 clone classes: while the positive correlation is strong for C, it is only weak or moderate for Java. Thus, we can conclude that when the fragment size increases, the number of line difference increases as well for all clone types, which is not entirely surprising.

Table XIII shows correlation (Spearman) results between fragments' percentage difference and the number of line differences for Pure Type-3 variants. We did not perform that analysis for Pure Type-2 variants because, for these types, the percentage difference is



	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98
Pure Type 3	242	128	77	98	19	75	17	46	36	10	52	21	18	26	23	36	16	17	10	11	5	12	9	14	4	16	7		
Pure Type 3 Consistent	76	109	39	42	17	51	24	41	35	15	47	13	19	40	15	28	14	19	22	14	27	16	23	16	4	12	12	8	9
Pure Type 3 Blind	414	157	114	55	47	98	36	48	24	18	47	12	29	19	20	12	15	28	8	4	16	5	6	2	6	5	4	5	4

Illustration 13 Similarity values for Type-3, Type-3 Consistent, and Type-3 Blind clone fragments

zero. Results indicate mostly a very weak or weak correlation between the two variables and only one moderate correlation. For instance, when we perform a further investigation for Pure Type-3 fragments, we discovered that some fragments differ by two lines with a similarity of 71%, while other fragments that also differ by two lines have a similarity of 98%. This example shows that it is important to report the number of line differences as well as similarity. Overall, correlation results indicate that the difference in pretty-printing versions of fragments and the difference in lines of code measure two different aspects of cloning, which suggest that they should both be used. What aspects of cloning they actually measure is an open question worth investigating in the future.

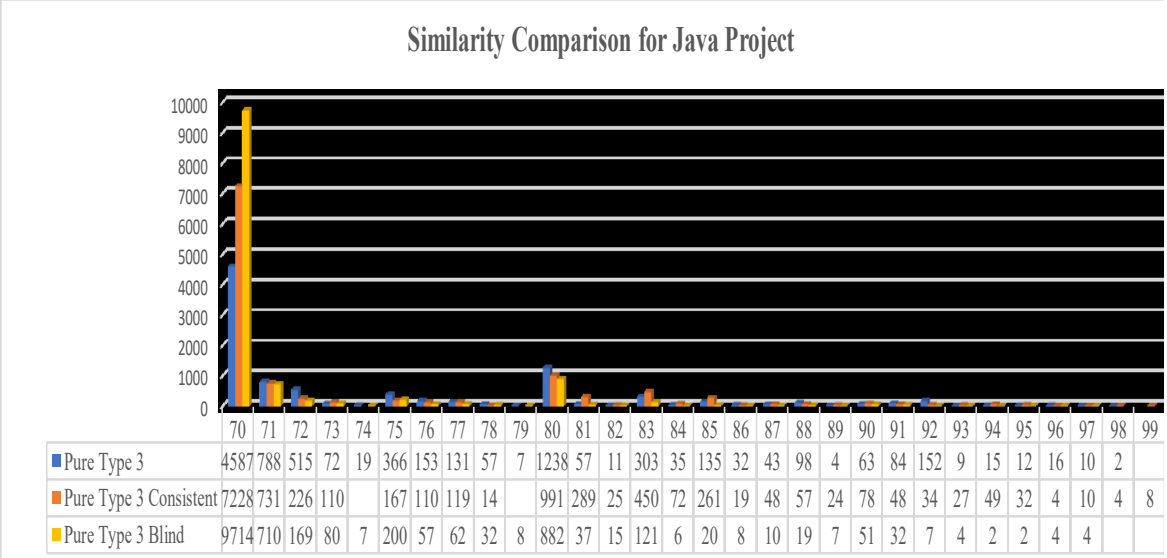
4.2.5 Number of occurrences of clone fragments for each similarity

In this section, we study the occurrences of clone fragments for various similarity values. Since Type-1, Pure Type-2 Consistent, and Pure Type-2 Blind have 100%

similarity, we did not analyze the corresponding fragments. Fragments for the other types are at least 70% similar.

For C projects, Illustration 13 shows similarity values on the x-axis and prevalence of fragments with such similarity values in the three Pure Type-3 clone classes on the y-axis. As expected, prevalence decreases as similarity increases. Also, when we compare fragment similarity (Illustration 13) and line differences of fragments (Illustration 9), there is no obvious relation to be found: e.g., when fragments similarity increases, fragments line difference does not necessarily increase or decrease. Indeed, when further investigating Pure Type-3 fragments, we discovered that some fragments differ by two lines with a similarity of 75%, while other fragments that also differ by two lines have a similarity of 96%.

We cannot find a relation between the number of line differences (Illustration 9) in a particular fragment and the fragment similarity (Illustration 13). We also performed the same analysis on the Java project; Illustration 14 has the same structure as Illustration 13. Again, prevalence decreases as similarity increases. Also, when we compare the results of Illustration 11 and Illustration 12 on the one hand, which show the number of occurrences of clone fragments for each number of line differences for Pure Type-3, Pure Type-3 Consistent, and Pure Type-3 Blind clones, with those of Illustration 14 on the other hand, which shows the number of occurrences of clone fragments for different similarity values for the same clone types, we observe that fragments similarity (Illustration 14) does not actually reflect how many lines differences particular fragments have (Illustration 11 and Illustration 12). Indeed, when further investigating those Pure Type-3 fragments, we



	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
Pure Type 3	4587	788	515	72	19	366	153	131	57	7	1238	57	11	303	35	135	32	43	98	4	63	84	152	9	15	12	16	10	2	
Pure Type 3 Consistent	7228	731	226	110		167	110	119	14		991	289	25	450	72	261	19	48	57	24	78	48	34	27	49	32	4	10	4	8
Pure Type 3 Blind	9714	710	169	80	7	200	57	62	32	8	882	37	15	121	6	20	8	10	19	7	51	32	7	4	2	2	4	4		

Illustration 14 Similarity values for Type-3, Type-3 Consistent, and Type-3 Blind clone fragments in Java Project.

discovered that some fragments differ by two lines with a similarity of 71%, while other fragments that also differ by two lines have a similarity of 98%.

We cannot find a relation between the number of line differences (Illustration 11 and Illustration 12) in a particular fragment and the fragment similarity (Illustration 14).

We computed the statistical correlation between similarity and number of line differences for Pure Type-3, Pure Type-3 Consistent and Pure Type-3 Blind clones. We observe weak negative correlations: -0.24 for Pure Type-3, -0.14 for Pure Type-3 Consistent, -0.14 for Pure Type-3 Blind for C projects, and -0.01 for Pure Type-3, -0.34 for Pure Type-3 Consistent, -0.18 for Pure Type-3 Blind for Java project. We conclude that similarity does not tell us exactly how many lines differ between clone fragments.

4.3 Threats to Validity

Like any study of this nature, results and conclusions are subject to validity threats. Validity threats are typically categorized into threats to conclusion validity, threats to internal validity, threats to construct validity, and threats to external validity [73].

We foresee some threats to conclusion validity. This threat corresponds to the inability to draw valid conclusions. The reason for this threat is related to insufficient data to draw logical conclusions. The percentage of redundant information of all clone types for the subject systems (from industry) is not representative of all systems and test suites, and the percentage of redundant information results is not generalizable. To overcome this threat, we used several subject systems in C and Java with varied sizes and complexities. We nevertheless do not make claims that the observations we make are generalizable to any software; more experiments of the kind we conducted are necessary. As we alluded to earlier in this manuscript, some important differences we observed between C test code

and Java test code may be due to various factors including the training of the engineers/developers contributing tests, the practices in the different teams, the specifics of the software under tests, the test requirements.

Concerning some internal validity threats to our work, the tool we used to detect clones could fail to retrieve some of the clones in the subject test suites. To ease this concern, we are aware of the effects of configurations on the performance of the tool, so we used multiple pre-defined configurations for NiCad. At the same time, there are configurations that meet the definitions of Type-1, Type-2 Consistent, Type-2 Blind, Type-3, Type-3 Consistent and Type-3 Blind clones that are used in many different publications [10, 49, 78]. In the end, we used the default configuration of NiCad since they have been shown to lead to both high precision and high recall at finding near-miss clones [68].

Construct validity is concerned with the relationship between theory and observation. We have used a procedure to remove overlapping clone classes and pairs from raw NiCad results; we have partly automated the procedure to reduce sources of mistakes. To reduce this threat, we have automated as much as possible, and we manually verified the outcome results on sample data to make sure that the procedure worked as intended. Another possible threat is represented by the analysis of results that are performed manually, but to reduce the threat, we verified and discussed the results with engineers of our industry partner.

There can be threats to external validity as well. These threats are related to the external aspects of the experiments that can limit the generalization of the results. This threat is primarily associated with the subject systems and test suites selection for performing the analysis and the test code clones derived from them. Our study is conducted solely in an

industry context. We do not claim the results are generalizable. We note that we also did not cherry-pick the industry projects; they were provided by our industry partner and are complete suites for the application logic they verify.

4.4 Conclusion

We set out to investigate what we suspected as a cause of increased costs of testing in the industry. By measuring the distribution of software clones in industry test code, we have analyzed and observed clone fragments detected by the NiCad clone detection tool in two C projects and one Java project that was overall made up of 286,168 and 791,384 lines of code (LOC), respectively, to answer our research questions.

We detected a high percentage of cloned test code: about 49% LOC of C test code and 73% LOC of Java test code. This is also equivalent to 36% of the C test cases and 94% of the Java test cases. Our results indicate that the projects we analyzed have a high level of redundancy, thereby leading to a high cost for the software testing activity. When we compare the clone ratio for the C and Java test code with previous work conducted by Tsantalis et al. [49], we find that the clone ratio in the test code of open source projects is between 14.3% and 81.9% (of LOC). This indicates that our case study is average in comparison. Also, Tsantalis et al.'s results [49] may be inflated compared to our results because we use the clone reduction procedure to remove the overlapped clone information from the clone detection tool results.

The study of distribution of clone classes' sizes shows that the vast majority of the clone classes, irrespective of the type, have a clone class size of two: two fragments are similar (or identical). We observed that while differences up to 30% (i.e., fragments in a clone class are at least 70% similar) may exist when comparing pretty-printed versions of

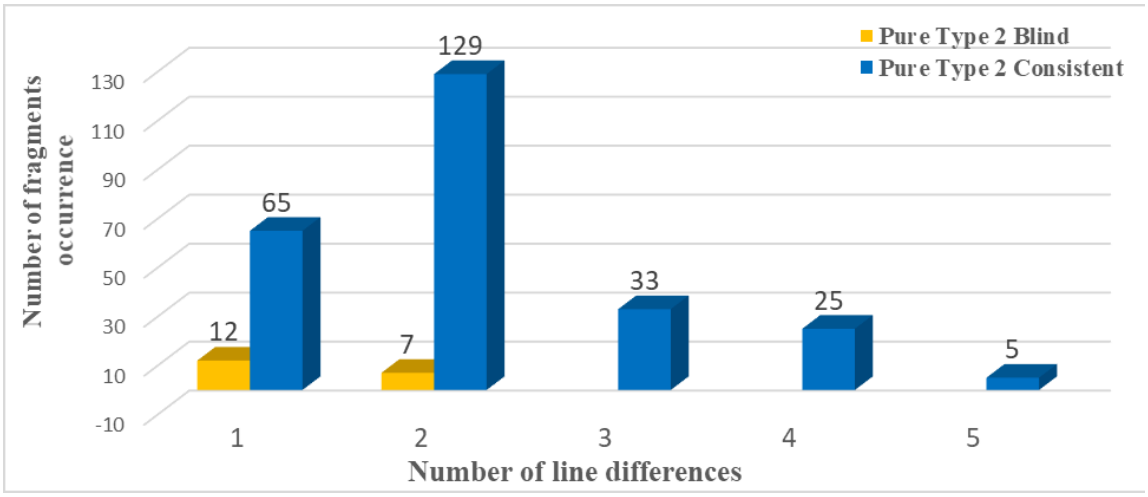


Illustration 15 Distribution of line differences between clone fragments for Type-2 Consistent and Type-2 Blind: C projects

code fragments, those differences actually happen is a very small (most often one or two) number of LOC in the original source code. The test code we analyzed shows a lot of similarities (49% and 73% shows some level of duplication for C and Java code, respectively), but duplications are varied and very localized.

Given that NiCad pre-processes source code into a pretty-printed version prior to identifying clones, clone information reported by NiCad (e.g., fragment similarity) is on the pretty-printed version of the code, not the original source code. The similarity percentage computed by NiCad does not tell us exactly how many lines of actual source code fragments are similar in clone classes. Differences between clone fragments, for instance in a Type-2 clone class, as reported by NiCad may therefore not be entirely representative of differences between the corresponding series of lines of source code. For instance, a 30% difference between two pretty-printed fragments may take place in one single line of the original source code; Conversely, 100% similar pretty-printed fragments, after renaming (e.g., Type-2) may be from original (source code) fragments that show

differences in many lines. We then considered it would be important to look at how information produced by NiCad, specifically fragment (size) and similarity, relate to differences in terms of LOC in the original source code. As one could expect, we observed a moderate to strong correlation between the fragments' size and the number of LOC difference, depending on the type of clone and the subject source code (Java or C): when the fragments' size increases, the number of LOC differences in the source code increases as well. On the other hand, the analysis of the statistical correlation between similarity and LOC difference is inconclusive. Similarity, as reported by NiCad, does not tell us exactly how many LOC differ between clone fragments in the original source code. We conclude, although our work should be replicated, that similarity and LOC difference may measure different aspects of test code cloning.

We note that we decided to post-process the information produced by NiCad to remove what we refer to as overlapping clone information, that is clone classes reported by NiCad with a more permissive clone type (e.g., Type-2) include clone classes obtained with a less permissive clone type (Type-1). We decided to remove overlapping clone information: e.g., all Type-1 clone classes that are also reported by NiCad as Type-2 clone classes are removed from the Type-2 clone class results. We felt that was a necessary step to precisely identify results for separate clone types; Since we cannot rule out, at the outset, that for instance trends on Type-1 clones may be different from trends on Type-2 clones, it is important to study clone type results separately for each clone type. One immediate consequence of our decision to remove overlapping clone data is that comparing our results with others should be done with care; if others have used data returned by NiCad without post-processing, it would be expected that their results would be inflated compared to ours.

Chapter 5: Procedure to Merge Clone Classes

Our investigation shows a clone fragment can belong to different clone classes under different clone types. This is different from the notion of overlapping between clone type results, which we introduced earlier (recall Section 3.1).

Test code duplication usually occurs because the testers are required to write test cases that do similar things. The test cases frequently exercise scenarios that are variations on the same objective, resulting in many copies of the same test codes. That means that test codes may require similar setup, teardown, logic or result verification logic (oracle). Therefore, when we use the clone detection tools to detect test code clones, similar test codes are reported under different clone classes/ pairs because these clone pairs/classes are classified according to the level of similarity between the fragments. For instance, suppose a Type 2 clone class has two fragment f_1 and f_2; f_1 and f_2 are 90% similar. Also, assume a Type 3 clone class has two fragment f_3 and f_4; f_3 and f_4 are at least 70% similar. Suppose f_1 and f_3 are the same fragment: f_1 is 90% similar to f_2 and at least 70% similar to f_4. One may want to refactor code for each clone class separately, i.e. attempt to refactor f_1 and f_2 separately from f_3 and f_4. We instead argue that merging such clone classes of various clone types but that share fragments before refactoring their fragments will offer more opportunities to improve test code.

In this chapter, we define and implement a Merge Clone Classes Procedure to benefit from broader refactoring opportunities. In this chapter, we first show an example of shared clone information between clone classes of different types (Section 5.1). Section 5.2 discusses the Merge Clone Classes procedure. Section 5.3 shows how we can apply the Merge Clone Classes procedure to NiCad results (after removing overlapping information),

and we present results in Section 5.4 of applying this procedure on the industry code. In Section 5.5, we conclude this chapter.

5.1 Illustrating Shared Clone Information

A clone fragment may be reported under different clone classes of different clone Types. This can be done, among other things, to renaming variables, different assignment data, assertions. Illustration 16 shows some of these reasons with three fragments (fragment A, fragment B, and fragment C) and the corresponding consistent and blind renaming strategies. When comparing the resulting fragments (i.e., fragment A, fragment B, and fragment C), one can identify that fragment A and B are not Type-1 clones since local variable m is renamed to n in fragment B; fragment A and C are not Type-1 clones either since the local variable m is assigned to different values in the two fragments; fragment B and C are not Type-1 clones for similar reasons. However, all the three fragments A, B, and C belong to a Type-3 clone class with a maximum of 30% differences. Since fragments A-Consistent and B-Consistent are 100% similar, fragments A and B belong to a Type-2 Consistent clone class. For similar reasons, accounting for small differences ("x7 x2 = 5;" vs "x7 x2 = -5;") which are less than 30% of other consistent fragments, fragments A, B and C belong to a Type-3 Consistent clone class. For similar reasons, fragments A, B, and C belong to a Type-3 Blind clone class, and fragments A and B belong to a Type-2 Blind clone class. In other words, fragments A, B, and C belong to a Type-3 none clone class, to a Type-3 Consistent clone class, and a Type-3 Blind clone class. Also, fragments A-Consistent, and B-Consistent will be identified as Type-2 Consistent, and fragments A-Blind, and B-Blind are identified as Type-2 Blind.

If we follow the standard definitions of clone types, fragments A and B belong to Type-2 Consistent, Type-2 Blind, Type-3 none, Type-3 Consistent, and Type-3 Blind clone classes. Fragments A, B, and C belong to Type-3 none, Type-3 Consistent, and Type-3 Blind clone classes. Since Type 3 clones are a combination for all clone types, one may decide to refactor clones using Type 3 only.

If we follow the Pure version of clone types, then CC_1 belongs to a Pure Type-2 Consistent clone class and CC_2 belongs to a Pure Type 3 clone Type. Either we follow the standard definitions of clone types or the Pure version of the clone types, clone classes have shared fragments and this may have impacts on refactoring activities. For instance,

`void TestCase-1() {`
`  int m = 5;`
`  int s = 0;`
`  int p = 1;`
`  for (int j=1; j<=m; j++) {`
`  s = s + j;`
`  p = p * j;`
`  foo(s, p);`
`  }`
`}`
	`void TestCase-2() {`
	`  int n = 5;`
	`  int s = 0;`
	`  int p = 1;`
	`  for (int j=1; j<=n; j++) {`
	`  s = s + j;`
	`  p = p * j;`
	`  foo(s, p);`
	`  }`
	`}`
		`void TestCase-3() {`
		`  int m = -5;`
		`  int s = 0;`
		`  int p = 1;`
		`  for (int j=1; j<=m; j++) {`
		`  s = s + j;`
		`  p = p * j;`
		`  foo(s, p);`
		`  }`
		`}`

Fragment A	**Fragment B**	**Fragment C**

`void x1() {`
`  x7 x2 = 5 ;`
`  x7 x3 = 0;`
`  x7 x4 = 1;`
`  for (x7 x5=1; x5<=x2; x5++)`
`{`
`  x3 = x3 + x5;`
`  x4 = x4 * x5;`
`  x6(x3, x4);`
`  }`
`}`
	`void x1() {`
	`  x7 x2 = 5 ;`
	`  x7 x3 = 0;`
	`  x7 x4 = 1;`
	`  for (x7 x5=1; x5<=x2; x5++)`
	`{`
	`  x3 = x3 + x5;`
	`  x4 = x4 * x5;`
	`  x6(x3, x4);`
	`  }`
	`}`
		`void x1 () {`
		`  x7 x2 = -5 ;`
		`  x7 x3 = 0;`
		`  x7 x4 = 1;`
		`  for (x7 x5=1; x5<=x2; x5++)`
		`{`
		`  x3 = x3 + x5;`
		`  x4 = x4 * x5;`
		`  x6(x3, x4);`
		`  }`
		`}`

Fragment A—consistent	**Fragment B—consistent**	**Fragment C—consistent**

`void x() {`
`  x x = 5 ;`
`  x x = 0;`
`  x x =1;`
`  for (x x=1; x<=x; x++) {`
`  x = x + x;`
`  x = x * x;`
`  x(x, x);`
`  }`
`}`
	`void x() {`
	`  x x = 5 ;`
	`  x x = 0;`
	`  x x =1;`
	`  for (x x=1; x<=x; x++) {`
	`  x = x + x;`
	`  x = x * x;`
	`  x(x, x);`
	`  }`
	`}`
		`void x() {`
		`  x x = -5 ;`
		`  x x = 0;`
		`  x x =1;`
		`  for (x x=1; x<=x; x++) {`
		`  x = x + x;`
		`  x = x * x;`
		`  x(x, x);`
		`  }`
		`}`

Fragment A—blind	**Fragment B—blind**	**Fragment C—blind**

Illustration 16 Example of clone fragments that are reported in different clone classes

suppose we decide to refactor clones by detecting Type-3 Consistent clones only. Using the above example, fragments A and B belong to Type-3 Consistent clone class (CC_1), and Fragments A, B, and C belong to Type-3 Consistent clone class (CC_2). CC_1 and CC_2 share two fragments A and B. One may want to refactor CC_1 and CC_2 separately; i.e. attempt to refactor fragment A and B separately from fragment C. Since refactoring CC_1 and CC_2 independently increases the effort of refactoring the clone fragments, we argue that merging such clone classes that share fragments prior to refactoring their fragments will offer more opportunities to improve test code. For instance, instead of refactoring CC_1 and CC_2 separately, one may merge CC_1 and CC_2 that share two fragments into one set of fragments before refactoring; consequently, fragments A, B, and C belong to the same clone class (CC_3). After that, we can refactor CC_3.

If we follow the standard definitions of clone types, we cannot easily distinguish the type of clone fragments in a Type 3 clone class. For instance, in a Type-3 clone class, one may find fragments A and B that in fact belong to a Type-2 Consistent class (CC_1), while fragments A, B, and C belong to a Type-3 Consistent clone class (CC_2). CC_1 and CC_2 share two fragments A and B. Also, CC_1 and CC_2 are both identified as Type-3 Consistent. Suppose for instance that we decide to refactor clones by detecting Type-3 Consistent clones only. One may refactor CC_1 and CC_2 separately; i.e. attempt to refactor fragment A and B separately from fragment C. It will then become difficult and therefore expensive to recognize that the refactored version(s) of fragments A and B may also be refactored with fragment C. Since refactoring CC_1 and CC_2 independently increases the effort of refactoring the clone fragments, we argue that merging such clone classes that share fragments prior to refactoring their fragments will offer more opportunities to improve test

code. As a result, we cannot analyze any relationship between clone types and refactoring decisions if the standard definitions of clone types are used.

Following the Pure version of clone types helps to analyze the effects of clone type on refactoring decisions and the relationship between clone types and the (test) patterns used during refactoring. In this study, we find the Pure version of clone types prior to merging clone classes because we want to study the relationship of clone types and the used (test) patterns and the effects of clone types on refactoring decisions.

Either we follow the standard definitions of clone types or the Pure version of the clone types, we argue to use the Merge Clone Classes procedure prior to refactoring clones because merging clone classes will provide more opportunities for refactoring test codes and reduce the effort of refactoring and maintaining test code.

5.2 A General Procedure to Merge Clone Classes

In this section, we will discuss the general principle of the Merge Clone Class algorithm we use to merge different clone classes that share clone fragments to obtain new clone classes for refactoring purposes. To merge clone classes that share a fragment, we suggest comparing code fragments in clone classes, to compare clone classes, and to compare clone data files as discussed below.

Principally, the procedure merges a set of clone fragments in a class that is obtained with a more permissive type with the set(s) of clone fragments in classes obtained with the less permissive type(s) when these classes have common fragments. The generated, merged clone classes will be considered during refactoring, instead of the initial clone classes.

For instance, suppose a Type 1 clone class (CC_1) has two fragment f_1 and f_2. Also, suppose a Type 2 clone class (CC_2) has two fragment f_3 and f_4. And suppose a Type 3 clone

```
Algorithm 4: Find if a clone fragment exists in another clone class
Input: two clone classes
Output: Boolean
function cloneFragmentExist(CC-1, CC-2)
// CC-1.Fragments is the collection of fragments in clone class CC-1
  for each cfi in CC-1.Fragments, do // cfi is a clone fragment in CC-1
    for each cfj ∈ CC-2.Fragments, do
      if (compareFragments(cfi, cfj)  then  return true;  end if
    end for
  end for

  return false;
end function
```

```
Algorithm 5. Comparison of two clone files
Input: two clone class files: file1, file2
Output: file1 with merged clone classes
function mergeDuplicatedClasses (file1, file2)
  // file1.cloneClasses is the set of clone classes in file1.
  for each cpi in file1.cloneClasses do
    for each cpj in file2.cloneClasses do
      if (cloneFragmentExist (cpi , cpj)), then
          // merge clone classes without duplicated fragments
          cpi.Fragments.appendFragments(cpj.Fragments)
          file2.cloneClasses.remove(cpj)
      end if
    end for
  end for
end function
```

Illustration 17 General Algorithms to Merge Clone Classes.

class (CC_3) has three fragment f_5, f_6 and f_7. Suppose f_1, f_3 and f_6 are the same fragments. To merge clone classes CC_1, CC_2, and CC_3 we proceed as follows: compare CC_1 and CC_2, since these clone classes share fragment f_1 and f_3, we merge CC_1 and CC_2, we obtain a clone class (CC_4) that has three fragments (f_1, f_2 and f_4); We then remove CC_2 from the Type 2 clone class; Second, we compare and merge clone classes CC_3 and CC_4, and we obtain a clone class (CC_5) that has five fragments (f_1, f_2, f_4, f_5 and f_7); We then remove CC_3 from the Type 3 clone class; Clone class CC_5 is used later to refactor clone classes.

The first step is to identify whether a clone fragment exists in another clone class. We consider that two fragments are equal if they have an equal start line, an equal end line, and an equal file name, as described in Section 3.1 (see algorithm 4 in Illustration 17).

In the second step, we compare two clone files, and if two clone classes share a clone fragment, we merge the clone classes into one new clone class with no duplicated fragments — algorithm 5 in Illustration 17.

5.3 Applying the Merge Clone Classes algorithm to NiCard Results

As discussed earlier, a shared clone fragment between different clone classes can potentially be found when comparing different types of clone class results. The Merge Clone Classes procedure can be applied either on the NiCad results directly or on the Pure version of the clone results. When we follow the standard definitions of clone types, Type-3 clone types are a combination of all clone types; then, we can directly apply the Merge Clone Classes procedure on Type-3 clone classes. However, the risk of applying the procedure directly on Type-3 clone classes is that we cannot distinguish the type of clone classes to which belong the Type-3 clone fragments. For instance, recall the example in Section 5.1 where fragments A and B belong to Type-2 Consistent class and fragment C belongs to Type-3 Consistent class; when we directly apply the merge clone classes procedure on the clone classes under Type-3 Consistent, we cannot identify if a particular clone fragment belongs to Type-2 Consistent or Type-3 Consistent clone type. The consequence of using the standard definitions of clone types is that we cannot analyze any relationship between clone types and refactoring decisions. If we follow the Pure version of clone types, we can analyze the effects of clone type on refactoring decisions and the relationship between clone type and the patterns that are used during refactoring. Either we follow the standard definitions of clone types or the Pure versions of the clone types, we argue against using the Merge Clone Classes procedure before refactoring clones because

the merging of clone classes will provide more opportunities for refactoring test code and reduce the effort of refactoring and maintaining test code.

We, therefore, devised the following procedure, relying on previously defined algorithms, to merge clone classes from XML files returned by NiCad, postprocessed to identify Pure clone classes, and identify clone classes. In the context of NiCad, we need the following steps, in the specified order (NiCad returns XML files and .xml file extensions are omitted from file names below):

1. Type-1 vs. Pure Type-2 Consistent: If a Type-1 clone class shares a fragment with a Pure Type-2 Consistent clone class, we merge the two classes. We call:

   ```
   mergeDuplicatedClasses(Type1, PureType2Cons)
   ```

2. Type-1 vs. Pure Type-2 Blind: : If a Type-1 clone class shares a fragment with a Pure Type-2 Blind clone class, we merge the two classes. We call:

   ```
   mergeDuplicatedClasses(Type1, PureType2Blind)
   ```

3. Type-1 vs. Pure Type 3 none: If a Type-1 clone class shares a fragment with a Pure Type-3 none clone class, we merge the two classes. We call:

   ```
   mergeDuplicatedClasses(Type1, PureType3)
   ```

4. Type-1 vs. Pure Type-3 Consistent: If a Type-1 clone class shares a fragment with a Pure Type-3 Consistent clone class, we merge the two classes. We call:

   ```
   mergeDuplicatedClasses(Type1, PureType3Cons)
   ```

5. Type-1 vs. Pure Type 3 Blind: If a Type-1 clone class shares a fragment with a Pure Type-3 Blind clone class, we merge the two classes. We call:

   ```
   mergeDuplicatedClasses(Type1, PureType3Blind)
   ```

6. Pure Type-2 Consistent vs. Pure Type-2 Blind: If a Pure Type-2 Consistent clone class shares a fragment with a Pure Type-2 Blind clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Cons, PureType2Blind)
```

7. Pure Type-2 Consistent vs. Pure Type-3 none: If a Pure Type-2 Consistent clone class shares a fragment with a Pure Type-3 none clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Cons, PureType3)
```

8. Pure Type-2 Consistent vs. Pure Type-3 Consistent: If a Pure Type-2 Consistent clone class shares a fragment with a Pure Type-3 Consistent clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Cons, PureType3Cons)
```

9. Pure Type-2 Consistent vs. Pure Type-3 Blind: If a Pure Type-2 Consistent clone class shares a fragment with a Pure Type-3 Blind clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Cons, PureType3Blind)
```

10. Pure Type-2 Blind vs. Pure Type-3 none: If a Pure Type-2 Blind clone class shares a fragment with a Pure Type-3 none clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Blind, PureType3)
```

11. Pure Type-2 Blind vs. Pure Type-3 Consistent: If a Pure Type-2 Blind clone class shares a fragment with a Pure Type-3 Consistent clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Blind, PureType3Cons)
```

12. Pure Type-2 Blind vs. Pure Type-3 Blind: If a Pure Type-2 Blind clone class shares a fragment with a Pure Type-3 Blind clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType2Blind,PureType3Blind)
```

13. Pure Type-3 none vs. Pure Type-3 Consistent: If a Pure Type-3 none clone class shares a fragment with a Pure Type-3 Consistent clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType3, PureType3Cons)
```

14. Pure Type-3 vs. Pure Type-3 Blind: If a Pure Type-3 none clone class shares a fragment with a Pure Type-3 Blind clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType3,PureType3Blind)
```

15. Pure Type-3 Consistent vs. Pure Type-3 Blind: If a Pure Type-3 Consistent clone class shares a fragment with a Pure Type-3 Blind clone class, we merge the two classes. We call:

```
mergeDuplicatedClasses(PureType3Cons, PureType3Blind)
```

A similar procedure can be followed if different threshold values are of interest. If a clone class shares a fragment with a clone class at a lower threshold value, then these clone classes are merged, and the fragments of the clone class obtained with the lower threshold value are excluded from the higher threshold value clone class set of fragments.

5.4 Results of Applying the Merge Clone Classes algorithm

We have applied the Merge Clone Classes algorithm only on the industry C test code, but not the Java test code. Prioritizing the refactoring of C test code was a corporate decision of our industry partner.

In this section, we present the results after applying the Merge Clone Classes algorithm for each test suite from the industry C test code as shown in Table XIV. For each test suite

the table has three groups of data: (i) initial results after using our clone reduction procedure, (ii) results after applying the Merge Clone Classes algorithm from results without duplication, (iii) percentages of reduction when comparing results before and after executing our merge clone classes procedure. For the first clone detection group, the table indicates the number of clone classes, the total number of fragments for each test suite, and the total fragments size. For the second clone detection group, the table indicates the number of clone classes, the total number of fragments for each test suite, the size of the total fragments. In the third clone detection group, the table indicates the reduction percentage for each of the number of clone classes, the total number of fragments for each test suite, and the size of the total fragments. For instance, for test suite TS 2, the clone detection results show 345 clone classes, 990 fragments for a cumulative of 34,491 LOC; After applying the Merge Clone Classes algorithm, we find 480 fragments in 126 clone classes for a cumulative of 20,665 LOC, which means the number of clone classes is reduced by 63.5%, the fragment's size is reduced by 51.5% and the number of fragments is reduced by about 40.1%. We found that the average clone class reduction for all the test suites is approximately 63%, and the average fragments reduction for all the test suites is equal to 54.4%, and the LOC reduction percentage is equal to 40.1%. Also, we noticed that test suites TS 21, TS 25, TS 31, TS 35, and TS 36 have a 0.0% reduction percentage for the clone classes, clone fragments, and total LOC. When we investigate these test suites, we found that the test suites' size is less than 500 LOC except for TS 25 where the test suite's size is equal to 2,321 LOC which are small sizes compared to other test suites. Also, we found that these test suites have a few or no clones. This explains why we have a 0.0% reduction percentage.

Moreover, we noticed that TS 18 has a LOC reduction percentage smaller than 10%. TS 18 has fewer than 1,000 LOC, and it has only a few duplications. This indicates that small size test suites are well maintained, and it has few or no duplications.

Table XIV Results after applying the Merge Clone Classes algorithm

Test Suite name	Number of Clone Classes	Number of fragments	Total LOC for Fragments	Number of Clone Classes	Number of fragments	Total LOC for Fragments	Number of Clone Classes	Number of fragments	Total LOC for Fragments
	(Before applying the Merge Clone Classes algorithm)			(After applying the Merge Clone Classes algorithm)			Reduction Percentage		
TS 1	11	77	1901	4	34	980	63.6%	55.8%	48.4%
TS 2	345	990	34491	126	480	20665	63.5%	51.5%	40.1%
TS 3	35	94	5533	13	39	3411	62.9%	58.5%	38.4%
TS 4	38	126	6263	6	58	3883	84.2%	54.0%	38.0%
TS 5	15	57	525	2	23	312	86.7%	59.6%	40.6%
TS 6	6	23	578	4	11	338	33.3%	52.2%	41.5%
TS 7	8	22	466	3	17	385	62.5%	22.7%	17.4%
TS 8	6	28	4431	2	13	2867	66.7%	53.6%	35.3%
TS 9	45	243	8330	19	108	4702	57.8%	55.6%	43.6%
TS 10	3	6	61	1	3	32	66.7%	50.0%	47.5%
TS 11	3	6	63	1	2	26	66.7%	66.7%	58.7%
TS 12	33	205	6437	15	90	3826	54.5%	56.1%	40.6%
TS 13	17	134	6647	12	55	3764	29.4%	59.0%	43.4%
TS 14	8	22	1334	2	6	646	75.0%	72.7%	51.6%
TS 15	29	101	12564	10	42	8225	65.5%	58.4%	34.5%
TS 16	19	78	4510	8	33	2661	57.9%	57.7%	41.0%
TS 17	15	51	2407	2	15	1078	86.7%	70.6%	55.2%
TS 18	8	21	141	6	19	131	25.0%	9.5%	7.1%
TS 19	3	7	48	2	5	39	33.3%	28.6%	18.8%
TS 20	52	210	6868	26	100	4470	50.0%	52.4%	34.9%
TS 21	0	0	0	0	0	0	0.0%	0.0%	0.0%
TS 22	45	138	7837	7	66	4874	84.4%	52.2%	37.8%
TS 23	22	60	2402	4	21	1279	81.8%	65.0%	46.8%
TS 24	2	5	215	1	2	98	50.0%	60.0%	54.4%
TS 25	1	3	22	1	3	22	0.0%	0.0%	0.0%
TS 26	49	220	8859	21	94	5085	57.1%	57.3%	42.6%
TS 27	10	33	415	5	10	226	50.0%	69.7%	45.5%
TS 28	56	221	9783	17	100	5417	69.6%	54.8%	44.6%
TS 29	20	80	2160	9	29	1195	55.0%	63.8%	44.7%
TS 30	5	11	258	1	2	57	80.0%	81.8%	77.9%
TS 31	1	1	13	1	1	13	0.0%	0.0%	0.0%
TS 32	9	18	557	1	7	308	88.9%	61.1%	44.7%
TS 33	27	83	1881	4	33	987	85.2%	60.2%	47.5%
TS 34	28	94	2306	23	58	1934	17.9%	38.3%	16.1%
TS 35	1	2	118	1	2	118	0.0%	0.0%	0.0%
TS 36	0	0	0	0	0	0	0.0%	0.0%	0.0%
Total	975	3470	140424	360	1581	84054	63.1%	54.4%	40.1%

After using the Merge Clone Classes algorithm, the clone classes are reduced by approximately 17-88%, and the number of fragments is reduced by approximately 9-81%, we conclude that using the Merge Clone Classes procedure prior to refactoring clones reduces the effort of refactoring and maintaining test codes.

5.5 Threats to Validity

We foresee some threats validity to our merging clone procedure. Validity threats are typically categorized into threats to conclusion validity, threats to internal validity, threats to construct validity, and threats to external validity [73].

Conclusion validity corresponds to the inability to draw valid conclusions. The reason for this threat is related to insufficient data to draw logical conclusions. The percentage of reduction of clone classes, fragments, and LOC for the test suites (from C industry test codes) is not representative of all systems and test suites, and the percentage of reduction results is not generalizable. To overcome this threat, we used several (36) test suites in C with varied sizes and complexities. We nevertheless do not make claims that the observations we make are generalizable to any software; more experiments of the kind we conducted are necessary.

Concerning some internal validity threats to our work, the tool we used to detect clones' instances could fail to retrieve some of the clones' examples in the test suites. To ease this concern, we are aware of the effects of configurations on the performance of the tool, so we used multiple pre-defined settings for NiCad. At the same time, some configurations meet the definitions of Type-1, Type-2 Consistent, Type-2 Blind, Type-3, Type-3 Consistent and Type-3 Blind clones that are used in many different publications [49, 65,

79-81]. In the end, we used the default configuration of NiCad since they have been shown to lead to both high precision and high recall at finding near-miss clones [68].

Construct validity is concerned with the relationship between theory and observation. We have used a procedure to remove duplicated clone classes and pairs from raw NiCad results; we have partly automated the procedure to reduce sources of mistakes. To minimize this threat, we have automated as much as possible, and we manually verified the outcome results on sample data to make sure that the procedure worked as intended.

There can be threats to external validity, as well. These threats are related to the external aspects of the experiments that can limit the generalization of the results. This threat is primarily associated with the test suites selection for performing the analysis and the test code clones derived from them. Our study is conducted solely in an industry context. We do not claim the results are generalizable. We note that we also did not cherry-pick the industry projects; they were provided by our industry partner and are complete suites for the application logic they verify.

5.6 Conclusion

In this chapter, we found that a clone fragment can belong to different clone classes under different clone types because of renaming variables, different assignments of data, assertions, at least one-line difference, and so many other reasons. We study the test code clones with a shared fragment; indeed, the test cases frequently exercise scenarios that are variations on the same objective, resulting in many copies of the same test codes. In other words, test code may require similar setup, teardown, logic or result verification logic (oracle). Consequently, the clone detection tools report similar test code under different clone classes/ pairs because these clone pairs/classes are classified according to the level

of similarity between the fragments. For instance, a clone fragment that belongs to a Type-1 clone class because of 100% similarity between the clone class's fragments can also belong to a Type-3 clone class because of at least 70% similarity between the clone class's fragments.

Given this analytical observation, we first provide an example that clone classes might share fragments. We then defined and implemented a general Merge Clone Classes Procedure to merge clone classes that share fragments and illustrate how it can be applied to results produced by NiCad, a popular and useful clone detection tool. Our results confirm that a clone fragment can belong to different clone classes under different clone types. After that, we experiment on 36 test suites from industry that are written in C.

In conclusion, the Merge Clone Classes algorithm shows that the total number of clone classes is reduced by approximately 63%, the average fragments reduction for all the test suites is equal to 54.4%, and the LOC reduction percentage is equal to 40.1%. This indicates that the Merge Clone Classes algorithm helps to clusters the clone fragments into clone classes to reduce the total number of clone classes, fragments and total LOC significantly when conducting refactoring. Also, since the total number of clone classes is reduced by approximately 63%, then the refactoring effort will also decrease and this should help maintain test code more efficiently.

Chapter 6: Results of Clone Refactoring on Industrial Case Study

Clone detection and refactoring are usually referred to as two different tasks; indeed, we find that it is required to integrate these two tasks to improve the maintainability of the production and test code and it is known that refactoring can take place with clone detection [2]. The integration between these two tasks is usually referred to as clone refactoring. A clone refactoring approach is a possible approach for clone management. In this chapter, we concentrate on managing the test clones by using the clone refactoring approach on test clones.

Recalling Chapter 5:, we found that many pieces of test code tend to do similar things because testers frequently write test code to exercise scenarios that are variations of each other, resulting in pieces of code that are similar to each other to various degrees. Consequently, a clone detection tool (such as NiCad) reports similar pieces of test code under different clone pairs/ classes for different levels of similarity between the fragments. We confirmed experminately that a clone fragment that belongs to Type-2 clone class because it is 90% similar to other fragments in the class may also belong to a Type-3 clone class if it is 70% similar to other fragments. This led us to merge clone classes using the Merge Clone Classes procedure discussed in Chapter 5:. We believe that the merged clone classes offer more opportunities for refactoring test code, improve test code and reduce the effort of refactoring and maintaining test code.

In this chapter, we aim to explain how the refactorability of test code is affected by different clone properties. To this end, we conduct an empirical study by analyzing the 360 clone classes that we obtained by using the Merge Clone Classes procedure on the code shared by our industry partner.

The rest of this chapter is organized as follows. Section 6.1 discusses the experiment design. Section 6.2 introduces and discusses the results of our empirical study. Furthermore, we present the threat to validity, followed by the conclusion in Sections 6.3 and 6.4, respectively.

6.1 Experiment Design

In our study, since we are interested in the management of the clone in test code using the clone refactoring approach, we firstly run NiCad and collect the results it produces from the industry C project, then we refine, post-process those results with our clone reduction algorithm to obtain the pure version of clone types, and then finally we refine the pure version of clone types with our merging clone classes algorithm, and we obtain 360 clone classes to be evaluated by the researcher and engineers in order to refactor them. For each clone class, we obtain the clone class characteristics such as the clone class's properties (clone class's size, clone fragment's size), the refactoring decision, and the relationship between the clone class's properties and refactoring decisionsWe performed another analysis to examine whether the clone class's size affects the refactoring decision. Illustration 22 shows the violin plots for the size of the refactorable versus the non-refactorable clone classes. As it is evident from the plots, the size of clone classes is small, with an average size of 5.4 and 3.1 for refactorable and non-refactorable clone classes respectively. The median value is equal to 3.5 and two for refactorable and non-refactorable clone classes respectively, and thus we conclude that the refactorable and the non-refactorable clone classes contain a small number of fragments. Also, the refactorable clone classes tend to have a larger average size compared to the non-refactorable clone classes. We applied a one-tailed variant of the Mann-Whitney U test on the sizes of the refactorable

and non-refactorable clone classes with the following null hypothesis: "the size of refactorable clone classes is smaller than the size of non-refactorable clone classes." The null hypothesis was rejected with significance at a 95 percent confidence level based on $p - value < 2.96\ X\ 10^{-7}$; and therefore, we can conclude that the size of refactorable clone classes is greater than the size of non-refactorable clone classes.

6.2 Experimental Results

In this section, we separately answer each of the research questions we presented in the Introduction. Table XVII presents an overview of the refactorability analysis results for each test suites.

6.2.1 Refactoring Decision versus Clone Class Properties

In this section, we examine the relationship between the refactoring decision and the clone class properties, using both the 217 and 143 sets of refactorings.

We identified 15 properties of fragments' clone classes (Table XV) that affect the refactoring decision. Table XV shows the number of clone fragments that have any one of the 15 properties (plus the percentage of clone fragments). We discover that properties that most frequently appear in the clone classes are "Variation in assigned input data" and "Variation in assertions and assigned input data." There are 665 (61.24%) clone fragments that have similar logic but different input data and assertions. The next two most frequent properties are "Identical test case" (11.81%) and "Similar teardown" (12.45%). We refactored all the fragments (123 + 401 + 264 + 135) that exhibit any of these four properties.

The clone class's properties allow us to understand the primary differences between the clone class's fragments. We will define each clone class's property in the following paragraphs:

1. "Identical test case": This means that the clone class's fragments are identical with 100% similarity.
2. "Variable rename": This means that the clone class's fragments are 100% similar, except for variable names.
3. "Variation in assigned input data": This means that the clone class's fragments have a similar logic, but the testers have created different test cases with different input to test several scenarios.
4. "Variation in assertions and assigned input data": This means that the clone class's fragments have a similar logic, but the testers have created different test cases with different input to test several scenarios, and there are different assertion in the test cases to validate the test case output(s).
5. "Different function call": This means that the clone class's fragments have a similar logic, but the testers have created different test cases with different function calls to test several scenarios.
6. "Variation in for-loop condition and assigned input data": This means that the clone class's fragments have a similar logic, but the testers have created different test cases with different input to test several scenarios, and there are different for-loop conditions in the test cases to test those scenarios.
7. "Similar setup": This means that the clone class's fragments have a similar setup logic, but test cases exercise several scenarios.

8. “Variation in switch statement cases”: This means that the clone class’s fragments have a similar logic, but the testers have created different test cases with different switch statement cases to test several scenarios.
9. “Similar logic but different print messages”: This means that the clone class’s fragments have a similar logic, but the testers have created different test cases with different print messages to print different information.
10. “Variation in assertions”: This means that the clone class’s fragments have a similar logic, but the testers have created different test cases with different assertions to validate the test case output(s).
11. “Variation in if-statement condition and assigned input data”: This means that the clone class’s fragments have a similar logic, but the testers have created different test cases with different inputs to test several scenarios, and there are differences in if-statement conditions in the test cases to test several scenarios.
12. “Similar teardown”: This means that the clone class’s fragments have a similar teardown logic, but test cases exercise several scenarios.
13. “Similar setup and teardown”: This means that the clone class’s fragments have a similar setup and teardown logics, but test cases exercise several scenarios.
14. “Variation in if-statement condition”: This means that the clone class’s fragments have a similar logic, but test cases include different if-statement conditions to test several scenarios.
15. “Similar test cases with missing cleanup function call”: This means that the clone class’s fragments have a similar logic, but the testers have created different test cases with a missing cleanup function call in some test cases.

6.2.2 Refactoring Principle vs Clone Class Properties

We have applied different refactoring patterns on those fragments during clones refactoring depending on the properties discussed previously. Table XV shows the association between the pattern being used and the properties of the clone classes.

We applied the ***Extract Method*** on each clone class which fragments are 100% similar (Property-1) to eliminate test code duplication, and we then applied the ***Pull Up Method*** (to the superclass). We also applied the ***Extract Method*** and ***Pull Up Method*** on the following clone classes' properties: a clone class's fragments with similar logic but different print message (Property-9), a clone class's fragments with a variable renaming (Property-2), and a clone class's fragments with a missing function call such as a call to a clean-up function (Property-15).

We applied the ***Test Utility Method*** to a clone class's fragments that have a similar test logic that is with at least one of the following clone class's properties: a clone class's fragments with similar setup (Property-7), a clone class's fragments with similar teardown (Property-12), a clone class's fragments with similar setup and similar teardown (Property-13), a clone class's fragments with variation in switch statement cases (Property-8), and a clone class's fragments with variation in if-statement condition (Property-14). We moved the common test logic into a Test Utility Method; after that, we can call this method from various tests within a single test suite. If there is any variation in the test cases, we can pass these variations as arguments to the Test Utility Method.

We applied the ***Parameterized Test*** pattern when a clone class's fragments have a similar test logic but with some variation: That is when a clone class's fragments show variations in assigned input data only (Property-3), show variations in assertions and

assigned input data (Property-4), have different function calls (Property-5), have variations in if-statement conditions and assigned input data (Property-11), have variations in for-loop conditions and assigned input data (Property-6), or have different assertions only (Property-10).

To apply the ***Parameterized Test*** pattern, we used the ***utility method*** to factor out the common logic and design the utility method so that it takes only the information that differs between fragments as arguments.

Table XV Properties of clone classes and association between patterns and clone class' properties

Pattern name	Property ID	Clone class properties	Number of clone fragments (Percentage)
Extract Method and Pull Up Method	Property-1	Identical test case	128 (11.81%)
	Property-2	Variable rename	17 (1.57%)
	Property-9	Similar logic but different print messages	13 (1.20%)
	Property-15	Similar test cases with missing cleanup function call	2 (0.18%)
Parameterized Test	Property-3	Variation in assigned input data only	401 (36.99%)
	Property-4	Variation in assertions and assigned input data	264 (24.35%)
	Property-5	Different function call	20 (1.85%)
	Property-6	Variation in for-loop condition and assigned input data	17 (1.57%)
	Property-10	Variation in assertions only	11 (1.01%)
	Property-11	Variation in if-statement condition and assigned input data	3 (0.28%)
Test Utility Method	Property-7	Similar setup	15 (1.38%)
	Property-8	Variation in switch statement cases	5 (0.46%)
	Property-12	Similar teardown	135 (12.45%)
	Property-13	Similar setup and teardown	51 (4.70%)
	Property-14	Variation in if-statement condition	2 (0.18%)

Table XVI Number of occurrences of each pattern and the properties of clone class

Validated by	Pattern name	Property ID	Clone class properties	Number of Clone classes Occurrences		
Engineers And Researcher	Extract Method and Pull Up Method	Property-1	Identical test case	14	Total: 19	Total: 54
		Property-2	Variable rename	2		
		Property-9	Similar logic but different print messages	2		
		Property-15	Similar test cases with missing cleanup function call	1		
	Parameterized Test	Property-3	Variation in assigned input data only	20	Total: 26	
		Property-4	Variation in assertions and assigned input data	2		
		Property-5	Different function call	0		
		Property-6	Variation in for-loop condition and assigned input data			
		Property-10	Variation in assertions only	4		
		Property-11	Variation in if-statement condition and assigned input data	0		
	Test Utility Method	Property-7	Similar setup	4	Total: 9	
		Property-8	Variation in switch statement cases	1		
		Property-12	Similar teardown	2		
		Property-13	Similar setup and teardown	1		
		Property-14	Variation in if-statement condition	1		
Only Researcher	Extract Method and Pull Up Method	Property-1	Identical test case	27	Total: 34	Total 99
		Property-2	Variable rename	5		
		Property-9	Similar logic but different print messages	2		
		Property-15	Similar test cases with missing cleanup function call	0		
	Parameterized Test	Property-3	Variation in assigned input data	26	Total: 49	
		Property-4	Variation in assertions and assigned input data	19		
		Property-5	Different function call	1		
		Property-6	Variation in for-loop condition and assigned input data	2		
		Property-10	Variation in assertions only	0		
		Property-11	Variation in if-statement condition and assigned input data	1		
	Test Utility Method	Property-7	Similar setup	2	Total: 16	
		Property-8	Variation in switch statement cases	0		
		Property-12	Similar teardown	10		
		Property-13	Similar setup and teardown	4		
		Property-14	Variation in if-statement condition	0		

Table XVI adds information about who validated each set of refactoring. The engineers and researcher decided to refactor four clone classes for Property-10, twenty clone classes for Property-3, and two clone classes for Property-4. In total, they applied the Parameterized Test pattern on twenty-six clone classes. The researcher has used the Parameterized Test patterns for Property-3, Property-4, and Property-10. In addition, they applied the Parameterized Test pattern for Property-5, Property-6, and Property-11. The researcher applied the Parameterized Test pattern on 49 clone classes in total.

The table shows that the most frequently used pattern is the Parameterized Test pattern (75 times in total). The Extract method and Pull up method are the second most commonly used pattern (53 times). The least used pattern is the Test Utility Method (25 times).

We can conclude by noting that the patterns we used to refactor test clones are relatively simple ones.

6.2.3 Reduction Ratio

In this section, we examine the reduction ratio achieved when refactoring clone classes as discussed in the previous section from the perspective of the LOC for each test suites. Since we are interested in how many clone classes and clone fragments are refactored, we calculated the ratio for the valid clone classes and clone fragments instead of the reduction ratio. For each test suite we analyzed, Table XVII shows the number of clone classes, the number of fragments and LOC in those classes before refactoring (i.e. result of Chapter 5:), the test suite size before refactoring, the number of valid clone classes and fragments along with their ratio, and the test suite size after refactoring along with its reduction percentage (in terms of LOC). For instance, TS-2 had 126 clone classes, made of a total of 480 fragments and those fragments amounted to a total of 20,665 LOC, and the initial size of the test suite was 87,818 LOC before refactoring; after refactoring TS-2 has 48 (38.1%) valid clone classes, which are made of a total of 385 (80.2%) clone fragments, and TS-2's size is reduced to 76,890 LOC (12.44%).

We inspected the 14 test suites that have 0% ratio for the clone classes, and fragments and 0% LOC reduction ratio: see highlighted rows in the table. All these test suites have a small number of clone tests, and these clones are either Pure Type-3, Pure Type-3 Consistent, or Pure Type-3 Blind clones. Recall that fragments in a Pure Type-3 clone class

Table XVII Reduction ratio for the clone classes, fragments, and LOC for each test suite.

Test Suite name	Clone Class	Fragment	Total LOC	Initial Test Suite size	Valid Clone Class (percentage)	Valid Clone Fragment (percentage)	Test Suite size After refactoring (reduction percentage)
TS-1	4	34	980	1978	2 (50.0%)	29 (85.3%)	1021 (48.38%)
TS-2	126	480	20665	87818	48 (38.1%)	385 (80.2%)	76890 (12.44%)
TS-3	13	39	3411	9749	8 (61.5%)	35 (89.7%)	6854 (29.70%)
TS-4	6	58	3883	8672	5 (83.3%)	56 (96.6%)	7289 (15.95%)
TS-5	2	23	312	581	2 (100.0%)	23 (100.0%)	296 (49.11%)
TS-6	4	11	338	2063	2 (50.0%)	6 (54.5%)	1939 (6.01%)
TS-7	3	17	385	630	3 (100.0%)	17 (100.0%)	290 (53.97%)
TS-8	2	13	2867	6619	1 (50.0%)	9 (69.2%)	6295 (4.89%)
TS-9	19	108	4702	9572	9 (47.4%)	83 (76.9%)	6089 (36.39%)
TS-10	1	3	32	1138	0 (0.0%)	0 (0.0%)	1138 (0.00%)
TS-11	1	2	26	638	0 (0.0%)	0 (0.0%)	638 (0.00%)
TS-12	15	90	3826	11536	4 (26.7%)	20 (22.2%)	9536 (17.34%)
TS-13	12	55	3764	7566	5 (41.7%)	39 (70.9%)	5429 (28.24%)
TS-14	2	6	646	3671	1 (50.0%)	4 (66.7%)	3348 (8.80%)
TS-15	10	42	8225	15306	5 (50.0%)	28 (66.7%)	13612 (11.07%)
TS-16	8	33	2661	11265	2 (25.0%)	5 (15.2%)	11077 (1.67%)
TS-17	2	15	1078	8106	0 (0.0%)	0 (0.0%)	8106 (0.00%)
TS-18	6	19	131	964	0 (0.0%)	0 (0.0%)	964 (0.00%)
TS-19	2	5	39	527	0 (0.0%)	0 (0.0%)	527 (0.00%)
TS-20	26	100	4470	22463	12 (46.2%)	54 (54.0%)	21165 (5.78%)
TS-21	0	0	0	48	0 (0.0%)	0 (0.0%)	48 (0.00%)
TS-22	7	66	4874	12303	3 (42.9%)	55 (83.3%)	11123 (9.59%)
TS-23	4	21	1279	2431	2 (50.0%)	17 (81.0%)	1535 (36.86%)
TS-24	1	2	98	478	0 (0.0%)	0 (0.0%)	478 (0.00%)
TS-25	1	3	22	2321	0 (0.0%)	0 (0.0%)	2321 (0.00%)
TS-26	21	94	5085	17030	9 (42.9%)	63 (67.0%)	15066 (11.53%)
TS-27	5	10	226	6667	0 (0.0%)	0 (0.0%)	6667 (0.00%)
TS-28	17	100	5417	20608	10 (58.8%)	85 (85.0%)	16019 (22.27%)
TS-29	9	29	1195	5141	6 (66.7%)	12 (41.4%)	4955 (3.62%)
TS-30	1	2	57	1100	0 (0.0%)	0 (0.0%)	1100 (0.00%)
TS-31	1	1	13	316	0 (0.0%)	0 (0.0%)	316 (0.00%)
TS-32	1	7	308	944	0 (0.0%)	0 (0.0%)	944 (0.00%)
TS-33	4	33	987	2246	2 (50.0%)	27 (81.8%)	1359 (39.49%)
TS-34	23	58	1934	2880	12 (52.2%)	32 (55.2%)	2413 (16.22%)
TS-35	1	2	118	317	0 (0.0%)	0 (0.0%)	317 (0.00%)
TS-36	0	0	0	476	0 (0.0%)	0 (0.0%)	476 (0.00%)
Total	360	1581	84054	286168	153 (42.5%)	1084 (68.6%)	247640 (13.46%)

in general tend to have large differences because of the renaming and a high threshold. In fact, we found that the most challenging clones from the refactorability point of view are all Pure Type-3 clones as we will discuss in section 6.2.4. Despite Type 3 clones having similar fragments to a certain extent (because of the threshold and renaming) and therefore structural similarities, to some extent, they are not semantically similar, and therefore, it did not make sense to refactor.

Also, we identify two test suites, TS-5 and TS-7, with a 100% ratio for clone classes and clone fragments. These test suites have a small number of Pure Type-3 or Pure Type-2 Consistent clone classes. The clone fragments exhibit similar logic but different input data, which make them good candidates for refactoring.

Most of the test suites have clone classes and fragment ratios over 50%. The overall ratio of valid clone class is equal to 42.5%, and the total ratio of valid fragment is equal to 68.6%. We conclude that we have a high percentage of refactorable clone fragments. Considering that the 14 test suites with 0% valid clone classes and fragments and the two test suites with 100% valid clone classes and fragments are outliers, we also calculated the overall ratio of valid clone classes and fragments after removing outliers; The overall ratio of valid clone classes and fragments become 44.6% and 71%, respectively.

The test suite LOC reduction is varied because the LOC reduction depends on the test suite size. We compare the LOC clone reduction percentage and the test suites' size, as shown in Illustration 18. The test suites are ranked on the x-axis in descending order of their size. We observe that the LOC reduction percentage fluctuates; for instance, TS-1 is a small test suite with a high LOC reduction percentage (48.8%) and TS-20 is the second-largest test suite with a low LOC reduction percentage (5.8%). Also, the largest test suite TS-2 has a slightly small LOC reduction percentage (12.4%). We examined these test suites that have a low LOC reduction percentage and large size. These test suites have a considerably large number of all Pure Type-3 clones. Although Pure Type 3 clone fragments tend to have similar fragments, to a certain extent (because of the threshold and renaming), and these fragments are structurally similar, to some extent, these fragments are not semantically similar, and therefore, it is very challenging to refactor them. From Illustration 18, there is no obvious relation between LOC reduction percentage and test suite size.

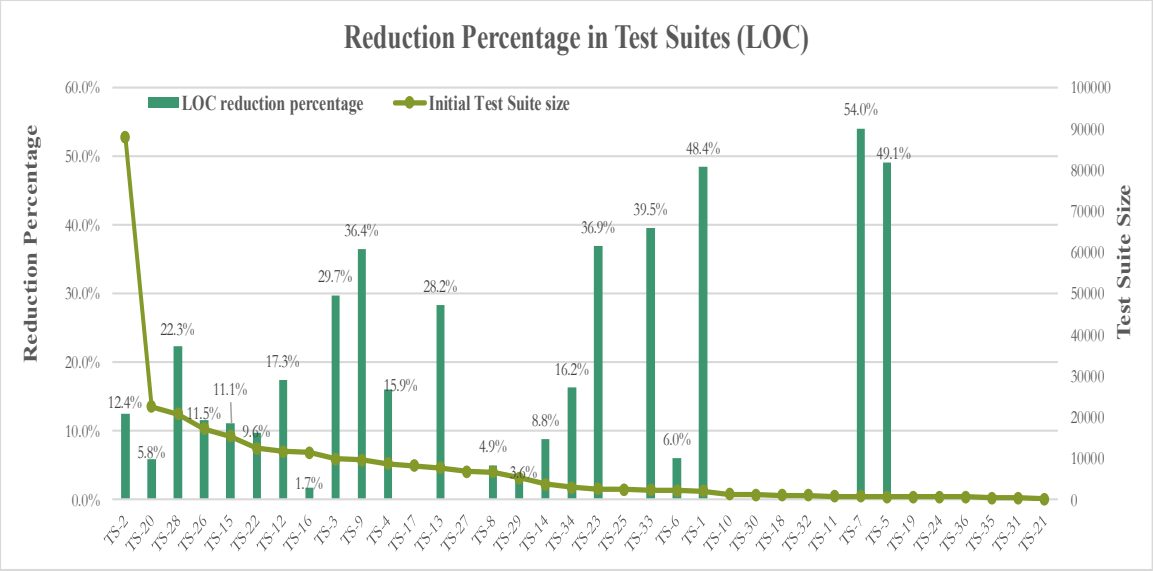


Illustration 18 Reduction ratio in test suites (LOC)

We used the Spearman correlation coefficients because the distributions are not normal, as confirmed by a Shapiro-Wilk normality test, to measure how strong of relationships exist between the LOC reduction and size of test suite. We find a statistical correlation coefficient of 0.92. It is higher than 0.7, which means that there is a very strong correlation between the total LOC reduction and the test suite size. When the test suite size increases, the overall LOC reduction increases as well in general.

6.2.4 Clone Type

In this section, we firstly investigate if the clone type has any effect on refactoring decisions. Secondly, we examine the relationship between the clone type and the clone class's properties. Table XVIII shows the number of refactorable clone fragments per the clone type. The percentage of refactorable clone fragments is what we refer to as the refactorability in the rest of this section: the higher the percentage, the higher the refactorability. It is evident that Type 1 clone fragments have a very high refactorability, with the highest percentage (100%) followed by Pure Type-2 Consistent (81.69%). All Pure Type-3 clones are considered the most challenging clones from the refactorability point of view.

NiCad detects clones by identifying similarities in pretty-printed lines based on a threshold value and renaming, and thus, NiCad detects a large number of Pure Type-3 clones fragments. By analyzing the results for Pure Type-3 Consistent and Pure Type-3 Blind clone fragments, we discovered that Pure Type 3 Consistent and Pure Type-3 Blind clone fragments have similar fragments to a specific extent (because of the threshold and renaming) and these fragments are structurally similar, to some extent; however, these fragments are not semantically similar, and thus, these fragments are very challenging to

refactor. Our results show that Pure Type-3 Blind has the lowest percentage of refactorability.

We perform another analysis to show the relationship between the clone type and the refactorability decision: Table XIX. For each clone class's property, the table shows the number of clone fragments for each clone type. For instance, for identical test cases (Property-1), there are 58 Type 1 clone fragments, 24 Pure Type-2 Consistent clone fragments, 38 Pure Type-3 clone fragments, two Pure Type-3 Consistent clone fragments, and six Pure Type-3 Blind clone fragments. The first observation we can make is that all Type 1 clone fragments have identical test cases (Property-1), which is not surprising. A second observation is that we found that there is a number of Pure Type-2 Consistent and Pure Type-3 clone classes that can be refactored as Type-1 clones (Property-1), that is they were refactored because their fragments are recognized as identical test cases (Property-1) which is essentially the property of Type-1 clone classes. We inspected these clone classes to find out some statements in these clone fragments are in different order, that the order of these identical individual statements does not matter for the correct execution of the fragments, and that therefore, by reordering these statements we obtain identical fragments

Table XVIII Number of refactorable fragments for each clone type

Clone Type	Initial Number of Fragments	Number of refactorable fragment (Percentage)
Type 1	58	58 (100.0%)
Pure Type-2 Blind	6	4 (66.67%)
Pure Type-2 Consistent	213	174 (81.69%)
Pure Type-3	820	591 (72.07%)
Pure Type-3 Consistent	223	144 (64.57%)
Pure Type-3 Blind	261	113 (43.30%)
Total	1581	1084 (68.56%)

which are then refactored similarly to Type-1 clone classes' fragments because we recognize Property-1. Another interesting observation is that all clone types except Type 1 have clone fragments with similar logic but a variation in input data (Property-3); the table illustrates that the most common properties for refactoring decision are variation in assigned input data (Property-3) and variation in assertions and assigned input data (Property-4). For instance, we found 94 clone fragments with "Variation in assigned input data" (Property-3) under Pure Type-2 Consistent and 176 clone fragments with the same reason under Pure Type-3.

Table XIX Clone types from a refactorability decision point view

Property ID	Clone class properties	Type 1	Pure Type-2 Blind	Pure Type-2 Consistent	Pure Type-3	Pure Type-3 Consistent	Pure Type-3 Blind
Property-1	Identical test case	58		24	38	2	6
Property-2	Variable rename			7	6	4	
Property-3	Variation in assigned input data only		4	94	176	88	39
Property-4	Variation in assertions and assigned input data			6	179	40	39
Property-5	Different function call			20			
Property-6	Variation in for-loop condition and assigned input data			12	3	2	
Property-7	Similar setup			7	8		
Property-8	Variation in switch statement cases			4	1		
Property-9	Similar logic but different print messages				10	3	
Property-10	Variation in assertions only				7	2	2
Property-11	Variation in if-statement condition and assigned input data				2		1
Property-12	Similar teardown				118	1	16
Property-13	Similar setup and teardown				41		10
Property-14	Variation in if-statement condition				2		
Property-15	Similar test cases with missing cleanup function call					2	
		58	4	174	591	144	113

The number of refactorable Pure Type-3 Consistent and Pure Type-3 Blind classes is much less than the number of refactorable Pure Type-3 classes. We believe this is because the renaming does not allow to recognize more similar fragments, especially when the threshold is smaller than 100%, potentially leading to more opportunities for refactoring, but in fact the fragment are not refactorable because syntactic similarity (with renaming and threshold) does not necessarily translate into semantic similarity (which is a requirement in order to refactor).

In conclusion, the common clone class property for refactoring Type 1 clones is identical test cases (Property-1); some Pure Type-2 Consistent and Pure Type-3 clones can be refactored just like Type-1 clones; Pure Type-3 clones are challenging to refactor because syntactic similarity does not necessarily means semantic similarity.

6.2.5 Size of Clone fragment

In this section, we examine whether the size of clone fragments affect the refactorability. Contrary to what we have done earlier in this manuscript, where the size of a class is its number of fragments, in this section, the size is measured in LOC. Since after merging, a clone class has fragments with varying numbers of LOC, we define the size of a clone class as the average value of the number of LOC of its fragments. In this section, we want to investigate if the size of a clone class (after merging) has an impact on the decision to refactor it. Illustration 19 shows the number of clone class occurrences (y-axis) for each clone class size (x-axis). For instance, we have refactored 72 clone classes of size 23 (LOC), i.e., 72 classes with an average fragment size of 23 (LOC). We find that we refactor more clone classes of size less than 37 LOC (the median value for the sample). By observing the bar chart, it seems that when the size of the clone class increases, its

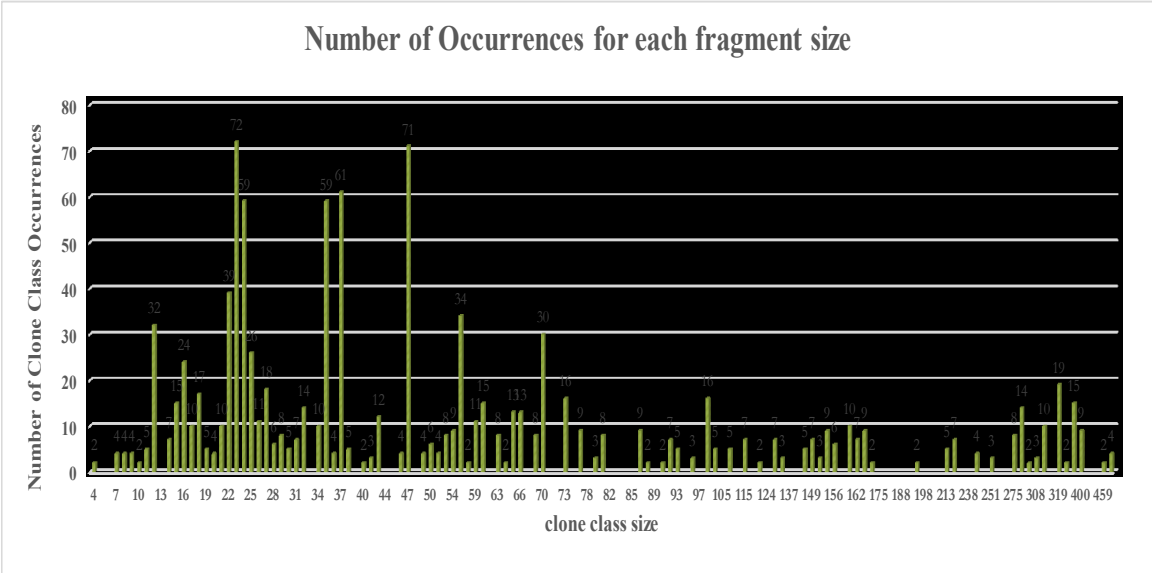


Illustration 19 Number of occurrences of clone classes with different sizes

refactorability decreases. We calculated the standard deviation to understand how spread out the class size is. We found that the standard deviation is equal to 91.22; this indicates that most of the clone class's size is not close to the mean size of 76.36.

We used the Spearman correlation coefficients because the distribution is not normal, as confirmed by a Shapiro-Wilk normality test, to measure how strong a relationship exists between the clone class size and the number of clone class occurrences. We find a statistical correlation coefficient of -0.31. We observe a weak negative correlation, which means that the clone class size and the number of clone class occurrences are relatively independent of one another. In other words, the size of a clone class does not indicate the refactorability of its fragments. We conclude that the clone class size does not tell us precisely if the clone class is more likely to be refactorable or not.

Also, we perform another analysis to examine whether the clone class's size affects the refactoring decision. Illustration 20 shows the violin plots for the size of the refactorable

versus the non-refactorable clone fragments. A violin plot is a method of plotting numeric data and it is often used to compare the distribution of data where the wider sections of the plot represents a higher probability of observations and the thinner sections correspond to a lower probability. Also, a violin plot shows the mean, median and range. As it is obvious from the plots, the average size of refactorable clone classes is slightly smaller than the average size of non-refactorable clone class; the averages are 76.36 and 77.53, respectively. The average is not an ideal presentation of the data, because the outliers influence it; and therefore, we evaluate the median values for refactorable and non-refactorable clone classes, we found that the median is equal to 37 and 52 for refactorable and non-refactorable clone fragments respectively. The median value shows that the fragment's size of refactorable clone classes is smaller than the fragment's size of the non-refactorable clone classes. We applied the one-tailed variant of the Mann-Whitney U test on the sizes of the refactorable and non-refactorable clone classes with the following null hypothesis: "the refactorable clone classes' size is greater than the non-refactorable clone classes' size." The null hypothesis was accepted with significance at a 95 percent confidence level

based on $p - value > 0.182$; and therefore, we can conclude that the refactorable clone classes' size is greater than the non-refactorable clone classes' size.

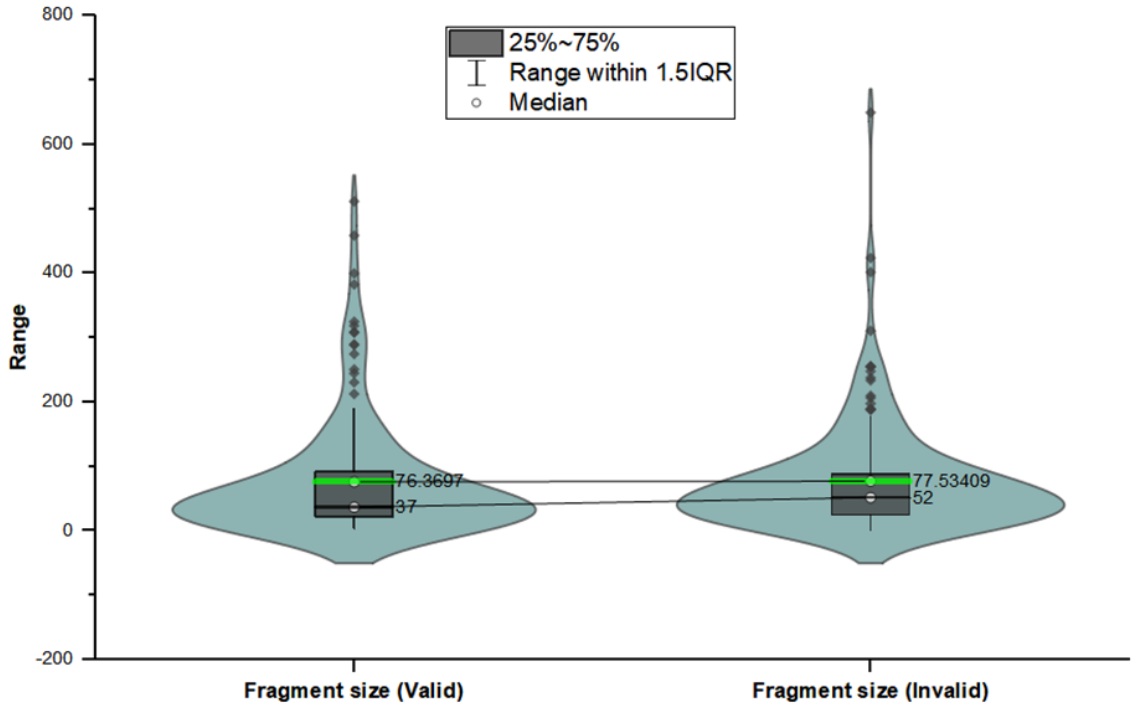


Illustration 20 Fragments' size of refactorable and non-refactorable clone classes.

6.2.6 Size of Clone class

In this section, we inspect if the size of a clone class will increase or decrease its refactorability. Each clone class has a different number of clone fragments, so in this section we consider the clone class's size as the number of its fragments: Illustration 21 shows the clone class size distribution. Since we are interested in refactorability, we only consider clone classes we refactored. The x-axis indicates clone class sizes and the y-axis indicates the number of occurrences of each clone class size. For instance, we refactored 39 clone classes of size two. When comparing clone class size and clone class size occurrences, we find that we refactored more clone classes of size less than four (the

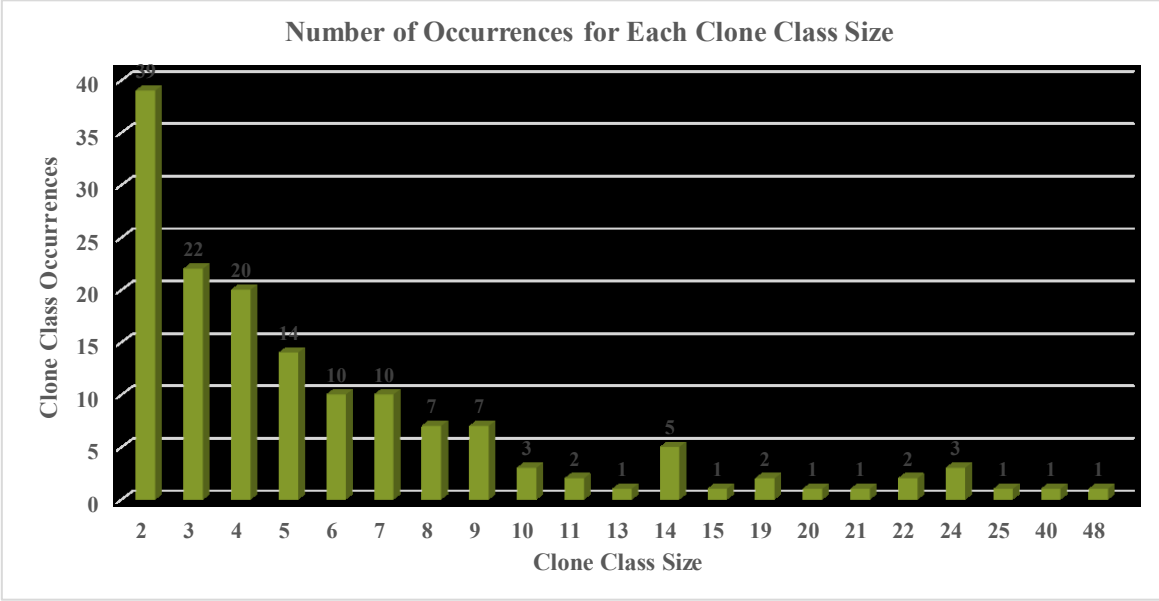


Illustration 21 Number of clone class occurrences for different clone class sizes

median for the sample). The bar chart strongly suggests that when the size of the clone class increases, the refactorability of clone classes decreases. We calculated the standard deviation to understand how to spread out the clone class size is. We found that the standard deviation is equal to six; this indicates that most of the clone classes' size is close to the mean which is equal to 5.43. We used the Spearman correlation coefficients because the distributions are not normal, as confirmed by a Shapiro-Wilk normality test, to measure how strong relationships exist between the size of a clone class and its refactorability. We find a statistical correlation coefficient of -0.61 which means that there is a negative strong correlation between the size of a clone class and its refactorability. When the size of a clone class increases, the clone class refactorability decreases as well in general.

We performed another analysis to examine whether the clone class's size affects the refactoring decision. Illustration 23 shows the violin plots for the size of the refactorable versus the non-refactorable clone classes. As it is evident from the plots, the size of clone

classes is small, with an average size of 5.4 and 3.1 for refactorable and non-refactorable clone classes respectively. The median value is equal to 3.5 and two for refactorable and non-refactorable clone classes respectively, and thus we conclude that the refactorable and the non-refactorable clone classes contain a small number of fragments. Also, the refactorable clone classes tend to have a larger average size compared to the non-refactorable clone classes. We applied a one-tailed variant of the Mann-Whitney U test on the sizes of the refactorable and non-refactorable clone classes with the following null hypothesis: "the size of refactorable clone classes is smaller than the size of non-refactorable clone classes." The null hypothesis was rejected with significance at a 95 percent confidence level based on $p - value < 2.96\,X\,10^{-7}$; and therefore, we can conclude that the size of refactorable clone classes is greater than the size of non-refactorable clone classes.

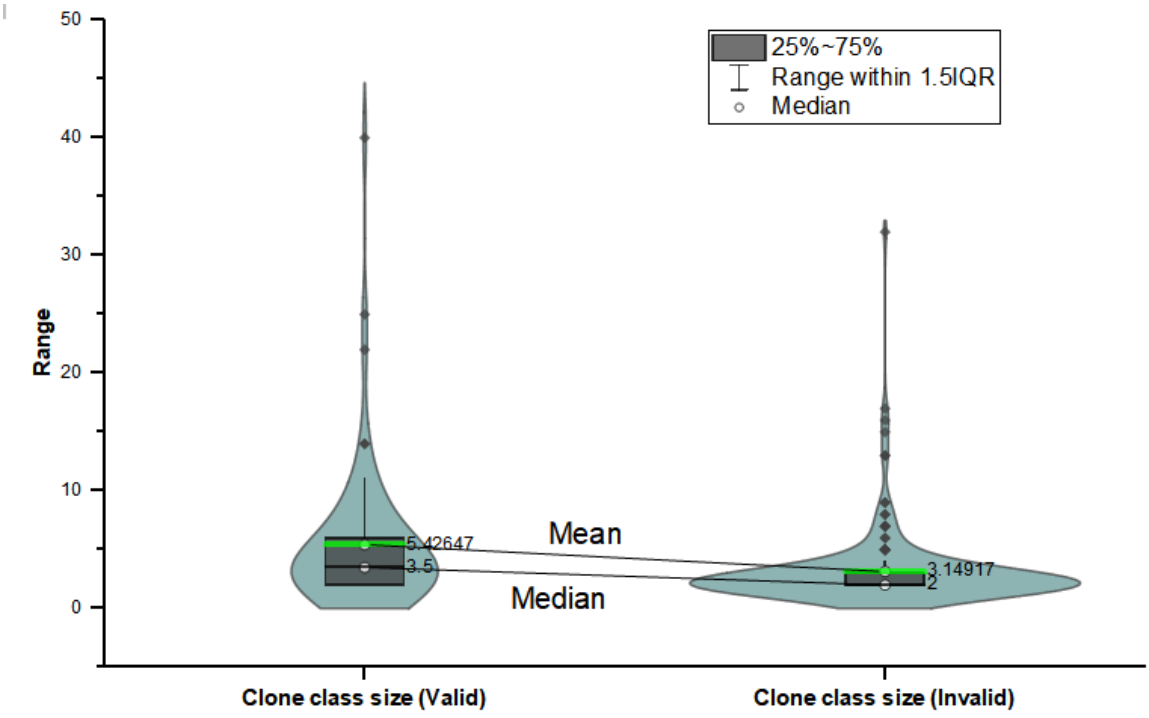


Illustration 22 Size of refactorable and non-refactorable clone classes.

6.3 Threats to Validity

Internal validity threats to this study are related to the configuration of the NiCad clone detection tool. To ease this concern, we are aware of the effects of configurations on the performance of the tool, so we used multiple pre-defined settings for NiCad. At the same time, some configurations meet the definitions of Type-1, Type-2 Consistent, Type-2 Blind, Type-3, Type-3 Consistent and Type-3 Blind clones that are used in many different publications. In the end, we used the default configuration of NiCad since they have been shown to lead to both high precision and high recall at finding near-miss clones [68]. Another threat to internal validity is the possibility of having an incomplete list of clone class properties to determine whether a clone class's fragments can be refactored. An incomplete list of clone class properties would result in some clone classes being erroneously determined as not refactorable, while they can be refactored in practice. To lessen this risk, we tested the clone class properties on 316 clone classes found in 36 test suites. All clone classes' fragments were successfully refactored without causing any compile errors or test failures and the resulting refactored tests achieved the same structural coverage as the initial test suite.

Another threat to internal validity is the possibility of missing some refactoring opportunities that would result in some clone classes being erroneously determined as not refactorable, while they could in fact be refactored. To lessen this risk, we have taken two steps: first, we manually assessed the refactorability of each clone class and provided a refactoring decision; second, we requested senior engineers with vast experience with the test code to analyze each clone class and validate our refactoring decisions. We have

successfully validated all clone classes using the above two steps to manually validate a clone class as a refactoring opportunity.

Conclusion validity corresponds to the inability to draw valid conclusions. The reason for this threat is related to insufficient data to draw logical conclusions. The clone classes' properties, the used patterns, and the association between the clone class's properties and pattern are not typical of all systems and test suites, and the result cannot be generalized. Also, the percentage of reduction of clone classes, fragments, and LOC for the test suites (from C industry test code) is not representative of all test suites, and the percentage of reduction results is not generalizable. To overcome this threat, we used 36 test suites in C with varied sizes and complexities. We nevertheless do not make claims that the observations we make are generalizable to any software; more experiments of the kind we conducted are necessary.

Construct validity is concerned with the relationship between theory and observation. We have refactored the clone classes from our merge clone classes procedure results; we have manually assessed the refactorability for each clone class, then manually refactored the valid clone classes, and executed the resulting test cases to make sure that the test execution has the same verdict as before (e.g. a test that used to pass/ fail still passes/ fails), and the refactored test suites achieve the same structural coverage of the application code being tested. To minimize this threat, we manually verified the outcome results of refactoring the clone classes to make sure that the test cases' behaviour is preserved and work as intended.

There can be threats to external validity, as well. These threats are related to the external aspects of the experiments that can limit the generalization of the results. This threat is

primarily associated with the test suites selection for performing the analysis and the test code clones derived from them. Our study is conducted solely in an industry context. We do not claim the results are generalizable. We note that we also did not cherry-pick the industry projects; they were provided by our industry partner and are complete suites for the application logic they verify.

6.4 Conclusion

We concentrate our study on managing test clones by refactoring them to help the developers to assess the refactorability of clone class and then refactor the clone class. We have collaborated with the engineers from Ericsson to manually evaluate and assess the refactorability and the results of refactoring the 360 clone classes that we obtained after using our merge clone classes procedure.

We defined 15 clone classes' properties that affect the refactoring decision and then we provide association between the refactoring decision and the clone class's properties. We applied these refactoring decisions on the clone classes from our industry case study, and we found that the most dominant clone classes' properties are "Variation in assigned input data" and "Variation in assertions and assigned input data." These two properties show that there are several clone fragments with similar logic but different input data or assertions. And we found that the most frequently used refactoring pattern is the Parameterized Test pattern. Also, this result confirms that test clones have several test cases with similar logic but different input data and assertions. Therefore, our finding shows that we need to refactor the test clones by applying different patterns to reduce the test code maintainability and allow the testers to add more test cases as required while reusing the test codes efficiently.

We investigated the reduction ratio for clone classes, fragments and LOC for each test suites to find out how much of the test suite size is reduced by refactoring test clones. We discover that most test suites have clone classes and fragment ratio of over 50% while the overall ratio of clone class is equal to 42.5%, and the overall ratio of fragment is equal to 68.6%. And therefore, we conclude that we have a high percentage of the refactorable clone fragments. Regarding LOC reduction, we discover that the LOC reduction percentage fluctuates, for instance, a large test suite may have a low LOC reduction while a small test suite may have a high LOC reduction percentage. To understand the fluctuation of LOC reduction percentage, we examine these test suites, and we find they have a considerably large number of all Pure Type-3 clones. These are very challenging clone fragments to refactor. We find that there is a very strong correlation between the total LOC reduction and the test suite size, and thus, we can conclude that when the test suite size increases, the overall LOC reduction increase as well for all test suites.

We found that Type-1 clones tend to be more refactorable than Pure Type-2 and Pure Type-3 clones. Pure Type-2 Consistent classes have the second highest refactorability, followed by Pure Type-3 clones. Pure Type-3 Blind clones are the most challenging clones to refactor. We discovered that some Pure Type-2 Consistent and Pure Type-3 clones can be refactored just like Type-1 clones, that is with similar refactoring patterns. For all clone types except Type 1, we refactor many clone classes with the property of similar logic but a variation in input data and assertions.

Our study shows that when the average size of the clone class (in LOC) increases, the refactorability of the clone class decreases. This was confirmed by the use of a statistical test: the correlation is statistically significant. We also compare the size of the refactorable

and non-refactorable clone classes, and we found that the fragment's size of non-refactorable clone classes is greater than the fragment size of the refactorable clone classes. We examined whether the number of fragments in a clone class will increase or decrease the refactorability of the clone class. Our investigation and statistical test show that when the size of the clone class is less than the clone classes' average, the refactorability of the clone classes increases. Also, we studied whether the clone class's size affects the refactoring decision. Our study shows that the size of refactorable clone classes is greater than the size of non-refactorable clone classes.

In conclusion, we believe engineers need to have precise information about clones, more precise than what a tool like NiCad reports including removing overlapping information and merging clone classes with similar fragments; as a result, engineers will have sufficient information to assess the refactorability of clone class and then refactor the clone class.

Chapter 7: Conclusions and Contributions

Many companies experience increasing costs of software verification, including Ericsson. The testers find it easy to add new test cases for every change or fix, and this causes duplications in test code which is typically referred to as (test) code clones. In this thesis, we endeavour to understand the presence of test code clones and manage the clones using clone refactoring, and therefore, we contribute to the clone detection, software testing, and clone refactoring fields. In the following paragraphs, we discuss our contributions.

Contributions 1: Our investigation shows that there is some overlap in clone detection information when a clone detection tool reports on different types of clones. The standard definitions of clone types allow overlapping information. We therefore study the level of such overlapped information and to what extent having such overlapped information or removing such overlapped information impacts conclusions one can empirically derive when conducting experiments on clone detection on source code. Since standard definitions of clone types allow clone detection results to overlap, we refine them to account for overlapping clone detection information. Essentially, we define, for each clone type, a pure clone type counterpart that avoids clone detection information of less permissive clone types. We then defined and implemented a generic Clone Reduction Procedure to remove overlapped clone information from clone detection tool results and illustrate how it can be applied to results produced by NiCad, a well-known clone detection tool that has shown to detect clone with high precision and recall. We performed an experimental study to confirm the presence of overlapping data. We selected, replicated and extended an experiment others have conducted with NiCad [1], using a total of nine

large Java subject programs and nine large C subject programs. Our experiment shows that clone detection results such as those produced by NiCad contain overlapped information, sometimes in tremendous proportions: on Java subjects, for instance, clone fragments obtained with a difference threshold of 10% are at least 80% overlapping with fragments obtained with a threshold of 0%.

The researchers and practitioners should be aware of overlapping clone detection information when empirically deriving conclusions on clone detection, and that tools should strive to remove overlapping clone detection information in their result reporting, and as such, provide accurate results to the user.

Contributions 2: We performed a case study with industry test code that aims at better understanding test clones. By measuring the distribution of software clones in industry test code, we have analyzed and observed clone fragments detected by NiCad in two C projects and one Java project, comprised of a total of 1,077,552 lines of code (LOC), obtained from our industry partner. We have detected a high percentage of clones in test code: 49% (LOC) of the entire C test code are clones, which is also equivalent to 36% of all the test cases; 73% LOC of the entire Java test code are clones, amounting to 94% of all the test cases.

Contributions 3: Our investigation shows that a clone fragment can belong to different clone classes under different clone types. The clone detection tools report similar test codes under different clone classes because these clone classes are classified according to the level of similarity between the fragments. For instance, a clone fragment that belongs to a Type-1 clone class because it is 100% similar to other clone class's fragments can also belong to a Type-3 clone class when it is at least 70% similar to yet other fragments. Thus, we defined and implemented a general Merge Clone Classes Procedure to merge clone

classes that share a fragment from the clone detection tool results and illustrate how it can be applied to results produced by NiCad.

We performed an experimental study using the 36 test suites written in C from our industry partner. Results confirm that a clone fragment can belong to different clone classes of different clone types. Also, the results show that when using the merging procedure the total number of clone classes is reduced by approximately 63%, the number of fragments is reduced by 54.4% on average, and the LOC reduction percentage is equal to 40.1%.

Contributions 4: Clone refactoring is a possible approach for clone management, and thus, we focus our study on refactoring test clones to maintain test code and reduce the size of test suites. We have collaborated with engineers from Ericsson to manually evaluate 360 clone classes that we obtained after using our merge clone classes procedure. We assessed the refactorability of these clone classes and then refactored them while preserving the system behaviour (i.e. no compile errors, same test verdicts, same achieved structural coverage of application code).

We identified 15 clone classes' properties that affect the refactoring decision and then we provided association between the refactoring decision and the clone class's properties. We applied these refactoring decisions on 360 clone classes, and we found 153 refactorable clone classes (42.5% of the classes are refactorable, corresponding to 68.6% of the fragments). We analyzed the 153 clone classes and their fragment, and we observed that:

- Most clones are due to variations in test input data or oracle assertions, and therefore that test logic in test clones is mostly identical.
- Type-1 clones tend to be more refactorable than Pure Type-2 and Pure Type-3 clones, Pure Type-2 Consistent clones are the second highest refactorable

clones, followed by Pure Type-3 clones. Pure Type-3 Blind clones are the most challenging clones for refactoring; Because of larger variations due to renaming and a higher threshold of differences, syntactic similarity in Pure Type-3 clone fragments does not necessarily translate into semantic similarity, which prevents refactoring.

- Some Pure Type-2 Consistent and Pure Type-3 clones can be refactored just like Type-1 clones, that is with a similar effort and similar refactoring patterns.
- When the test suite size increases, the overall LOC reduction due to clone refactoring increases as well for all test suites. When the average size (LOC) of the fragments in a clone class increases, the refactorability of the clone class decreases.
- The size (LOC) of fragments in non-refactorable clone classes is greater than the size of fragments in refactorable clone classes. The number of clone fragments in refactorable clone classes is greater than the number of fragments in non-refactorable clone classes.

Future work

Our work can be extended in several ways.

First, our study shows that there is overlapping information in clone detection results for different clone types. In our study, we only used NiCad to confirm the presence of overlapping information. As a direction for future work, one can apply the Remove Overlapping Clone Information procedure with other clone detection tools. Another future work is to extend our study to detect clones in industry test code and open-source test code by replicating our study with different projects and systems. Similarly, our merging clone

class procedure should be experimented on results provided by clone detection tools other than NiCad.

Clone refactoring on reduced clone detection information, i.e. after removing overlapping information and then merging clone classes, should be studied on other test suites, open source and from industry.

Since test code and application code are different (e.g., test code typically has a specific structure including a setup, the test, an oracle, and a test teardown), it may be worth replicating our study (e.g. overlapping, refactoring) on application code.

Last, it would be useful to implement an eclipse plugin to help developers to automatically assess the refactorability of clone classes using our association guidelines, and automatically provide suggestions to refactor specific clone classes. In terms of automation, it would be useful to integrate clone detection in automated continuous integration processes so test engineers do not contribute test clones.

Bibliography

[1] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software Maintenance and Evolution,* vol. 22, pp. 165-189, 2010.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] "Eclipse." http://www.eclipse.org (accessed 2007).

[4] C. K. Roy, "Detection and analysis of near-miss software clones," Doctor of Philosophy Thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, August 2009.

[5] "Java 2 SDK." http://java.sun.com (accessed 2007).

[6] "PostgreSQL." PostgreSQL (accessed 2007).

[7] "The Stuttgart Neuronal Network Simulator." http://www-ra.informatik.uni-tuebingen.de (accessed 2007).

[8] "Cook." http://miller.emu.id.au/pmiller/software/cook/ (accessed 2007.

[9] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, and G. Canfora, "Recommending refactorings based on team co-maintenance patterns," in Proceedings of the ACM/IEEE international conference on Automated software engineering, 2014.

[10] M. R. Islam and M. F. Zibran, "A Comparative Study on Vulnerabilities in Categories of Clones and Non-cloned Code," in Proceedings of the *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016, vol. 3, pp. 8-14.

[11] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider, "Comparing Software Bugs in Clone and Non-clone Code: An Empirical Study," *International Journal of Software Engineering and Knowledge Engineering,* vol. 27, pp. 1507-1527, 2017.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering,* vol. 33, no. 9, pp. 577-591, 2007.

[13] B. Beizer, *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold, 1990.

[14] M. Rieger, "Effective Clone Detection Without Language Barriers," Doctor of Philosophy, University of Bern, Switzerland, June 2005.

[15] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in Proceedings of the *Working Conference on Reverse Engineering*, 2006, pp. 19-28.

[16] B. S. Baker, "A Program for Identifying Duplicated Code," In Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:49–57, March 1992.

[17] 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 *IEEE International Conference on Program Comprehension*, 2008, pp. 172-181.

[18] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* vol. 28, no. 7, pp. 654-670, 2002.

[19] I. D. 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.

[20] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in Proceedings of the *Source Code Analysis and Manipulation* 2004, pp. 128-135.

[21] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience,* vol. 21, no. 7, pp. 739-755, 1991.

[22] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in Proceedings of the International Symposium on Static Analysis, Berlin, Heidelberg, 2001.

[23] J. Krinke, "Identifying similar code with program dependence graphs," in Proceedings of the *Conference on Reverse Engineering*, 2001, pp. 301-309.

[24] N. Davey, P. Barson, S. D. H. Field, R. Frank, and S. Tansley, "The Development of a Software Clone Detector," *International Journal of Applied Software Technology,* vol. 1, 1995.

[25] J. Mayrand, C. Leblanc and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in Proceedings of the *International Conference on Software Maintenance*, 1996, pp. 244-253.

[26] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in Proceedings of the *Annual International Conference on Automated Software Engineering*, 2001, pp. 107-114.

[27] J. R. Cordy, T. Dean, and N. Synytskyy, "Practical language-independent detection of near-miss clones," in Proceedings of the conference of the Centre for Advanced Studies on Collaborative research, Markham, Ontario, Canada, 2004.

[28] L. Moonen, "Generating Robust Parsers using Island Grammars," in Proceedings of the *Conference on Reverse Engineering*, Stuttgart, Germany, 2001, pp. 13–22.

[29] S. Nasehi, G. Sotudeh, and M. Gomrokchi, "Source code enhancement using reduction of duplicated code," in Proceedings of the *IASTED International Multi-Conference*, Innsbruck, Austria, 2007, pp. 192-197.

[30] S. R. Kosaraju, "Faster algorithms for the construction of parameterized suffix trees," in Proceedings of the *IEEE Annual Foundations of Computer Science*, 1995, pp. 631-638.

[31] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM,* pp. 262–272, 1976.

[32] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in Proceedings of the *International Conference on Software Maintenance*, Oxford, England,1999, pp. 109–118.

[33] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," Journal of *Automated Software Engineering,* vol. 3, no. 1, pp. 77-108, 1996.

[34] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in Proceedings of the Conference on Reverse Engineering, 2006.

[35] T. Kamiya. "The Official CCFinderX Website." http://www.ccfinder.net/ccfinderx.html (accessed November 2016).

[36] J. H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," in Proceedings of the conference of the Centre for Advanced Studies on Collaborative research: software engineering, Toronto, Ontario, Canada, 1993.

[37] A. Leitão, *Detection of redundant code using R2D2*. 2003, pp. 183-192.

[38] G. Meszaros, "XUnit Test Patterns: Refactoring Test Code," *Addison-Wesley,* 2007.

[39] G. Mostaeen, J. Svajlenko, B. Roy, C. K. Roy, and K. A. Schneider, "On the Use of Machine Learning Techniques Towards the Design of Cloud Based Automatic Code Clone Validation Tools," in *Proceedings of the IEEE 18th International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 155-164.

[40] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Journal of Empirical Software Engineering,* vol. 20, no. 4, pp. 1052-1094, 2014.

[41] F. Viertel, W. Brunotte, D. Strüber, and K. A. Schneider, "Detecting Security Vulnerabilities using Clone Detection and Community Knowledge," in Proceedings of the International Conference on Software Engineering & Knowledge Engineering, 2019.

[42] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "JDeodorant: Clone Refactoring," in Proceedings of the *IEEE/ACM International Conference on Software Engineering Companion*, 2016, pp. 613-616.

[43] B. S. Baker, "On finding duplication and near-duplication in large software systems," in Proceedings of the *Conference on Reverse Engineering*, 1995, pp. 86-95.

[44] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in Proceedings of the *International Conference on Software Maintenance*, 1997, pp. 314-321.

[45] Y. Dang, S. Ge, R. Huang, and D. Zhang, "Code clone detection experience at microsoft," in Proceedings of the *International Workshop on Software Clones*, 2011, pp. 63–64.

[46] E. Tüzün and E. Er, "A case study on applying clone technology to an industrial application framework," in Proceedings of the *International Workshop on Software Clones*, 2012, pp. 57-61.

[47] E. Merlo, T. Lavoie, P. Potvin, and P. Busnel, "Large scale multi-language clone analysis in a telecommunication industrial setting," in Proceedings of the *International Workshop on Software Clones)*, 2013, pp. 69-75.

[48] C. K. Roy and J. R. Cordy, "Are Scripting Languages Really Different?," in Proceedings of the *International Workshop on Software Clones*, New York, NY, USA, 2010, pp. 17-24.

[49] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the Refactorability of Software Clones," *IEEE Transactions on Software Engineering,* vol. 41, pp. 1055-1090, 2015.

[50] R. Koschke and S. Bazrafshan, "Software-Clone Rates in Open-Source Programs Written in C or C++," in Proceedings of the *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 1-7.

[51] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," *Journal of Systems and Software,* vol. 159, 2020.

[52] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. A. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Proceedings of the Working Conference on Reverse Engineering* , 1999, pp. 326-336.

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[54] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. A. Kontogiannis, "Measuring clone based reengineering opportunities," in Proceedings of the *International Software Metrics Symposium*, 1999, pp. 292-303.

[55] G. G. Koni-N'Sapu, "A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems," Diploma Thesis, Institute of Computer Science, University of Bern, 2001.

[56] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring Support Based on Code Clone Analysis," in Proceedings of the *International Conference on Product Focused Software Process Improvement*, Kausai Science City, Japan, 2004, pp. 220-233.

[57] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *Journal of Software Maintenance,* vol. 20, pp. 435-461, 11/01 2008.

[58] H. Li and S. Thompson, "Clone detection and removal for Erlang/OTP within a refactoring environment," in Proceedings of the *ACM SIGPLAN workshop on Partial evaluation and program manipulation* , New York, NY, USA, 2009, pp. 169-178.

[59] C. Brown and S. Thompson, "*Clone detection and elimination for Haskell*," in Proceedings of the *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2010, pp. 111-120.

[60] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Journal of Information and Software Technology,* vol. 54, no. 12, pp. 1297-1307, 2012.

[61] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in Proceedings of the *International Conference on Software Engineering* , 2007, pp. 96-105.

[62] F. Arcelli Fontana, M. Zanoni, and F. Zanoni, "A Duplicated Code Refactoring Advisor," in Proceedings of the *International Conference on Agile Software Development (XP 2015)*, Springer, May 2015 pp. 3-14.

[63] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?," in Proceedings of the *International Conference on Software Engineering*,, Florence, Italy, 2015.

[64] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," Queens University, Kingston, Ontario, Canada, TR 2007-541, 2007.

[65] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Journal of Science of Computer Programming,* vol. 74, no. 7, pp. 470-495, 2009.

[66] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Journal of Information and Software Technology,* vol. 55, no. 7, pp. 1165-1199, 2013.

[67] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in Proceedings of the *IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, pp. 36-43.

[68] C. K. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," in Proceedings of the *International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 157-166.

[69] J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools," in Proceedings of the *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 321-330.

[70] 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 *IEEE/ACM International Conference on Software Engineering* , 2016, pp. 1157-1168.

[71] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: A Token Based Large-Gap Clone Detector," in Proceedings of the *IEEE/ACM International Conference on Software Engineering* , 2018, pp. 1066-1077.

[72] R. K. Saha, C. K. Roy, K. A. Schneider, and D. Perry, "Understanding the evolution of Type-3 clones: An exploratory study," in Proceedings of the the Conference on Mining Software Repositories , San Francisco, CA, 2013.

[73] A. D. Lucia, F. Ferrucci, G. Tortora, and M. Tucci, *Empirical Experimentation in Software Engineering* (Emerging Methods, Technologies, and Process Management in Software Engineering). Wiley-IEEE Press, 2007, pp. 201-247.

[74] S. Prion and K. A. Haerling, "Making Sense of Methods and Measurement: Pearson Product-Moment Correlation Coefficient," *Journal of Clinical Simulation In Nursing,* vol. 10, no. 11, pp. 587-588, 2014.

[75] S. Prion and K. A. Haerling, "Making Sense of Methods and Measurement: Spearman-Rho Ranked-Order Correlation Coefficient," *Journal of Clinical Simulation In Nursing,* vol. 10, no. 10, pp. 535-536, 2014.

[76] B. Ratner, "The correlation coefficient: Its values range between +1/−1, or do they?," *Journal of Targeting, Measurement and Analysis for Marketing,* vol. 17, 2009.

[77] N. Mohd Razali and B. Yap, "Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests," *Journal of Statistical Modeling and Analytics,* vol. 2, 2011.

[78] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *Journal of ACM SIGAPP Applied Computing Review,* vol. 12, pp. 20-36, 2012.

[79] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Journal of Science of Computer Programming,* vol. 95, pp. 445-468, 2014.

[80] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," in Proceedings of the *Conference on Reverse Engineering*, 2008, pp. 81-90. S

[81] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating Code Clone Genealogies at Release Level: An Empirical Study," in Proceedings of the *IEEE Conference on Source Code Analysis and Manipulation*, 2010, pp. 87-96.

Appendix A Extending the Taxonomy of Clone Detection Types to Account for Overlapping Information in Clone Detection Results

The bar chart figures below help graphically compare NiCad results before and after overlap reduction as well as reduction percentages for clone pairs, clone classes and clone fragments, respectively, for the subjects of the previous study. (Results for new subjects are similar as illustrated by Table VII and Table VIII.)

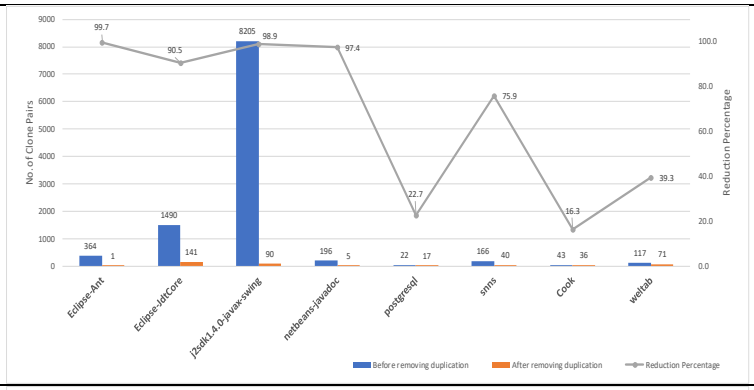
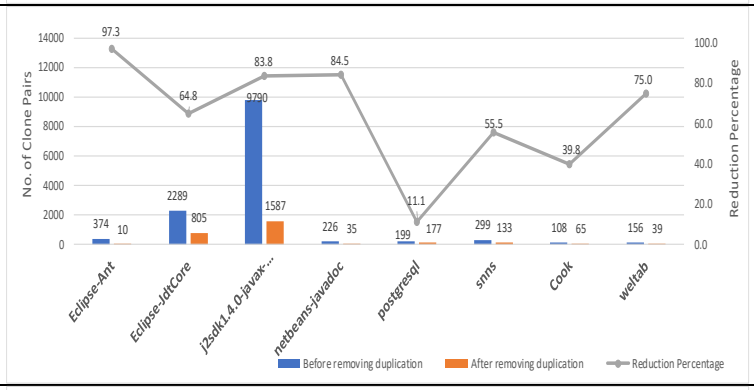
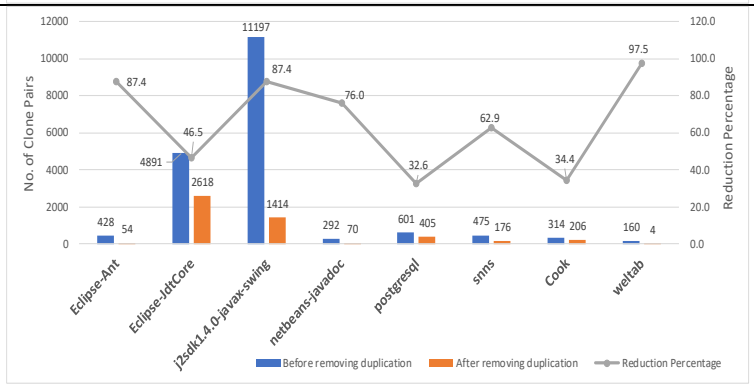
10%	
20%	
30%	

Illustration 23 Clone pairs comparison at various threshold values for different subjects (recall such data was not available in previously published work).

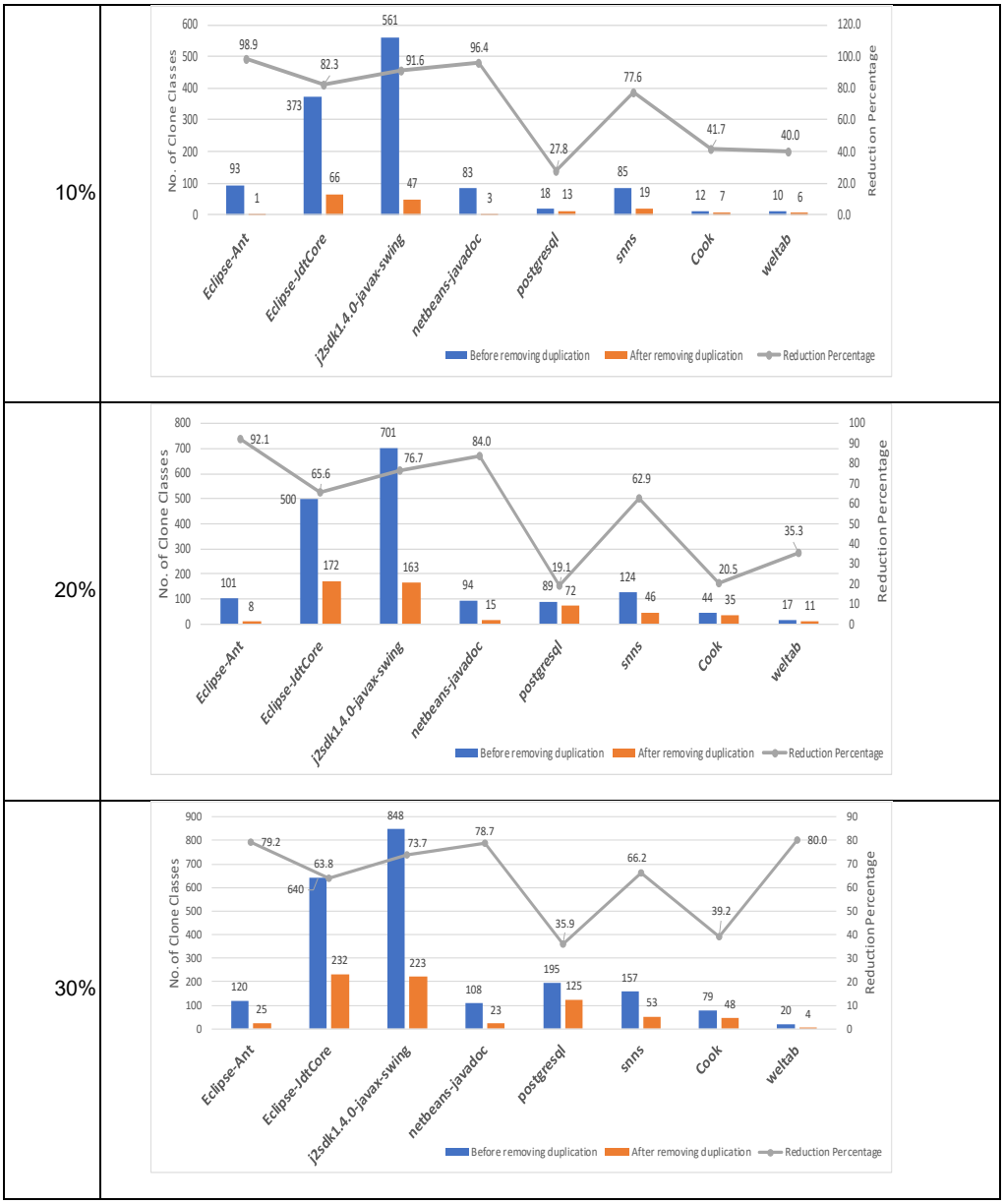


Illustration 24 Clone classes comparison at various threshold values for different subjects (recall such data was not available in previously published work).

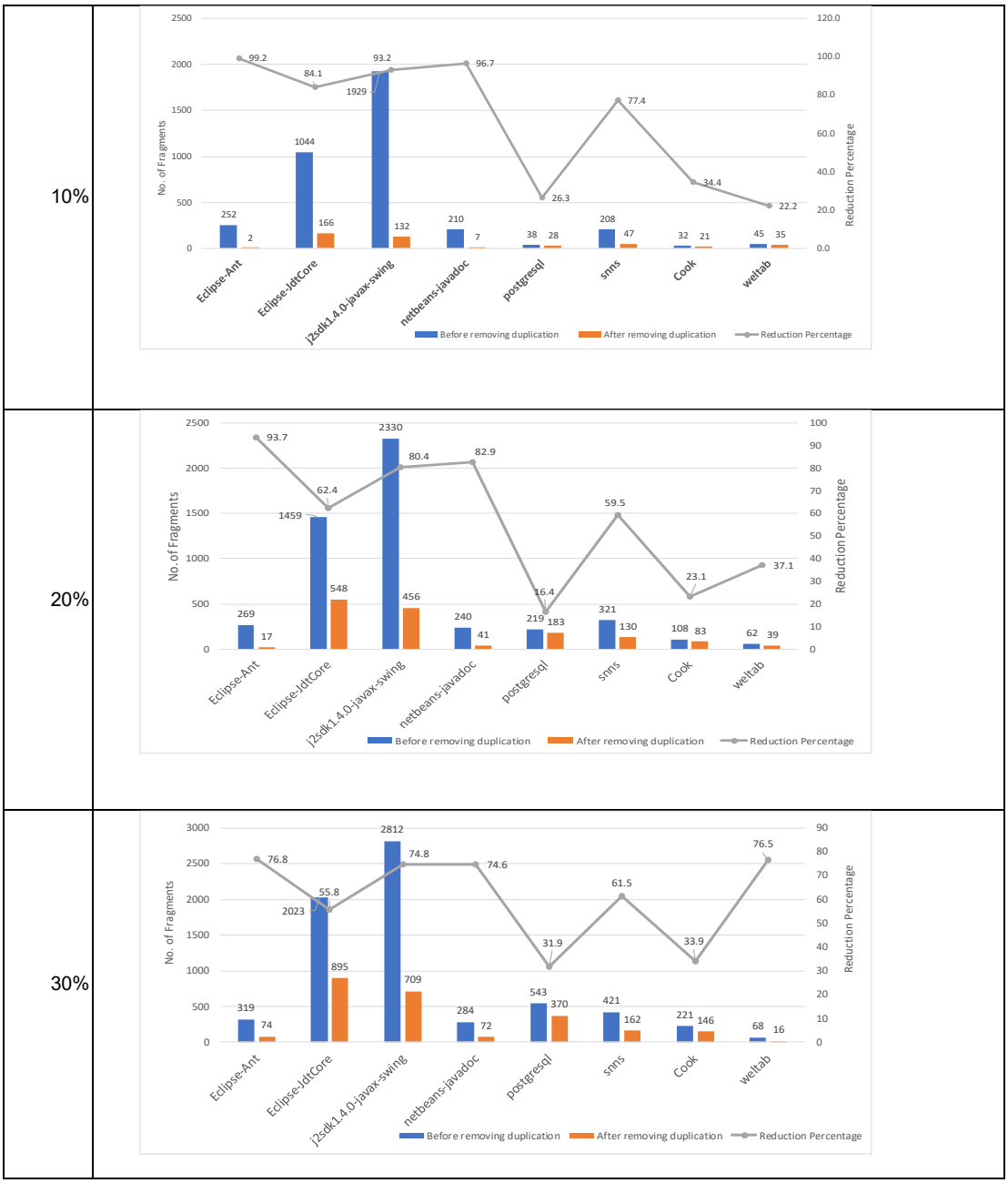


Illustration 25 Clone fragments comparison at various threshold values for different subjects (recall such data was not available in previously published work).

Appendix B An Analysis of Complex Industry Test Code using Clone analysis

The below table shows the clone class size distribution for various pure clone class types for Java project. (Results for new subjects are similar as illustrated by Illustration 6)

Clone Type \Clone class size	2	3	4	5	6	7	8	9	10	11	12	13	14
Type 1	730	399	24	9	26	5	5	5	10	2	5	3	4
Pure Type 2 Consistent	420	100	67	42	27	17	19	13	6	7	9	8	4
Pure Type 2 Blind	34	18	9	6	11	7	1	3	3	2	4		
Pure Type 3	596	214	63	40	32	6	9	8	8	5	4	2	2
Pure Type 3 Consistent	359	107	49	29	29	16	10	12	8	6	3	7	8
Pure Type 3 Blind	202	90	38	20	15	10	12	6	10	1	1	3	3
Clone Type \Clone class size	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
Type 1			3		2			1	2			2	
Pure Type 2 Consistent	4	4	1	2	3	1	2	4	1	4	2	1	1
Pure Type 2 Blind	1		1	3		1		3		1		3	
Pure Type 3	3	1	3	1		1	1	2	1		2		2
Pure Type 3 Consistent	3	6	2	3	1	2	1	3	1	2	2	3	1
Pure Type 3 Blind	3	2	1	2	3	1	2	2		2		2	
Clone Type \Clone class size	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**38**	**39**	**40**	**41**
Type 1	2						2					2	
Pure Type 2 Consistent	1			2			1		1				1
Pure Type 2 Blind	2							2		1			1
Pure Type 3	1					1	1	2		1	1	1	
Pure Type 3 Consistent	1	1	1	2	1	1					1		
Pure Type 3 Blind	1	2	1			1		2	1		1		
Clone Type \Clone class size	**42**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**55**	**60**	**61**	**62**
Type 1		2				1							
Pure Type 2 Consistent	1	1	1	1	1		1				1		
Pure Type 2 Blind					1				1			1	
Pure Type 3		1			2			1					
Pure Type 3 Consistent	1		1			1				1			
Pure Type 3 Blind	1								1				1
Clone Type \Clone class size	**63**	**66**	**68**	**70**	**72**	**73**	**74**	**76**	**77**	**79**	**85**	**89**	**91**
Type 1				1					1			1	
Pure Type 2 Consistent							1	1					3
Pure Type 2 Blind										1			
Pure Type 3												1	
Pure Type 3 Consistent			2		1	1					1		
Pure Type 3 Blind	1	1					1				1		
Clone Type \Clone class size	**93**	**96**	**101**	**102**	**103**	**104**	**105**	**106**	**108**	**109**	**110**	**111**	**113**
Type 1	2				2		1						
Pure Type 2 Consistent		1					2	1	1				
Pure Type 2 Blind											1	1	
Pure Type 3					1			1					1
Pure Type 3 Consistent			1	1		1							1
Pure Type 3 Blind										1			
Clone Type \Clone class size	**119**	**127**	**130**	**133**	**140**	**148**	**149**	**151**	**163**	**164**	**182**	**188**	**194**
Type 1											2		
Pure Type 2 Consistent					1	1		1					1
Pure Type 2 Blind													
Pure Type 3			1								1		
Pure Type 3 Consistent		1					1			1		1	
Pure Type 3 Blind	1			1					1				

Clone Type \Clone class size	196	204	206	208	211	219	221	223	238	239	240	247	248
Type 1				1									
Pure Type 2 Consistent					1	1	1				1		1
Pure Type 2 Blind									1				
Pure Type 3	1		1									1	
Pure Type 3 Consistent		1						1		1			1
Pure Type 3 Blind													

Clone Type \Clone class size	276	301	313	319	322	349	359	461	475	513	531	570	625
Type 1			1										
Pure Type 2 Consistent	1				3			1					
Pure Type 2 Blind				1					1		1	1	
Pure Type 3			1										
Pure Type 3 Consistent		1				1	1			1			1
Pure Type 3 Blind												1	

Clone Type \Clone class size	632	886	926	1562	1663	2050	3129	4844
Type 1								
Pure Type 2 Consistent								
Pure Type 2 Blind								
Pure Type 3				1		1		
Pure Type 3 Consistent		1	1		1			
Pure Type 3 Blind	1						1	1