FEASIBILITY OF DETECTING CODE CLONES USING INTERFACE SIMILARITY

MD RAKIB HOSSAIN
Master of Science in Software Engineering,
Institute of Information Technology
University of Dhaka
Class Roll: MSSE 0502
Session: 2017-2018
Registration Number : 2012-212-073

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology,University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Master of Science in Software Engineering

Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

FEASIBILITY OF DETECTING CODE CLONES USING INTERFACE SIMILARITY

MD RAKIB HOSSAIN

Approved:

Signature *Date*

Student: Md Rakib Hossain

Supervisor: Dr. Kazi Muheymin-Us-Sakib
Professor
Institute of Information Technology
University of Dhaka

To *Hosne Ara Begum*, my mother
who has always been there for me and inspired me

Abstract

Code clone refers to identical code fragments in a code repository. Developers usually repeat similar interface, for example, similarity of the return type, method name and parameter types while cloning a code fragment. A hypothesis states that when two methods contain similar interfaces, it is expected that those methods contain same logical implementation and perform similar behavior. Generic approaches of detecting clones are to compare each code fragment to each other with similarity function. If the similarity is higher than a given threshold, code fragments are reported as clones. These approaches are known as Complete Search approaches, which cause a prohibited polynomial comparison that is not effective in large code repositories. In order to reduce candidate code fragment comparison, the relationship between code clone and interface similarity may be helpful in clone detection.

The aim of this research is to develop a new code clone detection approach that can detect clones with reduced candidate comparison by investigating the relationship between code clone and interface similarity. This relationship is established by conducting an exploratory study where cloned method pairs are detected in code repositories. Then, interface information is extracted from source code, and several interface similarities are measured using this information such as two methods are similar if those contain same return type. The experimental corpus contains three different types of code repositories with 35, 109 and 24,558 Java projects respectively. The detected clone pairs in three code repositories are

231411, 242992 and 130226 respectively. It shows that on average 79.65% intra-project and 69.44% inter-project clones contain similar interfaces. Besides, the average similarity of interfaces in exact clone (Type-1), renamed (Type-2) and gapped (Type-3) clones are 100%, 69.35% and 67.34% respectively. These results are evident that code clone is strongly related to interface similarities.

Based on the relationship between code clones and interface similarities an Interface Driven Code Clone Detection (IDCCD) approach is developed. During clone detection, IDCCD tokenizes the method blocks from the source files. For those block, interface information is extracted and indexed with mapped tokens for quick retrieving. Then, similar interfaces are retrieved from that index and compared with a similarity function for detecting clones. IDCCD detects clones with the reduced candidate comparison since it only compares code fragments that have similar interfaces. It is observed that compared to Complete Search approach IDCCD overall reduces 53.04% candidate comparison detecting the same number of clones. IDCCD's recall is measured by using a clone detector evaluation framework called BigCloneEval and precision is identified by manually validating a sample of clones. The results represent that IDCCD attains 95% recall and 84% precision that is close to other clone detector's performance.

Acknowledgments

"*All praises are due to Allah*"

First and Foremost, I express my gratitude to Allah, The Almighty and The Lord of The World, for giving me this opportunity and granting me the ability to continue my research work effectively.

I want to express my significant appreciation and profound respect to my supervisor, Professor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka. This research could not be successful without his support, motivation, efforts and immense knowledge. He has been relentless in his efforts to bring the best out of me. His continuous guidance, while doing the research and writing the thesis, has enabled me to complete it successfully.

I would like to thank my thesis reviewers Dr. Muhammad Masroor Ali and Dr. Md. Saidur Rahman for their valuable feedbacks which have enabled to enhance the quality of the thesis

I am also thankful to the DSSE Student Research Group, Institute of Information Technology, University of Dhaka, for their continuous support and valuable feedback that helped me in improving my research work.

I am also thankful to Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No 56.00.0000.028.33.079.17-223 (Part-1) -772, Date 20-06-2017.

List of Publications

1. "Md Rakib Hossain Misu and Kazi Sakib. Interface driven code clone detection." *In Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, Nanjing, China, December 4-8, 2017, pages 747-748, IEEE.

2. "Md Rakib Hossain Misu, Abdus Satter and Kazi Sakib. An Exploratory Study on Interface Similarities in Code Clones." *In Proceedings of 24th the Asia-Pacific Software Engineering Conference Workshops, (APSECW)*, Nanjing, China, December 4-8, 2017, pages 126-133, IEEE.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Copying and pasting of source code is a common task during software development. Apart from copying-pasting, code reusability can be done through the unintentional analogous functionality of the code, automatic code generation and plagiarism with or without minor modifications. Thus, similar pieces of code that propagate within or between software systems are known as code clones [2]. Previous studies show that a typical software system can contain a significant amount of (between 7% to 23%) cloned code [3]. In many ways, code clones are detrimental to software evolution and maintenance. For example, if a bug is present in a code fragment, all fragments analogous to that may contain the same bug. To detect that bug, all those code fragments should be tested [4]. Besides, cloned codes increase the workload while enhancing and adapting new features. So, code clone detection and management are necessary to maintain software quality, prevent and locate new bugs, reduce development cost and risk [5].

Over many years, various tools and techniques have been proposed to detect code clone [6]. However, in large code repositories, most of the techniques take huge execution time that is not expected in software development. This is because the basic way of detecting clones is needed to compare each code fragment to each other. This technique is known as Complete Search approach.

While code reusing, not only similar pieces of code, method interface such as method name, return type and parameter types may also be repeated with the code fragment. If two methods contain similar interfaces, it is very likely those perform same functionalities either entirely or at least partially [7]. When those methods contain same interface and perform analogous functionalities, it indicates that these methods should be semantic or syntactic code clone to each other [7]. It is evident that there is a relation between code clones and interface similarities. Since similar interfaces are repeated with similar pieces of codes, this relationship can be helpful for code clone detection. In this research, the relationship between code clones and interface similarity is established through an exploratory study and further this relationship is applied to code clone detection.

This chapter presents the motivation of detecting code clones using interface similarities. Issues of the existing code clone detection techniques are also demonstrated. It then formulates the research questions and provides a guideline towards answering those questions. The contribution and achievement of this research are also discussed. Finally, the organization of this thesis is outlined for giving a guideline to the readers.

1.1 Motivation

An elementary way of detecting cloned code fragments is to measure the degree of similarity between the code fragments using various similarity functions such as Jaccard similarity, Cosine similarity etc. [7]. The higher the similarity, the more the chances of the code fragments to be cloned. To identify clones in a large code repository, each code fragment is needed to be compared against all code fragments that cause a prohibitively $O(n^2)$ time complexity [8]. Figure 1.1 represents the candidate method comparison in 35 Apache Java projects while detecting clones (Appendix A represents details about the projects). For example, for a project

Junit the number of code fragments (in this case method block) is 274. To detect clones, it needs to make 274^2=75,076 candidate comparisons. From Figure 1.1, it has been observed that candidate method comparison exponentially increases with the number of methods present in those projects (red line). It implies an algorithmic challenge for all the techniques to perform the comparison in a faster and efficient way. Most of the techniques do not work fast for making the comparison in a large code repository. Two obvious reasons are responsible for it such as the computational complexity of the detection problem itself and the consideration of all code fragments for a single clone detection. This challenge, along with modern use cases and todays large systems make clone detection in large code repository a difficult problem.

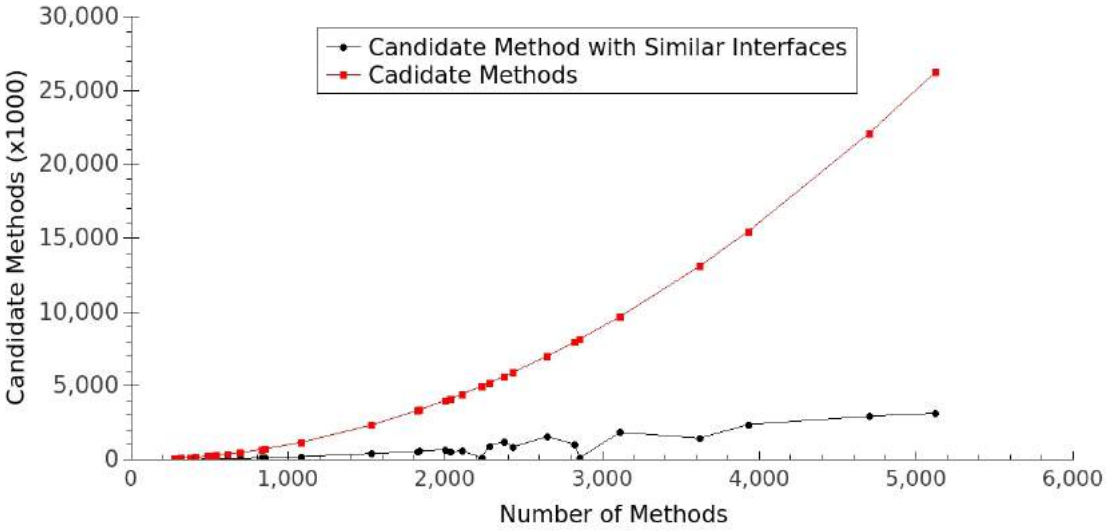


Figure 1.1. Growth in number of candidate comparisons with the increase in the number of method blocks

However, a common replication that appears in large software repositories is similar method interface. It refers the return type, method names and parameter types of a method (shown in Figure 1.2). Sometimes, developers repeat similar method interfaces across the code repositories during code reusing. This kind of

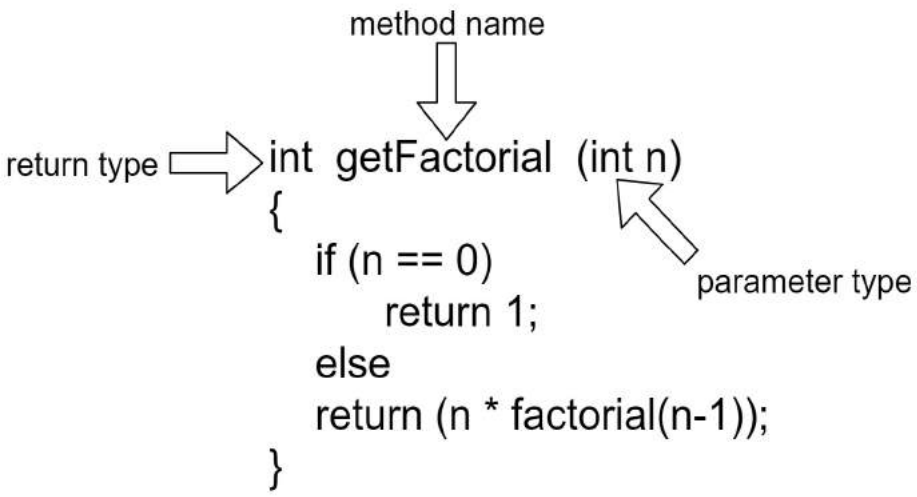


Figure 1.2. Method Interface Example

replication can be useful in many ways, for example, it is a hypothesis that if two methods contain the similar interfaces, it is very likely that those methods perform same functionalities either entirely or at least partially. If those methods contain same interface and perform analogous functionalities, it exhibits that these methods should be semantic or syntactic code clone to each other [7]. But verifying that hypothesis is very challenging because there are various types of clones such as inter-project clone, intra-project clone, exact clone (Type-1), renamed clone (Type-2) etc. and for each type of clone measuring the interface similarity is also difficult since multiple similarity combinations can be possible. Besides, an exploratory study [7] shows that 80% of the target projects contained repeated similar interfaces. An interface from a non-trivial method to repeat itself across a large repository is around 25% [7]. The following example is the evidence of that claims. For example Listing 1.1 and 1.2 represents two method blocks collected from project *rhino*[1] and *stanford-nlp*[2]. These two method blocks represent a clone pair where both contain same syntactical implementation in the method body and similar interface such as same return type, parameter type and method names.

[1]rhino/org/mozilla/javascript/ast/AstNode.java
[2]stanford-nlp/src/edu/stanford/nlp/trees/Tree.java

Listing 1.1: Method Block From Project *rhino*

```
1 public String makeIndent(int indent) {
2   StringBuilder sb = new StringBuilder();
3   for (int i = 0; i < indent; i++) {
4     sb.append("␣␣");
5   }
6   return sb.toString();
7 }
```

Listing 1.2: Method Block From Project *stanford-nlp*

```
1 private static String makeIndentString(int indent) {
2   StringBuilder sb = new StringBuilder(indent);
3   for (int i = 0; i < indentIncr; i++) {
4     sb.append('␣');
5   }
6   return sb.toString();
7 }
```

Another example is that Listing 1.3, 1.4, 1.5 and 1.6 represent four method blocks m_1[3], m_2 [4], m_3[5], m_4[6] collected from three different projects such as *Apache pmd*, *Apache Lucene* and *stanford-nlp* respectively. It is observed that all those four method blocks are clone to each other and interestingly those method blocks contain similar interfaces. These examples infer that code clones (in this case method block) contain interface similarities. So, it is possible that while detecting clones, comparison of method blocks with similar interface can reduce the number of candidate comparison (shown in Figure 1.1 with black line).

[3]pmd/net/sourceforge/pmd/lang/cpp/ast/CppParserTokenManager.java
[4]stanford-nlp/src/edu/stanford/nlp/ling/tokensregex/TokenSequenceParserTokenManager.java
[5]lucene/org/apache/lucene/queryParser/QueryParserTokenManager.java
[6]stanford-nlp/src/edu/stanford/nlp/trees/tregex/TregexParserTokenManager.java

Listing 1.3: Method Block (m_1)

```
1 int jjStopAtPos(int pos, int
    kind){
2   jjmatchedKind = kind;
3   jjmatchedPos = pos;
4   return pos + 1;
5 }
```

Listing 1.4: Method Block (m_2)

```
1 int jjStopAtPos(int pos, int
    kind){
2   jjmatchedKind = kind;
3   jjmatchedPos = pos;
4   return pos + 1;
5 }
```

Listing 1.5: Method Block (m_3)

```
1  int jjStopAtPos(int pos, int
     kind){
2  jjmatchedKind = kind;
3  jjmatchedPos = pos;
4  return pos + 1;
5  }
```

Listing 1.6: Method Block (m_4)

```
1  int jjStopAtPos(int pos, int
     kind){
2  jjmatchedKind = kind;
3  jjmatchedPos = pos;
4  return pos + 1;
5  }
```

For example, in order to detect clones in project *Junit* containing 274 methods, it is needed to compare only 10,810 method blocks that contain similar interfaces (shown in Figure 1.1). To the best of the author's knowledges, the relationship between code clone and interface similarity have not been identified and applied on code clone detection.

1.2 Issues in State-of-the-Art Approaches

Numbers of clone detection tools and techniques have been proposed in the literature. These tools and techniques differ from many aspects such as what type of detection algorithm is used, how source code is represented to operate and how many types of code clone can be detected. Roy et at have performed a comprehensive survey focusing the strength and limitation of various clone detection

techniques [9]. According to Roy et al, clone detection techniques can be categorized into four types, for example, textual, lexical, syntactic and semantic [9]. Textual or string-based techniques use the source code with little or no transformation and use string matching algorithms for detecting clones. These techniques vary from one another based on the string matching algorithms and the granularity level of matching units. For example, Marcus et al used the Latent Semantic Indexing (LSI) algorithm for detecting high-level concept clones [10]. Similarly, Cordy et al also applied the Longest Common Subsequence (LCS) algorithm in their tool called NiCad [11] for an efficient text line comparison to find near miss match clones. However, text-based approaches are not efficient enough because string matching algorithms do not work faster in larger dataset and face time complexity issues [9].

Lexical or token based approaches first transform the source code into a sequence of lexical tokens with a predefined granularity level. Next, the comparison is done among the tokens. The transformation of the tokens is done by using the lexical analyzer to parse the source code and rule-based transformation is applied to generate a stream of the tokens. The tokenization is useful to detect code fragments that have different syntax but have a similar meaning. Most popular clone detectors, for example, Dup [12], CCFinder [13] etc. are built based on token-based techniques. Syntactic or tree-based techniques detect clones by finding similar subtrees. In tree-based techniques, first, the source code are parsed into its Abstract Syntax Tree (AST) representation. Next, various graph or tree matching algorithms such as suffix tree matching, AST pattern matching are used to detect similar subtrees finding potential clones. Jiang et al proposed a novel approach for matching similar trees in their tool Deckard [14]. They computed a certain characteristic vector from the AST and used Locality Sensitive Hashing (LSH) to cluster similar vectors based on Euclidean distance metric for detecting clones. However, tree-based techniques face scalability issues as parse trees

contain a lot of information consuming high memory [9].

Semantic-based techniques include Program Dependency Graph (PDG)-based techniques and the metrics based techniques. These techniques detect clones using semantic information such as method call, control flow etc. instead of using syntactical information. PDG-based techniques are similar to tree-based techniques but these approaches contain control flow and data flow information that conveys more semantic information rather than AST. So, PDG based techniques are more robust to detect clone. Komondoor et al is the pioneer for applying PDG-based techniques in code clone detection [15]. They represented a program source code into a dependency graph and then transformed the code clone detection problem to finding a subgraph problem over PDG. The extension of their work was done by Krinke et al [16] to show how pattern matching algorithm can be useful to detect maximal similar subgraph in more efficient way. However, similar to tree-based techniques PDG-based approaches are not scalable and faster in large code repositories [9]. Metrics-based techniques classify the source code using a set of metrics such as cyclomatic complexity, function points, lines of code, etc. A study was performed by Mayrand at el. applying the metrics-based approach in code clone detection [17]. They calculated twenty one metrics in each function unit of a program and detected functions with similar metrics values as clones. It was found that metrics-based approaches are more effective for detecting clones in source code with high-level granularity such as file level or class level.

All these approaches are not scalable and cannot make the code fragment comparison faster in a large code repository. However, Sajnani et al has introduced a token based approach called SourcererCC [18] that is scalable for large code repository. They have used sub block overlapping and token positioning heuristic to reduce code fragments for efficient comparison. Similar to SourcererCC [18], Jefferey et al, has recently proposed a technique called CloneWorks [19] that is also scalable. Although both techniques used sub-block overlapping heuristic, the

key difference is that SourcererCC [18] is developed as a token based approach and CloneWorks [19] is flexible for any source code representation. As both techniques never applied interface similarity heuristic to reduce candidate code fragment comparison, it can be applied to reduce candidate comparison during clone detection.

1.3 Research Questions

To detect code clone, based on the relationship between interface similarities and clones, first the relationship should be investigated. To do so, an exploratory study has been performed that has identified the relationship of interface similarities with all types of clones. No early work has been found that has applied interface similarity heuristic to detect code clones and reduced the candidate comparison. Existing approaches both SourcererCC [18] and CloneWorks [19] use sub-block overlapping heuristic to minimize the candidate comparison but this minimization does not make the comparison process faster for large code repositories. If a strongly positive relation between interface similarities and code clones is presented, it can be useful for both reducing and optimizing the candidate comparison while detecting clones. This is because, the number of similar interfaces in a code repository is smaller than the total number of interfaces [7]. So, it helps to develop a new lightweight and faster code clone detection tool based on the relationship between interface similarity and code clones. As a result, the process of detecting code clone in large code repositories will be faster. Thus, it leads to the following Research Questions (RQ).

- **Research Question 1 (RQ1):** How strong the relationship is between code clones and interface similarity?

  To answer this question following steps are adopted.

  (a) A large, medium and small source code repositories, containing various projects, is required to be built for conducting the exploratory study.

For each project from the code repositories, various types of method clone pairs including inter-project, intra-project, exact clone (Type-1) etc. are identified using existing code clone detectors such as SourcererCC [18] and NiCad [20].

(b) For each clone pair, interface information such as return type, method name, and parameter types is extracted from each clone pair. Keywords are also identified from method name. Finally, using this extracted interface information various types of interface similarities are measured by satisfying similarity conditions. The frequency of how many clones satisfy each condition is calculated and constructed a conditional distribution. These results are used for establishing the desired relationship.

- **Research Question 2 (RQ2):** Can interface similarity minimize the candidate code fragment comparison in code clone detection?

  The following steps are carried out to answer this research question.

  (a) An Interface Driven Code Clone Detection (IDCCD) approach is developed based on the relationship of interface similarities and clones. Twenty four open source Apache Java projects are randomly selected as experimental dataset to conduct a candidate comparison minimization experiment.

  (b) Three evaluation metrics such as number of clones detected, total number of candidates compared, total number of tokens compared are chosen. This is because these metrics help to describe the minimization of candidate code fragments. How much minimization is possible by IDCCD while detecting clones is also measured. A Complete Search approach is also developed for comparative result analysis.

- **Research Question 3 (RQ3):** How accurate the Interface Driven Code Clone detection approach against the state-of the-art works?

In order to answer this question following steps are performed

(a) To measure the accuracy of Interface Driven Code Clone Detection approach, recall and precession are used as evaluation metrics. Existing clone detection benchmark BigCloneBench [21] and evaluation framework BigCloneEval [22] are used to calculate the recall of IDCCD. This is because, BigCloneBench [21] and BigCloneEval [22] contain large code repository with millions of true clones and false clones. Besides, it produces clone report for each type of clone.

(b) Precision is measured manually validating a random sample of detected clones for each tool. For each tool, randomly 300 sample clone pairs are selected. These sample clones are selected from those clones that are reported by each tool in the recall measurement experiment. Finally, a comparative result is presented that shows the accuracy comparison among the clone detectors.

1.4 Contribution and Achievement

This research presents a novel solution named as Interface Driven Code Clone Detection (IDCCD) to address the questions that have been raised in the previous section. Through an exploratory study, the relationship between code clone and interface similarity has been established. Based on this relationship IDCCD detects code clones with reduced candidate comparison. In a nutshell, the major contributions of this research work include:-

- The relationship between code clones and interface similarities has not been investigated yet. In this research work, this relationship has been established through an exploratory study. Firstly, cloned methods are detected in code repositories. Then, interface information such as return type, method name

and parameter types is extracted from source code and interface similarities are measured using that information. The experimental results exhibit that code clone is strongly related to interface similarities. (Chapter-4)

- Based on the established relationship, a new light weight Interface Driven Code Clone Detection (IDCCD) approach is developed. IDCCD detects clones with the reduced candidate code fragment comparison since it only compares those code fragments that contain similar interfaces. This is verified by a candidate code fragment minimization experiment. Reduction of candidate code fragment comparisons enables IDCCD to be an effective clone detection approach. (Chapter-5)

- With reduced candidate comparison, the performance to IDCCD is also near to the sate-of-the-art clone detectors. The accuracy is measured by clone detector evaluation framework called BigCloneEval [22]. Compared to other popular clone detectors such as NiCad [20], SourcererCC [18] and CloneWorks [19], IDCCD gains acceptable recall and precision. (Chapter-5)

1.5 Organization of the Thesis

This section provides an overview about the remaining chapters of this thesis. The chapters are organized as follows-

Chapter 2: Background Study

In this chapter, the preliminary topics for understanding the work have been described. The terms and terminologies of code clone related study such as Code Fragment, Code Clone, Clone Pair, Method Clone etc. are mentioned with examples. Besides, various types of clones, for example, Intra-project clone, Inter-project clone, exact clone (Type-1), renamed clone (Type-2) etc. are also presented with real examples. At the end of this chapter, several source code processing

techniques including Keyword decomposition, Stop Words Removal, Stemming and constructing Inverted Index are also discussed.

Chapter 3: Literature Review

This chapter demonstrates existing code clone detection approaches and theirs categories such as Text-based, Tree-based, Token-based code clone detection. From each category, pros and cons of some well known code clone detectors are mentioned. Clone detector evaluation benchmarks and frameworks are also discussed at the end of this chapter for better understanding of the research.

Chapter 4: Relationship between Interface and Clones

In this chapter, an exploratory study is performed to identify the relationship between code clones and interface similarities. The overview of study design and the description of experiment dataset for the study are exhaustively described. For each type of clone, the relationship of clone and interface similarity is quantified by answering three sub research questions. The outcome and the threats to validity of the exploratory study are clearly mentioned at the last part of this chapter.

Chapter 5: Interface Driven Code Clone Detection

This chapter exhibits newly proposed approach called Interface Drive Code Clone Detection (IDCCD). Firstly, the methodology and the implementation of IDCCD is described. Then, a candidate code fragment comparison minimization experiment is conducted to show how IDCCD reduces the candidate comparison while detecting clones. At the last part of this chapter, a comparative study is provided that represents the performance and accuracy of IDCCD against some popular code clone detectors such as SourcererCC [18], CloneWorks [19] etc.

Chapter 6: Conclusion This chapter first summarizes the overall thesis. It then discusses the answer of the three research questions. Lastly, the chapter concludes the thesis with considerable future remarks.

Chapter 2

Background Study

Identical code fragments, in a code repository, are known as code clones. Because of many issues, code clone has been a topic of research for a long time. A large number of research works have been performed in this area. This chapter comprises the initial terms and terminologies associated with code clone-related research. First few sections discussed the basic definition and terminologies of code clones such as Code Fragment, Clone Pair Method Clone etc. Next, definitions of various types of clones are given. Issues and important aspects of code clone detection are discussed briefly and different types of clone detection tool evaluation metrics are mentioned. Besides, various aspects of source code processing such as Keyword Decomposition, Stop Words Removal etc. are also elaborately discussed at the end of this chapter.

2.1 Terminology of Code Clones

Within this thesis, some well-known code clone related terms and terminologies are used widely. In this section, definitions of those terms and terminologies are provided with examples.

2.1.1 Code Fragment

A successive segment of source codes is known as a code fragment. It is specified by the triple (*f, s, e*), where *f* represents the source file, *s* represents the line number from where the method starts on and the line at which it ends is represented by *e* [9]. For an example, source file *Example.java* contains a code fragment shown in Listing 2.1. It is observed that code fragment starts from Line-1 and ends at Line-5. So, this code fragment is represented by the triple {*Example.java,1,5*}.

Listing 2.1: Code Fragment Example (1)

```
1  if (a >= b ) {
2  c = d + b ; // Comment 1
3  d = d +1;
4  } else
5  c =d - a ; // Comment 2
```

2.1.2 Code Clone

If two or more code fragments in a software system's code-base are identical or nearly similar to one another, these are known as code clones [9]. Listing 2.2 and Listing 2.3 represent two code fragments Clone Example (a) and Clone Example (b) respectively. By observing these code fragments, it is intuitive that both the code fragments are identical to each other that makes those code clone.

2.1.3 Clone Pair

A pair of code fragments that are similar or identical to each other is called clone pair [9]. For example, code fragments from Listing 2.2 and 2.3 represent a clone pair.

Listing 2.2: Clone Example (a)

```
public int getSum(int n){
  int sum=0;
  while(n>=0){
    sum+=n
    n--
  }
  return sum;
}
```

Listing 2.3: Clone Example (b)

```
public int getResult(int r){
  int result=0;
  while(r>=0){
    result+=n
    r--
  }
  return result;
}
```

2.1.4 Method Clone

If two methods are cloned, these are specified as method clone. It is expressed by the triple $\{m_1, m_2, t\}$, where m_1 and m_2 represent the similar methods, and the clone type is specified by t [9]. Listing 2.4 and Listing 2.5 represent two code fragments (methods) Clone (m_1) and Clone (m_2). It is observed that both the code fragments represent a code clone and contain java methods called *swap*. Therefore, this clone exhibits a method clone that is represented by the triple: (Clone (m_1), Clone (m_2), *Type-1*).

Listing 2.4: Clone (m_1)

```
void swap(int i, int j) {
  int temp =i;
  i=j;
  j=temp;
}
```

Listing 2.5: Clone (m_2)

```
void swap(int i, int j) {
  int temp =i;
  i=j;
  j=temp;
}
```

2.1.5 Subject System

A subject system refers a code repository with different types of projects. These projects vary from one another by application domains and the number of files and Line of Code (LOC). Clone detection is performed on each project or all project of the subject systems [9].

2.2 Types of Code Clone

This section provides an exhaustive description of code clone types with examples. Based on the project dependency, code clones are categorized into two types such as *Project Dependent* and *Project Independent*. Each type is also categorized into multiple subtypes. Definition and real examples of those types are discussed in the following subsections.

- Project Dependent
  (a) Intra-project Clone
  (b) Inter-project Clone
- Project Independent
  (a) Type-1 (Exact Clone)
  (b) Type-2 (Renamed Clone)
  (c) Type-3 (Gapped Clone)
  (d) Type-4 (Semantic Clone)

2.2.1 Intra-Project Clone

A clone pair where the code fragments are found in the same project is known as Intra-project clone. Intra-project clones occur when developers reuse similar code fragments within the same project through copying and pasting with or without modifications. For example, Listing 2.6 and Listing 2.7 show two code fragments (methods) Intra-project (m_1) and Intra-project (m_2) collected from project *jfreechart*[1] respectively. It is observed that both the code fragments represent a clone pair within the same project. Since those code fragments are found in the same project, these code fragments create an intra-project clone pair.

Listing 2.6: Code Fragment Intra-project (m_1)

```
1 public int hashCode() {
2   int result = 39;
3   result = HashUtilities.hashCode(result, getToolTipText());
4   result = HashUtilities.hashCode(result, getURLText());
5   return result;
6 }
```

Listing 2.7: Code Fragment Intra-project (m_2)

```
1 public int hashCode() {
2   int result = 39;
3   result = HashUtilities.hashCode(result, getToolTipText());
4   result = HashUtilities.hashCode(result, getURLText());
5   return result;
6 }
```

[1]jfreechart/org/jfree/chart/entity/AxisEntity.java –jfreechart/org/jfree/chart/entity/PlotEntity.java

2.2.2 Inter-Project Clone

Inter-project clone refers a clone pair that contains code fragments from different projects instead of a single project. It can occur either intentionally or unintentionally. Generally, it occurs when developers reuse a code fragment copying from a project and pasting into other projects with or without little modification. An example of an inter-project clone is provided with Listing 2.8 and Listing 2.9 that represent two code fragments Inter-project (m_1) and Inter-project (m_2) collected from project *berkeleyparser*[2] and *stanford-nlp*[3] respectively. Since, both the code fragments are found from two different projects, those code fragments produce an inter-project clone pair between those two projects.

Listing 2.8: Code Fragment Inter-project (m_1)

```
1 public static double[] subtract(double[] a, double[] b) {
2   double[] c = new double[a.length];
3   for (int i = 0; i < a.length; i++)
4     c[i] = a[i] - b[i];
5   return c;
6 }
```

Listing 2.9: Code Fragment Inter-project (m_2)

```
1 public static double[] add(double[] a, double c) {
2   double[] result = new double[a.length];
3   for (int i = 0; i < a.length; i++) {
4     result[i] = a[i] + c;
5   }
6 return result;
7 }
```

[2]berkeleyparser/edu/berkeley/nlp/util/ArrayUtil.java
[3]stanford-nlp/src/edu/stanford/nlp/math/ArrayMath.java

2.2.3 Type-1 or Exact Clone

Syntactically identical code fragments, except for differences in white-space, layout and comments are known as Type-1 clone [9]. This type of clone occurs when developers perform copying and pasting of code fragment without modifications. Listing 2.10 and Listing 2.11 represent two code fragments (methods) Type-1 (m_1) and Type-1 (m_2) respectively, collected from project *Apache ant*[4]. These two code fragments represent two methods with the same return type, method name and parameter types. However, it is observed that code fragment Type-1 (m_2) contains a comment (Line:2). If this comment is disregarded, these two code fragments become identical and create a Type-1 clone pair.

Listing 2.10: Code Fragment Type-1 (m_1)

```
1 public void setClasspath(Path classpath) {
2   if (this.classpath == null) {
3     this.classpath = classpath;
4   } else {
5     this.classpath.append(classpath);
6   }
7 }
```

Listing 2.11: Code Fragment Type-1 (m_2)

```
1 public void setClasspath(Path classpath) {
2   if (this.classpath == null) {//check null
3     this.classpath = classpath;
4   } else {
5     this.classpath.append(classpath);
6   }
7 }
```

[4](ant/util/ClasspathUtils.java) and (ant/taskdefs/Property.java)

2.2.4 Type-2 or Renamed Clone

Syntactically identical code fragments, except for differences in white-space, layout, comments, identifier names and literal values [9] are considered as Type-2 clones. These clones are mainly originated from Type-1 clones due to renaming variable identifier or changing data types and literal values. Listing 2.12 and Listing 2.13 represent two code fragments (methods) Type-2 (m_1) and Type-2 (m_2) respectively, collected from project *eclipse-jdtcore*[5]. It is observed that code fragment Type-2 (m_1) contains a variable type *StringLiteral* (Line:2-4). However, in code fragment Type-2 (m_2) the corresponding variable is renamed as *PackageDeclaration* (Line:2-4). Because of renaming this variable, these two fragments create a Type-2 clone pair. This is because if this renaming is ignored, then these two code fragments become identical.

Listing 2.12: Code Fragment Type-2 (m_1)

```
1 boolean equalSubtrees(Object other) {
2   if (!(other instanceof StringLiteral)) {
3     return false; }
4     StringLiteral o = (StringLiteral) other;
5   return ASTNode.equals(getEscapedValue(), o.getEscapedValue());
6 }
```

Listing 2.13: Code Fragment Type-2 (m_2)

```
1 boolean equalSubtrees(Object other) {
2   if (!(other instanceof PackageDeclaration)) {
3     return false; }
4   PackageDeclaration o = (PackageDeclaration) other;
5   return ASTNode.equalNodes(getName(), o.getName());
6 }
```

[5](jdt/core/dom/StringLiteral.java) and (jdt/core/dom/PackageDeclaration.java)

2.2.5 Type-3 or Gapped Clone

Syntactically identical code fragments which differ at the statement level such as statements added, modified or removed with respect to each other, in addition to Type-1 and Type-2 clone differences, are known as Type-3 clones [9]. These clones originate from Type-1 and Type-2 clones due to addition, modification and deletion of source code in statement level. Type-3 clones are also called gapped or updated clones [9]. Listing 2.14 and Listing 2.15 represent two code fragments (methods) Type-3 (m_1) and Type-3 (m_2) extracted from project *Apache poi*[6]. It is seen that code fragment Type-3 (m_1) contains a statement at Line-5. However, in code fragment Type-3 (m_2) corresponding line is absent. Because of deleting this line from code fragment Type-3 (m_2), these two fragments create a Type-3 clone pair. The reason is that if that line is not deleted from code fragment Type-3 (m_2), then these two code fragments become identical.

Listing 2.14: Code Fragment Type-3 (m_1)

```
1 public int serialize(int offset, byte[] data){
2   int pos = 0;
3   LittleEndian.putShort(data, 0 + offset, sid);
4   LittleEndian.putShort(data, 2 + offset,(getRecordSize() - 4));
5   LittleEndian.putShort(data, 4 + offset + pos, field_1_axisType);
6   return getRecordSize();}
```

Listing 2.15: Code Fragment Type-3 (m_2)

```
1 public int serialize(int offset, byte[] data){
2   int pos = 0;
3   LittleEndian.putShort(data, 0 + offset, sid);
4   LittleEndian.putShort(data, 2 + offset, (getRecordSize() - 4));
5   return getRecordSize();}
```

[6](apache/poi/hssf/record/DatRecord.java)and(apache/poi/hssf/record/PlotAreaRecord.java)

2.2.6 Type-4 or Semantic Clone

Syntactically dissimilar code fragments that perform the same task but are implemented in different ways are called Type-4 code clones [9]. Therefore, these Type-4 clones are also known as semantic clones [9]. These clones are created unintentionally when developers implement the same functionality with their own approach without following traditional approaches.

Listing 2.16 and Listing 2.17 represent two code fragments (methods) Type-4 (m_1) and Type-4 (m_2) respectively. It is shown that code fragment Type-4 (m_1) and Type-4 (m_2) both perform the same task such as finding the factorial for a given number. However, the implementation of both methods syntactically dissimilar but these methods exhibit same behavior and functionality. So, these code fragments create a Type-4 clone pair.

Listing 2.16: Code Fragment Type-4 (m_1)

```
1 public int getFactorial(int n) {
2   if (n == 0) return 1;
3   else return n * factorial(n - 1);
4 }
```

Listing 2.17: Code Fragment Type-4 (m_2)

```
1 public int getFactorial(int n) {
2   int result = 1;
3   for (int i = 1; i <= n; i++) {
4     result *= i;
5   }
6   return result;
7 }
```

2.3 Why and How Code Clones Occur

There are many reasons for which code clones occur. A brief description of those reasons are mentioned below.

(a) Cloning as a Way to Reuse: The most common way of reusing code is the *copy-past-modify* programming practice [23]. It is observed that developers first fork a code repository of an existing solution and then add or modify the code to adapt to their requirements. Sometimes developers face difficulties to understand the software system of a new domain. These impose them to use the example oriented programming by copying and adapting the code that is already written in the system [8].

(b) Cloning for Maintenance Benefits: In many cases, code clones can be useful, for example, code reusing. Developers are often asked to reuse the existing code by copying and adapting it according to the new product requirements. If a code portion is well tested, reusing that code fragment through cloning reduces software maintenance efforts and costs [24]. In many real-time applications such as financial product, a monolithic code is well preferred instead of a modularized code because in those application function calls are very expensive [8].

(c) Limitation of Programming Languages/Frameworks: Some programming languages lack sufficient abstraction mechanism such as PHP, JavaScript etc. Because of this limitation of the programming languages, code clones are introduced. Many procedural languages are in a lack of some features such as inheritance, generic types, templates and parameter passing that make it difficult to write reusable code [25]. Sometimes clones can also be generated by the use of a specific framework. It uses a code generator for automatically producing *getter* or *setter* codes, for example, Java class attribute *getter* and *setter* codes [8].

(d) Software Development Practices: The practice of software development often influences how the code is written. In real life software develop-

ment, developers are always under a pressure to release the product within a tight deadline. These deadlines impose the developers to implement features through copying-pasting existing code and adapting to current needs [26]. In some companies, developers do not have write access to the existing code. They are not allowed to modify the existing code for reusing. Therefore, they have the only option to reuse that code through copying and pasting [8].

(e) Accidental Cloning: Sometimes code clones can occur accidentally. Developers often implement same functionality by writing same codes since they write the same algorithmic logic to implement the same functionality. Sometimes, it needs to perform a sequence of tasks for using library and Application Program Interface (API). This sequence of tasks is done through some order of method calls. As a result, such library and API usage may introduce accidental clones [23] [8].

2.4 Issues of Code Clones

There are some affects of code clones on software maintainability, reusability and quality. Some important issues of code cloning are listed below.

(a) Hidden bug propagation: If a code fragment contains a bug, all other code fragments that are cloned from or similar to this code fragment should contain the same bug [27]. So, cloning of that code fragment increases the probability of bug propagation [8].

(b) Unintentional inconsistencies: While making code clones, the process seems to be error-prone that introduces new bugs [28]. The reason is that after cloning a code fragment sometimes developers have to perform some modifications to adapt the code in their projects. Thus, the modification often leads to incomplete and inconsistent changes in the cloned code fragments [8].

(c) Difficult to Bug Resolving: When a bug is found within a code frag-

ment, all of its similar or cloned copies are need to be checked in order to resolve that bug. It seems the bug is already present at the time of reusing that code fragment [8].

(d) Software Size: Software size is also increased by code cloning. This is another issue that occurs for system and hardware constraints [29]. With the increasing size of the software, hardware should be upgraded [8].

(e) Lack of Inheritance: The tendency of code cloning may break design abstraction or indicate the lack of inheritance presented among the codes [8]. So, sometime it is difficult to reuse the parts of cloned code.

2.5 Applications of Code Clone Detection

There are many aspects for which clone detection are necessary. A few of those are described below.

(a) Detecting Plagiarism: The process of detecting a plagiarized portion of source code within the code repository is known as plagiarism detection [30] [31]. It is considered to be one of the major areas where clone detection tools and techniques can be used [8].

(b) Mining API or Library: Frequent usage of cloned files or large code fragments sometimes helps to mine potential candidates for forming reusable library or Application Program Interface (API) [32] [33] [8].

(c) Code Search: Researchers recently used clone detection in code search [34]. Real-time code clone detection is used to improve the performance of code search [35]. From the code search query, initially similar codes fragments are queried in order to improve the recall of code search performing clone detection [36] [8].

(d) License Violation and Copyright Infringement: Many proprietary companies have its own software copyright. Even many open source foundation

such as Apache has its copyright to prevent unauthorized copying of their source codes. Software copyright or license violation is one of the most popular usages of code clone detectors [37] [8].

(e) Reverse Engineering Product Line Architecture: Another modern use of clone detector is in the reverse engineering and product line architecture [8]. The main purpose is to find commonalities in the code repositories to reverse engineer the common features in the product and its derivatives. For example, to identify the issues of increasing fragmentation of Linux derivatives, fifteen clone detectors are used by the researchers in Linux code base to detect clones [38].

(f) Analyzing Software Provenance: The origin of the software components can be detected by code clone detection [39]. The provenance of software entities is recovered for technical and ethical concerns using the code clone detection in large software repositories [38].

(g) Analyzing Multi-version Program: Clone detection is one of the techniques to match program element across multiple version of the program. Thus, multi-version program analysis is carried out for analysis [40] [8].

(h) Understanding Program Comprehension: In order to understand program behavior, clone detection can also be useful. For example, when the behavior and functionality of a code fragment are known, it helps to comprehend the functionalities of other classes containing the cloned code of that fragment [41] [8].

2.6 Evaluation Metrics for Clone Detectors

Many metrics are used to evaluate clone detectors and also compare those detectors against each other. To measure the accuracy of a code clone detector, most popular metrics are *Precession* and *Recall* [8]. These metrics are calculated based on the presence of *true clone* and *false clone* in a code repository.

A *true clone* is a code fragment that indeed an exact copy or near copy of another code fragment. On the other hand, a *false clone* is a code fragment that is not the actual copy of another code fragment. Based on the presence of *true clone* and *false clone* how *Precession* and *Recall* are measured in the context of code clone detection, is mentioned below.

(a) Precision: *Precision* is the ratio of the candidate clones reported by a clone detector which are *true clones* not *false clones*. Equation 2.1 represents how precision is measured in the context of code clone detection. It is expected that clone detector should not detect instances of code blocks which are not clones [21].

$$Precision = \frac{true\ clones \cap detected\ clones}{detected\ clones} \tag{2.1}$$

(b) Recall: *Recall* is the ratio of the clones within a source repository that a detector is able to detect. How *Recall* is measured while evaluating clone detectors is represented by Equation 2.2. Clone detector should be able to detect most of the clones in a subject system [21].

$$Recall = \frac{true\ clones \cap detected\ clones}{true\ clones} \tag{2.2}$$

2.7 Source Code Processing

This section deals with various aspects of source code processing such as *Keyword Decomposition*, *Stop Word Removal*, *Stemming* and *Inverted Index Construction*. Before detecting clones these processes are performed on source code repository. For each process, a detailed description with examples is provided in the following subsections.

2.7.1 Keyword Decomposition

In order to name the source code entities such as package, interface, class, variable and method name etc. almost all Object Oriented Programming (OOP) languages follow recognized name conventions. The most widely used naming convention is CamelCase naming convention [41]. It refers to concatenating more than one words without space where the first letter of each word must be in uppercase form. This is also called Upper CamelCase form. Sometimes the first letter of the first word is written in lower case letter that makes a variation known as Lower CamelCase. However, in some cases, words are concatenating with a separator such as underscore, dot, dash etc.

Source code contains a collection of words in form of CamelCase naming convention, for example, a variable name may be *numberOfIteration*. To analyze the context of the code, keywords decomposition is used to split the words from Camel Case names. So, after spiting *"numberOfIteration"* by following CamalCase naming convention, three words are found such as *"number"*, *"of"* and *"iteration"*, To understand how keywords decomposition works, some examples of keyword decomposition is provided in Table 2.1.

Table 2.1: Example of Keyword Decomposition

Decomposition Based On	**Keywords**	**Processed Keywords**
CamelCase Letter	numberOfItems	number,of, items
Seperator Character	student_roll	student,roll
Number and Special Character Removal	getEmployee_1	get,employee

2.7.2 Stop Word Removal

All keywords, presented in the source code, are not equally important to analyze. This is because some words have no effects to understand the semantics of the source code. For example, almost all cases articles, prepositions and conjunctions, within a sentence, contain less semantic meanings. Since these words occur in all

source code documents, semantic differences among the source code documents cannot be differentiated. These types of words are known as *Stop Words* [42]. It refers a set of non-informative words that do not contain much information about the document. In order to analyze and distinguish the semantics of source code documents, it is necessary to remove those Stop Words.

Although Stop words can be language and task dependent, there is a set of common keywords that are considered as stop words in all scenarios. Some examples of semantically non-selective stop words are mentioned in Reuters-RCV1 [1]. Table 2.2 represents those stop words.

Table 2.2: Sample Stop Words from Reuters-RCV [1]

Stop Words							
a	an	and	are	as	at	by	for
from	has	he	in	is	it	its	of
on	that	the	to	was	were	will	with

2.7.3 Stemming

After keyword decomposition and stop words removal, significant keywords are found that contain semantic information for a code fragment. Due to grammatical issues, those keywords may appear in multiple forms, for example, *add, added, adding* etc. The semantics of a word with multiple forms is similar but multiple forms are used to represent distinguished contexts. It is useful to have all the keywords in the same forms while processing the code fragments.

To do so, an well-known text processing approach, *Stemming* is used [43]. Stemming is a process that reduces the inflected words into their root or base forms, sometimes known as stems. For example, after performing stemming *do, did, done, doing* are transformed into its root form *do*. As stemming is necessary to process and analyze words, many algorithms such as Lovins Stemmer [44], Paice Stemmer [45] and Porter Stemmer [46] [43] are developed to improve the

stemming process. Among these, Porter Steamer is the best well known in the research community as it shows good performance in many empirical studies [46]

While stemming, five sequential phases of word reduction are performed by Porter Steamer algorithm. In each phase, several rules are applied that reduced the longest matching suffix. For example, phase one contains four rules represented in Table 2.3 where the first rule shows that when a word contains suffix *"sses"* it will be transformed into *"ss"* by reducing the suffix. So, while steaming the word *"discusses"*, it is transformed into *"discuss"* according to the first rule. Some more examples are also shown in Table 2.3.

Table 2.3: Rules from Porter Stemmer First Phase

No	**Rules**	**Example**
1	$SSES \rightarrow SS$	$caresses \rightarrow caress$
2	$IES \rightarrow I$	$ponies \rightarrow poni$
3	$SS \rightarrow SS$	$caress \rightarrow caress$
4	$S \rightarrow$	$cats \rightarrow cats$

2.7.4 Index Construction

Once all the keywords and entities are extracted from the source codes, this information is required to store in a structured format for quickly retrieving. In order to do that many data processing approaches are developed in the literature. The most popular data processing technique, for faster retrieval, is constructing an inverted index.

2.7.4.1 Inverted Index

An inverted index comprises a set of unique documents where each document contains a set of keywords extracted from the code fragments. It refers that each code fragment is represented by a document which contains all unique keywords appeared in that code fragment. This index can be constructed in one sequential scan of the code fragments. Once the inverted index created, the query *'find those*

documents where keyword X is present' can now be answered in a single random access [8]. For example, it is similar to get the pages in a book by searching the index at the back of a book instead of looking up the words on each page of the book. Various approaches and libraries are developed to build an inverted index from documents. Among those approaches, Apache Lucene has been widely used text search library [47]. Apache Lucene, an open source full-text search library, provides faster search responses by constructing the inverted index. It is widely used in research community [48] because of its simple implementation and usages. Unlike other searching libraries, Lucene provides high performance as it first searches into the inverted index instead of searching the whole data directly. An overview of Lucene indexing is provided in Figure 2.1.

2.7.4.2 Inverted Index Construction

The core unit of an inverted index is document. A document consists of a set of unique fields. Each field has a name and value. First, the number of fields and the name of those fields are specified before creating document since this helps to identify the possible searching criteria [49]. For example, for a method block code fragment, a document can be created with many fields such as method identifier, return type, parameter type, keywords etc. For example, let Doc_1, Doc_2 and Doc_3 are documents in the document collections that are created from three method blocks. The descriptions of those documents are represented in

Figure 2.1. Overview of Lucene Indexing

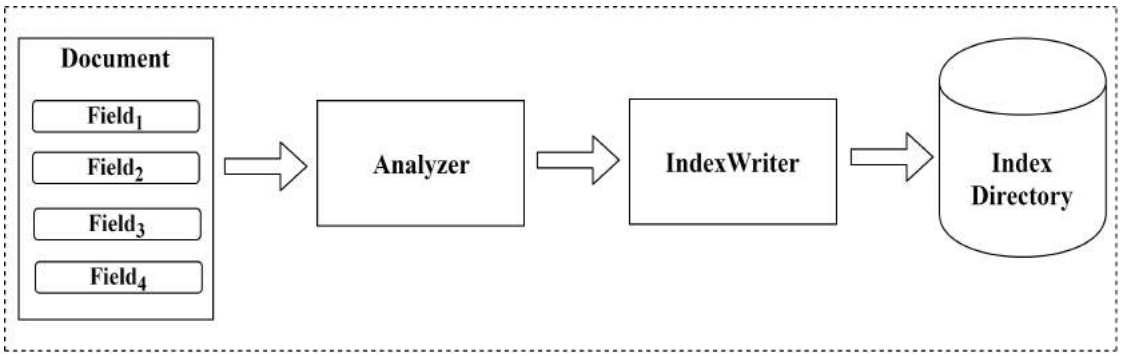


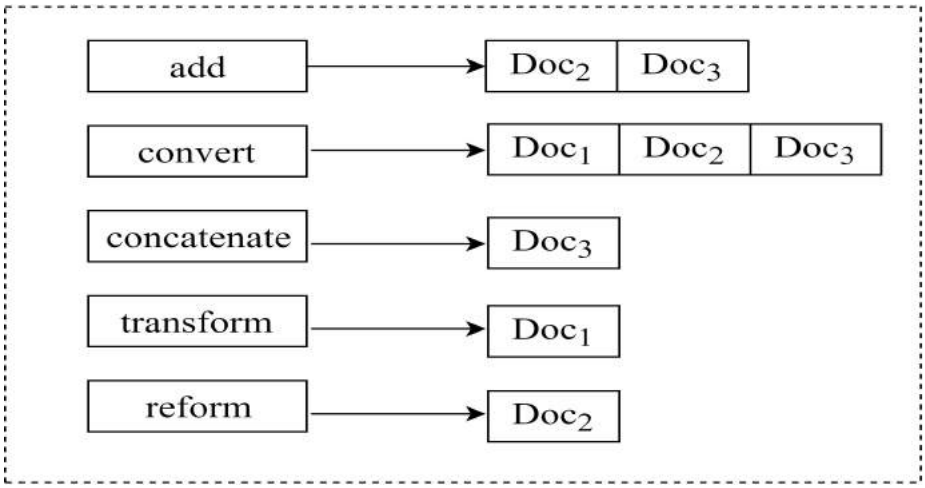
Table 2.4: Sample Documents

Documents	$field_1$	$field_2$	$field_3$	$field_4$
	method id	keywords	return type	parameter types
Doc_1	1	convert, transform	double	int
Doc_2	2	convert, reform, add	int	int, double
Doc_3	3	convert, concatenate, add	String	String, String

Table 2.4. Each document has four fields $field_1$ (method id), $field_2$ (Keywords), $field_3$ (return type), $field_4$ (parameter types). During indexing, Lucene first process the values associated with the fields and then constructs the inverted index is shown in Figure 2.2. Table 2.4 exhibits five unique values for the $field_2$ (Keywords) such as *convert, transform, reform, concatenate and add*. So, for the $field_2$ (Keywords), the inverted index contains five entries for mapping those three documents based on the $field_2$ (Keywords). Therefore, the inverted index maps the keyword *"convert"* to all the three documents Doc_1, Doc_2 and Doc_3, as all the documents contain this keyword in $field_2$. Similarly, keyword *"add"* is mapped against Doc_2 and Doc_3 as only those documents contain this keyword in $field_2$.

However, before constructing the index, all documents are analyzed by a module called *Analyzer* shown in Figure 2.1. After that, another module *IndexWritter* constructs the actual inverted index over all the documents by following the process mentioned above. *IndexWritter* also helps open and edit the index as required. Depending on the resource and usage, the index can be stored in the file system or main memory. After successfully creating the index, now searching can be done over the index. Using the previously defined fields, queries are created to search for related documents from the index. The search results provide a set of similar documents, those satisfy the given queries.

Figure 2.2. Inverted Index for Table 2.4 based on $field_2$ (keywords)



2.8 Summary

A brief description of code clone, its types are discussed in this chapter. It also deals with how and why code clones occur. The affects of code clones in software maintenance and development are also discussed. The evaluation metrics for code clone detector are also presented. Besides, some aspects of source code processing are also discussed at the end of this chapter. The next chapter exhibits the relevant existing works that have been performed in the literature.

Chapter 3

Literature Review

Code Clone is a way of code reusing having both positive and negative impacts on a software system. These impacts certainly make clone detection an active area of research in the software engineering research community. Over the last couple of years, many works have been performed on code clone. Among those, several studies have been found which proposed code clone detection techniques [9] having various categories such as Text-based [50] [51], Tree-based [28] [14] clone detection techniques etc. Empirical and comparative studies have also been conducted [9]. Besides, recent studies are also performed on developing benchmark and evaluation frameworks for evaluating the accuracy and the performance of code clone detection techniques. Code search tools and techniques are also strongly influenced by clone detection approaches [21] [22]. Considering all of those works, this chapter discusses significant code clone detection techniques and their drawbacks. As code search is related to clone detection, code search techniques are briefly discussed. The evaluation benchmark and framework for clone detectors are also elaborately described at the end of this chapter.

3.1 Code Clone Detection Techniques

Many clone detection approaches have been proposed throughout the last decade. Based on algorithm and source code representation, these approaches differ from each other. Roy et al. have performed a comprehensive survey focusing the strength and limitation of various clone detection approaches [9]. According to Roy et al. clone detection techniques can be categorized into various types such as Textual or String-based, Lexical or Token-based, Syntactic or Tree-based, Semantic-based or Program Dependency Graph (PDG) based and Metrics based techniques [15] [16]. A short description of each type approach is mentioned in the following Table 3.1. The following subsections also briefly describe those approaches.

Table 3.1: Clone Detection Tools and Techniques

Approach	Tool	Short Description
Text-based	Johnson [52] [53] [54]	Hashing of strings per line, then textual comparison
	DupLoc [50]	Hashing of strings per line, then visual comparison using dot plots
	NiCad [20] [11]	Syntactic pretty-printing with flexible code normalization and filtering, then textual comparison with thresholds
Token-based	CCFinder [13]	Token normalizations, then suffix-tree based search
	SourcererCC [18] [55]	Toke based large scale code clone detector with sub-block overlapping heuristic
	CloneWorks [19] [56]	Fast and flexible clone detector by following SourcererCC methodology
Tree-based	CloneDr [28]	Hashing of syntax trees and tree comparison
	Deckard [14]	Metrics for syntax trees and metric vector comparison with hashing
	JSCD [57]	Tree-based clone detector for JavaScript by following Deckard methodology
PDG-based	Komondoor [16]	Matching similar subgraphs in PDGs with program slicing comparison
	Duplix [15]	Approximative search for similar subgraphs in PDGs
Metrics-based	CloneD [17]	Hashing of syntax trees and tree comparison
	Kostas [58]	Metrics for syntax trees and metric vector comparison with hashing

3.2 Text-based Techniques

String-based techniques use the source code with little or no transformation and perform string matching algorithms for detecting clones. These techniques vary from one another based on the string matching algorithms and the granularity level of matching units. For example, Johnson [52], Duplo [50] and NiCad [20] are popular string or text-based clone detection tools. Each approach is also briefly discussed in the following subsections.

3.2.1 Johnson

Johnson is considered to be one of the oldest code clone detection techniques [53]. It is a simple string based clone detection technique that represents the source code as text without any modification while detecting clones. It performs clone detection in several steps. First, for each source file, special characters are discarded from the source code. A set of the sub-strings is generated to cover the source code with a group of lines. Next, those sub-strings that have the same sequence of characters and lines are identified. Those sequences are then hashed as a window. Then, a sliding window technique is applied which explores the sequence of lines with the same hash values incrementally. The detected clones have the same number of lines. So, in order to detect clones with different lengths, the sliding window technique is repeatedly performed with different lengths.

This technique shows good performance for detecting exact clones but is incapable of identifying gapped clones [54]. Besides, in the large code repository, sliding window and hashing based approach face memory and time complexity issues.

3.2.2 Duploc

Duploc is a textual clone detection technique proposed by Ducasse et al [50]. It is actually scattered or dot plot based clone detection technique. Scatter or dot plot is a two-dimensional chart where the axes represent the source entities. Firstly, source code is normalized by removing white-spaces, special characters, and comments. In this approach, the primary comparison entities are lines of the source code. Each line of the source code is hashed using a traditional hashing function. Two lines of source code are considered to be similar if those have same hash values. After that, for each pair of similar hash values, a dot is plotted in the coordinate *(x,y)* where x and y are represented equal hash values of two different lines of source code.

For a specific code fragment, the plotted dots make a straight line diagonally if all pairs of lines have equal hash values. Thus, this dot plot is useful for visualizing clones that represent by the diagonals of the plot. Interestingly, diagonals of dot plot with fewer gaps also represent the near mismatch clones such as Type-3 clones. Later this approach is automated by using dynamic string pattern matching. Further, this approach is also extended by Wettel et al. [51] in order to identify near mismatch clone by utilizing the dot plots. Their approach first identifies lines with same hash values, and then makes chain together for the neighboring lines for detecting Type-3 clones. Although this approach is effective for visualizing the detected clones, it fails to detect clones in a large code base. This is because, creating and comparing dot plots for each code fragment in a large code base are very time and memory consuming.

3.2.3 NiCad

NiCad [20] is a popular and modern code clone detection tool. It takes advantage of both text-based and tree-based approaches. Although apparently it is a hybrid

approach (including text and tree), it is considered as text/string-based approach. The reason is that it performs textual comparison while detecting the clones. The methodology of NiCad comprises three main steps such as parsing, normalization, and comparison. In the parsing step, source files are parsed to extract all code fragments with a predefined granularity such as block, function, or class. Each code fragments are then transformed into a pretty-printed standard form [11]. Comments are removed and line breaks, spaces are also normalized to make the code fragments as Type-1 clones with exposing textually identical fragments.

In the next step, extracted code fragments are normalized, abstracted and filtered to make an effective comparison. For example, code fragments are transformed by removal of the declaration, standard parametric forms and renaming. Finally, the extracted and normalized code fragments are compared line by line using Longest Common Sub-sequence (LCS) algorithm with an optimization to detect code clones. Various thresholds are applied in order to perform that comparison which makes the approach flexible. For example, a threshold of 0.0 is applicable for detecting exact clones, 0.1 threshold is used for those clones that differ by up to 10% of its normalized lines. Comparing to other tools NiCad creates clone classes as part of comparison instead of clustering individually. Besides, it only supports a few programming languages such as C, Java and C#.

Since string matching algorithms do not work fast for a large dataset, these techniques face time complexity issues. For example, NiCad uses LCS algorithm that can only compare two potential candidates at a time. Since each potential candidate clone needs to be compared with other candidates, making the comparisons using LCS is very expensive.

3.3 Token-based Techniques

Token-based techniques first transform the source code into a sequence of a lexical token with a predefined granularity level. Next, the comparison is made between the tokens. The transformation of the token is done by using a lexical analyzer to parse the source code, and rule-based transformation is applied to generate a stream of tokens. Most popular clone detectors, for example, CCFinder [13] and SourcererCC [18], CloneWorks [19] are developed based on token-based techniques. The following subsections briefly describe those techniques.

3.3.1 CCFinder

CCFinder is a token based clone detection approach developed by Kamiya et al. [13]. It takes source files as input and provides clone pairs as outputs. The whole clone detection process is comprised of four steps such as lexical analysis, transformation, match detection and formatting. In the first step, lines of the source files are divided into tokens corresponding to a lexical rule of its programming language. All the tokens are concatenated into a single token sequence. It helps to find clones in multiple files using the same way. Next, the white spaces, including new line, tab and comments are also removed. Then, in the transformation step, tokens are transformed such as added, removed, or changed based on the transformation rules. Various types of transformation are used in this process. For example, some rules are, removing the package name, removing initialization list, separating class definitions and removing accessibility keywords from the source codes etc.

Besides, identifiers of variables, types and constants are replaced with a specific token. This makes same code fragments to be a clone with different variable's and identifier's names such as renamed clones. Then matching is performed among the tokens using sub-string matching and an equivalent pair of tokens with the same

sub-string is reported as clones. Finally, in the formatting step, each location of clone pair is transformed into line numbers on the original source files. Being a token based approach, this approach detects clones with low computational complexity. However, due to the sub-string matching of tokens, it fails to detect near mismatch clones.

3.3.2 SourcererCC

SourcererCC is a large scale code clone detector proposed by Sajnani et al. [18]. It has two steps such as partial index creation and clone detection. In the partial index creation phase, the source files are first parsed into code fragments and then tokenized those code fragments. This tokenization is performed with a simple scanner that is aware of the token and the block semantics for a given language. Then, using these tokens an inverted index was built that maps the tokens to its code fragment. An inverted index is an index data structure sorting and mapping from content such as numbers, words and tokens to its location in a document or in a set of documents (in this case, code fragments). Comparing to other approaches, SourcererCC does not create an index of all tokens, instead it applies filtering heuristic called sub-block overlapping filtering to develop a partial index for only a subset of the tokens in each block.

Next, in the detection phase, SourcererCC [55] first iterates through all the code fragments and retrieves the candidate code fragments by querying into the partial index. Since partial index uses filtering heuristic, only those tokens within the sub-block are used for querying in the partial index. This process reduces the number of candidate code fragments for the comparison of the detection process. After retrieving all the candidate code fragments, another filtering heuristic namely token position filtering is applied for optimization. This filtering exploits the ordering of the tokens in code fragments to measure an upper and lower bound of similarity scores between the query and the candidate code fragments. The can-

didate code fragments those upper bounds fall below the similarity threshold are ignored without further processing. On the other hand, the candidates with lower bounds similarity threshold are accepted for matching. All processes are continued until all the code fragments are located. By following the above-mentioned processes it can only detect clones in few programming languages such as C, Java, C# and Pythons.

To use sub block overlapping heuristic, first it needs to compute a Global Term Position (GTP). GTP represents all the terms and its frequency presented in a code repository. Then while indexing the sub-block overlapping of code fragments, it sorts each code block by following the order of GTP. Even while detecting clones, it performs the sorting again for each code fragments. For a small and medium size code repository, GTP computing and sorting each code block have lower complexity. However, for large dataset, GTP computing and sorting for every code block take huge memory and overhead time. Because of this, it does not work faster in large code repositories.

3.3.3 CloneWorks

CloneWorks is another fast and flexible clone detection tool developed by Svajlenko et al. [19]. It comprises two components, for example, flexible input builder and clone detector. First source files are extracted by the input builders and transformed those into a set of term representations. It gives flexibility to the users for the processing of the source code including transformation, normalization, filtering applied at the code fragment in term level. The users are also allowed to plug in their own custom term processor. After processing the code fragments into terms, these term representation is fed into the clone detector.

Clone detector first creates an inverted index following the same process used in SourcererCC [18]. It applies sub-block overlapping heuristic while building the index. Then it retrieves all the code fragments from the index by querying only

the sub-block of the code fragments. Then, clone detection is performed using modified Jaccard similarity. This metric takes two code fragments and computes the minimum term intersection ratio. A pair of code fragments is reported as clone if those satisfy a minimum similarity threshold. Similar to SourcererCC, CloneWorks also performs the same GTP computing and sorting. Therefore, it shows the same complexity issues while GTP computing and sorting.

3.4 Tree-based Techniques

Tree-based techniques detect clones by finding similar sub-trees. In tree-based techniques, first, the source code is parsed into its Abstract Syntax Tree (AST) representation. Next, various graph or tree matching algorithms are used to detect similar sub-trees. Most popular tree-based approaches are CloneDr [28], Deckard [14], and JSCD [57] etc. A brief description of these approaches is given below.

3.4.1 CloneDr

CloneDr is considered to be the pioneering of tree-based clone detection tool proposed by Baxter et al. [28]. This approach works in three steps such as finding sub-tree clones, finding clone sequence and generalizing clones. In the first step, the source code is transformed into AST representation. Now for a specific AST of N nodes, finding sub-tree clones requires O (N^3) comparison. In case of a large software repository of M lines of code contains N=10*M AST nodes. The complexity of the computation is prohibitively large. To resolve this problem, a hashing approach is applied into the sub-trees of AST. First, the sub-trees are categorized with the same hash values. Now, comparisons are made only with those sub-trees that have same hash values.

In order to find clone sequences, a list structure is made. In this list structure, each item is related to a sequence of the structure. It stores the hash value of

each sub-tree element of the corresponding sequence. This list structure makes the process efficient to compute the hash code of any sub-sequence. Now each pair of the sub-tree that contains a sequence of nodes search for the maximum length of the possible sequence which encompassed a clone. After detecting all near-mismatch clones, clone generalization process is performed. It first visits the parent of already detected clones and checks whether the parents of that node is a near mismatch clone too. The advantage of using this technique is any near mismatch clones should be assembled from some set of the exact sub-clones. Therefore, no near-miss clones will be missed. However, as this approach compares AST of code fragments, it requires high memory and inefficient comparison to detect clones in lower complexity.

3.4.2 Deckard

Deckard [14] is one of the popular tree-based clone detection approaches. On the contemporary time, it introduced a novel approach for detecting code clones by performing efficient similar tree matching. The uniqueness of the approach is to compute a characteristic vector from the structural information of the source codes such as Abstract Syntax Tree (AST). It uses a hashing algorithm called Locality Sensitivity Hashing (LCS) to cluster the similar vectors that are reported as clones. The first step of this approach is an automatic selection of a parser that is generated from the language grammar. Then the parser transforms the source code into its AST representation. Syntactical information is captured from the AST that is used to produce a fixed dimension set of vectors. There remains a mapping between the source code, AST and the vectors. Finally, the vectors are clustered by computing the Euclidean distance among the vectors. The clusters of similar vectors are reported as clones. The computation of the characteristic vectors from the AST needs high memory. So, in large code repository Deckard [14] also faces memory consumption and huge execution time issues.

3.4.3 JSCD

Most of the code clone studies are actually conducted on the subject systems of statically typed languages such as Java, C# etc. Generally, the nature of both statically and dynamically typed languages influence the developers how they duplicate code. Unlike statically typed languages such as Java, dynamically typed languages, for example, JavaScript does not support any method overloading features. Besides, expression and declaration of functional statement do not specify parameter types and because of that those are less likely to be functional level clones. In spite of having various nature of statically and dynamically typed languages only a few studies have discussed with the code clone in dynamically typed languages. JavaScript-based code clone studies are limited. The reason is that most of the feature of JavaScript is extremely dynamic such code generation in run-time, having on module system, embedding inside HTML. So, detecting code clones in JavaScript application has many challenges.

JSCD [57] is a recent code clone detector that is capable of detecting code clones in JavaScript applications. This tool is a tree-based code clone detector inspired by Deckard [14]. Like Deckard [14], JSCD first parses the JavaScript source code into its AST representation. For parsing JavaScript files it uses SAFE [59] analysis frameworks that are specially designed for JavaScript. It is capable of parsing both standalone JavaScript file and embedded JavaScript inside the HTML file. Then from the approximate structure of the source code like AST, it identifies the characteristic vectors that are a fixed dimension integer vector. These integers represent the frequency of the nodes existed within the AST. Now, Locality Sensitive Hashing (LSH) is used to cluster similar vectors by their Euclidean distances into clones. Here LCS is used since it hashes two similar vectors to the same hash values with the high probability. On the other hand, it hashes two distant vectors to the same hash values with low probability. Finally, it detects clones those have similar hash values of the characteristic vectors.

While these techniques are precise, it does not always work faster as parse trees are rich in information and hence consume a high amount of memory. Apart, from Deckard and JSCD cannot detect intra-project and inter-project clones.

3.5 Program Dependency Graph Based Techniques

It is similar to tree-based or AST based techniques but the difference is that it represents the source code in the form of a Program Dependency Graph (PDG). It contains control flow and data flow information that conveys more semantic information rather than AST. So, PDG based techniques are more robust to detect gapped or updated clones. Komondoor et al. are the pioneer for applying PDG based techniques in code clone detection [15]. They represented a program source code into a dependency graph and use a variation of program slicing techniques that transformed the code clone detection problem to identify a subgraph problem over PDG. The usefulness of this approach is to detect clones that do not occur as contiguous text in the program such as non-contagious clone.

A similar approach is also followed by Krinke et al [16]. They first represented the source code into a grain program dependency graph and then tried to identify similar subgraph structures which are extracted from duplicated code. To present those duplicate codes to the user, a mapping was used to retrieve the code from the identified similar subgraphs. However, similar to AST based techniques, the above-discussed PDG based techniques do not work faster in large code repositories because of consuming a high amount of memory.

3.6 Metrics Based Techniques

In metrics based techniques, a set of metrics are gathered for code fragments. Then those metrics are compared for detecting clones instead of comparing code or AST. Generally, the source codes are transformed into AST or Control Flow Graph

(CFG) from which various metrics are identified such as cyclomatic complexity, function points, lines of code, etc. A study was performed by Mayrand at el. by applying the metrics-based approach in code clone detection [17]. First metrics are measured from names, layout, expressions, and control flow of functions. They calculated total twenty one metrics in each function unit of a program and detected functions with similar metric values as clones.

Two different approaches for detecting clones using metrics were proposed by Kontogiannis et al [58]. The first approach is to perform a direct comparison of metric values as a representative for similarity at the granularity of begin-end blocks. The other approach is based on dynamic programming that compares the begin-end blocks for each statement using minimum edit distance. Thus, pairs with a small edit distance are likely to be code clones. It is found that metrics-based approaches are more effective for detecting clones in source code with high-level granularity such as file level or class level. However, the drawback of metrics based approaches is that calculating those metrics are required extra efforts and time. Besides, comparing those metrics for all code fragments also occurs polynomial comparison.

3.7 Code Search

Code search refers to finding or retrieving code snippiest with a view to reusing code in software development and maintenance. According to a study, it is considered to be one of the most common practices to the software engineers during software development. Code search can be performed in two type of repositories such as local repository or in a large inter project code repository. Usually, developers search code in local code repository to identify or fix bugs, detect or refactor code smells etc. But developers search code in large code repository in order to reuse existing code or to know how a specific type of implementation is already

done. Thus, these reduce the time and efforts during development. With the increasing movement of open source code, code search has become more popular as software development practices. Existing code search techniques include Keyword Based Code Search (KBCS) [60], Semantic Based Code Search (SBCS) [61], Test Driven Code Search (TDCS) [62] and Interface Driven Code Search (IDCS) [63]. Among these techniques, Interface Driven Code Search is closely related to our works since IDCS retrieves similar code fragments based on its interface. A brief description of those techniques is mentioned below.

3.7.1 Keyword Based Code Search (KBCS)

KBCS helps developers to retrieve reusable code fragments that are related some specific keywords. In KBCS, the source codes and its meta information such as source file name, comments inside the codes, commit messages etc. are processed as plain text document. From this text, keywords are extracted and index those keywords following the traditional information retrial approaches. Then, relevant code fragments are retried from a code repository by querying over the previously built index. JSearch [64], Thesaurus Based Query Expansion, Sourcerer [65], is a significant work related to KBCS.

3.7.2 Semantic Based Code Search (SBCS)

As open source software development is getting popular, it contributes a significant amount of code available on the web. So, before writing new codes it is useful to search the codes that developers want to implement. This is because perhaps that implementation has already done by other. But, the implementation found by code searching, often does not satisfy developer's needs. Developers need to modify that implementation according to their project requirements. It is so difficult and tedious to understating others implementation and adapting that implementation with proper modification. Therefore, the semantic of the source code should be

matched with developer code search query [61]. This helps developers understand the context and semantic of the retrieved code and required minor modification to adapt those retrieved codes. Various code search techniques are proposed to search semantically relevant code fragments from a large code repositories.

3.7.3 Test Driven Code Search (TDCS)

Sometimes developers are much interested in the behavior of the desired code fragments. The behavior of a code fragment exhibits when it is compiled and run against various inputs. In this case, test cases can be used to identify the behavior in the local context for that code fragment [62] because test suit specifies the dynamic behaviors of the code fragments. Thus, in TDCS test cases are used as search interface and retrieve the codes fragments by identifying their dynamic behaviors through running the code fragments against the given test cases.

3.7.4 Interface Driven Code Search (IDCS)

In order to extend code search techniques, researchers have proposed a sophisticated approach called Interface Driven Code Search (IDCS) [63]. It allows users to search code in code libraries, by using interface information. IDCS first crawls all the methods from the code libraries and extracts interfaces for indexing similar methods. It can be performed by using existing code search tools such as Sourcerer [65]. However, while indexing, ICDS cannot select proper terms for similar codes with the analogous functionality.

Recently a study [66] has improved IDCS performance by indexing similar methods under appropriate terms. The study has shown that if two methods contain similar return types and parameters, most of the time those perform similar functionality omitting the keyword they contain in method names. However, IDCS differs clone detection since it only finds relevant code based on an interface not the method clones.

3.8 Exploratory Study

In large code repositories, code clone occurs because of code reusing. Apart from code clones, interface and functional repetition can be possible. In order to identify the effects of these repetitions, two exploratory studies have been performed. Details of those studies are described in the following subsections.

3.8.1 Interface Redundancy

Interface Redundancy (IR) represents the repetition of whole method interface, for example, return type, method name, and parameters types across the software corpus. Paula et al. have first introduced it in an exploratory study [7]. Their study is much focused on exploring the impacts and effects of IR in code search. They extracted 380,000 methods from a code corpus called SF100 in a relational database by Sourcerer [65]. To identify redundant interfaces, they have performed IDCS on the databse in two steps. First, IDCS is performed without query expansion. Then, again it is performed with Automatic Query Expansion (AQE) [67] to overcome the keyword related vocabulary mismatch and different thesaurus [68] problems, for example, types of lists contained within the Java API. The results have shown that 80% project of the targeted repositories contain redundant interfaces. Besides, it is found that the knowledge of redundant interface in code repositories improves the performance of code search especially in IDCS. Additionally, it is observed that IR has diverged from traditional code cloning since in their study only 0.002% IR is related to method clones [7].

Although, IR related study is similar to this interface similarity study, it differs from many aspects. In IR based study, researchers only considered the public interfaces only in a medium subject system.The relationship between IR and code clones is not clearly discussed.

3.8.2 Functional Redundancy

In large code repository, the probability of reappearing a function in multiple projects is high [69]. For recent code reuse and repair, this type of Functional Redundancy (FR) is expected. FR is considered to be important since it represents the amount of code rewritten across the project either intentionally or unintentionally. Although Interface Redundancy exhibits the repetition of the whole interface, it still cannot guarantee that two functions with similar interface express analogous behavior. So, in order to measure the degree for repetition of functional behavior, researchers performed another exploratory study.

First, method pairs are collected from a code repository based on IR. Then, it is checked whether each pair of method perform similar semantic behaviors that refers whether those pair of method are FR or only IR. For each method, they generate random input according to their parameter types. Then both the methods from a pair are run against the same inputs. By analyzing and comparing the outputs, they identify the semantic of those methods. Thus, they took decision whether those methods express similar behaviors or not. This is named as method profiling.

Experimental results show that from 68 Java projects, researchers found almost 1000 pair of methods. 41.17% projects contain functional redundancy. Besides, 93% of the method pairs are executed successfully while method profiling. Most of the methods with functional redundancy are diverse from textual code clones.

3.9 Benchmark and Evaluation Framework

Many code clone detection tools and approaches have been developed in the last decade. Software maintenance and evaluation have ameliorated by these tools through detecting and tracking code clones. Before using these tools, evaluating the performance of these tools is very important since the evaluation expose the

capability of these tools. Generally the performance of tools is evaluated by a metrics called recall as it shows the ratio of number of detected clones and actual clones that a tool is able to detect. So, to measure the recall of clone detectors, various benchmark are developed. A brief description of those benchmarks and frameworks is mentioned below.

3.9.1 Bellons Framework

To compare the evaluation performance of six clone detection tools, an experiment was conducted by Bellon et al [70]. First clones are reported by the six clone detector with a minimum 6 lines configuration parameters in a corpus that contained C and Java programs varied from 11K SLOC to 235K SLOC. Each clone pair reported by the tools is called candidate clone pair. To evaluate the performance of these tools, they needed to decide whether these candidate clone pairs are actual clones or not. So, they should have known all reference clone pairs that are actually existed within the corpus. The traditional way of making such reference clones is making a union of reported candidate clones from various clone detector or making the intersection of candidate clone pairs reported by the detectors.

However, these approaches have deficiencies as using the candidate clones for making the reference clones, the clone detectors should have required precision 1. Instead of following this way, Bellon et al [71] manually made the reference clone pairs that known as Bellons Framework. They selected 2% candidate clone pairs from total 325,935 clone pairs that are reported by all the detectors. To make the selection process unbiased, they followed an automatic process to make sure all tools have equal candidate clones within the selected 2% clones. After that, they classified the selected clone pairs and manually injected those clone into the corpus. It took in total 77 hours to manually classify those clones. These injected clones are not disclosed to the clone detectors and the detectors were run within

Table 3.2: Symbol and Description of Mutation Operators

Name	**Mutation Description**	**Clone Type**
mCW	Changes in whitespace	Type-I
mCC	Changes in comments	Type-I
mCF	Changes in formatting	Type-I
mSRI	Systematic renaming of identifiers	Type-II
mARI	Arbitrary renaming of identifiers	Type-II
mRPE	Replacement of identifiers with expressions	Type-II
mSIL	Small insertions within a line	Type-III
mSDL	Small deletions within a line	Type-III
mILs	Insertions of one or more lines	Type-III
mDLs	Deletions of one or more lines	Type-III
mMLs	Modifications of whole line(s)	
mRDs	Reordering of declaration statements	Type-III
mROS	Reordering of other statements	Type-IV
mCR	Replacing one type of control by another	Type-IV

the injected corpus. Clones reported by the tools are checked whether those tools are capable of detecting those injected clones. Thus, it helped them to evaluate the recall of those detectors more efficiently.

3.9.2 Mutation Injection Framework

Manual classification and injection of reference clones within a large corpus is tedious and time-consuming that makes it very difficult. On that contemporary time no studies were found that report the reliability of judges who verified the detected candidate clones. No studies provided the report of recall and precision of the detectors separately for various types of clones. To ameliorate this problem a mutation based framework was developed by Chanchal K et al. [72]. This framework is able to measure and compare recall and precision of the detectors automatically and efficiently for all types of clones. The framework is developed in two stages first a list of mutation operators are defined for all type of clones. Next the evaluation framework is built for tool comparison.

Mutation Operators: For each type of fine grain clones Chanchal K et al. [73] defined some specific types of mutation operators. By injecting these mutation operators, only a certain type of artificial clones can be created. A complete list of the mutation operators is shown in the Table 2.2.

Evaluation Framework: It contains two phases such as Generation phase and Evaluation Phase. In the Generation phase artificial clones are created using the mutation analysis. Then these artificial clones are injected into the code base by following the automatic process. Next in the evaluation phase it is determined how efficiently the known clone pairs are detected by the tools.

3.9.3 BigCloneBench

Previously, most of the clone detectors were evaluated on small code repositories. In that time, the common benchmark was to identify clones using those clone detectors that performed best. Then, it was compared with the output of the newly proposed detector. Thus, the performance of newly proposed detector was measured [21]. The problem of this approach is that the benchmark only includes those clones, the participating tools are able to detect. Therefore, it builds a benchmark that gives unfair advantages over other tools those did not contribute to the benchmark. Recently, clone detection is performed on large code repositories. So, evaluating a clone detectors on large code repositories is a challenging task since the absence of a complete evaluation benchmark and framework for big data clone detection approach.

To resolve this problem a big data clone benchmark called BigCloneBench is developed by the researchers [74]. BigCloneBench is comprised of known true and false clones mined from the big data inter-project repository IJaDataset 2.0 [1]. IJaDataset 2.0 dataset contains 24,558 open source Java projects, crawled from Google Code, GitHub and SourceForge, as subject systems with 365MLOC. In-

[1]IJaDataset 2.0, http://secold.org/ projects/seclone

stead of using any clone detectors, researcher performed mining on IJaDataset for identifying clones of frequently implemented functionalities. First, various searching heuristics are applied to mine code fragments in IJaDataset which would implement targeted functionalities. Then the candidate clone fragments are then manually tagged as true or false positive clones of the targeted functionalities by the judges. After the tagging process, the benchmark is populated with the true and false clones identified in the tagging process. Each clone is classified into its type and the syntactic similarity is measured. The current version of BigCloneBench contains 47 tagged functionalities and 8 million clones with 6 millions true positive and 2 millions false positive clones.

3.9.4 BigCloneEval

BigCloneBench is capable of measuring the recall of clone detectors. However, the comparison of recall among the clone detectors is also required when it is needed to evaluate the performance of newly proposed clone detectors against the existing detectors. In this case BigCloneBench is not accessible enough to make the comparison of recall among the clone detectors.

In order to make BigCloneBench more flexible researchers have developed a framework called BigCloneEval [22] for evaluating clone detection tools using BigCloneBench. BigCloneEval makes easy to evaluate the recall of clone detection with the recall of other tools. Like BigCloneBench, BigCloneEval also includes the IJDataset 2.0 repository that contains 24,558 open source Java projects with 3 million source files and 365MLOC. All types of clones such as Type-1 (T1), Type-2 (T2) , Type-3 (T3) and Type-4 (T4) are present in BigCloneEval. However, there is no agreement on, when a clone is no longer syntactically similar. As a result, it is difficult to separate T3 and T4 clones [21]. So, researchers categorized IJDataset's T3 clones into various catagories such as Very Strongly Type-3 (VST3) and Strongly Type-3 (ST3) based on the syntactic similarity. VST3 clones contain

90% syntactic similarity and ST3 clones have 70-90% syntactical similarity.

To use BigCloneEval, users just have to set some configuration parameters instead of writing any code. The execution of the candidate clone detector for IJaDataSet is maintained by BigCloneEval. It also manages the possible scalability constraint of the tools using deterministic input partitioning. It tracks the detected clones and efficiently determines which of the reference clones in BigCloneBench the tool was able to detect. It also produces tool evaluation report that summarizes recall per clone type including inter-project, intra-project, Type-1, Type-2, etc. Besides, users are also allowed to specify their own clone detection algorithm with a plugged in architecture.

3.10 Summary

Clone detection helps the developers to maintain software quality, prevent bug propagation, late propagation, clone tacking etc. Many code clone related tools, techniques and the benchmark of evaluation are proposed in the literature, for example, Text based, Tree based token based techniques etc. The summary of state of art works such as clone detection approaches, code search techniques and evaluation frameworks and benchmark, are discussed in this chapter. However, these research works do not analyze the effects and impact of interface similarities in code clones. The following chapter discusses the relationship between interface similarity and code clones.

Chapter 4

Relationship of Interface Similarities in Code Clones

Code clone is one of the most popular code reusing techniques where similar pieces of code are replicated within or between code repositories. Interface similarity is a kind of replication that refers to the similarity of method names, return types and parameter types which repeat across the code repositories. Two methods with similar interfaces are prone to be cloned if those perform analogous functions either entirely or at least partially [7]. An exploratory study is performed in this chapter, to explore the relationship and effects of interface similarity in code clones. It is investigated whether interface similarity can be helpful for code clone detection. Firstly, cloned methods are detected in code repositories. Then, interface information is extracted from source code and several interface similarities are measured using this. The experimental corpus contains three different types of code repositories with 35, 109 and 24,558 Java projects respectively. The detected clone pairs in three code repositories are, on average 231411, 242992 and 130226 respectively. Promising results are found as it shows on average 79.65% intra-project and 69.44% inter-project clones containing similar interfaces. Besides, the average similarity of interfaces in Type-1, Type-2 and Type-3 clones are

100%, 69.35% and 67.34% respectively. These results prove the strong relationship between code clones and interface similarities. Next, this relationship is applied to designing and developing interface driven clone detection tools.

4.1 Introduction

Code clones are pairs of code fragments that are identical to each other and replicated within or between the code repositories. It occurs when developers reuse code through copying and pasting, automatic code generation or plagiarism with or without modifications [2]. Another common replication that appears in large software repositories is method interface. It refers to the method's return type, method names and parameter types that repeat exactly or similarly across the code repositories. Such an example is provided with Listing 4.1 and Listing 4.2. It represents two code fragments, A[1] and B[2] extracted as a clone pair from two different Java projects *berkeleyparser* and *stanfordnlp* respectively. It is observed that same code fragments are repeated into two different projects and those two code fragments also contain similar interfaces such as return type (*String*), parameter types (*String*) and method names (*doubleArrayToFloatArray*). It shows both types of replication such as repetition of method interface and clones occurred in those code repositories during development.

Besides, it is a hypothesis that if two methods contain similar interfaces, it is very likely those perform same functionalities either entirely or at least partially[7]. When those methods contain the same interface and perform similar functionalities, it indicates that these methods should be semantic or syntactic code clone to each other [7].

[1]berkeleyparser/blob/master/src/edu/berkeley/nlp/util/ArrayUtil.java
[2]stanfordnlp/CoreNLP/blob/master/src/edu/stanford/nlp/math/ArrayMath.java

Listing 4.1: Code Fragment (A)

```
1  public static float[][] doubleArrayToFloatArray(double a[][]) {
2    float[][] result = new float[a.length][];
3    for (int i = 0; i < a.length; i++) {
4      result[i] = new float[a[i].length];
5      for (int j = 0; j < a[i].length; j++) {
6        result[i][j] = (float) a[i][j];
7      }
8    }
9    return result;
10 }
```

Listing 4.2: Code Fragment (B)

```
1  public static float[][] doubleArrayToFloatArray(double[][] a) {
2    float[][] result = new float[a.length][];
3    for (int i = 0; i < a.length; i++) {
4      result[i] = new float[a[i].length];
5      for (int j = 0; j < a[i].length; j++) {
6        result[i][j] = (float) a[i][j];
7      }
8    }
9    return result;
10 }
```

For establishing the relationship between interface and clones, it is required to verify the above mentioned hypothesis. However, it is very difficult since various types of clones and interface similarities can be possible. It is also a challenging task to measure those similarities in different types of code clones such as intra-project, inter-project, Type-1, Type-2 and Type-3 [7]. So, if interface similarity is

significantly related to code clone, this may be helpful for clone detection, tracking and management.

Throughout the last decade, various clone detection techniques are proposed. According to Roy et al. [9] clone detection techniques, using various representation, includes Textual [20], AST based [14], Token based [18] and Dependency graph based [15] etc. Although these tools are focused on clone detection, the effect and relationship of interfaces in code clones have never been analyzed. Several code search techniques such as Keyword Based Code Search (KBCS) [60] and Interface Driven Code Search (IDCS) [63] provide code searching with method interface information but never investigated the relationship between code clones and interfaces. A detailed description of those tools and approaches is already provided in the previous chapter. However, a recent study [7] has discussed the effect of Interface Redundancy (IR) in code search but has not clearly mentioned the effect and impact of interface similarities in different types of code clones.

To explore the relationship and effects of interfaces similarity in code clones, an exploratory study is conducted on three subject systems categorized as *Small, Medium and Large* based on the number of projects. The subject systems contain 35, 109 and 24,558 open source Java projects crawled from SourgeForge, GitHub, Google Code etc. Firstly, various types of method clone lists are detected by using two prominent clone detection tools NiCad [20] and SourcererCC [18]. After that, interface information, for example, return type, method name, parameter types is extracted for each method clone from its source code. For every interface, keywords and related words are also identified by extracting its method name. Finally, using this extracted information various types of interface similarities are found by satisfying seven proposed similarity conditions. The percentage of how many clones satisfy each condition is measured. These results are used for establishing the desired relationship. More specifically, the aim of this study is to seek the answer of the $\mathbf{RQ_1}$ posted in Chapter 1. To investigate the relationship of

interface similarities in code clones, the $\mathbf{RQ_1}$ is substituted in the following three Sub-Research Questions (SRQ).

$\mathbf{SRQ_1}$:What percentage of interface similarities occur in intra-project and inter-project method clones with similarity combinations?

$\mathbf{SRQ_2}$: Are the intensities of interface similarity different in various types of clones and which clone-type(s) have higher possibilities to be detected by using interface similarity?

$\mathbf{SRQ_3}$: How does interface similarity relate to code clone detection? More specifically, how many code clones occur due to interface similarity?

The following subsections of this chapter provide an exhaustive description of study design, experimental dataset and answering the sub research questions through result analysis.

4.2 Overview of Study Design

The aim of this study is to investigate the relationship, impact and effect of interfaces similarity in code clones. To do so, three types of subject systems are chosen such as *Small, Medium* and *Large* as the experimental dataset. Next, two near accurate clone detection tools SourcererCC [18] and NiCad [20] are used to identify all types of clones in those subject systems. While detecting the clones the granularity of those detection tools was set to method level because it is needed to get those clones that are methods and contain complete interfaces. Next, For each clone, interface information is extracted from its source code. The overview of this study design is depicted in Figure 4.1. However, a detailed description of the experimental dataset, several types of clone detection processes and details of interface extraction from those clones are described in the following sections.

4.2.1 Experimental Dataset Selection

Based on the number of projects, the experimental dataset of this study is categorized into three types such as *Small*, *Medium* and *Large* subject system. A brief description of each type is described below.

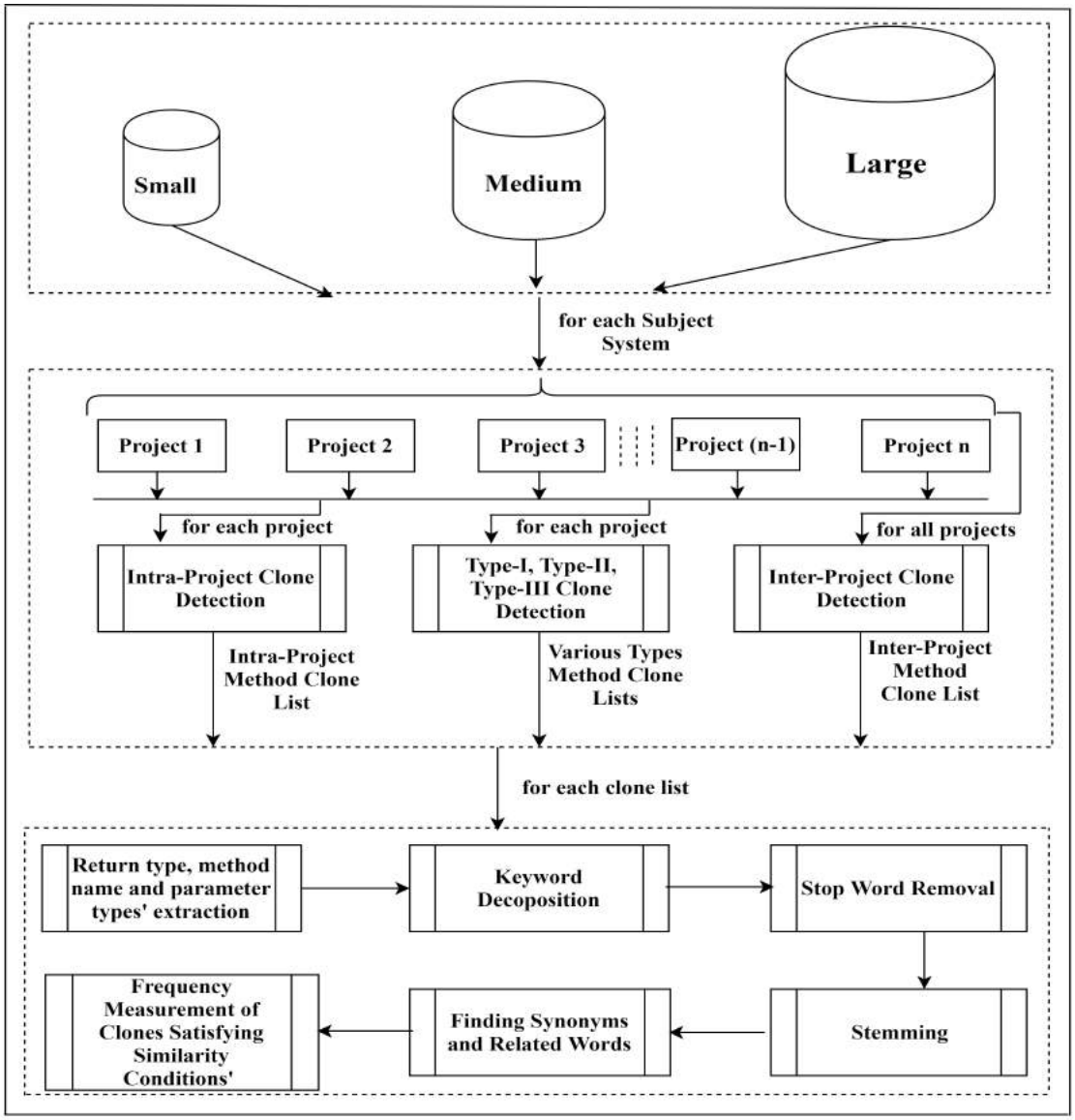


Figure 4.1. Overview of Study Design

4.2.1.1 Small Subject System

Popular 35 open source Apache Java projects are selected as *Small* subject system. These projects vary based on their size and functionalities. Those function-

alities include natural language processing libraries, network systems, distributed and server systems, machine learning, search and database systems, etc. These projects are considered to be highly used and popular in their respective domains. Therefore, it represents a dataset, with various sizes and domains, which makes this study unbiased towards any particular software system. These projects are also used in several code clone related studies [18] [75]. Table 4.1 represents the summary of *Small* subject system. However, a detailed description of those projects are available in this link[3].

4.2.1.2 Medium Subject System

To develop a statistically sound test data generation approach, Fraser and Arcuri [76] have arbitrarily chosen 100 open source Java projects from SourceForge[4]. This is because, most of the novel test data generation approaches are proposed without proper empirical studies. As a result, case studies on a specific topic are susceptible to be either small or biased to a specific kind of software system [76]. Their approach results in a benchmark called SF100 as a representative of the open source projects. This benchmark was previously used in several studies [68] [77]. Later, it has been extended by adding 10 most downloaded projects from SourceForge. In this study, the extended version of SF100 benchmark called SF110 [78] is used as a *Medium* subject system. However, before using SF110, a project namely *Liferay Portal* has been removed from it. This is because that project, containing 8,335 Java files and 1,552,597 LOC, creates extreme value problem for statistical analysis. The summary of the *Medium* subject system is represented in Table 4.1. Moreover, detailed description of SF110 dataset is also publicly available [5].

[3]https://projects.apache.org/
[4]http://sourceforge.net
[5]http://www.evosuite.org/experimental-data/sf110/

Table 4.1: Summary of Subject Systems

Features	Subject System		
	Small	Medium	Large
Name	Apache	SF110	IJaDataset 2.0
Java Projects	35	109	24,558
Java Files	13,122	19,492	2,078,126
LOC	1,711,237	3,630,723	300,000,000

4.2.1.3 Large Subject System

IJaDataset-2.0[6] provides a large Java inter-project source code repository that covers almost 24k projects crawled from GitHub, GoogleCode and SourceForge. In many earlier studies, it has been used in scalability testing of clone detection tools such as SourcererCC [18] and CloneWorks [19]. Besides, IJaDataset-2.0 dataset is also used to create a clone detection benchmark called BigCloneBench [21]. Later, based on BigCloneBench [21] researchers developed a clone detection evaluation framework namely BigCloneEval [22]. Here the actual IJaDataset-2.0 is modified for making it compatible with BigCloneEval [22] that has been used by clone detection tools for accuracy and scalability evaluation such as SourcererCC [18]. In this study, the modified version of IJaDataset-2.0 and the clone evaluation framework BigCloneEval [22] are used as a *Large* subject system. The summary of the *Large* subject system is listed in Table 4.1. A detailed description of IJaDataset-2.0 is provided with this link[7].

4.2.2 Code Clone Detection

To detect code clones in the above described subject systems, two popular token-based clone detection tools SourcererCC [18] and NiCad [20] are used. The reasons for using token-based techniques are that these tools give high recall in comparison to any other existing tools and are scalable for large code repository. In this

[6]IJaDataset 2.0, http://secold.org/ projects/seclone
[7]https://github.com/clonebench/BigCloneBench

experiment, SourcererCC [18] has been used for detecting intra-project and inter-project clones. On the other hand, various types such as Type-1, Type-2, and Type-3 clones are identified by NiCad [20]. Clone detection process is performed for each type of subject system according to the following phases.

Configuration Setup: In the beginning, similarity threshold for both tools is set to 80% with method level granularity. This configuration has been set to get the maximum number of true clones. This is because with this configuration, SourcererCC [18] and NiCad [20] achieved maximum recall and precision in several studies [18], [7]. Besides, the minimum lines a clone may contain is set to 6. The reason is that it helps to detect only those clones that have a concrete implementation. This configuration ignores the `getter` and `setter` methods of Java class attributes and also omits the abstract methods presented in the Java interfaces.

Intra-project Clone Detection: Next, with the above configuration SourcererCC [18] is run on each project of a subject system. It provides a list of intra-project method clones for every project of that subject system.

Inter-project Clone Detection: After that, inter-project clones are also detected by running SourcererCC [18] across all the projects of that subject system with the same configuration. However, SourcererCC [18] does not provide inter-project clones separately and includes intra-project clones. So, each clone that occurs in intra-project clone list is excluded from the list. As a result, it provides a list of inter-project method clones.

Type-1, Type-2 and Type-3 Clone Detection: Finally, for every project of the subject system, NiCad [20] is run with the same configuration to identify various types of clones. However, before using NiCad [20], it is a concerned that

Table 4.2: Summary of Detected Clones

Clone Type	Subject System			Total
	Small	Medium	Large	
Intra-project	55,095	82,264	47,847	185,206
Inter-project	2,352	20,342	17,264	39,958
Type-1	9,099	4,826	47,567	61,492
Type-2	53,459	45,980	4,233	103,672
Type-3	111,406	89,580	13,315	214,301
Total	231,411	242,992	130,226	604,629

NiCad [20] generates Type-2 clones including Type-1 clones. It detects Type-3 clones containing both Type-1 and Type-2 clones.

In this study, it is required to identify each type of clone separately. So, the Type-2 clones which are exactly matched with Type-1 method clones are excluded from Type-2 clones. Similarly, redundant Type-1 and Type-2 clones are excluded from Type-3 clones. Eventually, it results in method clones of Type-1, Type-2 and Type-3 for each project of that subject system. A complete list for the number of detected clones is shown in Table 4.2.

4.2.3 Interface Extraction

For every method clone, the source code of those methods are collected from their respective projects. After that, the source codes of each method are transformed into its Abstract Syntax Tree (AST) representation by using *Eclipse's ASTParser* [8]. Next, interface information such as return type, method name and parameter types are extracted by traversing AST nodes. In a method clone, each method name may contain single or multiple keywords that describe its functionality. For further analysis it needs to extract those keywords. This keyword extraction process contains multiple steps such as keyword decomposition, stop word removal and stemming. These topics are also elaborately discussed in Chapter 2. A short description of each step is mentioned below in the context of this experiment.

[8]https://github.com/eclipse/eclipse.jdt.core

Keyword Decomposition: Programming languages have their own naming conventions to represent the source code entities such as method names, variable names etc. As an Object Oriented Programming Language CamelCase naming convention is used while developing Java projects. Since all the projects used in the study are developed in Java, these keywords are also extracted by following Java *CamelCase*[9] method naming convention. Example and complete description of keywords decomposition are mentioned in Chapter 2.

Stop Word Removal: In this step, irrelevant English and Java stop-words like "and", "any", "but", etc. are removed from the method name. It is performed by using a standard stop-word list[10]. It provides extracted keywords into the next step. A detailed description of stop word removal is provided in Chapter 2.

Stemming: This step is responsible for transforming the keywords into its root form that is known as stemming. Stemming can be done based on various algorithms and tools. Here, *Porter Stemmer* is used. An example of the stemming process is to stem keyword "running" to root word "run". An exhaustive description on Stemming is mentioned in Chapter 2.

Finding Synonyms and Related Words: For each root word, synonyms are identified by using the MIT Java Wordnet Interface JWI [79]. JWI also provides some related word phrases for each root word. Those phrases are also merged with the synonym words. Since the root words are extracted from method names, the synonyms and related words are collected based on the verb form of those root words. Finally, for each method clone pair, three types of information such as return type, parameter types, set of keywords including synonyms and related words are retrieved. Various types of interface similarity conditions are constructed with the combination of interface information. Since three types of interface information such as return type, parameter types and keywords from method name, are

[9]https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html
[10]http://www.ranks.nl/stopwords

available for a pair of clones, it is possible for occurring (2^3-1)=7 types of similarity combinations for each pair of clone. For example, one of the conditions is that two interfaces are similar if their return types are similar. Each condition is represented by a unique identifier such as S_1 represents the condition of return type similarity of two methods in a method clone. A complete list of seven types of similarity conditions is mentioned in Table 4.3.

Table 4.3. List of Seven Interface Similarity Conditions

Identifier	Similarity Conditions
S_1	Return types are same
S_2	Types of parameters are same (At least one)
S_3	Keywords are similar (At least one)
S_4	Return types and types of parameters are same ($S_1 \cap S_2$)
S_5	Return types and keywords are similar ($S_1 \cap S_3$)
S_6	Keywords and types of parameters are similar ($S_2 \cap S_3$)
S_7	Return types, keywords and types of parameters are similar ($S_1 \cap S_2 \cap S_3$)

4.2.4 Frequency Measurement

First, frequencies of how many clones satisfy each similarity condition are measured. Since the relationship between code clones and interface is more likely to establish the relationship between two categorical variables, those frequencies are used to construct marginal and conditional distribution between clone and interface similarities. For example, in the *Small* subject system for a single project, detected clone lists such as intra-project, inter-project, Type-1, Type-2 and Type-3 are taken. Then, for each clone list, how many clones satisfy each similarity condition are counted and obtained the frequency to make the marginal distribution for each type of similarity condition. Then using the marginal distribution, the conditional distribution of *clone given interface similarities* is built for each type of clones. This process is repeated for both *Medium* and *Large* subject system. However, it is noted that the number and types of parameters received by

a method are more important than the order in which those have appeared. So, while satisfying the similarity conditions associated with parameters, the ordering of those parameters are ignored. Here, interfaces `int read(String, int, int)` and `int read(int, int, String)` are considered to be similar based on the number of parameters and types ignoring the ordering of those parameters.

4.3 Study Result

In this section, the experimental result analysis of code clone and its relationship with interfaces are presented. The results are found by satisfying seven similarity conditions such as S_1, S_2, S_3, S_4, S_5, S_6 and S_7. So, the answers of those sub research questions (mentioned in Section 4.1) are drawn based on the fulfilling of those similarity conditions.

SRQ_1: What percentage of interface similarities occur in intra-project and inter-project method clones with similarity combinations?

To answer this question, the results are presented with respect to intra-project and inter-project method clones. The numbers of intra-project clones found in *Small*, *Medium* and *Large* subject system are 55,095, 82,264 and 47,847 respectively that amounts to a total of 185,206 method clones. In the *Small* subject system, all 35 projects contain a considerable number of clones. However, in *Medium* and *Large* subject system only 96 and 1,841 projects include method clones. This is because in those subject system, most of the cloned code fragments do not satisfy at least 80% similarity threshold. Moreover, some cloned code fragments contain less than six line of codes that violate the minimum 6 line configuration setup.

On the other hand, the number of detected inter-project clones in *Small*, *Medium* and *Large* subject system are 2,352, 20,342 and 17,264 respectively. It

amounts to a total of 39,958 inter-project method clones. The frequency of interface similarities in intra-project and inter-project clones in each subject system is shown in Table 4.4 and Table 4.5

Table 4.4. Interface Similarities in Intra-Project Clones

Similarity Conditions	Subject System			Average(%)
	Small	Medium	Large	
S_1	87.13%	88.82%	99.28%	91.75%
S_2	92.34%	91.55%	98.90%	94.26%
S_3	86.57%	82.30%	98.95%	89.27%
S_4	80.84%	81.00%	98.28%	86.71%
S_5	77.83%	75.07%	98.41%	83.77%
S_6	79.90%	75.78%	98.22%	84.63%
S_7	72.20%	69.00%	97.74%	79.65%

Table 4.5. Interface Similarities in Inter-Project Clones

Similarity Conditions	Subject System			Average(%)
	Small	Medium	Large	
S_1	94.07%	76.06%	97.49%	89.21%
S_2	91.38%	99.80%	94.22%	95.13%
S_3	89.56%	38.16%	95.11%	74.28%
S_4	87.59%	76.02%	92.73%	85.45%
S_5	85.32%	37.72%	92.90%	71.98%
S_6	83.68%	38.15%	91.24%	71.02%
S_7	80.68%	37.72%	89.92%	69.44%

It is observed that the average rate of fulfilling each type of similarity condition is almost consistent in all subject systems. It is an indication that interface similarity is related to method clone. Especially, on average above 90% of the clones satisfy only return type and parameter types based similarity conditions such as S_1, S_2, and S_4. This is because if two methods contain similar return types and parameters, it means those take similar input and provide same output. Most of the time these methods are considered to be clones as those perform similar functionality omitting the keyword those contain in method names [66]. It infers that approximately above 90% intra-project and inter-project method clones contain similar return type and parameter types. As keywords have less significant influ-

ences on method functionality comparing to its return type and parameter types, 89.27%, 83.77%, 84.63% and 79.65% intra-project clones satisfy conditions S_3, S_5, S_6 and S_7 respectively. In comparing to intra-project clones, similarity conditions S_3, S_5 and S_6 are not satisfied by a significant number of inter-project clones. Since the usage of the inappropriate naming convention, improper keywords and generic type prevents method clones satisfying similarity conditions S_3, S_5 and S_6. Only 74.28%, 71.98% and 71.02% inter-project clones meet S_3, S_5 and S_6 conditions respectively that infer on average 69.44% inter-project clone contains similar keywords from method names, return type and parameter types.

From Table 4.4 and Table 4.5, it is observed that conditions associated to keyword such as S_3, S_5 and S_6 are not satisfied by a significant number of clones. Besides, conditions associated to return type and parameter types such as S_1, S_2 and S_4 are not satisfied by almost 10-15% clones. To identify the reasons why those clones did not satisfy the interface similarity conditions, source code of those clones are manually inspected. Some obvious reasons are found by this inspection. A brief description of each reason is mentioned below.

Usage of Various Naming Convention: The first reason is usage of various naming convention. In some clones, one of the method names is not written by following Camel Case naming convention. Some clones contain Pascal Case and Snake case or underscore case naming convention. While splitting method name by following Camel Case naming convention, keywords cannot be extracted. As a result, clones fail to fulfill conditions S_3, S_5, S_6 and S_7.

Usage of Improper Keywords: Another significant reason is usage of improper keywords in method names. For example, a pair of method clone is shown in the Listing 4.3 and 4.4 as code fragments C[11] and D[12] respectively. In this experiment, both the code fragments of Listing 4.3 and 4.4 are detected as inter-project method clone. These code fragments can be found from two different Java

[11]https://github.com/apache/nutch/blob/master/src/java/org/apache/nutch/util/StringUtil.java
[12]https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/NativeGlobal.java

projects *Apache Nutch* and *Mozila Rhino* respectively. Here, code fragments C and D contain two interfaces such as `int charToNibble (char c)` (Code Fragment (C), Line-1) and `int unHex (char c)`(Code Fragment (D), Line-1) respectively.

In the code fragment D, while defining the interface, developers use improper keywords such as *unHex* in the method name that does not describe its functionality. So, correct keywords cannot be extracted from its method name. As a result, while satisfying the similarity conditions proper keywords cannot be drawn that prevents these clones satisfying conditions S_3, S_5, S_6 and S_7 despite having similar return and parameter types.

Return and Parameter Type Mismatch: The second reason is type mismatch problem. In some clones, the return type and parameter types are not exactly the same, but those are considered to be cloned since their coding structure and functionality are similar. For example, Listing 4.5 and Listing 4.6 represent two code fragments E[13] and F[14] as an inter-project method clone where the code fragments of E and F are collected from project *stanford-nlp* and *berkeleyparser* respectively. In this method clone, code fragments E and F contain interfaces `double[] add(double[] a, double[] b)` and `float[] pairwiseAdd(float[] a, float[] b)`. Both method namely *add* and *pairwiseAdd* from code fragment E and F take two arrays as input and perform pairwise addition for each element of those two input arrays along with return the summation array as the output.

However, those two methods perform similar functionality on two different type of data such as double and float. These methods also contain the similar syntactic implementation in the method body. But, in this study, these interfaces are considered to be dissimilar. The reason is that while satisfying interface similarity conditions, only those clone pairs are counted that contain same return type and parameter types.

[13]stanfordnlp/CoreNLP/blob/master/src/edu/stanford/nlp/math/ArrayMath.java
[14]berkeleyparser/blob/master/src/edu/berkeley/nlp/util/ArrayUtil.java

Listing 4.3: Code Fragment (C)

```
private static final int charToNibble(char c) {
  if (c >= '0' && c <= '9') {
    return c - '0';
  } else if (c >= 'a' && c <= 'f') {
    return 0xa + (c - 'a');
  } else if (c >= 'A' && c <= 'F') {
    return 0xA + (c - 'A');
  } else
    return -1;}
```

Listing 4.4: Code Fragment (D)

```
private static int unHex(char c) {
  if ('A' <= c && c <= 'F') {
    return c - 'A' + 10;
  } else if ('a' <= c && c <= 'f') {
    return c - 'a' + 10;
  } else if ('0' <= c && c <= '9') {
    return c - '0';
  } else
    return -1;
}
```

Although, in this example the clone pair containing interface double[] add(double[] a, double[] b) and float[] pairwiseAdd(float[] a, float[] b), have similar keywords in method names, return types and parameter types are not exactly same.

Usage of Generic Types: Type mismatch problems have also occurred in some clones for the usages of generic return type and parameter types. For

example Listing 4.7 and Listing 4.8 show two code fragments G[15] and F[16] as an inter-project method clone. Code fragments G and F are collected from project *pdm* and *eclipse-jdtcore* respectively. Code fragments G and F contain interface boolean areSemanticEquals(T[] a, T[] b) and boolean equalArraysOrNull(int[] a, int[] b). It is observed that code fragment G contains generic type in its parameters where code fragment F contains integer type parameters. So, while satisfying similarity conditions related type, generic type is not correspondent to the concrete type. As a result, a significant number of clones (both intra-project and inter-project clones) does not satisfy those similarity conditions S_3, S_5, S_6 and S_7 adequately.

Listing 4.5: Code Fragment (E)

```
1  public static double[] add(double[] a, double[] b) {
2    double[] result = new double[a.length];
3    for (int i = 0; i < a.length; i++) {
4      result[i] = a[i] + b[i];
5    }
6    return result;}
```

Listing 4.6: Code Fragment (F)

```
1  public static float[] pairwiseAdd(float[] a, float[] b) {
2    float[] result = new float[a.length];
3    for (int i = 0; i < a.length; i++) {
4      result[i] = a[i] + b[i];
5    }
6    return result;
7  }
```

[15]pmd/pmd/blob/master/pmd-core/src/main/java/net/sourceforge/pmd/util/CollectionUtil.java
[16]eclipse/org.eclipse.jdt.core/model/org/eclipse/jdt/internal/core/util/Util.java

Listing 4.7: Code Fragment (G)

```
public static <T> boolean areSemanticEquals(T[] a, T[] b) {
  if (a == null) { return isEmpty(b); }
  if (b == null) { return isEmpty(a); }
  if (a.length != b.length) return false;
  for (int i=0; i<a.length; i++) {
    if (!areEqual(a[i], b[i]))
    return false;
  }
  return true;
}
```

Listing 4.8: Code Fragment (H)

```
public static boolean equalArraysOrNull(int[] a, int[] b) {
  if (a == b)
    return true;
  if (a == null || b == null)
    return false;
  int len = a.length;
    if (len != b.length)
    return false;
  for (int i = 0; i < len; ++i) {
    if (a[i] != b[i])
    return false;
  }
  return true;
}
```

SRQ_2: Are the intensities of interface similarity different in various types of clones and which clone-type(s) have higher possibilities to be detected by using interface similarity?

This research question is answered with respect to Type-1, Type-2 and Type-3 method clones. The total number of detected Type-1, Type-2 and Type-3 method clones in three subject systems are 61492, 103,672 and 214,301 respectively. Table 4.6, Table 4.7 and Table 4.8 represent the summary of satisfying interface similarities by Type-1, Type-2 and Type-3 clones respectively. From Table 4.6, it is observed that conditions S_1, S_2, S_3, S_4, S_5, S_6 and S_7 are satisfied by Type-1 clones are 100%. It is consistent with Type-1 method clone definition. Since syntactical modifications do not occur in Type-1 method clone, interfaces of those clones always remain similar.

Table 4.6. Interface Similarities in Type-1 Clones

Similarity Conditions	Subject System			Average(%)
	Small	Medium	Large	
S_1	100%	100%	100%	100
S_2	100%	100%	100%	100
S_3	100%	100%	100%	100
S_4	100%	100%	100%	100
S_5	100%	100%	100%	100
S_6	100%	100%	100%	100
S_7	100%	100%	100%	100

Table 4.7. Interface Similarities in Type-2 Clones

Similarity Conditions	Subject System			Average(%)
	Small	Medium	Large	
S_1	98.72%	81.81%	83.26%	87.93%
S_2	98.02%	87.33%	87.27%	90.87%
S_3	89.79%	82.19%	82.00%	84.66%
S_4	96.88%	69.47%	71.76%	79.37%
S_5	88.57%	71.63%	71.74%	77.31%
S_6	87.88%	70.79%	70.76%	76.48%
S_7	86.79%	60.48%	60.76%	69.35%

From Table 4.7 and Table 4.8, it is found that on average above 85% Type-2 and Type-3 method clones satisfy return type and parameter types based similarity conditions such as condition S_1, S_2, and S_4. However, a significant number of Type-2 and Type-3 clones satisfy conditions S_7. On average 69.35% Type-2 and 67.34% Type-3 clones fulfill S_7 conditions respectively. As renaming of identifiers and literal values occur in Type-2 clones and some addition or deletion of statement occur in Type-3 method clones, some method clones fail to fulfill some similarity conditions. For example, in a Type-2 method clone interfaces are `List<String> getContent(File)` and `ArrayList<String> getContent(File)` respectively. However, by following Java polymorphism features, these two methods contain similar interfaces, but here these are considered dissimilar. This is because that exact matching is performed to measure the return type similarity. Besides, inappropriate naming convention, type mismatch problem and usage of generic type prevent on average 25% to 30% Type-2 and Type-3 clones satisfying similarity conditions. It infers that approximately 69.35% Type-2 and 67.34% Type-3 clones contain similar keywords from method name, return and parameter types. So, the intensity of interface similarity is higher in Type-1 compared to Type-2 and Type-3 clones.

From the experimental result, it has been analyzed that all types of method clones can be detected using interface similarities. There is a 100% probability that Type-1 method clones can be identified by performing exact interface information matching. Above 85% Type-2 and Type-3 method clones can be detected by interface similarities since 87.93% Type-2 and 87.31% Type-3 clones contain similar interfaces. However, while detecting Type-2 and Type-3 clones, more than 85% clone can be detected by incorporating to the inappropriate naming convention, type mismatch problem and usage of the generic types.

Table 4.8. Interface Similarities in Type-3 Clones

Similarity Conditions	Subject System			Average(%)
	Small	Medium	Large	
S_1	91.46%	85.68%	84.78%	87.31%
S_2	88.21%	85.93%	85.16%	86.44%
S_3	88.62%	77.94%	83.38%	83.31%
S_4	81.97%	73.84%	72.25%	76.02%
S_5	82.28%	69.71%	75.93%	75.98%
S_6	80.70%	68.35%	71.69%	73.58%
S_7	75.67%	60.80%	65.55%	67.34%

$\mathbf{SRQ_3}$: How does interface similarity relate to code clone detection? More specifically, how many code clones occur due to interface similarity?

In this question, the relationship between classical code clone and interface similarity has been investigated. The main question is how many clones occur due to interface similarity. In this case, only the intra-project clones are considered because these are the method clones that are implemented by the developers for each project. Besides, for interface similarity measurement, only those clones are considered that satisfy similarity condition S_7. This is because it ensures both methods in each clone pair contain same return types, at least one keyword from the method name and one parameter type. Table 4.9 represents the number of intra-project method clones in each subject system. It also provides both the numbers of clones that satisfy and do not satisfy S_7 similarity condition. It is found that out of 1,85,260 intra-project method clones 43,749 clones do not contain similar interfaces that refer to only 21.44% clones do not satisfy S_7 similarity condition. Because of the inappropriate naming convention, type mismatch and generic type matching problem, these clones fail to satisfy similarity condition that is discussed while answering sub-research questions $\mathbf{SRQ_1}$ and $\mathbf{SRQ_2}$. On the other hand, it is observed that in total 1,41,475 method clones contain a similar interface. It infers that 78.56% clones occur due to interface similarity. It is a

very important result since it is the evidence that interface similarity may have a significant relationship to classical method clone detection.

Table 4.9. Intra-Project Clones Satisfying Condition S_7

Clones	Subject System			
	Small	Medium	Large	**Total**
Intra Project	55095	82264	47847	**1,85,206**
Satisfy S_7	38754	55939	46764	**1,41,457**
Percentage (%)	70.34	67.99	97.36	**78.56%**
Do not Satisfy S_7	16341	26325	1083	**43,749**
Percentage (%)	29.65	32.00	2.26	**21.44%**

4.4 Threats to Validity

The number of detected clones may increase or decrease with the variation of detection parameters used by SourcererCC [18] and NiCad [11]. In an exhaustive study, [80] it is observed that clone detection tools are affected by confounding configuration parameter choice problem. So, SourcererCC [18] and NiCad [11] are also affected by this problem. However, here standard configuration settings for clone detection such as 80% similarity with minimum 6 lines are used by SourcererCC [18] and NiCad [11] that prevents both tools getting false positive clones and ignoring the `getter`, `setter`, and abstract methods of Java class attributes.

In this study, NiCad [11] has been used for detecting Type-1, Type-2 and Type-3 clones. So, any other clone detection tools can provide different experimental results. In [18], it has been shown that for detecting three types of clone, NiCad [11] is very accurate in comparison with other clone detection tools. Similarly, in some previous clone detection studies [18], it is observed that SourcererCC [18] is also very accurate for detecting intra-project and inter-project clones in large code repositories. So, the clones detected by SourcererCC [18] and NiCad [11] have the significant impact on the results of this study.

The experimental results may vary if the subject systems are changed. However, subject systems used in this experiment are enough to take a complete decision on the relationship and effects of interface similarities in code clones. The reason is that the candidate systems differ in terms of the application domains, size, revision and the presences of various types clones. These subject systems contain 35, 109 and 24,558 open source Java Projects extracted from SourceForge, Git-Hub, Google Code etc. So, for instance, these projects can be representative of a small, medium and large company's local repositories. These are also used in many code clone studies [18]. The experimental results implied from this subject system should be statistically sound.

Reproducibility: All the necessary artifacts of this study are publicly available[17]. For generating the statistical results to verify the claims of this study, these artifacts include the source code of subject systems, all types of detected clones such as intra-project, inter-project, Type-1, Type-2 and Type-3, raw data generated in the study, and the source code of analyzing interface similarity[18].

4.5 Summary

The relationship between method clones and interfaces have never been studied before. In this study, an exploratory study has been performed to investigate the relationship, impact and effects of interfaces in code clones. In the first step, three types of subject systems (*Small*, *Medium* and *Large*) are selected as the experimental dataset that contains 35, 109 and 24,558 open source Java projects respectively. Next, two token based clone detection tools such as SourcererCC [18] and NiCad [11] are used to identify all types of clones such as intra-project, inter-project, Type-1, Type-2 and Type-3 within the subject systems. The total number of clones found are 231411, 242992 and 130226 respectively. After that,

[17]https://github.com/MisuBeImp/APSEC-2017-Paper-Artifacts
[18]https://github.com/MisuBeImp/CloneInterfaceSimilarityDetector

interface information such as return type, method name and parameter types of those clones are extracted from the source code. Finally, interfaces similarity is measured in method clones by satisfying similarity conditions.

The experimental result analysis shows that on average 79.65% intra-project clones and 69.44% inter-project clones contain similar interface return types and at least one root word and at least one parameter types are similar. Besides, 100% Type-1, 69.35% Type-2 and 67.34% Type-3 clone contain similar interfaces. However, use of inappropriate naming convention, generic type and type mismatch problem prevent some clones satisfying interface similarities which ushers new research directions. Nevertheless, the findings will help to design new interface driven code clone detection tool.

Chapter 5

Interface Driven Code Clone Detection

Two identical code fragments are know as code clone. It is a common code reusing technique that occurs when developers replicate similar pieces of code fragments within or between software repositories. By doing this, developers also repeat method interfaces such as method name, return type and parameter types. Two methods are prone to be cloned when those have similar interfaces and perform similar functionalities [7]. So, it indicates that there is a relationship between code clones and interface similarities. Based on this relationship, a new Interface Driven Code Clone Detection (IDCCD) technique is proposed that can detect clones by using method interface similarities. Firstly, the method blocks are tokenized from the source files. For those method block tokens, interface information is extracted and indexed with mapped tokens for quickly retrieving. Then similar interfaces are queried from that index and compared those with a similarity function for detecting clones. IDCCD is evaluated with other state of the art techniques by using BigCloneEval [22] frameworks. The experimental results show that IDCCD performs similar comparing to other existing tools with reduced candidate comparison.

5.1 Introduction

Code clones refer to identical code fragments within a code repository. Due to having several negative impact of code clones such as propagating bugs, increasing software size and lacking inheritance etc. it is required to detect and manage code clones [9]. Similar to code clones, another replication happens when developers repeat method interfaces such as method name, return and parameter types. In the previous chapter, an exploratory study has been performed where it is observed that interface similarity is strongly related to code clones. So, there is a potentiality that this relationship will be conducive to develop a clone detection tool and technique.

Clone detection tools and techniques differ from many aspects such as what type of detection algorithm is used, how the source code is represented to operate and how various clones can be detected. Textual based techniques [9] use string matching algorithms that are perfect for detecting exact clones but do not work faster for larger dataset, and thus, fail to detect all possible clones. For example, researchers applied the Longest Common Subsequence (LCS) algorithm in their tool called NiCad [11] for an efficient text line comparison to find nearly mismatched clones. AST based techniques [14] are useful for refactoring of clones, but those techniques may not efficient for large code repository as parse trees require high memory. Token based techniques gain high recall but may yield clones which are not syntactically complete.

A token based approach has also been introduced called SourcererCC [18] that works faster for large code repositories. Sub block overlapping and token positioning heuristic are used to reduce code fragments for efficient comparison. Similar to SourcererCC [18], a flexible technique CloneWorks has been proposed for various types of clones [19]. Both SourcererCC [18] and CloneWorks [19] compute sub-block overlapping tokens for indexing sub-blocks of tokens that performs overhead

execution time before making actual method fragment comparison.

In order to reduce the candidate comparison while detecting clones, a light weight Interface Driven Code Clone Detection (IDCCD) technique is proposed based on the relationship of interface similarity and code clones. Firstly, source files are tokenized into method blocks and interface information such as keywords, return and parameter types is extracted. An inverted index is built using that interface information and mapped into the method block tokens. For each method block, similar interfaces are queried from that index and compare those with a similarity function. Finally, pairs of method blocks are reported as code clone if those satisfy user given similarity threshold.

The approach was evaluated into two phases. Firstly, a candidate code fragment comparison minimization experiment was conducted to show that IDCCD was capable of detecting clones with a reduced candidate comparison. Next, the accuracy of IDCCD was measured through the usage of existing evaluation benchmark and frameworks. The performance of IDCCD was also compared against the popular code clone detectors such as CloneWorks [19], SourcererCC [18], NiCad [20] etc. The evaluation results are promising to use interface similarities for detecting clones. A detailed description of the proposed approach and its evaluation results comparing to state of the art techniques are mentioned in the later portions of this chapter.

5.2 Overview of Interface Driven Code Clone Detection (IDCCD)

Figure 5.1 represents the overview of the IDCCD. It comprises of three phases such as (i) Token Generation, (ii) Interface Index Creation and (iii) Clone Detection. A brief description of each step is mentioned below.

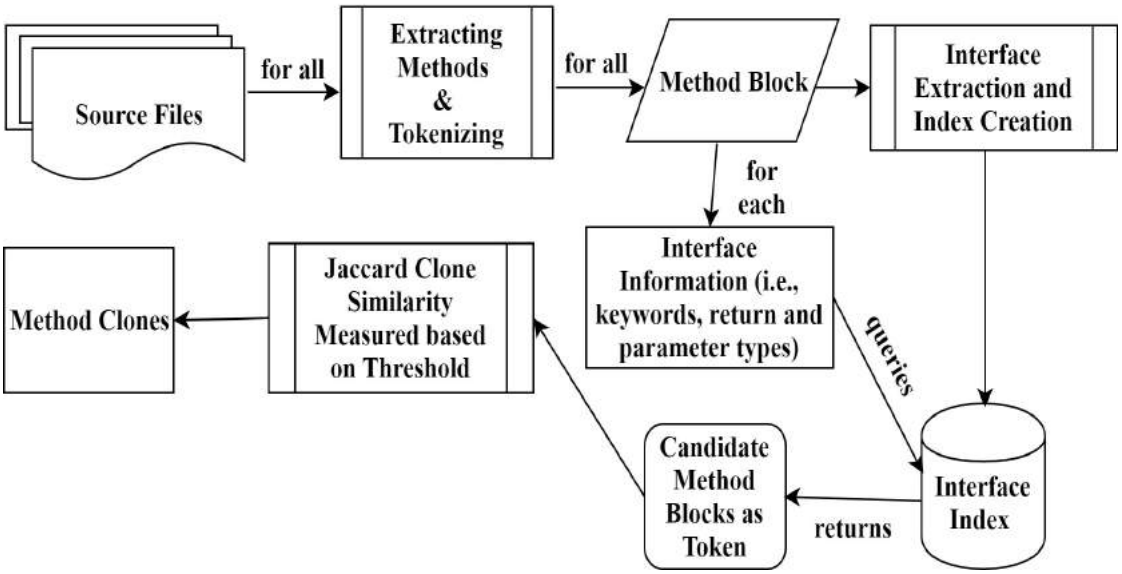


Figure 5.1. Overview of Interface Driven Code Clone Detection (IDCCD)

5.2.1 Token Generation

In token generation phase, first source files are transformed into its AST representation by using *Eclipse ASTParser*[1]. Method blocks are then extracted by traversing the AST nodes. Next, extracted method blocks are tokenized into a stream of tokens by the below steps.

- **Removing Comments:** In order to alleviate the affects of comment in natural language, first comments are removed from the method blocks.
- **Lexing:** Next, it lexes the code snippets of the method blocks without comments into a token stream.
- **Removing Punctuations:** Since punctuations do not contain significant semantics, those are also removed from the token stream.
- **Post Processing:** Camel Case tokens, Pascal Case tokens and underscore tokens are also transformed into lowercase letters. It mitigates the different code writing conventions. The frequency of each unique token is also measured. Currently, this token generation works for both Java and C#.

[1]https://github.com/eclipse/eclipse.jdt.core

To understand how the token stream is generated from the method blocks, an example is provided. Figure 5.2 represents a method block extracted from a java source file. Within this method block, firstly, the comments are removed. Then a token stream is generated for each word and symbol splitting by space. From this token stream, punctuations and special characters are also discarded. Now, these tokens are transformed into lower case letters and calculate the frequency of each unique token. The token stream is represented as a set of unique token where each token is represented with its frequency. For example, in the method block, the token *if* is appeared four times (shown in Figure 5.2). So, it represented as in a form of *(if-4)*. Figure 5.3 shows the final output of token generation phase for a single method block.

```
public static boolean equal(double[][] a, double[][] b) {
    if (a == null) {
            return (b == null);
     }
     if (b == null) {
            return false;  // already know 'a' isn't null
     }
      if (a.length != b.length) {
            return false;
       }
       for (int i = 0; i < a.length; i++) {
            if (!Arrays.equals(a[i], b[i])) {
                return false;
            }
        }
        return true;
    }
}
```

Figure 5.2. A Method Block From Java Source File

```
{(0-1),(a-5),(arrays-1),(b-5),(boolean-1),(double-2),(equal-1),(equals-1),
(false-3),(for-1),(i-5),(if-4),(int-1),(length-3),(null-3),(public-1),(return-5),
(static-1),(true-1)}
```

Figure 5.3. Token Stream Generated from the Method Shown in Figure 5.2

5.2.2 Interface Index Creation

In the index creation step, interface information, for example, method name, return type and parameter types are extracted from method blocks. After that, keywords are extracted from the method names by removing stop words and performing stemming. A detailed description of stop word removing and stemming is provided in Chapter 2. Synonyms and related words of those keywords are identified by using standard WordNet Library [79]. For each method block it provides the following interface information -

(a) Return type.

(b) Keywords (Extracted from method name including a set of synonyms and related words)

(c) Parameter types.

This information is used to build an inverted interface index. An inverted index is a data structure for sorting and mapping from a content, for example words, numbers, tokens etc. to its location in a document or in a set of documents. In inverted interface index, document considered to be the smallest unit. A document contains one or more fields to store the value in the index. In this case, document should contain all the interface information as fields. It maps tokens of the method block with this interface index. A detailed description of how inverted index is created from the method blocks is mentioned in Chapter 2.

5.2.3 Clone Detection

In clone detection, for each method block, IDCCD retrieves the candidate method block identifiers from the interface index by querying. Since the interface index is constructed using interface information such as keywords, return and parameter types, only the interfaces of those method blocks are used to query into the index.

The interface index only results those method block identifiers that are similar to query. Then using those identifiers, token streams are collected from the token list. As a result, detection comparison is performed among the method block tokens for which interfaces are similar. After getting all candidate method block tokens, clone detection is performed by modified Jaccard similarity metric shown in Equation 5.1 that is also used by CloneWorks [19]. It first takes a pair of method block token stream, for example, t_1 and t_2, and computes minimum token intersection ratio. A pair of method blocks is reported as method clone if these token stream intersection ratio satisfies user given similarity threshold U_t. For example, if the similarity score between two token streams is greater than or equal to the user given similarity threshold ($Similarity(t_1,t_2) \geq U_t$), those are reported as clone pair.

$$Similarity(t_1, t_2) = \frac{|t_1 \cap t_2|}{max(|t_1|, |t_2|)} = min(\frac{|t_1 \cap t_2|}{|t_1|}, \frac{|t_1 \cap t_2|}{|t_2|}) \tag{5.1}$$

5.3 Implementation of IDCCD

In this section, a detailed description about the implementation and execution steps of IDCCD are provided. It is developed as an open source Java project that can be downloaded from GitHub/MisuBeImp[2]. As mentioned above, IDCCD has three phases such as (1) Token Generation, (2) Interface Index Creation and (3) Clone Detection. For each phase, an individual component is developed and packaged into Java jar file such as (i) *Tokenizer.jar*, (ii) *Indexer.jar* and (iii) *Detector.jar* respectively. To work collaboratively, these modules need to take configuration parameters from property files. The implementation of each component and the configuration files are described below.

[2]https://github.com/MisuBeImp/IDCCD

5.3.1 Tokenizer.jar

The responsibility of Tokenizer.jar is to read the source files from respective projects and parse these into method blocks. After that, interface of those method blocks are extracted and the source codes are transformed into a stream of tokens. To understand how the Tokenizer.jar works, an example is provided.

As mentioned earlier, Token Generation phase takes each source file as input and parses all method blocks from the source files. So, for a given project directory path, source files are extracted into method blocks by Tokenizer.jar from the location where those files are presented. For each method block, various meta data are collected such as method block identifier, method name, source file path, start line and end line of that method block. For each method block, this data is stored into a *headers.file* where in each line, meta information is differentiated by a delimiter such as comma (,). For example, Figure 5.4 represents a snapshot of *headers.file*. Here each line contains four meta data in the form of following line - ***method_id, method_name, source_file_path, start_line, end_line.***

```
0,defineClassFromData,/home/misu/Data/sample/ant/org/apache/tools/ant/loader/AntClassLoader2.java,72,79
1,getJarManifest,/home/misu/Data/sample/ant/org/apache/tools/ant/loader/AntClassLoader2.java,92,105
2,definePackage,/home/misu/Data/sample/ant/org/apache/tools/ant/loader/AntClassLoader2.java,117,139
3,definePackage,/home/misu/Data/sample/ant/org/apache/tools/ant/loader/AntClassLoader2.java,149,223
4,addPathFile,/home/misu/Data/sample/ant/org/apache/tools/ant/loader/AntClassLoader2.java,236,299
5,addDefaultExclude,/home/misu/Data/sample/ant/org/apache/tools/ant/DirectoryScanner.java,516,522
6,setIncludes,/home/misu/Data/sample/ant/org/apache/tools/ant/DirectoryScanner.java,640,649
7,setExcludes,/home/misu/Data/sample/ant/org/apache/tools/ant/DirectoryScanner.java,663,672
8,addExcludes,/home/misu/Data/sample/ant/org/apache/tools/ant/DirectoryScanner.java,687,703
9,normalizePattern,/home/misu/Data/sample/ant/org/apache/tools/ant/DirectoryScanner.java,714,721
```

Figure 5.4. Snapshot of headers.file

From the source code of method blocks, this module extracts the interface of those methods and collects the interface information such as return type, keywords and related words from method name and parameter types. Then, source file identifier, method block identifies and interface information are stored into an *interfaces.file*. A snapshot of *interfaces.file* is provided in Figure 5.5. In *interfaces.file*, for each method block interface information are stored in a single line

and the source file identifier, method identifier and method name are separated by comma (,) delimiter but return type, parameter types, keywords and related words are separated by double hash (##) delimiter in the form of following line - ***file_id, method_id, method_name, return_type ## [parameter_types] ## [keywords]## [related words]***

```
4,0,defineClassFromData,Class##[File, byte[], String]##[data, defin]##[]
4,1,getJarManifest,Manifest##[File]##[manifest, jar]##[bump_around, evidence, manifest, certify,
4,2,definePackage,void##[File, String]##[defin, packag]##[]
4,3,definePackage,void##[File, String, Manifest]##[defin, packag]##[]
4,4,addPathFile,void##[File]##[add, path, file]##[add, tote_up, charge, tot_up, add_together, cor
8,5,addDefaultExclude,boolean##[String]##[add, exclud]##[add, tote_up, tot_up, add_together, cont
8,6,setIncludes,void##[String[]]##[set, includ]##[prepare, dress, localize, correct, jell, do, ge
8,7,setExcludes,void##[String[]]##[set, exclud]##[prepare, dress, localize, correct, jell, do, ge
8,8,addExcludes,void##[String[]]##[add, exclud]##[add, tote_up, tot_up, add_together, contribute,
8,9,normalizePattern,String##[String]##[normal, pattern]##[pattern, model]
```

Figure 5.5. Snapshot of interfaces.file

The source code of the methods is transformed into a stream of tokens. How the stream of tokens is generated, it is previously described in the above section. Now, the source file identifier, method identifier and stream of tokens are stored in a *tokens.file*. Figure 5.6 provides a snapshot of *tokens.file*. In *tokens.file*, for each method block, tokens are stored in a single line. The source file identifier, method block identifier are separated by comma (,) and the tokens differentiated by double hash (##) delimiter and enclosed inside curly brackets such as {bag of tokens}. In the stream of tokens, each token is separated by a comma (,) and represented by a pair token and frequency that is *(token term-frequency)*.

file_id, method_id ##{(token-frequency),(token-frequency)...}

```
4,0##{(0-1),(byte-1),(class-1),(classdata-3),(classname-3),(container-2),(defineclass-1),(definecl
4,1##{(close-1),(container-3),(file-1),(finally-1),(getjarmanifest-1),(getmanifest-1),(if-2),(ioex
4,2##{(0-1),(1-1),(classindex-3),(classname-3),(container-3),(definepackage-3),(else-1),(file-1),(
4,3##{(attributes-2),(catch-1),(container-2),(definepackage-2),(e-1),(equalsignorecase-1),(file-1)
4,4##{(8-1),(abspathplustimeandlength-3),(addpathfile-3),(ant-3),(apache-3),(are-1),(baseurl-2),(b
8,5##{(1-1),(add-1),(adddefaultexclude-1),(boolean-1),(defaultexcludes-2),(false-1),(if-1),(indexo
8,6##{(0-1),(else-1),(for-1),(i-5),(if-1),(includes-8),(int-1),(length-2),(new-1),(normalizepatter
8,7##{(0-1),(else-1),(excludes-8),(for-1),(i-5),(if-1),(int-1),(length-2),(new-1),(normalizepatter
8,8##{(0-5),(addexcludes-1),(arraycopy-1),(else-1),(excludes-14),(for-1),(i-5),(if-2),(int-1),(len
8,9##{(endswith-1),(file-3),(if-1),(normalizepattern-1),(optional-1),(p-2),(pattern-4),(private-1)
```

Figure 5.6. Snapshot of tokens.file

In order to execute *Tokenizer.jar*, user have to run the following command.

>>java -jar Tokenizer.jar tokenizer.properties

It is observed that *Tokenizer.jar* accepts an argument. This argument represents a file name called *tokenizer.properties*. It contains ten parameters which are listed below.

1. **maximumLine:** Maximum number of lines to be considered in a method.
2. **minimumLine:** Minimum number of lines to be considered in a method.
3. **maximumToken:** Maximum number of tokens to be considered in a method.
4. **minimumToken:** Minimum number of tokens to be considered in a method.
5. **headerFilePath:** File path where the *headers.file* will be created.
6. **tokenFilePath:** File path where the *tokens.file* will be generated.
7. **interfaceFilePath:** File path where the *interfaces.file* will be stored.
8. **dictonaryPath:** File path for using the Word Net dictionary.
9. **projectDirectoryPath:** Base directory path for sample projects.
10. **projectName:** Name of a project from the sample projects.

A snapshot of *tokenizer.properties* is shown in Figure 5.7.

```
1  maximumLine =0
2  minimumLine =6
3  maximumToken = 0
4  minimumToken = 0
5  headerFilePath=/home/misu/Data/headers.file
6  tokenFilePath=/home/misu/Data/tokens.file
7  interfaceFilePath=/home/misu/Data/interfaces.file
8  dictonaryPath=/home/misu/SoftwareReEngineeringProject/wn3.1.dict/dict
9  projectDirectoryPath=/home/misu/Data/sample
10 projectName=hadoop-mapred
```

Figure 5.7. Snapshot of tokenizer.properties

5.3.2 Indexer.jar

After extracting the interface and generating tokens for the corpus, *Indexer.jar* takes the *interface.file* as input. It reads each line from *interface.file* in order to construct an inverted index for each interface information for each method block. The command to execute the Indexer.jar is as follows

>>java -jar Indexer.jar indexer.properties

Here Indexer.jar expects one argument as a file name called *indexer.properties*. This file contains the configuration parameters for *Indexer.jar*. Figure 5.8 represents a snapshot of the *indexer.properties* file. It includes two parameters that are listed below. After index building, clone detection is performed by *Detector.jar*.

1. **indexDirectory:** Directory path where the index will be constructed.
2. **interfaceFilePath:** File path where the *interfaces.file* is located.

```
1 indexDirectory=/home/misu/Data/index
2 interfaceFilePath=/home/misu/Data/interfaces.file
```

Figure 5.8. Snapshot of indexer.properties

5.3.3 Detector.jar

Once the index is created, now *Detector.jar* is launched. It takes three files as input such as *header.file*, *interface.file*, *tokens.file*. It also gets a similarity threshold between 0 to 1 and a similarity condition from S_1 to S_7 (mentioned in Chapter 4). To identify all clones, it queries into the index for each interface and retrieves candidates of similar interface based on the given similarity condition. Then, it performs comparison among the candidate tokens for those interface based on the user defined similarity threshold. Finally it produces three output files such as *outputClonePair.file*, *outPutCloneCode.file* and *outputComparison.file*.

The command to execute the *Detector.jar* is as follows:

>>java -jar Detector.jar detector.properties

Similar to *Indexer.jar*, *Detector.jar* also expects one argument as a file name called *detector.properties*. This file contains the configuration parameters for the *detector.jar*. Figure 5.9 represents a snapshot of *detector.properties* file. It includes nine parameters listed below.

1. **indexDirectory:** Directory path where the inverted index has been built.
2. **interfaceFilePath:** File path where the *interfaces.file* is presented.
3. **headerFilePath:** File path where the *header.file* is located.
4. **tokenFilePath:** File path where the *token.file* is stored.
5. **outputClonePair:** File path where detected clone pairs will be stored.
6. **outputCloneCode:** File path where detected clone code will be found.
7. **outputComparison:** File path where candidate comparison will be found.
8. **similarityThreashold:** A threshold value between 0 to 1 inclusive.
9. **similarityCondition:** A preferable similarity condition between S_1 to S_7 (mentioned in Chapter 4 in Table 4.3)

```
1 indexDirectory=/home/misu/Data/index
2 interfaceFilePath=/home/misu/Data/interfaces.file
3 headerFilePath=/home/misu/Data/headers.file
4 tokenFilePath=/home/misu/Data/tokens.file
5 outputClonePair=/home/misu/Data/clonePairIDCCD.file
6 outputCloneCode=/home/misu/Data/cloneCodeIDCCD.file
7 outputComparison=/home/misu/Data/Compare/comparisonIDCCD.txt
8 similarityThreashold=0.80
9 similarityCondition=S7
```

Figure 5.9. Snapshot of detector.properties

5.3.4 Evaluation

The evaluation of IDCCD is conducted based on the research questions RQ2 and RQ3 posted in Chapter 1. Those questions are also mentioned below.

- RQ2: Can interface similarity minimize the candidate code fragment comparison in code clone detection?
- RQ3: What is the accuracy of Interface Driven Code Clone Detection approach against the state-of the-art works?

In order to answer the RQ2, a candidate comparison minimization experiment is performed. On the other hand, to answer the RQ3, recall and precision are used as evaluation metrics to measure the accuracy of IDCCD. In the following sections, the candidate comparison minimization experiment and accuracy measurement of IDCCD are described exhaustively.

5.4 Candidate Comparison Minimization Experiment

The aim of this experiment is to identify whether IDCCD reduces the number of comparison while detecting code clones. To measure reduce number of comparison, the following three metrics are used.

1. The number of clones detected.
2. The total number of candidates compared.
3. The total number of tokens compared.

To compare the candidate comparison minimization, a Complete Search (CS) approach is also developed that detects clones comparing all candidates to each other. The experimental dataset is introduced. Then, how those metrics are calculated

for both IDCCD and Complete Search approach along with their comparison are also mentioned.

5.4.1 Subject System

For the candidate comparison minimization experiment, twenty four Apache Java projects are chosen from thirty five projects randomly. These projects are also used as *Small* subject system in the exploratory study discussed in the previous chapter. These projects are selected since those are varied from various sizes and functionalities which includes database system, natural languages processing library, machine learning, distributed system, network system etc.

Some projects are highly popular in their respective domain. Therefore, such subject systems provide helps to avoid a potential bias towards a specific software system. The detailed description of those projects such as number of source files, LOC and number of methods and clones can be found in Appendix.

5.4.2 Procedure

In order to calculate the above mentioned metrics, it is required to identify how many clones are detected. Besides, while detecting clones, it is also needed to count how many candidates are compared and how many tokens are compared. To do so, at first, the configuration parameters are set for IDCCD. Here, standard configuration parameters are used such as method level granularity, minimum 6 lines with 80% similarity threshold. Now for each project, IDCCD is run in order to detect clones with that configuration parameters. While detecting clones, the total number of candidates detected and the total number of tokens compared are also calculated.

For comparative study, a Complete Search (CS) approach is also developed that can detect clones comparing all candidates to each other. The standard configuration parameters with the method level granularity are also set for this

Complete Search approach. After that for each project, this Complete Search approach is also run to detect clones and at the same time the total number of candidates detected and the total number of tokens compared are also measured.

In order to know how much minimization of candidate comparison is performed by the IDCCD, both the results of IDCCD and Complete Search approach are compared for each metric.

5.4.3 Result Analysis

This section provides the experimental results for candidate comparison minimization experiment. As mentioned above, three metrics are used to evaluate the candidate comparison minimization done by both IDCCD and Complete Search approach. For each metric, the following subsections discuss the comparative results.

5.4.3.1 Number of Clones Detected

For each project, the number of detected clones are varied based on the project size and configuration parameters. Figure 5.10 represents the comparison of detected clones. The Complete Search approach detected the maximum number of clones (3,307 clone pairs) in project *cloud9* and the minimum number of clones (13 clone pairs) are detected in project *cocoon*. Figure 5.10 depicts that the proposed approach IDCCD was able to detect almost the same amount of clones in both projects *cloud9* (3,303 clone pairs) and *cocoon* (13 clone pairs). For rest of the projects, the number of detected clones are almost similar for both IDCCD and Complete Search approach. Since Complete Search approach looked for all the candidates for detecting a single clone, detected clones are consider to be true clones. Comparing to Complete Search approach, IDCCD also shows the same performance by detecting the same number of true clones (shown in Figure 5.10).

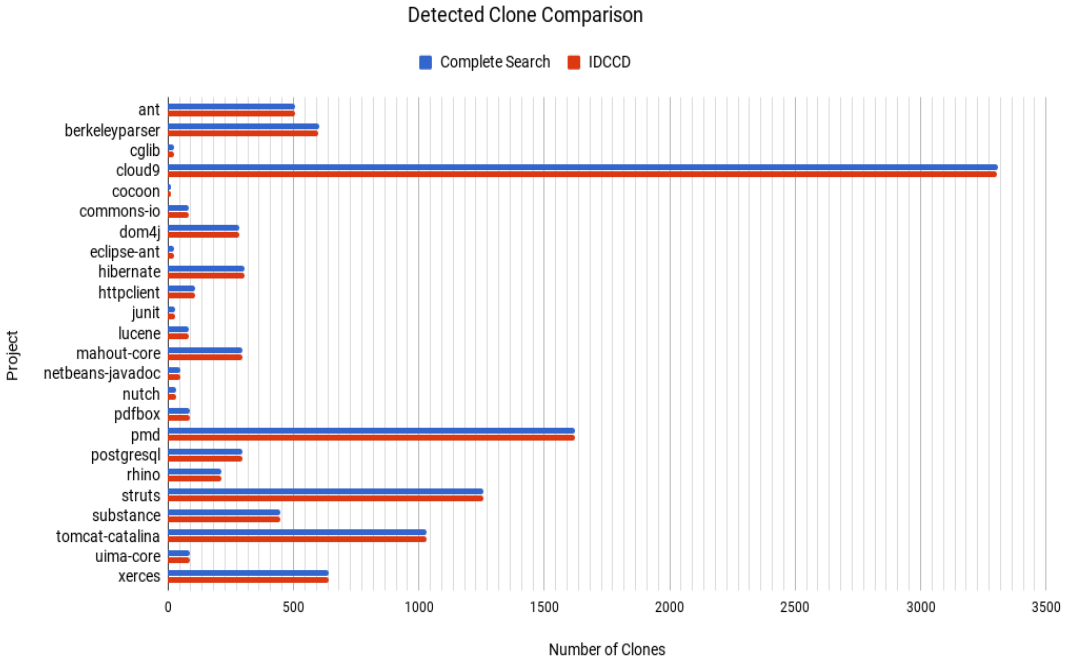


Figure 5.10. Comparison of Detected Clones

5.4.3.2 Number of Candidates Compared

Candidate code fragments refer to those code fragments (in this case method blocks) that are needed to compare for detecting clones. In Complete Search approach all candidate method blocks are compared to each other to detect a single clone. However, in IDCCD, only those candidate method blocks are compared to each other that contain similar interfaces. Figure 5.11 shows number of candidate method block comparison while detecting clones by IDCCD and Complete Search approach. For Complete Search approach, it is observed that the maximum and minimum number of candidate method blocks are compared in project *Apache ant* (3,457,135 candidates compared) and *Junit* (25,425 candidates compared). However, on average 46.43% and 57.48% candidate method block comparison are reduced by IDCCD for these two projects *Apache ant* (1,851,926 candidates compared) and *Junit* (10,810 candidates compared) respectively. Although IDCCD detects same number of clones comparing to Complete Search approach has de-

tected (shown in Figure 5.10). The maximum reduction is done in project *pmd* with 62.50% reduction and the minimum reduction is found in project *cocoon* with reduced 34.45% candidate comparison in detecting clones.

From Figure 5.11, it is also analyzed that considering all projects IDCCD reduced almost 53.04% of the candidates for detecting the same number of clones. This is because, most of the clones occur with similar interfaces. While detecting clones, IDCCD needs not require considering all candidates, rather it performed the comparison with those code fragment containing similar interfaces. Thus, it reduces the number of candidates. So, it infers that comparing to Complete Search approach, IDCCD is able to detect the same number of clones with reduced number of candidate comparisons.

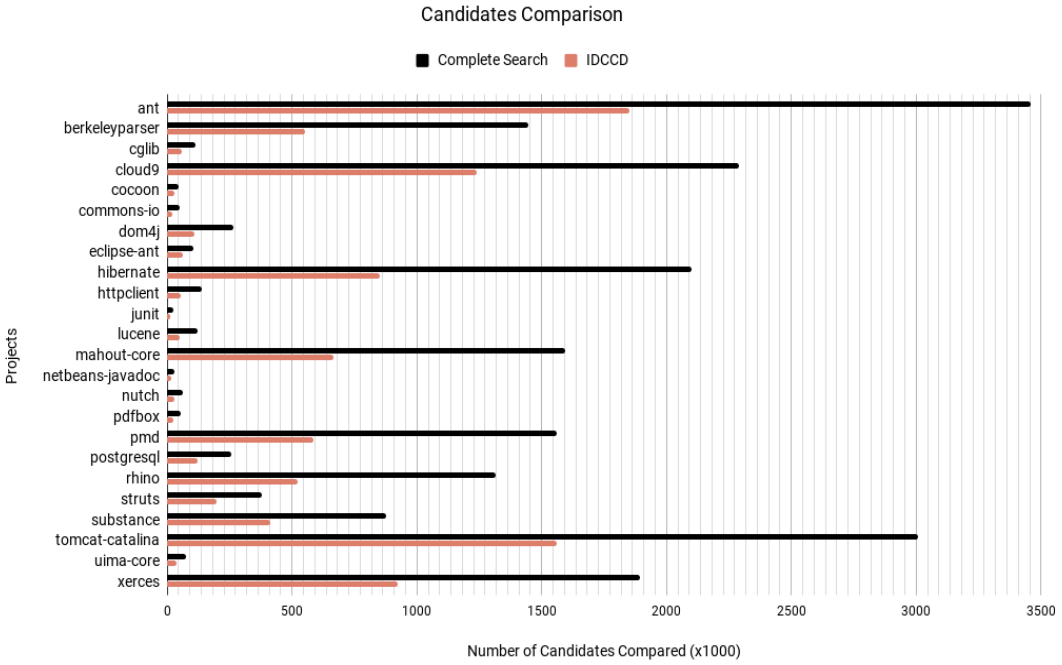


Figure 5.11. Comparison of Candidate Code Fragments

5.4.3.3 Number of Tokens Compared

It is previously discussed (Section 5.2.1) that every candidate method block contains a list of tokens with its frequencies. Therefore, while detecting clones in

Complete Search approach, tokens from each method blocks are required to compare with the other method block's tokens. This is because, to measure the similarity among those tokens, it is needed to compare those tokens with each others. However, in IDCCD, similarity is calculated by using modified Jaccard Similarity (mentioned in Section 5.2.3) where the intersection of unique tokens are calculated based on its minimum frequency.

Figure 5.12 shows a comparative result for token comparison between the Complete Search approach and IDCCD. It represents number of tokens compared while detecting clones by both the Complete Search approach and IDCCD. In case of Complete Search approach, maximum and minimum number of tokens are compared in project *Apache ant* (3,331,811 tokens compared) and project *Junit* (30,817 tokens compared) respectively. However, in order to detect the same number of clones (shown in Figure 5.10) IDCCD reduces 43.12% and 52.88% of the token comparison for those two projects *Apache ant* (1,895,268 tokens compared) and project *Junit* (14,522 tokens compared) respectively. The maximum token comparison reduction 57.64% is found in project *berkeleyparser* and the minimum 30.23% token comparison reduction is done in project *cocoon*. The reason is that, IDCCD only compares candidate method block tokens that contain similar interfaces. As IDCCD needs reduced number of candidate method block comparison to detect clones, the number of token comparison is also reduced. This is because, less candidate method blocks contain less tokens. Therefore, Figure 5.12 depicts that for each project IDCCD performs less token comparison comparing to Complete Search approach.

5.5 Accuracy Measurement

In this section, the evaluation process of IDCCD is provided in detail. Here, recall and precision are used as evaluation metrics. Firstly, the recall of IDCCD is mea-

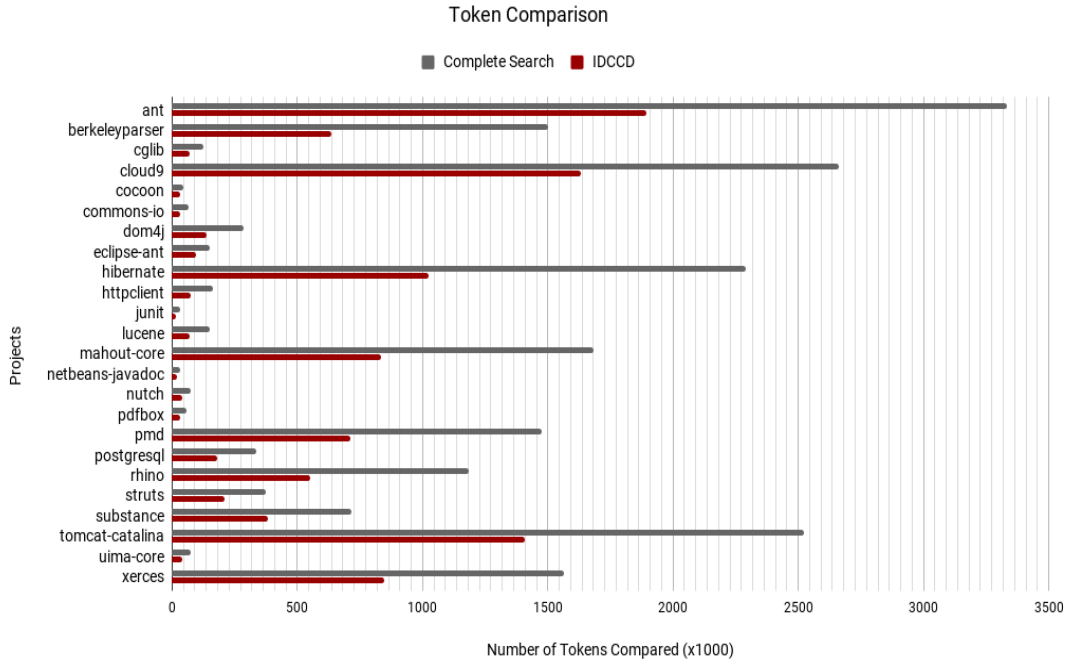


Figure 5.12. Comparison of Tokens

sured using a clone detection benchmark and framework called BigCloneBench [21] and BigCloneEval [22] respectively. The performance of IDCCD is also compared against some popular clone detectors such as CloneWorks [19], SourcererCC [18], NiCad [20] and Deckard [14]. However, the precision of IDDCD is measured by manually validating some statistically sound samples of its output from the recall experiment. Before discussing the precision and recall measurement procedures, a brief description of BigCloneBench [21] including BigCloneEval [22] is described in the following subsections.

5.5.1 BigCloneBench and BigCloneEval

BigCloneBench [21] is a big data benchmark that is developed to calculate the recall and precision of clone detectors in large code repository. BigCloneBench [21] consists of millions of known true and false clones mined from the big data inter-project repository called Big IJaDataset. IJaDataset contains almost 24,558

projects with 365MLOC. With a view to identifying clones of frequently implemented functionalities, BigCloneBench is built by mining targeted functionalities from the IJaDataset. The current version of BigCloneBench contains forty seven tagged functionalities and eight million clones with six million true clones and two million false clones. A detailed description of BigCloneBench can be found in Chapter at 3 Section 3.9.

To make BigCloneBench more flexible researchers have developed a framework called BigCloneEval [22] for evaluating clone detection tools using BigCloneBench. Using BigCloneEval, it is easy to evaluate the recall of clone detectors comparing with the recall of other tools. Like BigCloneBench, BigCloneEval also includes the Big IJDataset repository. All types of clones such as Type-1 (T1), Type-2 (T2) , Type-3 (T3) and Type-4 (T4) are presented in BigCloneEval. However, there is no agreement on, when a clone is no longer syntactically similar. As a result, it is difficult to separate T3 and T4 clones [21]. So, researchers categorized IJDataset's T3 clones into various catagories such as Very Strongly Type-3 (VST3) and Strongly Type-3 (ST3) based on the syntactical similarity. VST3 clones contain 90% syntactic similarity and ST3 clones have 70-90% syntactical similarity. Table 5.1 represents the number of clones is presented in BigCloneBench. A complete description of BigCloneEval is also mentioned in in Chapter 3 at Section 3.9.

Table 5.1. BigCloneEval Clone Summary

Clone Type	T1	T2	VST3	ST3	Intra-Project	Inter-Project
Clone Pairs	48,116	4,234	4,577	9,569	47,852	17,265

5.5.2 Recall Measurement

This section provides a complete description of recall measurement for IDCCD using the BigCloneEval [22]. Recall is measured based on the presence of true clones and is the ratio of true clones that a clone detector is able to detect within a

code repository. In BigCloneEval [22], recall is calculated using Equation 5.2 where B_t represents the true clones existed in the BigCloneEval [22] and D_t exhibits the true clones detected by the IDCCD. The following sections provide the procedure of recall measurement and the outcomes.

$$Recall = \frac{B_t \cap D_t}{B_t} \tag{5.2}$$

5.5.2.1 Procedure

At first, standard configuration parameters are set for IDCCD such as method level granularity with minimum 6 lines and 80% similarity threshold. Next, IDCCD is run in BigCloneEval's VM with an average workstation, for example, 3.26GHz quad-core i7, 8GB ram and 500GB drive. Then recall is calculated for each type of clones. For comparative result analysis, CloneWorks [19], SourcererCC [18], NiCad [20] are also run in BigCloneEval's VM with the same configuration parameters and for all clone detectors the recall measurement report is also generated by BigCloneEval [22]. Finally, the recall of each detector is compared to each other.

5.5.2.2 Result Analysis

Recall measured by BigCloneEval[22] is summarized in Table 5.2. It is summarized intra-project, inter-project, T1, T2 and T3 category including VST3, ST3. IDCCD has perfect detection of T1 clones with 100% recall. For detecting T2 clones, it achieved near perfect recall of 98%. In VST3 and ST3, it gained 95% and 86% recall similar to other tools. For intra-project and inter-project clones IDCCD performs well and is able to gain on average 96% and 94% recall. Comparing to other tools, IDCCD achieves the second best overall 95% recall, with NiCad taking the lead with 99% overall recall (shown in Table 5.3).

However, NiCad [20] used LCS algorithm that can only compare two potential clones at a time. Since each potential clone needs to be compared with others,

Table 5.2. BigCloneEval Recall Measurements

Clone Detectors	All Clones				Intra-Project Clones				Inter-Project Clones			
	T1	T2	VST3	ST3	T1	T2	VST3	ST3	T1	T2	VST3	ST3
IDCCD	100	98	95	86	100	100	96	89	100	99	95	83
CloneWorks	100	99	98	94	100	100	97	95	100	99	98	93
SourcererCC	100	98	93	61	100	99	99	86	100	97	86	48
NiCad	100	100	100	95	100	100	100	99	100	100	100	93
Deckard	60	58	62	31	59	60	76	31	64	58	46	30

making the comparisons using LCS is very expensive. On the other hand, SourcererCC [18] and CloneWorks [19] need overhead calculation and execution time to index sub-block overlapping. In IDCCD, no extra calculation is needed, since the number of similar interfaces in a code repository is smaller than the total number of interfaces [7], the use of query in the interface index reduces the number of candidate method block comparison with lower time complexity comparing to NiCad [11], SourcererCC [18] and CloneWorks [19].

5.5.3 Precision Measurement

In this section the precision of IDCCD is measured and performed its comparison against the existing clone detection tools. Although IDCCD recall is measured by BigCloneEval [22] clone detection framework there is no existing framework for measuring precision. Measuring precision is remained as an open challenge since there is no standard benchmark or methodology. Therefore, typical precision measurement approach is followed to estimate the precision of IDCCD and other clone detection tools, for example, manually validating a random sample of detected clones for each tool. This manual process is also used for evaluating SourcererCC's [18] precision. The same manual process was also executed for IDCCD's precision measurement.

Here, for each tool, arbitrarily 300 sample clone pairs are selected. These sample clones are selected from those clones that are reported by each tool in the recall measurement experiment. These clone pairs are reviewed manually by three

reviewers. Those clones are equally distributed to the reviewers. As a result, each reviewer got 100 clone pairs for each tool to validate it. Moreover, the reviewers are remained blind of the source of those clone pairs. They were only familiar with the clone definition and were asked to validate those clones according to their opinions.

Accumulating the reviewers judgment, it is found that IDCCD's precision is 84%. However, it gets the second best precision where CloneWorks [19] takes the first position with precision 87%. The summary of precision for all tools is shown in Table 5.3 and constructed it with the overall recall reported by BigCloneEval [22]. Here, only the T1, T2, T3, VST3 and ST3 clones are included. IDCCD shows a good balance of recall and precision with the use of interface similarities before comparison process. CloneWorks [19] uses the sub-block filtering with the flexibility of its configuration parameters that helps it to attain the high precision 87%. SourcererCC [18] also achieved 83% precision that is close to CloneWroks [19] and IDCCD's performance. NiCad [11] gets a precision of 56%, this is because it uses normalized and flexible threshold. However, with the same configuration setting, NiCad [11] also has a very strong recall 99%. Deckard [14] reported some clones that are not similar. As a result, it shows poor performance with 28% precision. The reason for its poor performance is the relaxation of its configuration parameters that is used to detect more Type-3 clones. The F-measure value is also calculated by using the over all precision and recall values shown in Figure 5.13.

Table 5.3: Recall and Precision Summary

Clone Detector	Recall	Precision	F-Measure
IDCCD	95	84	89
CloneWorks	98	87	92
SourcererCC	88	83	85
NiCad	99	56	72
Deckard	53	28	37

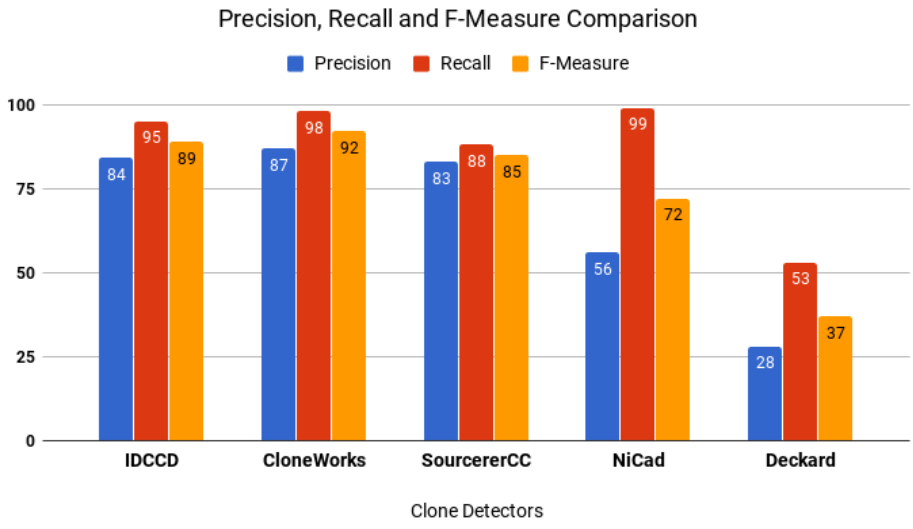


Figure 5.13. Comparison of Precision, Recall and F-Measure

5.5.4 Revisiting the Research Questions

In this section, Research Questions RQ2 and RQ3 are revisited, posed in the Introduction (Chapter 1), and discussed how IDCCDs clone detection approach answers these.

Research Question 2 (RQ2): Can interface similarity minimize the candidate code fragment comparison in code clone detection?

Comparing the code fragments (in this case method blocks) that have similar interfaces reduces the candidate code fragment comparison in IDCCD. While detecting clones, IDCCD first quires into the interface index for retrieving those code fragments containing similar interfaces. As the number of redundant interface is less than the number of total interface [7], the index only provides a few number of candidates. So, the actual comparison is performed among those few candidates. Since IDCCD uses modified Jaccard Similarity as its similarity functions for determining similar code fragments, it needs not to compare all tokens to each other (mentioned in Section 5.2). The empirical evidence of reduction in the number of candidate comparisons is shown in Figure 5.10, 5.11 and 5.12. These evidences infer that, while detecting clones, IDCCD is capable of reducing the candidate

code fragment comparisons based on the interface similarity.

Research Question 3 (RQ3): How accurate the Interface Driven Code Clone Detection approach against the state-of-the-art works?

As mentioned earlier, IDCCD detects clones based on the interface similarities. For a given code fragment (in this case method block), IDCCD retrieves all those code fragments that contain similar interface from the interface index. However, while detecting clones, IDCCD calculates the similarity using modified Jaccard similarity function (mentioned in Section 5.2) from the stream of tokens for those code fragments. It is observed that IDCCD gains an acceptable average recall of 95% comparing to highest recall achiever NiCad [11] with 99%. Although NiCad [11] achieved 99% recall, its precision is not so well with 56% and it needs more candidate code fragment comparison as it uses LCS in its detecting algorithm. On the other hand, precision of IDCCD is 84% that is close to the top precision gainer CloneWorks [19] with 87% (shown in Figure 5.13). These are the evidence that the accuracy of IDCCD is acceptable and near to state-of-the-art clone detection techniques.

5.6 Threats to Validity

The number of clones detected, candidates compared and tokens compared may increase or decrease with the variation of detection parameters used by Interface Driven Code Clone Detection (IDCCD) approach. In an empirical study, Wang et al. [80] have observed that clone detection studies and tools are affected by confounding configuration parameter choice problem. To be a clone detection approach, there is no exception for IDCCD.

While identifying the number of clones detected, candidate compared and tokens compared by the IDCCD and Complete Search approach, the standard configuration parameters are chosen for all clone detectors such as minimum 6 line

with 80% similarity with method level granularity. In measuring the recall and precision by the other clone detectors, the same configurations are also used. This is because, with this configuration those clone detectors provided good results [75] [71] [2].

There are some threats in precision measurement. The subject system (in this case Big IJDataset), configuration parameters and the targeted use cases can have a significant impact on the calculated precision. Besides, the selection of reviewers, the experience and the reliability of reviewers are also an important factor in precision measurement. In order to alleviate this the validation process is distributed into three reviewers. The reviewers were only familiar with the clone definition and remained blind about the source of each clone pair. So, conducting this reviewing process in a blind manner ensured no subjective biasness towards any specific clone detectors.

5.7 Summary

Code clone detection using interface similarity has never been performed before. In this chapter, an Interface Driven Code Clone Detection (IDCCD) approach is developed that can detect clones by using interface information such as method name, return and parameter types. After tokenizing method blocks and indexing its interface information, IDCCD compares those method blocks that have similar interfaces. A pair of method blocks are reported as clone satisfying a similarity threshold. To verify whether IDCCD is capable of reducing the candidate comparison, a candidate comparison minimization experiment was conducted. The experimental results show that comparing to Complete Search approach IDCCD detects same amount of clone with reduced candidate comparison and token comparison. Besides, the accuracy of IDCCD is measured by the clone evaluation framework BigCloneEval. The evaluation result represents IDCCD gained 95%

recall and 84% precision that is close to other clone detector's performance. The summary of this research work is discussed in the next chapter.

Chapter 6

Conclusion

Identical code fragments in a code repository are known as code clones. Because of many issues such as propagating bug, increasing workload etc. code clone detection is necessary for software maintenance and evolution. A rudimentary approach of clone detection is to compare each code fragment to each other with similarity function but this approach is not effective for large code repository as it takes polynomial time complexity. Sometimes, it is observed that code fragments are repeated with similar interfaces such as similar return type, method name and parameter types. Two code fragments with similar interfaces indicate that they should perform analogous behavior and contain similar logical implementation. It states that code clones and interface similarities are related to each other. If this relationship is strong enough, it can be beneficial to code clone detection. This chapter describes the summary of the relationship between code clones and interface similarity. Afterwards, how this relationship reduces the candidate code fragment comparison are also discussed and the accuracy of Interface Driven Code Clone Detection approach is also summarized. Finally, this chapter concludes the research work by outlining the possible future directions.

6.1 Relationship between Code Clones and Interface Similarity

A preliminary approach for code clone detection is to compare each code fragments to each other using similarity functions such as Cosine similarity, Jaccard Similarity etc. For such pair of code fragments, if the similarity is high that pair of code fragments is considered to be a code clone. This approach is also known as Complete Search approach. The main drawback of this approach is that it causes a prohibited polynomial comparison which is not efficient for detecting clones in large code repository.

Apart from code clones, another replication, redundant or similar interface, can be occurred in large code repositories. It refers the similarity of interface information such as return type, method name and parameter types. It is found that in a large code repository almost 25% interfaces are similar and appeared as redundant [7]. Besides, there is a hypothesis that when two methods contain similar interfaces it is expected that those methods contain same logical implementation and perform similar behavior. If those methods have similar interfaces and exhibit same implementation, those are considered to be clone. So, it express that there is a relationship between code clones and interface similarity.

To establish that relationship, an exploratory study is conducted on various subject systems containing 35, 109 and 24,558 open source Java projects respectively. Two popular clone detectors NiCad [11] and SourcererCC [18] are used to identify all types of clones such as intra-project, inter-project, Type-1, Type-2 and Type-3 in each subject systems. The total number of clones found in those subject systems are 231411, 242992 and 130226 respectively. After that, interface information such as return type, method name and parameter types of those clones is extracted from the source code. Finally, interface similarity is measured in those method clones by satisfying similarity conditions.

The experimental result shows that on average 79.65% intra-project clones and 69.44% inter-project clones contain similar interface, for example, similarity of return type, at least one keyword and at least one parameter type. Besides, 100% Type-1, 69.35% Type-2 and 67.34% Type-3 clone contain similar interfaces. However, use of inappropriate naming convention, generic type and type mismatch problem prevent some clones satisfying interface similarities. The findings also help to design new interface driven code clone detection tool with reduced candidate comparison.

6.2 Candidate Comparison Reduction based on Interface Similarity

Based on the relationship between code clones and interface similarities an Interface Driven Code Clone Detection (IDCCD) approach is proposed. To verify whether IDCCD reduces the candidate code fragment comparison while detecting clones, a candidate comparison minimization experiment was performed. In first step, randomly twenty four projects are chosen from thirty five Apache Java projects. For each project, IDCCD and Complete Search approach were run with standard configuration parameters such as method level granularity with minimum 6 lines and 80% similarity threshold. Three types of metrics, for example number of clones detected, number of candidates compared and number of tokens compared are calculated for both approaches.

It is observed that both Complete Search approach and IDCCD detected the same number of clones in those projects. However, compared to Complete search approach IDCCD overall reduces 53.04% candidate comparison detecting the same number of clones. Similarly IDCCD also reduces the token comparison on average 46.73%. This is because, instead of considering all code fragments, IDCCD only compares those code fragments that have similar interfaces. As it reduces the

number of candidate comparison, the number of token comparison also reduces since each candidate contain a list of unique tokens and less candidates contain less tokens. So, it infers that compared to Complete Search approach, IDCCD reduces both candidate comparison and token comparison during clone detection.

6.3 Accuracy of Interface Driven Code Clone Detection

Candidate comparison minimization experiment exhibits that Interface Driven Code Clone Detection (IDCCD) approach reduces the comparison among the code fragments compared to Complete Search approach. However, the accuracy of IDCCD is also required to evaluate against the existing clone detectors. In order to evaluate the performance of IDCCD, precision and recall are calculated as evaluation metrics. To measure the recall of IDCCD against the popular clone detectors, a clone evaluation framework BigCloneEval [22] is used. BigCloneEval includes a becnhmark called BigCloneBench [21] uses a inter-project dataset Big IJDataset. It has above 24,558 Java projects with and 365MLOC. BigCloneEval contains eight millions true clones and two millions false clones that are used to automatically calculate clone detectors recall. IDCCD and other clone detectors were run in BigCloneEval's VM with the standard configuration parameters. The clone reports show that IDCCD attains on average 95% recall and attain the second best position taking lead the NiCad with 99% recall.

Precision is calculated by manually validating a sample of clones from those clones detected in recall measurement. For IDCCD and each clone detectors, 300 sample of clone pairs are manually validated by three reviewers where each reviewer got 100 clone pairs. Considering the reviewers judgment, IDCCD gain 84% precision taking lead to CloneWorks [19] with 87% precision.

6.4 Future Direction

In this research work, an Interface Driven Code Clone Detection approach is developed based on the similarity between code clones and interfaces. However, there have some future scopes that can be performed to improve its effectiveness. The following points address the potential future directions derived from this work.

- Since IDCCD reduces candidate code fragment comparison and token comparison, it can be developed as a scalable code clone detection technique. Testing its scalability against other scalable clone detection techniques is a possible future work.

- Code clone and interface similarity relation can be used to detect semantic clones (Type-4). Semantic clones express similar behaviors with different logical implementation. As similar interface may exhibit analogous behaviors, semantic clones can be detected through testing two code fragments with identical random inputs. Analyzing the outputs, their behavior can be compared that expresses whether those code fragments are semantic clones or not.

Although, the relationship between clone and interface similarity ameliorate clone detection techniques through IDCCD, this relationship can be beneficial to other research works.

- The relationship between code clones and interface similarity can be beneficial to improving the Interface Driven Code Search (IDCS) approaches since it searches similar code fragments based on its interface information.

- Interface Driven Code Clone detection approach can be applied to detect cross-language clone detection. Various strongly typed languages such as Java and C#, contain similar syntactical programming style. So, it is possible to occur interface similarity in a code fragment that is implemented

using both Java and C#. So, using the interface similarity, cross-language clone can be detected.

- Since IDCCD detects clones with reduced candidate comparison, it can be capable of detecting clones with minimum time complexity. So, this approach may be applicable for real time clone tacking and monitoring.

Appendix A

Small Subject System

Project	Java Files	LOC	Number of Methods	Number Of Clones
ant	690	86421	3113	718
berkeleyparser	314	57905	2033	693
cglib	194	13668	539	27
cloud9	464	56766	2376	3364
cocoon	84	10377	376	20
commons-io	103	8673	410	81
dom4j	166	17854	853	492
eclipse-ant	178	16103	522	23
eclipse-jdtcore	741	98111	3931	3952
hadoop-hdfs	341	69990	2857	3804
hadoop-mapred	524	64018	2232	339
hibernate	820	72387	2429	363
httpclient	289	18022	696	120
j2sdk1.4.0-javax-swing	538	102835	4698	9488
jfreechart	594	93461	3622	7234
junit	175	6728	274	26

Project	Java Files	LOC	Number of Methods	Number Of Clones
log4j	207	20611	1030	2196
lucene	160	15670	610	94
mahout-core	704	53366	1998	309
mason	302	35934	1615	2369
netbeans-javadoc	101	9580	316	50
nutch	153	12243	408	39
pdfbox	124	13936	384	65
pig	641	84770	2824	3918
pmd	749	59978	2106	3273
poi	380	47780	1837	5508
postgresql	187	23507	832	309
rhino	210	55070	1825	221
stanford-nlp	752	210222	5122	1475
struts	277	24755	1081	1882
substance	360	47361	1528	456
synapse-core	468	41612	1212	242
tomcat-catalina	376	73397	2650	1009
uima-core	141	11942	487	214
xerces	615	76184	2280	732

Appendix B

Medium Subject System

Project	Java Files	LOC	Project	Java Files	LOC
1_tullibee	20	3236	26_jipa	3	392
2_a4j	45	3602	27_gangup	95	11088
3_gaj	14	320	28_greencow	1	6
4_rif	15	953	29_apbsmem	50	4404
5_templateit	19	2463	30_bpmail	37	1681
6_jnfe	68	2096	31_xisemele	56	1805
7_sfmis	19	1288	32_httpanalyzer	19	3580
8_gfarcegestionfa	50	3662	33_javaviewcontrol	17	4617
9_falselight	8	372	34_sbmlreader2	6	498
10_water-simulator	49	6502	35_corina	349	41290
11_imsmart	20	1039	36_schemaspy	72	9974
12_dsachat	32	3860	37_petsoar	76	2255
13_jdbacl	126	18404	38_javabullboard	44	8295
14_omjstate	23	593	39_diffi	10	524
15_beanbin	88	3788	40_glengineer	35	2990
16_templatedetails	1	391	41_follow	60	4812
17_inspirento	36	2427	42_asphodel	24	691
18_jsecurity	296	13133	43_lilith	295	46188
19_jmca	25	14882	44_summa	584	69314
20_nekomud	10	363	45_lotus	54	1028
21_geo-google	62	6623	46_nutzenportfolio	84	8268
22_byuic	12	5868	47_dvd-homevideo	9	2911
23_jwbf	69	5848	48_resources4j	14	1242
24_saxpath	16	2619	49_diebierse	20	1888
25_jni-inchi	24	1156	50_biff	3	2097

Project	Java Files	LOC	Project	Java Files	LOC
51_jiprof	113	13950	81_javathena	53	10492
52_lagoon	81	9954	82_ipcalculator	10	2684
53_shp2kml	4	266	83_xbus	203	23507
54_db-everywhere	104	7125	84_ifx-framework	3872	117324
55_lavalamp	54	1474	85_shop	34	3886
56_jhandballmoves	73	5345	86_at-robots2-j	231	9459
57_hft-bomberman	135	8397	87_jaw-br	30	4851
58_fps370	8	1506	88_jopenchart	48	3996
59_mygrid	37	3317	89_jiggler	184	20072
60_sugar	37	3120	90_dcparseargs	6	204
61_noen	408	18867	91_classviewer	7	1467
62_dom4j	173	18209	92_jcvi-javacommon	619	45495
63_objectexplorer	88	6988	93_quickserver	152	16040
64_jtailgui	44	2020	94_jclo	3	387
65_gsftp	17	2327	95_celwars2009	11	2876
66_openjms	624	54879	96_heal	184	22521
67_gae-app-manager	8	411	97_feudalismgame	36	3515
68_biblestudy	21	2265	98_trans-locator	5	357
69_lhamacaw	108	22750	99_newzgrabber	39	5868
70_echodep	81	15396	100_jgaap	17	1009
71_ext4j	45	1892	101_netweaver	204	24380
72_battlecry	11	2522	102_squirrel-sql	1151	122843
73_fim1	70	10294	103_sweethome3d	185	68703
74_fixsuite	25	2665	104_vuze	3303	517165
75_openhre	135	8355	105_freemind	468	62141
76_dash-framework	22	241	106_checkstyle	169	19573
77_io-project	19	698	107_weka	1031	243787
78_caloriecount	684	61542	109_pdfsam	369	35904
79_twfbplayer	104	7240	110_firebird	258	40004
80_wheelwebtool	113	16275			

Appendix C

Detected Clone Comparison

Project	Complete Search	IDCCD	None Detected
cloud9	3307	3303	4
pmd	1622	1622	0
struts	1257	1257	0
tomcat-catalina	1031	1029	2
xerces	639	639	0
berkeleyparser	601	597	4
ant	508	508	0
substance	449	449	0
hibernate	307	307	0
mahout-core	297	297	0
postgresql	297	297	0
dom4j	284	283	1
rhino	212	211	1
httpclient	109	109	0
uima-core	88	88	0
pdfbox	86	86	0
lucene	82	82	0
commons-io	81	81	0
netbeans-javadoc	47	47	0
nutch	31	31	0
junit	30	30	0
eclipse-ant	24	23	1
cglib	23	23	0
cocoon	13	13	0

Appendix D

Candidate Comparison

Project	Complete Search	IDCCD	Reduced Candidate Comparison	Reduction
ant	3457135	1851926	1605209	46.43%
tomcat-catalina	3007378	1561091	1446287	48.09%
cloud9	2290870	1238172	1052698	45.95%
hibernate	2102275	852997	1249278	59.43%
xerces	1892485	924898	967587	51.13%
mahout-core	1594005	664620	929385	58.31%
pmd	1560261	585164	975097	62.50%
berkeleyparser	1445850	550852	894998	61.90%
rhino	1314631	522283	792348	60.27%
substance	875826	410903	464923	53.08%
struts	378885	197091	181794	47.98%
dom4j	263175	106824	156351	59.41%
postgresql	255255	121962	133293	52.22%
httpclient	138601	55097	83504	60.25%
lucene	119805	49942	69863	58.31%
cglib	111628	56568	55060	49.32%
eclipse-ant	102831	61212	41619	40.47%
uima-core	76245	37681	38564	50.58%
nutch	60726	29249	31477	51.83%
pdfbox	53628	24524	29104	54.27%
commons-io	50403	20476	29927	59.38%
cocoon	45150	29596	15554	34.45%
netbeans-javadoc	30135	16855	13280	44.07%
junit	25425	10810	14615	57.48%

Appendix E

Token Comparison

Project	Complete Search	IDCCD	Reduced Token Comparison	Reduction
ant	3331811	1895268	1436543	43.12%
cloud9	2661963	1632930	1029033	38.66%
tomcat-catalina	2524047	1408799	1115248	44.18%
hibernate	2290597	1023638	1266959	55.31%
mahout-core	1680479	832204	848275	50.48%
xerces	1565048	847869	717179	45.82%
berkeleyparser	1499580	635236	864344	57.64%
pmd	1475746	711012	764734	51.82%
rhino	1183407	549993	633414	53.52%
substance	715620	380679	334941	46.80%
struts	372813	208545	164268	44.06%
postgresql	334041	181179	152862	45.76%
dom4j	284864	135768	149096	52.34%
httpclient	163987	75844	88143	53.75%
eclipse-ant	148908	96023	52885	35.52%
lucene	148726	68983	79743	53.62%
cglib	124230	69363	54867	44.17%
nutch	75099	38867	36232	48.25%
uima-core	73221	41792	31429	42.92%
commons-io	64369	31099	33270	51.69%
pdfbox	58561	30530	28031	47.87%
cocoon	46681	32568	14113	30.23%
netbeans-javadoc	33757	20724	13033	38.61%
junit	30817	14522	16295	52.88%

Bibliography

[1] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "RCV1: A new benchmark collection for text categorization research," *Journal of Machine Learning Research*, vol. 5, pp. 361–397, 2004.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[3] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, pp. 81–90, 2008.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[5] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pp. 18–33, 2014.

[6] D. Rattan, R. K. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information & Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[7] A. C. de Paula, E. Guerra, C. V. Lopes, H. Sajnani, and O. A. L. Lemos, "An exploratory study of interface redundancy in code repositories," in *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pp. 107–116, 2016.

[8] H. Sajnani, *Large-Scale Code Clone Detection*. PhD thesis, University of California, Irvine, USA, 2016.

[9] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[10] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pp. 107–114, 2001.

[11] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pp. 219–220, 2011.

[12] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *2nd Working Conference on Reverse Engineering, WCRE '95, Toronto, Canada, July 14-16, 1995*, pp. 86–95, 1995.

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

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 96–105, 2007.

[15] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, pp. 40–56, 2001.

[16] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, pp. 301–309, 2001.

[17] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, p. 244, 1996.

[18] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 1157–1168, 2016.

[19] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pp. 27–30, 2017.

[20] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pp. 172–181, 2008.

[21] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 476–480, 2014.

[22] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pp. 596–600, 2016.

[23] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[24] F. Khomh, M. D. Penta, and Y. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 75–84, 2009.

[25] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, "Beyond templates: a study of clones in the STL and some general implications," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pp. 451–459, 2005.

[26] J. R. Cordy, "Comprehending reality - practical barriers to industrial adoption of software maintenance automation," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pp. 196–206, 2003.

[27] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pp. 289–302, 2004.

[28] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pp. 368–377, 1998.

[29] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *8th IEEE International Software Metrics Symposium (METRICS 2002), 4-7 June 2002, Ottawa, Canada*, p. 87, 2002.

[30] U. Manber, "Finding similar files in a large file system," in *USENIX Winter 1994 Technical Conference, San Francisco, California, January 17-21, 1994, Conference Proceedings*, pp. 1–10, 1994.

[31] H. T. Jankowitz, "Detecting plagiarism in student pascal programs," *Comput. J.*, vol. 31, no. 1, pp. 1–8, 1988.

[32] E. Burd and M. Munro, "Investigating the maintenance implications of the replication of code," in *ICSM*, p. 322, 1997.

[33] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries - an empirical study on 13, 000 projects," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pp. 387–391, 2012.

[34] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "SHINOBI: A tool for automatic code clone detection in the IDE," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 313–314, 2009.

[35] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pp. 23–27, 2011.

[36] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland, "Shuffling and randomization for scalable source code clone detection," in *Proceeding of the 6th International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012*, pp. 82–83, 2012.

[37] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pp. 309–318, 2012.

[38] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pp. 357–366, 2012.

[39] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle, "Software bertillonage: finding the provenance of an entity," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pp. 183–192, 2011.

[40] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pp. 58–64, 2006.

[41] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pp. 201–210, 2015.

[42] J. Yang, Y. Jiang, A. G. Hauptmann, and C. Ngo, "Evaluating bag-of-visual-words representations in scene classification," in *Proceedings of the 9th ACM SIGMM International Workshop on Multimedia Information Retrieval, MIR 2007, Augsburg, Bavaria, Germany, September 24-29, 2007*, pp. 197–206, 2007.

[43] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 40, no. 3, pp. 211–218, 2006.

[44] J. B. Lovins, "Development of a stemming algorithm," *Mech. Translat. & Comp. Linguistics*, vol. 11, no. 1-2, pp. 22–31, 1968.

[45] W. B. Frakes and C. J. Fox, "Strength and similarity of affix removal stemming algorithms," *SIGIR Forum*, vol. 37, no. 1, pp. 26–30, 2003.

[46] P. Willett, "The porter stemming algorithm: then and now," *Program*, vol. 40, no. 3, pp. 219–223, 2006.

[47] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in action: covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[48] M. Borg, P. Runeson, J. Johansson, and M. Mäntylä, "A replicated study on duplicate detection: using apache lucene to search among android defects," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pp. 8:1–8:4, 2014.

[49] B. Milosavljevic, D. Boberic, and D. Surla, "Retrieval of bibliographic records using apache lucene," *The Electronic Library*, vol. 28, no. 4, pp. 525–539, 2010.

[50] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, pp. 109–118, 1999.

[51] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005), 25-29 September 2005, Timisoara, Romania*, pp. 63–70, 2005.

[52] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, October 24-28, 1993, Toronto, Ontario, Canada, 2 Volumes*, pp. 171–183, 1993.

[53] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 3, 1994, Toronto, Ontario, Canada*, p. 32, 1994.

[54] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, pp. 120–126, 1994.

[55] V. Saini, H. Sajnani, J. Kim, and C. V. Lopes, "Sourcerercc and sourcerercci: tools to detect clones in batch mode and during software development," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pp. 597–600, 2016.

[56] J. Svajlenko and C. K. Roy, "Cloneworks: a fast and flexible large-scale near-miss clone detection tool," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pp. 177–179, 2017.

[57] W. T. Cheung, S. Ryu, and S. Kim, "Development nature matters: An empirical study of code clones in javascript applications," *Empirical Software Engineering*, vol. 21, no. 2, pp. 517–564, 2016.

[58] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Autom. Softw. Eng.*, vol. 3, no. 1/2, pp. 77–108, 1996.

[59] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, p. 96, Citeseer, 2012.

[60] W. B. Frakes and B. A. Nejmeh, "Software reuse through information retrieval," *SIGIR Forum*, vol. 21, no. 1-2, pp. 30–36, 1987.

[61] S. P. Reiss, "Semantics-based code search," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pp. 243–253, 2009.

[62] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher, "Codegenie: : a tool for test-driven source code search," in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pp. 917–918, 2007.

[63] A. M. Zaremski and J. M. Wing, "Signature matching: A tool for using software libraries," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 2, pp. 146–170, 1995.

[64] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 905–908, 2006.

[65] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, 2014.

[66] A. Satter and K. Sakib, "A similarity-based method retrieval technique to improve effectiveness in code search," in *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*, pp. 39:1–39:3, 2017.

[67] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Comput. Surv.*, vol. 44, no. 1, pp. 1:1–1:50, 2012.

[68] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pp. 212–221, 2014.

[69] M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos, "An exploratory study of functional redundancy in code repositories," in *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*, pp. 31–40, 2017.

[70] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[71] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pp. 321–330, 2014.

[72] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pp. 157–166, 2009.

[73] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *Proceeding of the 7th International Workshop on Software Clones, IWSC 2013, San Francisco, CA, USA, May 19, 2013*, pp. 8–9, 2013.

[74] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pp. 131–140, 2015.

[75] H. Sajnani, V. Saini, and C. V. Lopes, "A comparative study of bug patterns in java cloned and non-cloned code," in *14th IEEE International Working*

Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014, pp. 21–30, 2014.

[76] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pp. 178–188, 2012.

[77] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?," in *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, pp. 41–50, 2015.

[78] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, 2014.

[79] M. A. Finlayson, "Java libraries for accessing the princeton wordnet: Comparison and evaluation," in *Proceedings of the Seventh Global Wordnet Conference, GWC 2014, Tartu, Estonia, January 25-29, 2014*, pp. 78–85, 2014.

[80] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 455–465, 2013.